A Type-Theoretic Approach to Smart-Contract Safety

Stian Lybech

Dissertation submitted to the Department of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

March 7, 2025

Thesis Committee:

Luca Aceto, Supervisor
Professor, Reykjavík University, Iceland

Mohammad Hamdaqa, Supervisor
Assistant Professor, Polytechnique Montreal, Canada

Daniele Gorla, External supervisor
Associate Professor, Sapienza University of Rome, Italy

Silvia Crafa, Thesis committee member
Associate Professor, University of Padova, Italy

Ettore Merlo, Thesis committee member
Professor, Polytechnique Montreal, Canada

Laura Bocchi, Examiner
Reader, University of Kent, UK

Maurizio Murgia, Examiner
Assistant Professor, Gran Sasso Science Institute, Italy

ISBN 978-9935-539-60-1 · printed version
ISBN 978-9935-539-61-8 · electronic version
0000-0001-8219-2285 · Stian Lybech

A Type-Theoretic Approach to Smart-Contract Safety

Stian Lybech

Abstract

Type systems are routinely employed in many modern programming languages to statically ensure various notions of runtime safety. We explore issues of typability and notions of safety in two different fields:

Firstly, we focus on process calculi with composite channel names, where the type of a channel must somehow be derived from the types of its constituents. This collection of results includes a simple type system for the ρ-calculus, along with some results of expressivity w.r.t. the π-calculus; a generic type system for the Higher-Order Ψ-calculus, extending a similar type system for the 'first-order' Ψ-calculus; and a simple type system for ${}^{e}\pi$, which aims to highlight a connexion to type structures from class-based/object-oriented languages.

Secondly, we focus on the language TinySol, which models core features of the smart-contract language Solidity. Smart contracts are immutable programs with publicly visible code, that run atop a blockchain and are used to manage financial assets of users. Guided by insights from our work in process calculi, we develop type systems for ensuring three different properties: non-interference, call-integrity, and absence of out-of-gas exceptions.

Lastly, we seek to tackle some of the shortcomings of the conventional, syntactic approach to type soundness, which had become evident in our previous developments. In particular, we study a peculiar construct in Solidity, known as the *fallback function*, which is untypable by syntactic type rules. Hence, we turn to a *semantic* approach to type soundness which allows type safety to be shown, even in cases where well-typedness cannot be proved by ordinary syntactic type rules. We use this approach to propose a method by which type safety may be recovered, even for contracts containing fallback functions, by allowing the programmer to supply a manual proof of type-safety for untypable pieces of code. This method does not depend on specific features of the fallback function, or even of TinySol or Solidity, and it may therefore also be developed for other smart-contract languages.

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

T.S. Eliot, *Four Quartets*

Acknowledgments

This dissertation represents the conclusion of a journey that began nine years ago, when I decided to leave Greenland and move to Denmark to study computer science, and thence to Iceland for my PhD. I would like to express my gratitude towards a selection of people, who have helped me along the way:

- To Hans Hüttel, who first introduced me to operational semantics, type systems and process calculi, and who inspired me to pursue a PhD within these related fields.
- To Luca Aceto, my main supervisor, who always was ready with advice, support and an abundance of constructive comments on my writings (which were often sent in the oddest of hours).
- To Mohammad Hamdaqa, my secondary supervisor, for bringing me on to the project, and for trusting my idea to take a type theoretic approach.
- To Daniele Gorla, for our great collaboration, which carried this project much further than I think, I would otherwise have been able to go.

Thanks also to my friend, Jordian Farahani, for our late-night talks, for your encouragement and optimism, and your contagious belief that everything will work out in the end. And to Ann-Sophie Bader and Luisa Vogel for all our adventures, great and small, when I most needed a break from the seemingly endless proofs. You made me feel at home in Iceland.

The research presented in this dissertation was supported by the Icelandic Research Fund, grant no. 218202-05(1-3). I also received financial support from the Department of Computer Science at Reykjavik University during the final months of writing of this thesis.

Publications

The thesis contains the published papers listed below. They appear as Chapters 3, 4, 6 and 7. Some of them are journal versions of shorter papers that have previously appeared elsewhere. Details about the publication history of each chapter are given in a chapter footnote following its title.

- Stian Lybech. The Reflective Higher-Order Calculus: Encodability, Typability and Separation. *Information and Computation*, 297:105138, 2024. ISSN 0890-5401. doi:10.1016/j.ic.2024.105138.
  Contribution: author of the paper.

- Hans Hüttel, Stian Lybech, Alex R. Bendixen, and Bjarke B. Bojesen. A Generic Type System for Higher-Order Ψ-calculi. *Information and Computation*, page 105190, 2024. ISSN 0890-5401. doi:10.1016/j.ic.2024.105190.
  Contribution: co-author of the paper (all sections and proofs) and principal author of section 5.3 on the ρ-calculus.

- Luca Aceto, Daniele Gorla, and Stian Lybech. A Sound Type System for Secure Currency Flow. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:27, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-341-6. doi:10.4230/LIPIcs.ECOOP.2024.1.
  Contribution: principal author of the paper.

- Luca Aceto, Daniele Gorla, Stian Lybech, and Mohammad Hamdaqa. Preventing Out-of-Gas Exceptions by Typing. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. REoCAS Colloquium in Honor of Rocco De Nicola*, pages 409–426, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-73709-1. doi:10.1007/978-3-031-73709-1_25.
  Contribution: principal author of the paper.

The thesis furthermore contains the following, currently unpublished, material, appearing in in Chapters 5 and 8:

- Luca Aceto, Daniele Gorla, and Stian Lybech. Typing composite subjects. 2024. URL https://arxiv.org/abs/2411.13732.
  Contribution: principal author of the paper.
- Stian Lybech. *A Semantic Approach to Security-Type Safety*.
  Contribution: Author of the chapter.

Contents

Part I

Preliminaries

1 Introduction

Type systems are a cornerstone in the field of static analysis techniques for ensuring program safety. Within the discipline of *data types*, the notion of safety relates to properties of the data that flow through the program and are stored in its memory. For example, in an imperative setting, a typical property could state: '*the value stored in variable x is always one from set $\mathcal{V}$.*' We express such properties by using *types* to denote the different sets of values, and by assigning types to the various syntactic constructs that may *contain* such values. We then judge a program to be *safe* w.r.t. the restrictions imposed by our type assignments, if all the properties remain true for all execution steps of the program. A *type system* is then a proof system, consisting usually of syntax-directed inference rules, by which we can statically decide, i.e. without actually *running* the program, whether it in fact *will* obey the type restrictions for all its execution steps.

Type systems have proved immensely useful in preventing program errors, and many modern programming languages are equipped with a type system that allows the programmer to statically check his programs in various ways, such as preventing unwanted data flows, incorrect data conversions, unauthorised data access, shared pointers, and many other forms of runtime errors. However, type systems seem particularly relevant in a setting where such errors can have catastrophic consequences, and where updating the program after deployment to correct the errors would be difficult or outright impossible. Such is for example the case in the setting of *smart contracts*, which are immutable programs that run atop a blockchain and are used to manage financial assets of users, including storing, transferring, receiving, and controlling access to these assets. In the present thesis we investigate how type systems may be used to ensure program safety in such a setting.

1.1 Blockchains and smart contracts

Although our ultimate goal is to use type systems to ensure safety properties of smart contracts, we shall work with a variety of languages, not all of which are directly related to smart-contract languages. Thus, detailed knowledge of the blockchain architecture is not required for understanding our results in general, and the necessary

details will be mentioned where needed. However, as the field of blockchains and smart contracts is relatively new, compared to the other programming models we consider, we shall provide a brief overview here at the outset.

A *blockchain* is an immutable, distributed data structure, which first and foremost is used to implement digital currency; see e.g. [116] for an overview of the architecture. The original blockchain platform, called *Bitcoin*, was described in a whitepaper [87] from 2009, authored by one or more unknown persons writing under the pseudonym 'Satoshi Nakamoto.' It came in the wake of the financial crisis of 2007, which had highlighted the fragility of a financial system, where trust is based on central authorities (e.g. banks) to verify transactions. As an alternative, Nakamoto proposed a decentralised (peer-to-peer based) system, where the users themselves are collectively responsible for verifying transactions. It was named *blockchain*, because validated transactions are collected into blocks, with each block containing a pointer to the previous block, similar to a linked list, and the structure thus contains the entire validated transaction history of the network.

A blockchain network can abstractly be viewed as a replicated state machine, where the current state records the distribution of assets (currency) amongst users of the network, and each participating node in the network executes a transition function f, updating the state s_i to state s_{i+1}. Most importantly, this implies two key features, which are characteristic of blockchains; namely *immutability* and *public visibility*. The entire transaction history is visible to all users, which also allows them to validate the steps that produced the current state; and once a block has been validated and has become part of the chain, it cannot be changed.

The state update can be simple currency transfers between users of the network, but, additionally, many newer blockchain platforms also allow transactions to invoke *code* supplied by the users, in the form of *smart contracts*. The idea of smart contracts predates the blockchain technology, as it goes back to 1997, to Nick Szabo [120], who proposed that features of ordinary 'pen and paper' legal contracts could be replaced by computer programs. The idea is to provide, in programmatic form, the equivalent of a legally binding contract, yet without requiring a legal system, or some other third-party trusted authority, to enforce its execution. Instead, the smart contract simply executes itself (or, in the context of blockchains, is executed by all validator nodes in the network as part of the state transition) when the conditions in the contract are met.

The original proposal of smart contracts by Szabo does not mention any specific platform on which they might be implemented, but with the advent of blockchain technology, with its focus on anonymity and verification of transactions in a 'trustless' environment, a form of smart contracts eventually came to be added as a layer on top of several blockchain platforms. In particular, the Ethereum platform introduced a Turing-complete, low-level, imperative language called EVM (Ethereum Virtual Machine language), and a high-level, Java-like language called Solidity [41]. Since then, a plethora of other blockchain platforms have appeared, each with their own,

particular smart-contract language; see [14] for an overview and comparison of the six most well-known languages. However, Solidity remains the language that has seen the most widespread adoption.

A core tenet of blockchain technology is to minimise the needed level of *trust* between users; thus users should not be assumed to trust each other, nor the programs (smart contracts) themselves. The code of a smart contract is therefore deployed onto the blockchain itself, simultaneously rendering it immutable and publicly visible. The benefit of this approach is twofold:

1. Due to public visibility, any user can (at least in principle) verify for himself that the contract will do exactly what he expects it to do; no more and no less. He does not need to trust any claims made by the creator of the contract regarding its behaviour.

2. Due to immutability, the user can be assured that the contract creator cannot suddenly change the contract. Thus, if the user has already verified the behaviour of a contract once, he will not need to do it again at a later stage.

However, the combination of public visibility and immutability is a doubled-edged sword:

1. Public visibility of the code means that any *vulnerabilities* in a contract may be easily found and exploited by malicious users.

2. Immutability of the code means that the contract creator cannot create and deploy a *fix* for a vulnerability, even if it is known.[1]

This combination has resulted in large financial losses in the past (see e.g. [77] for a list of examples on the Ethereum platform). Hence, it is clearly desirable that a smart contract should be safe to execute *before* it is deployed onto the blockchain.

Not surprisingly, this situation has spurred a substantial research effort within the formal methods community on developing formal techniques to prove safety properties of smart contracts; see [124; 115] for comprehensive surveys. The literature on the subject is vast, with a multitude of approaches being applied. This includes model checking and writing contracts in specialised modelling languages, that compile to the desired target language [28; 29; 95; 21; 13; 39]; specialised static-analysis tools [40; 68; 114; 125] and formal models of the semantics of the smart-contract language [51; 55; 66]; verification using theorem provers [56; 78]; and smart-contract synthesis [79; 31]. The list is by no means exhaustive.

[1] Some blockchain architectures, such as Ethereum, have features that do allow a contract to be updated, for example to provide a fix for a vulnerability. This can be done either through the pointer-to-implementation pattern [38], or by self-destructing the original contract and replacing it with an updated version at the same address. However, it is generally discouraged to change a contract, since doing so contradicts the desirable properties of immutability and public visibility.

Several of the aforementioned approaches rely on contracts or properties being written in a separate specification language, which then compiles to the actual smart contract language (the target language); the point being that the specification language rules out certain classes of errors or vulnerabilities by design. This is impractical, because it requires the developer to learn yet another language, which furthermore may produce only a partial implementation, so that the developer still has to write parts of the contract in the smart-contract language. Rather than using a single language, the developer now has to use two (or more, if several different specification languages ward against different classes of errors).

In contrast, the type system approach seeks to rule out vulnerabilities directly at the level of the target language itself, or a suitable model thereof, thereby making its adoption easier. Two notable examples are the works by Crafa et al. [33] and Hu et al. [58], both of which develop type systems for a 'core part' of the smart-contract language Solidity. In the former, the authors present a standard type system for preventing runtime errors resulting from attempts to access non-existing contract members; and in the latter, the authors create a type system to enforce security policies on the data flows. However, there is also a vast literature on type systems for languages outside of the field of smart contracts, which ensure properties that may also be desirable in a smart contract setting. In the present thesis we shall therefore consider a variety of languages and properties.

1.2 Contents and contributions

The contents of this dissertation consist of four published papers [76; 64; 4; 6], interspersed with some unpublished material. Each chapter investigates a different problem, but the common theme unifying them is an application of a type system to discipline the data flow in a computational model. Hence, we begin in Chapter 2 by giving a high-level overview of the core concepts and underlying intuitions of type systems. These concepts will reappear in various guises in each of the following chapters.

As previously mentioned, we shall work with a variety of languages, not all of which are directly related to smart contracts. In some cases, a programming construct may be better understood in a simpler setting, where its features stand out clearer, and then later be transferred to a more complex setting by means of an encoding. Hence, issues of encodability and relative expressiveness of languages are a secondary, recurring theme in several of the chapters.

Chapters 3–8 contain the core contributions of this thesis. They can be divided into two groups:

- The contributions in the first group, presented in chapters 3–5, focus on a family of computational models known as process calculi. They are small, theoretical models with a primary focus on concurrency, such as is found in threaded or

distributed applications. This is first and foremost seen in the fact that the primary construct for program composition in these languages is the so-called 'parallel composition' operator, written $P_1 \mid P_2$, which denotes that the processes (or programs, or threads) P_1 and P_2 are executing in parallel. However, these languages also have a clear connexion to imperative, class-based or object-oriented languages, which we make explicit in Chapter 5. Hence, they are also well-suited to study imperative features in a minimalistic setting.

- The contributions in the second group, presented in chapters 6–8, focus on a particular modelling language for Solidity smart contracts, called TinySol, originally proposed by Bartoletti et al. [12]. It is an imperative, class-based language, and already much more complex than the aforementioned process calculi, despite still only being a model of the full Solidity language [41]. A distinguishing feature of this language is that method calls are of the form

$$\texttt{call}\ e_1.f(\tilde{e})\$e_2$$

i.e. they carry an extra parameter e_2, which is the amount of currency transferred along with the call. This is the mechanism by which currency is transferred between contracts (corresponding to classes), and it implies that a currency flow is always also a *control flow*.

In the remainder of this section, we shall review and relate the contributions of each individual chapter. We shall present the results in the order given above, to emphasise how ideas appearing in one setting are later transformed and reapplied in a different setting. As several of the results pertain to particular features of different computational models, we shall also have to give a few details about some of the languages. A full description is given at the start of each of the respective chapters.

1.2.1 Typing reflection

In Chapter 3 we study the ρ-calculus (Reflective Higher-Order calculus), a language superficially resembling the π-calculus [83; 86], but equipped with a form of *reflection*.[2] This gives the ρ-calculus some unusual features, notably, structured names, runtime generation of free names, and the lack of an operator for scoping visibility of names, all of which pose some interesting difficulties for proofs of encodability and type system soundness.

Very briefly, communication between processes in the ρ-calculus is on named channels, as in the π-calculus, but the channel names themselves are not *atomic*

[2]The ρ-calculus is also the theoretical foundation for the language *Rholang* (cf. https://rholang.org), which is a smart-contract language focused on concurrency. However, it has not seen widespread adoption.

identifiers. Instead, they are built from a syntax, which is the same as the syntax for processes; hence, if P is a process, then $\ulcorner P \urcorner$ is a name. Communication is on the form

$$\ulcorner P_1 \urcorner \langle\!| P_2 |\!\rangle \mid \ulcorner P_1 \urcorner (y).P_3 \to P_3\{\ulcorner P_2 \urcorner / y\}$$

meaning that the process P_2 is *quoted*, thereby turning it into the name $\ulcorner P_2 \urcorner$, which is sent along the channel $\ulcorner P_1 \urcorner$, whence it is received and substituted for y within the continuation P_3. Lastly, a quoted process can be turned back into a running process by the *drop* operator, $\urcorner y \ulcorner$. This combination of quote and drop gives the ρ-calculus higher-order characteristics as a by-product.

In the original presentation of this language [80], the authors claimed that the ρ-calculus can encode the π-calculus, claiming full abstraction up to weak, barbed bisimilarity as correctness criterion for their encoding. However, we give two counter-examples contradicting this claim. Then, we give a new encoding and prove its correctness, using a set of encodability criteria close to those proposed by Gorla in [48], but with some adaptations necessitated by the fact that the ρ-calculus uses *structured* channel names, which furthermore can be composed at runtime, rather than atomic identifiers. We also prove a separation result, showing that there cannot exist an encoding of the ρ-calculus into the π-calculus satisfying the same correctness criteria. This result is interesting, because it shows that the reflective capabilities of the ρ-calculus make it more expressive than the π-calculus.

There is a rich body of research on type systems for the π-calculus; e.g. [83; 126; 100; 37] to name but a few classic works. However, the aforementioned separation result means that we cannot hope to obtain typability in the ρ-calculus through an encoding into the π-calculus. Instead we create a type system for the ρ-calculus, similar to the classic simple type system for correct channel usage by Milner [83]. This type system is capable of typing certain 'well-behaved' terms, which in particular include those that are encodings of π-calculus terms.

This result is in itself hardly surprising, since we already know that the π-calculus is typable. Hence, we would expect ρ-calculus terms, that *behave like* π-calculus processes, to be typable as well. The interesting point here is rather the limitations we have to impose on ρ-calculus processes to ensure that the subject-reduction theorem can be proved.[3] Specifically, we must require that a process never, during its course of evaluation, will generate a name of a particular shape.

Name-generation in the ρ-calculus is comparable to pointer arithmetics in an imperative setting, so this requirement can be viewed as a saying that a program must never generate a pointer to a certain, limited area of the address space. This property is not a purely syntactical property of the program, and it cannot be ensured by the type system. Hence, this provides a glimpse of the limitations of the purely

[3]The Subject-Reduction theorem is the main theorem to be shown in the so-called 'syntactic approach to type soundness' that we follow in most of the chapters, cf. Chapter 2.

syntactical approach to types that we employ in chapters 3–7. We return to this matter in a different setting in Chapter 8.

1.2.2 A generic type system

The Ψ-calculus [18; 19] is an 'abstract process calculus' or framework, in the sense that some parts of its syntax and semantics, known as parameters, are left unspecified. By making different choices for these unspecified parts, subject to a few constraints, one can obtain concrete instances of the language. In this way, many well-known process calculi can be obtained as instances, and the Ψ-calculus thus allows all these to be treated in a uniform way. The Higher-Order Ψ-calculus (HOΨ) [98] is then an extension of the 'first-order' Ψ-calculus with a construct for process mobility, which further allows higher-order process calculi to be instantiated.

In [61], Hüttel created a generic type system for the 'first-order' Ψ-calculus, which allows concrete type systems for Ψ-calculus instances to be obtained as instances of the type system. In Chapter 4, we extend this work to the Higher-Order Ψ-calculus. This result is interesting for at least three reasons:

- Firstly, the fact that some of the syntactic categories of HOΨ are left unspecified (specifically, terms, conditions and assertions) means that we have to impose certain constraints on the syntactical structure of these sets, in order to ensure that the subject-reduction theorem holds. These constraints are interesting in their own right, because they, like the constraint we had to impose on name generation in the ρ-calculus, yield insights into the syntactical features a language must seemingly possess in order to be typable by the syntactic approach.

- Secondly, we use the generic type system to create instances of other type systems, including one for termination in the Higher-Order π-calculus (HOπ) [36]; and one for non-interference in HOπ, inspired by the security model of Goguen and Meseguer [47]. Both of these ensure properties that are also desirable to have in smart contracts, and we return to both of them in chapters 6–7.

- Thirdly, it turns out that HOΨ, unlike the π-calculus, in fact *is* able to encode the ρ-calculus, as we also show in this chapter. This means we can also instantiate the generic type system to obtain a type system for the ρ-calculus, which is different from the one we created in Chapter 3. In particular, it imposes a different kind of restriction on the runtime-generated names; namely, that name-equivalent names must have the same type. As was the case in Chapter 3, the type system cannot by itself ensure that this property holds for a ρ-calculus program, so this requirement, in effect, means that the programmer must know in advance all the names that *might* be generated at runtime.

1.2.3 Type correspondence

In Chapter 5, we focus on a third process calculus, ${}^{e}\pi$, which is an extension of the π-calculus with polyadic synchronisation vectors. This means that communication in ${}^{e}\pi$ is of the form

$$x_1 \cdot \ldots \cdot x_n\langle v\rangle \mid x_1 \cdot \ldots \cdot x_n(y).P \rightarrow P\{v/y\}$$

where each of the names x_i in the so-called 'subject vector' must match, in order for the communication to succeed. In other words, ${}^{e}\pi$ uses *composite* channel names, rather than just single atomic identifiers. This gives it some similarity to the ρ-calculus, although channels in ${}^{e}\pi$ are only built from a simple composition of names, which are themselves atomic identifiers.

The Ψ-calculus can be instantiated to languages with composite channel names such as ${}^{e}\pi$, and the generic type system for the (first-order) Ψ-calculus can therefore also be instantiated to obtain type systems for such calculi. However, all the examples save one in [61] are for calculi with single, atomic channels, and the sole exception (a type system for a language with subject vectors of length 2) does not seem adequate as a type system for ${}^{e}\pi$, where subject vectors can be of arbitrary length.

On the other hand, there is an obvious similarity between outputs on composite subjects and 'name-spaced' method calls, such as those found in class-based and object-oriented languages. The ${}^{e}\pi$ output construct $c \cdot f\langle v\rangle$, which sends out the value v on the composite channel $c \cdot f$, is reminiscent of a method call

$$\texttt{call } c.f(v)$$

where the method or procedure f is declared in a name space c, which e.g. could be a class name. A method *definition* would then obviously correspond to an *input*, $c \cdot f(y).P$, where P is the method body. Using this intuition, we create a simple, class-based imperative language called WC (**W**hile with **C**lasses), along with an encoding of WC into ${}^{e}\pi$. Besides method calls, we also have *field declarations*, which are represented as *outputs*, and with reads then corresponding to inputs. Hence, interestingly, fields (declarations and reads), and methods (declarations and calls) can both be represented by the same constructs in ${}^{e}\pi$, but used oppositely.

There does exist a type system for ${}^{e}\pi$, created by Carbone in [26, Chapter 6.5], but the structure of the types does not bear any particular semblance to the usual structures of object-oriented languages. We therefore create a new type system with named types, which resemble *interfaces* with signatures for interface members. Then we carry the correspondence with object-oriented languages further, by giving an 'expectable' type system for WC, wherein classes are given interface types, and then showing that these types can be encoded in the type language for ${}^{e}\pi$, such that well-typed WC-terms yield well-typed ${}^{e}\pi$-processes, when they are passed through the encoding. This comparison contributes to understanding the relationship between our types and conventional types for object-oriented languages.

1.2.4 Types for currency flows

In Chapter 6, we study a safety property for Solidity smart contracts known as *call integrity* [51]. This property is desirable, because contracts satisfying it are immune to reentrancy attacks, which are a form of attack that has resulted in large financial losses in the past. The property essentially states that a contract C satisfies call integrity, if, for any two execution contexts, all method calls emanating *from* C will be exactly the same. In other words, the two call-traces, when restricted to calls from C, must be equal. This is important, because method calls are also used to transfer *currency* between contracts. Thus, the property ensures that changing the execution context (which might be controlled by an attacker) cannot affect the *currency flows* from C.

Unfortunately, *showing* that a contract satisfies call integrity is difficult, because the definition involves a quantification over all possible execution contexts. However, we notice a similarity between call integrity and another, well-known property, namely *non-interference* [47]. Both involve a notion of one part of a program being unaffected by changes in another part, although the notion of 'effect' differs: specifically, non-interference speaks about the *memory* being unaffected, rather than the execution trace. This initially means that the two properties are incomparable, as we show in the chapter. However, surprisingly, we also show that call integrity *can* be approximated by a type system that *also* ensures non-interference. This is possible precisely because a control-flow in this setting *also* is a currency-flow, which therefore involves reading from, and writing to, the memory; specifically to the `balance` associated with each contract, which is a special field that holds the amount of currency currently stored in each contract.

The call integrity property is originally formulated for the language EVM, which is the low-level bytecode language of the Ethereum blockchain architecture. However, to make it easier to work with, we first reformulate it in terms of the language TinySol [12], which models aspects of the Solidity language. In particular, it is able to model reentrancy attacks. The type system we then create is an adaptation of a classic type system created by Volpano, Smith and Irvine in [130], using types with security levels to control the data flow. This type system is originally formulated for a small, imperative language, without any class-based/object-oriented features, whereas TinySol represents contracts with a class-like structure. Thus, an important part of our development lies in determining an appropriate type structure for contracts, such that the non-interference theorem would still hold.

It turns out that contracts need a pair-type, consisting of both an interface component describing the signatures of the contract members (as in `WC`), *and* a security level component. Contracts exist at specific addresses, which can be passed as values, and the members of a contract are accessed through this address reference. The security level can therefore be thought of as the type of the *address* itself, and the interface as the type of the structure *stored at* the address.

Although not explicitly mentioned in the chapter, this realisation goes back to one of the instantiations of type systems for the Ψ-calculus in Chapter 4, mentioned above, which also concerned non-interference and security types. The version of TinySol used in this chapter is little more than WC, with every method having an extra, mandatory argument for the value parameter, which is easily encodable in 'plain WC,' and this could then further be represented in ${}^{e}\pi$, via the encoding from Chapter 5. When viewed at the level of ${}^{e}\pi$, it is clear that each name (corresponding to addresses, field names and method names) has a dual nature, since names in ${}^{e}\pi$ can be used both as channels on their own, and in compositions with other names to build composite channels. The two-component type of addresses can be considered a reflection of this observation.

1.2.5 Termination by Typing

In Chapter 7, we consider a different, desirable property of smart contracts, namely termination. As previously mentioned, all validator nodes must execute the code of a smart contract, if it is invoked in a transaction. This creates a problem, if the smart-contract language is Turing-complete, since it would allow a malicious user to create a smart contract containing an infinite loop, which, if invoked by a transaction, would mean that the state-update function would never terminate. This would effectively cause a denial-of-service on the network.

The Ethereum blockchain architecture includes a Turing-complete smart-contract language (EVM, and the high-level language Solidity), so to counter the adverse effects of infinite loops, it uses a mechanism called *gas*, which essentially is a fee the user must pay to schedule a transaction, and which is proportional to the number of computational steps required to *execute* the transaction. Hence, the user must pay (in advance) for the number of computational steps he requests to have executed. In this way, if the transaction exceeds the number of steps that have been paid for, the transaction is aborted, and its effects are rolled back, but the user has still paid for the execution. This prevents infinite loops, and it also gives users an incentive to schedule shorter transactions.

The gas construct solves the problem for the network, but it simultaneously creates a new problem for the users: One user may wish to invoke functionality in a smart contract created by another user, which therefore requires him to know in advance how many computation steps it will need. If he does not supply sufficient gas, the transaction will abort, even if it eventually *would have* terminated, if more gas had been supplied; and even though any excess gas is paid back to the user, when the transaction completes, it still might not be desirable to pay too much in advance, since this excess amount cannot be used for other purposes while the transaction is running. Furthermore, a user might be faced with a choice between several contracts that all seemingly provide the same functionality, but nevertheless might differ in the number of steps they require. In that situation, it is therefore clearly desirable for the

user to be able to assess which one will require the least amount of computational steps.

To model this problem in TinySol, we first extend the language with a gas mechanism, and equip it with a new, small-step operational semantics to be able to reason about the number of computational steps. Then we devise a type system which ensures termination of well-typed contracts. This type system is based on one by Deng and Sangiorgi [37] for the π-calculus, and also using the same technique as the instance of the Ψ-calculus type system from Chapter 4. It works by assigning positive integers to statements, and ensuring that this number always decreases after a computational step. Thus, the integer associated to a method call acts as an upper bound on the number of required computational steps, which therefore also means that we can use this number to provide an upper bound on the amount of gas required to execute the method call.

The type system is quite straightforward, as it reuses most of the structure from the type system of Chapter 6, which again is based on the one from Chapter 5 for `WC`. The downside is that it disallows all forms of recursive method calls, since the number associated to a method is part of the type of that method. Hence, if a method f requires at most n steps, then its body must be typable as using at most $n-1$ steps, and it therefore cannot contain calls to methods using more than $n-1$ steps, which therefore excludes calls to f itself. This limitation is inherited from the type system in [37], but that paper also provides refinements of this basic type system, which seek to relax this restriction. Thus, our work in this chapter is only a first step in this direction. It opens an avenue of future work on similarly extending the type system for TinySol, which likewise may be guided by the encoding from Chapter 5.

1.2.6 Semantic Typing

In Chapter 8, we extend TinySol with two new programming constructs, *delegate calls* and *fallback functions*, and we provide another small-step operational semantics for these constructs, extending the work from Chapter 7. We also provide a new version of the type system from Chapter 6 with a type rule for delegate calls, and we incorporate the type rules for stacks from the type system in Chapter 7.

In these developments, we also tackle a separate issue with the subtyping of interfaces used in the type systems of the aforementioned chapters: The subtyping relation in these type systems all use *structural* subtyping, meaning that an interface I_1 is judged as a subtype of another interface I_2, if I_2 is a substructure of I_1. This can lead to infinite recursions, if e.g. a method signature in I_2 itself can take an address of type I_1 as an input parameter. We solve this problem in Chapter 8 by instead using explicit inheritance, which also leads to a much simpler subtyping relation.

Delegate calls and fallback functions are needed to model the so-called Parity Wallet attack [77], which involves an unwanted data flow to a variable, that should not have been modifiable by an outside user. Hence, this attack *should* be preventable

by using the same techniques as the type system with security types from Chapter 6. However, unfortunately, it is not possible to extend our type system with a type rule for fallback functions. This construct is untypable by ordinary, syntactic type rules, because it admits a limited form of reflection into the language, which causes problems w.r.t. typability, just as we saw in the case of the type systems for the ρ-calculus in Chapter 3 and Chapter 4. Again, we encounter a limitation of the approach to type soundness, known as the *syntactic approach*, which we have hitherto employed. This approach is straightforward and well-suited to handle safety properties that are essentially syntactic in nature, but is equally ill-suited when this is not the case.

In Chapter 8 we therefore employ a different approach, sometimes called the *semantic* approach [23; 9; 8; 7; 67; 123]. The key idea is that the semantics of the types (i.e. their *meaning*) is defined separately, and the type rules are then *proved admissible*, i.e. proved to preserve the semantics of the types, rather than simply given *a priori*. Types and type judgments are thus more akin to logical predicates, rather than just 'tags' on various syntactical constructs, and any piece of code that can be shown to behave *in accordance* with the constraints imposed by the types (predicates) are therefore *safe*, regardless of whether this is inferred by the type rules or shown by a separate 'manual' proof.

Using the semantic approach, we can take advantage of the immutable nature of the blockchain and allow contract creators to supply 'manual proofs' of safety for usages of untypable constructs such as the fallback function, along with the code itself as a form of 'proof-carrying code' [88]. This gives us a mechanism by which users of such code can still obtain guarantees about its safety, even when the code cannot be judged type-safe.

Our approach to semantic typing is based on coinduction, following the work of Caires [23]. This allows us to adapt the work of Pous and Sangiorgi [106] on enhancements of the bisimulation proof-technique, to create *up-to techniques for typing interpretations*, which can be used to reduce the size of manual proofs of type safety. These techniques allow proofs to be smaller, and thereby both easier to find and to verify, which is important for the practical applicability of our approach. The work by Pous and Sangiorgi is formulated in the field of process calculi, so this also serves as another example of how tools and techniques from this area may be used in the (seemingly) quite different setting of an imperative, class-based smart-contract language. Thus, Chapter 8 ties together the strands of all the previous chapters.

* * *

In this chapter, we have presented a broad overview of our results, and how they relate to each other. To summarise, our main results are (1) a simple type system for the ρ-calculus, which was motivated by the encodability and separation results w.r.t. the π-calculus; (2) a generic type system for HOΨ, which provides a guiding structure for our later developments of type systems; (3) a typed encoding of the language `WC`

into ${}^{e}\pi$, which relates our work on process calculi to the class-based/object-oriented, imperative paradigm; (4) a type system with security types for TinySol, which ensures non-interference and also can be used to ensure call-integrity; (5) a small-step operational semantics for TinySol, extended with a gas mechanism, and a simple type system for preventing out-of-gas exceptions; and finally (6) a semantics of security types for a version of TinySol extended with delegate calls and fallback functions. As type systems are the common theme in our results, we shall give a short, high-level introduction to this topic in the next chapter, before embarking on the detailed presentation of our results.

2 Some background on type systems

> *Any type system should be seen as a compositional, decidable, and (usually incomplete) proof system for a specialized logic.*
>
> Luís Caires [23]

Type systems are routinely employed in most programming languages to rule out terms, that are well-*formed* according to the syntax, but nevertheless might not be well-*behaved*, in the sense that they *might* give rise to a runtime error. They are usually divided into *typing disciplines* based on what the types represent; e.g. data types, behavioural types, dependent types, and so forth. In this dissertation we focus specifically on type systems for *data types*, in which base types represent sets of data, and the notions of safety, that we are interested in, are some form of invariant on how these data are used during the execution of a program.

Despite the differences in the type systems we shall present, they nevertheless share a common structure, which we shall review in the present chapter at a high level, to provide the reader with a view of the intuitions and guiding principles behind these type systems. For a more thorough introduction to type systems see e.g. [101].

2.1 Types and safety

Assume we have a programming language $\mathcal{L} \triangleq (\mathcal{P}, \rightarrow)$, where $\mathcal{P}$ is the set of programs, ranged over by P, and $\rightarrow$ is the transition relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$, giving the operational semantics in terms of a transition system. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. An example of such a language could be the following:[1]

$P ::= \texttt{halt}$	terminate the thread
$\mid P_1 \texttt{ par } P_2$	parallel execution
$\mid \texttt{func } f(x) \texttt{ \{ } P \texttt{ \}}$	function declaration
$\mid \texttt{call } f(z)$	function call
$\mid \texttt{new } x \texttt{ in } P$	private name declaration
$\mid \texttt{rep } P$	repeated execution

[1] This language is of course just the asynchronous π-calculus [83] with a slightly modified syntax.

However, we do not need to assume any specific details about the syntax, semantics or even the programming paradigm for the purpose of this high-level description.

We shall assume a *language of types* $\mathcal{T}$, ranged over by T. Exactly what it makes sense to ascribe types to will depend on the specific features of the language, as well as the intended notion of safety. In an imperative language, types would usually be given to variables, function names, and other named constructs, such as records or classes. In a functional language, they would be given to variable names and function abstractions, and so on. In the case of $\mathcal{L}$ above, we assign types to the set of identifier names $\mathcal{N}$ occurring in the programs (variable names and function names), ranged over by f, x, y, z.

Next, we need the notion of a *type environment*

$$\Gamma \in \mathcal{E} ::= \mathcal{N} \rightharpoonup \mathcal{T}$$

to record assumptions about the types of names. It is defined as a partial function from names to types, which can be viewed as a list of tuples $(x_1, T_1), \ldots, (x_n, T_n)$, where the x_i are pairwise distinct, alternatively written as $x_1 : T_1, \ldots, x_n : T_n$. Thus we denote by $\Gamma, x : T$ the type environment Γ extended with the assumption that x has type T.

The notion of a *runtime error* is captured by an *error predicate* $\mathsf{Wrong}_\Gamma(P)$, which is defined syntactically and depends on Γ; i.e. we should be able to conclude whether $\mathsf{Wrong}_\Gamma(P)$ holds *now* for any P, by inspecting only the *syntax* of P (and the assumptions in Γ). As an example, suppose the set $\mathcal{N}$ is partitioned into *function names* $\mathcal{N}_f$, and *value names* $\mathcal{N}_v$. We distinguish them by assigning a type nil to value names, and we add types to the declarations of new names, i.e. we now write `new` $x : T$ `in` P to indicate that x is of type T. A simple example of an error predicate could then be

$$\begin{aligned}
\mathsf{Wrong}_\Gamma(\texttt{halt}) &\triangleq \mathsf{F} \\
\mathsf{Wrong}_\Gamma(\texttt{func}\ f(x)\ \{\ P\ \}) &\triangleq \Gamma(f) = \mathsf{nil} \\
\mathsf{Wrong}_\Gamma(\texttt{call}\ f(z)) &\triangleq \Gamma(f) = \mathsf{nil} \\
\mathsf{Wrong}_\Gamma(P_1\ \texttt{par}\ P_2) &\triangleq \mathsf{Wrong}_\Gamma(P_1) \vee \mathsf{Wrong}_\Gamma(P_2) \\
\mathsf{Wrong}_\Gamma(\texttt{new}\ x : T\ \texttt{in}\ P) &\triangleq \mathsf{Wrong}_{\Gamma, x:T}(P) \\
\mathsf{Wrong}_\Gamma(\texttt{rep}\ P) &\triangleq \mathsf{Wrong}_\Gamma(P)
\end{aligned}$$

The predicate expresses that it is wrong to use a value-name as a function. Thence we can define the *now-safe* predicate

$$\mathsf{NSafe}_\Gamma(P) \triangleq \neg\mathsf{Wrong}_\Gamma(P)$$

which expresses that P is safe to execute for at least *one step*. Thus, if $\mathsf{NSafe}_\Gamma(P)$ holds, we say that P is *now-safe*. We can now define a family of step-indexed safety predicates n-$\mathsf{Safe}_\Gamma(P)$, such that $(n+1)$-$\mathsf{Safe}_\Gamma(P)$ implies

1. $\mathsf{NSafe}_\Gamma(P)$, and
2. $P \to P' \implies n\text{-}\mathsf{Safe}_\Gamma(P')$

and with $1\text{-}\mathsf{Safe}_\Gamma(P) \triangleq \mathsf{NSafe}_\Gamma(P)$. Thus $n\text{-}\mathsf{Safe}_\Gamma(P)$ expresses that P is safe to execute for up to n steps. Then, in the limit where $n \to \infty$, we obtain a set of terms that will be safe to execute for *all* steps. If $\infty\text{-}\mathsf{Safe}_\Gamma(P)$ holds, then P is *invariantly* now-safe, hence we say that it is *runtime safe*, or simply just *safe*. We shall denote this as $\mathsf{Safe}_\Gamma(P)$, defined thus:

$$\mathsf{Safe}_\Gamma(P) \triangleq P \to^* P' \wedge \mathsf{NSafe}_\Gamma(P')$$

Now, for each n, let $n\text{-}\mathsf{Safe}_\Gamma$ denote the *largest* set of programs satisfying the $n\text{-}\mathsf{Safe}_\Gamma(\cdot)$ predicate, i.e.

$$n\text{-}\mathsf{Safe}_\Gamma \triangleq \{\, P \mid n\text{-}\mathsf{Safe}_\Gamma(P) \,\}$$

and with $\mathsf{Safe}_\Gamma \triangleq \{\, P \mid \mathsf{Safe}_\Gamma(P) \,\}$. Then it should be clear that

$$\mathcal{P} \supseteq 1\text{-}\mathsf{Safe}_\Gamma \supseteq 2\text{-}\mathsf{Safe}_\Gamma \supseteq \ldots \supseteq n\text{-}\mathsf{Safe}_\Gamma \supseteq \ldots \supseteq \mathsf{Safe}_\Gamma$$

2.2 Typing interpretations

Another way to view the matter is to consider a Γ-indexed family of sets of programs $\mathcal{R}_\Gamma$, that all satisfy the type assumptions in Γ. Following Caires [23] we shall call such a set a *typing interpretation* of Γ; the basic idea is that the semantics (i.e. meaning) of a type can be expressed as the set of terms that behave according to that type. The definition of $\mathcal{R}_\Gamma$ will be coinductive, closely mirroring the definition of the $n\text{-}\mathsf{Safe}_\Gamma(\cdot)$ predicate above; i.e. something like the following:

Definition 1 (Typing interpretation)**.** A *typing interpretation* $\mathcal{R}_\Gamma$ is a Γ-indexed family of sets of programs, such that
$P \in \mathcal{R}_\Gamma \implies$

1. $\mathsf{NSafe}_\Gamma(P)$, and
2. $P \to P' \implies P' \in \mathcal{R}_\Gamma$. ■

Other conditions may also have to be added, depending on specific features of the language. For example, if the transition is on a declaration of a new variable x of type T, it would have to be appended to the type environment, and in that case we would instead have to find a *new* typing interpretation for this extended type environment, such that it contains the reduct P'.

The definition of a typing interpretation for some Γ can also be given in terms of a monotonic endofunction f_Γ (parametrised by Γ) over the complete lattice $(2^{\mathcal{P}}, \subseteq)$, defined thus:

Definition 2 (Function f)**.** Let $\mathcal{R} \subseteq \mathcal{P}$ and let $f_\Gamma : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ be defined such that $P \in f_\Gamma(\mathcal{R})$ iff

1. $\mathsf{NSafe}_\Gamma(P)$, and
2. $P \to P' \implies P' \in \mathcal{R}$. ■

Lemma 1. *f_Γ is monotonic.*

Proof. We must show that

$$\mathcal{R}_1 \subseteq \mathcal{R}_2 \implies f_\Gamma(\mathcal{R}_1) \subseteq f_\Gamma(\mathcal{R}_2).$$

Thus we assume that $\mathcal{R}_1 \subseteq \mathcal{R}_2$, and we must now show that

$$\forall P \,.\, P \in f_\Gamma(\mathcal{R}_1) \implies P \in f_\Gamma(\mathcal{R}_2).$$

Therefore, assume that $P \in f_\Gamma(\mathcal{R}_1)$. We then check the conditions for $P \in f_\Gamma(\mathcal{R}_2)$:

1. $\mathsf{NSafe}_\Gamma(P)$. We know that $P \in f_\Gamma(\mathcal{R}_1)$, which implies that $\mathsf{NSafe}_\Gamma(P)$ (condition 1). Thus this holds.
2. $P \to P' \implies P' \in \mathcal{R}_2$. We know that $P \in f_\Gamma(\mathcal{R}_1)$, which implies that $P' \in \mathcal{R}_1$ (condition 2). By assumption, $\mathcal{R}_1 \subseteq \mathcal{R}_2$, so we conclude that $P' \in \mathcal{R}_2$. Thus this holds. □

This proof would of course have to be extended for any extra conditions added to Definition 1.

We now have that $\mathcal{R}$ is a typing interpretation for Γ precisely when $\mathcal{R} \subseteq f_\Gamma(\mathcal{R})$, i.e. when $\mathcal{R}$ is a post-fixed point of f_Γ. Now, by Tarski's fixed-point theorem (see e.g. [60]), the greatest fixed-point for f_Γ, written $\nu\mathcal{R}_\Gamma$, is given by

$$\nu\mathcal{R}_\Gamma \triangleq \bigcup \{ \mathcal{R} \in 2^{\mathcal{P}} \mid \mathcal{R} \subseteq f_\Gamma(\mathcal{R}) \}$$

i.e. by the union of all the post-fixed points of f_Γ. We say $\nu\mathcal{R}_\Gamma$ is the *typing* of Γ.

Even in this very generic setting, we can use this definition to show a few properties of our typing interpretations:

Corollary 1. *For any Γ, $\nu\mathcal{R}_\Gamma$ is the largest typing interpretation of Γ.*

Proof. We must show two things:

1. That for any choice of Γ, every $\mathcal{R}_\Gamma$ is contained in $\nu\mathcal{R}_\Gamma$. This follows directly from the definition of $\nu\mathcal{R}_\Gamma$.

2. $\nu\mathcal{R}_\Gamma$ is itself a typing interpretation. Here we check the conditions of Definition 1: If $P \in \nu\mathcal{R}_\Gamma$ then we know there exists a $\mathcal{R}_\Gamma$ such that $P \in \mathcal{R}_\Gamma$. We can therefore conclude:

   a) Since $P \in \mathcal{R}_\Gamma$ we know that $\mathsf{NSafe}_\Gamma(P)$.

   b) If $P \to P'$ then we know that $P' \in \mathcal{R}_\Gamma$. Since $\mathcal{R}_\Gamma \subseteq \nu\mathcal{R}_\Gamma$, we therefore conclude that also $P' \in \nu\mathcal{R}_\Gamma$. □

We note that this formulation of $\nu\mathcal{R}_\Gamma$ is very reminiscent of the definition of bisimilarity over a transition system as the largest bisimulation.

Conversely, we also have that the *least* fixed point is the intersection of all pre-fixed points of f_Γ. This yields the empty set $\emptyset$, which then is the *least typing interpretation* for any Γ. We can also easily show this by checking the conditions of Definition 1, since both of them hold vacuously. Then, since $\forall \mathcal{R} \subseteq \mathcal{P}$, we have that $\emptyset \subseteq \mathcal{R}$, and we therefore have that $\emptyset$ is the least set of programs satisfying the conditions in Definition 1. Thus, the least typing interpretation is rather uninteresting, since all it tells us is that we may be able to find two typing interpretations for the same Γ which nevertheless have no elements in common.

Corollary 2 (Runtime safety)**.** *For any* Γ*, if* $P \in \nu\mathcal{R}_\Gamma$ *then* $\mathsf{Safe}_\Gamma(P)$.

Proof. From $P \in \nu\mathcal{R}_\Gamma$, and by the definition of $\nu\mathcal{R}_\Gamma$, we know there exists a typing interpretation $\mathcal{R}_\Gamma$ such that $P \in \mathcal{R}_\Gamma$. We shall first show a slightly different statement:

$$P \in \mathcal{R}_\Gamma \wedge P \to^* P' \implies \mathsf{NSafe}_\Gamma(P')$$

Now assume that $P \to^* P'$. Then we know there exists a transition sequence of length $n \geq 0$ such that

$$P = P_0 \to \ldots \to P_n = P'$$

We proceed by induction on the length of the transition sequence.

- Case $n = 0$ (base case): Here $P_0 = P$, and by Condition 1 of Definition 1, we have that $P \in \mathcal{R}_\Gamma \implies \mathsf{NSafe}_\Gamma(P)$.

- Inductive case: Assume the statement holds for all k, such that $0 < k < n$. We show that it holds for $k + 1$ as well.

  By Condition 2 of Definition 1, we have that $P_k \to P_{k+1} \implies P_{k+1} \in \mathcal{R}_\Gamma$, and by Condition 1 we have that $P_{k+1} \in \mathcal{R}_\Gamma \implies \mathsf{NSafe}_\Gamma(P_{k+1})$.

Hence, $P \in \mathcal{R}_\Gamma$ implies that $\mathsf{NSafe}_\Gamma(P')$ holds for any P' such that $P \to^* P'$, which is precisely the definition of $\mathsf{Safe}_\Gamma(P)$. □

We can also show the converse of Corollary 2:

Corollary 3. *For any* Γ*, if* $\mathsf{Safe}_\Gamma(P)$ *then* $P \in \nu\mathcal{R}_\Gamma$.

Proof. Assume $\mathsf{Safe}_\Gamma(P)$; then by definition, $P \in \mathsf{Safe}_\Gamma$. We then show that Safe_Γ is a typing interpretation by checking the conditions of Definition 1:

Pick any $Q \in \mathsf{Safe}_\Gamma$. Then from the definition of Safe_Γ, we know that $\mathsf{Safe}_\Gamma(Q)$ holds.

- From the definition of $\mathsf{Safe}_\Gamma(\cdot)$, we know that $\mathsf{Safe}_\Gamma(Q)$ implies $\mathsf{NSafe}_\Gamma(Q)$. This satisfies the first condition.
- From the definition of $\mathsf{Safe}_\Gamma(\cdot)$, we know that $\mathsf{Safe}_\Gamma(Q)$ implies that if $Q \to Q'$, then $\mathsf{Safe}_\Gamma(Q')$ holds as well. This satisfies the second condition.

Thus we conclude that Safe_Γ indeed is a typing interpretation. By Definition 1, we therefore also have that $\mathsf{Safe}_\Gamma \subseteq \nu\mathcal{R}_\Gamma$. □

By Corollary 2 and Corollary 3 we thus get that $\nu\mathcal{R}_\Gamma \subseteq \mathsf{Safe}_\Gamma$ and $\mathsf{Safe}_\Gamma \subseteq \nu\mathcal{R}_\Gamma$, and therefore that $\nu\mathcal{R}_\Gamma = \mathsf{Safe}_\Gamma$. This is indeed what we would expect a sensible definition of the largest typing interpretation to mean.

2.3 Well-typedness

While generic, the results in the preceding section give us a proof technique for showing that a term P is safe, relative to the set of typing assumptions in Γ. By Corollary 2, we just need to find a typing interpretation $\mathcal{R}_\Gamma$ such that $P \in \mathcal{R}_\Gamma$. However, this task may be difficult, since even the least typing interpretation containing P may be large, and we have to check that the conditions hold for every element in our candidate typing interpretation. This is of course problematic, since we want an efficient and automatic decision procedure for checking the safety of our programs.

Rather than using typing interpretations directly, the core idea in a *type system* is instead to define another, *decidable* predicate, which we for now shall refer to as $\mathsf{WellTyped}_\Gamma(P)$. If $\mathsf{WellTyped}_\Gamma(P)$ holds, we say that P is *well-typed*; and, conversely, if $\neg\mathsf{WellTyped}_\Gamma(P)$ holds, we say that P is *ill-typed*.

The idea is that the predicate $\mathsf{WellTyped}_\Gamma(P)$ must be defined such that it is a *sound approximation* to the set of programs that are runtime safe. Like above, we can also consider $\mathsf{WellTyped}_\Gamma$ to be the largest set of programs for which the predicate $\mathsf{WellTyped}_\Gamma(\cdot)$ holds; i.e.

$$\mathsf{WellTyped}_\Gamma \triangleq \left\{ P \mid \mathsf{WellTyped}_\Gamma(P) \right\}$$

Now note the following:

- The fact that it is a *sound* approximation means precisely that $\mathsf{WellTyped}_\Gamma \subseteq \mathsf{Safe}_\Gamma$, or equivalently that $\mathsf{WellTyped}_\Gamma \subseteq \nu\mathcal{R}_\Gamma$.

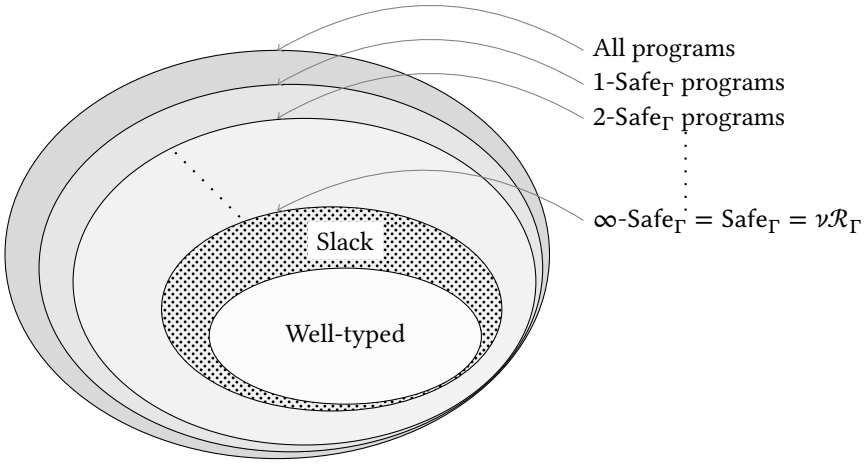

Figure 2.1: The relationship between the sets of *safe* and *well-typed* programs for a sound, but incomplete, type system. The difference between these two sets constitute the *slack* of the type system, here depicted as the hatched area.

- If instead it were a *complete* approximation, then we would have that $\mathsf{Safe}_\Gamma \subseteq \mathsf{WellTyped}_\Gamma$, which would mean that $\mathsf{WellTyped}_\Gamma$ also might contain terms that are *not* in Safe_Γ, thereby defeating the purpose of the type system.
- If it were *both* sound and complete, we would have that $\mathsf{WellTyped}_\Gamma \subseteq \mathsf{Safe}_\Gamma$ and $\mathsf{Safe}_\Gamma \subseteq \mathsf{WellTyped}_\Gamma$, and thus that $\mathsf{WellTyped}_\Gamma = \mathsf{Safe}_\Gamma$, which is rarely possible, lest the type system would become undecidable.[2]

Precisely because $\mathsf{WellTyped}_\Gamma$ is only a *sound* approximation to Safe_Γ, we will therefore generally have that $\mathsf{WellTyped}_\Gamma \cap \mathsf{Safe}_\Gamma \neq \emptyset$, and the difference between these two sets, $\mathsf{Safe}_\Gamma \setminus \mathsf{WellTyped}_\Gamma$, is commonly referred to as the *slack* of the type system. We illustrate this concept in figure 2.1

Another way to characterise the notion of slack is the following: Within systems of formal reasoning, we distinguish between statements that are *provable* and statements that are *true*. Statements that are *true* are those that can be considered true with respect to some structure; whilst the *provable* statements are those for which we can construct a proof using a system of deduction, i.e. a system of inference rules. This difference is captured by the use of two different turnstile symbols:

[2]Setting $\mathsf{WellTyped}_\Gamma$ to be the empty set of programs would trivially yield a sound approximation of Safe_Γ. On the other hand, defining $\mathsf{WellTyped}_\Gamma$ as the set of *all* programs would give us a complete approximation of Safe_Γ. However, neither of those approximations would be typically meaningful, since we expect at least some program to be well-typed and some other to be ill-typed.

- The *syntactic* consequence $\vdash$ represents provability; i.e. $\Gamma \vdash \varphi$ reads *the statement φ is provable, given* Γ.
- The *semantic* consequence $\vDash$ indicates truth; i.e. $\Gamma \vDash \varphi$ reads *the statement φ is true, given* Γ.

To further illustrate the difference between *provable* and *true* statements, we note that it is possible for a statement to be provable, but at the same time not be true. In this case, we say that the proof system is *unsound*. Conversely, we say that a proof system is *sound* if it is the case that every provable statement is also true, as illustrated by the following implication:

$$\Gamma \vdash \varphi \implies \Gamma \vDash \varphi$$

which has a counterpart in the form of *completeness*, that holds when it is the case that every statement that is true also has a proof, illustrated as follows:

$$\Gamma \vDash \varphi \implies \Gamma \vdash \varphi$$

In summary, we say that a deductive system is *sound* if it only allows us to prove statements that are true; and *complete* if every true statement can be proved using the system. This gives rise to a formal characterisation of the slack of a type system, as the programs that are *not well-typed*, but nevertheless are *well-behaved*; i.e. runtime safe. In terms of provability and truth, if we consider the statements φ to be about the safety of programs, the slack of the type system consists of the statements that are true (here $\mathsf{Safe}_\Gamma(P)$), but are not provable with the rules of the system (here, by the predicate $\mathsf{WellTyped}_\Gamma(P)$).

2.4 Syntactic typing

The common approach to type soundness, sometimes known as *syntactic typing*, was introduced by Wright and Felleisen in [133]. It begins by defining the $\mathsf{WellTyped}_\Gamma(P)$ predicate as a typing relation $\vdash \subseteq \mathcal{E} \times \mathcal{P}$ between type environments and terms, which is given in terms of a set of syntax directed, compositional inference rules of the form

$$[\textsc{t-rule}]\frac{\Gamma \vdash P_1 \quad \dots \quad \Gamma \vdash P_k}{\Gamma \vdash \mathrm{op}\,(P_1, \dots, P_k)}$$

Here, the statement $\Gamma \vdash P$ is known as a (syntactic) *type judgment* and reads *P is well-typed, given* Γ.[3] We recognise the use of the single turnstile symbol here in the type judgement, since it precisely denotes that we can *prove* the statement $\Gamma \vdash P$ by using the rules defining the typing relation $\vdash$.

[3]Note that, if we were giving explicit types to programs, then the statement would instead be $\Gamma \vdash P : T$, and we would say that P is well-typed (relative to Γ) if it can be given any type T.

The syntactic typing approach relies on two results to show soundness for the type system. The first is variously known as *safety* or *progress*,[4] and is of the following general form:

Theorem 1 (Safety). $\Gamma \vdash P \implies \mathsf{NSafe}_\Gamma(P)$.

This is proved by showing that the set 1-Safe_Γ of all now-safe terms is closed forward under the rules for type judgments. Thus we proceed by induction in the rules for type judgments, by checking at each step that the $\mathsf{NSafe}_\Gamma(\cdot)$ predicate is satisfied (i.e. that the $\mathsf{Wrong}_\Gamma(P)$ predicate does *not* apply). By this result, we know that, if $\Gamma \vdash P$, then at least $\mathsf{NSafe}_\Gamma(P)$ holds too. In terms of figure 2.1, P is at least within the set of 1-Safe_Γ programs.

The second result is variously known as *subject reduction* or *preservation*, and it states that the property of well-typedness is preserved by the semantics:

Theorem 2 (Preservation). $\Gamma \vdash P \wedge P \to P' \implies \Gamma \vdash P'$.

This result assures us that a well-typed term cannot reduce to an *ill-typed* term. The proof proceeds again by induction, but this time on the rules defining the operational semantics; i.e. the relation $\to \subseteq \mathcal{P} \times \mathcal{P}$. It is somewhat more involved than the other theorem, as it usually (depending on the language and the type system) requires a number of auxiliary lemmas; most notably *strengthening* and *weakening* of the type environment. Let $\mathrm{fn}(P)$ denote the set of *free names* of P:

Lemma 2 (Strengthening). $\Gamma, x : T \vdash P \wedge x \notin \mathrm{fn}(P) \implies \Gamma \vdash P$.

The strengthening lemma states that we can throw away unused type assumptions; i.e. assumptions about the type of a name, if that name does not occur free in the program we are typing. Thus we 'strengthen' Γ by discarding unused assumptions. Its converse, or counterpart, is the weakening lemma:

Lemma 3 (Weakening). $\Gamma \vdash P \wedge x \notin \mathrm{fn}(P) \implies \Gamma, x : T \vdash P$.

It states that we can add assumptions about types of names, if those names do not occur free in the program we are currently typing. Thus we 'weaken' Γ by adding more assumptions than we actually need. Both of the aforementioned two lemmas are usually simple to prove by induction on the type rules. Other lemmas may also be needed, depending on the language and exact formulation of the semantics.

On a side note, it might here be worth noting an interesting property of weakening and strengthening in light of our previous discussion of typing interpretations: We speak of *weakening* of a type environment, when we extend it with extra type

[4] The name *progress* comes from a particular application of type systems, where the notion of now-safety checks that the program is not stuck. Hence, the theorem ensures that the program can make progress, by taking at least one more computational step.

information, e.g. $\Gamma, x : T$, and conversely of *strengthening*, when we remove an unused type entry. However, when we consider the largest typing interpretation for some Γ, then weakening the type environment will correspond to *strengthening* the typing interpretation, in the sense that $\nu\mathcal{R}_{\Gamma,x:T} \subseteq \nu\mathcal{R}_{\Gamma}$. This is quite intuitive, since the type entries in Γ can be regarded as *constraints* on the usage of names, and thus adding more constraints can at most *remove* some programs from the set of programs that use all names in accordance with these constraints. Weakening (of the type environment) thus resembles a property of *monotonicity* of a certain function on the set of typing interpretations.

The preservation theorem does not mention the $\mathsf{Safe}_{\Gamma}(\cdot)$ predicate, but by combining the two theorems, we obtain the desired result of type system soundness:

Corollary 4 (Soundness)**.** $\Gamma \vdash P \implies \mathsf{Safe}_{\Gamma}(P)$.

Proof. By the definition of $\mathsf{Safe}_{\Gamma}(P)$, we have that $P \to P'$ and $\mathsf{NSafe}_{\Gamma}(P')$. We shall first show a slightly different statement:

$$\Gamma \vdash P \wedge P \to^* P' \implies \mathsf{NSafe}_{\Gamma}(P')$$

Now assume that $P \to^* P'$. Then we know there exists a transition sequence of length $n \geq 0$ such that

$$P = P_0 \to \ldots \to P_n = P'$$

We proceed by induction on the length of the transition sequence.

- Base case ($n = 0$): Here $P = P'$, since no transitions are performed. Then by Theorem 1 (Safety) we know that $\Gamma \vdash P \implies \mathsf{NSafe}_{\Gamma}(P)$
- Inductive case: Assume the statement holds for all k, such that $0 \leq k \leq n$. We show that it holds for $k + 1$ as well.

  We know $\Gamma \vdash P_k$. If $P_k \to P_{k+1}$, then by Theorem 2 (Preservation) we also have that $\Gamma \vdash P_{k+1}$, and by Theorem 1 (Safety) that $\mathsf{NSafe}_{\Gamma}(P_{k+1})$.

Hence, $\Gamma \vdash P$ implies that $\mathsf{NSafe}_{\Gamma}(P')$ holds for any P' such that $P \to P'$, which is precisely the definition of $\mathsf{Safe}_{\Gamma}(P)$. □

Notice the similarity between the proof of Corollary 4 and Corollary 2 above. The Soundness result thus expresses that well-typedness is preserved by all transition steps, and as well-typedness implies now-safety, we therefore get that well-typedness implies *invariant* now-safety, which is the definition of safety.

* * *

In this chapter, we have described the general concepts and structure of the syntactic approach to type soundness. The reasoning is quite intuitive and straightforward, which is one of the great advantages of this approach. We shall see these concepts appearing in slightly different guises throughout Chapters 3–7.

However, the presentation has also illustrated a drawback of the syntactic approach: There will always be programs P that reside in the slack of a syntactic type system. This is a fundamental limitation, since the typing relation $\vdash$ can only provide a *sound* approximation to the set Safe_Γ of runtime safe programs.

Furthermore, even in case we can formally *prove* that P in fact *would* be safe to execute, for example by directly giving a typing interpretation $\mathcal{R}_\Gamma$ such that $P \in \mathcal{R}_\Gamma$, we cannot readily use this knowledge to make any program *containing* P typable. That is, we cannot use the type system to reason about 'unsafe code' such as P, even if we could encapsulate P in an abstraction and formally show that the abstraction actually *is* safe to use (i.e. that it still preserves the invariant(s) of the safety predicate). This is a limitation, since abstractions (procedures, functions, classes etc.) are common in programming: a syntactic type system cannot easily be used to reason at the level of abstractions, since it analyses the *syntax*. Thus, we cannot use abstractions to 'hide' (i.e. encapsulate) the unsafe portions of the code without breaking well-typedness. We shall return to this matter in Chapter 8.

Part II

Contributions

3 The Reflective Higher-Order Calculus: Encodability, Typability and Separation[1]

3.1 Introduction

Process calculi are formalisms for modelling and reasoning about concurrent and distributed computations; a prominent example is the π-calculus of Milner, Parrow and Walker [86; 83]. These languages commonly begin by assuming a *countably infinite set of atomic names* $\mathcal{N}$, ranged over by x, y, z. This is not an unreasonable assumption for most purposes, but it does leave open the question of how this set of names actually should be interpreted, e.g. if we were to create an implementation of the π-calculus or one of its variants [126; 103; 42].

A similar issue arises with the scoping operator $(\boldsymbol{\nu} x)P$, which is used to declare a new name x with visibility limited to P. Here the question becomes how in practice to implement this limited visibility. One possibility is to use the Barendregt convention [11] and choose all bound names such that they are unique, but this, in turn, begs the question of how we should choose a new name x, such that it is actually ensured to be unique. For a process modelling a program running on a single computer, this can easily be solved, e.g. with a counter; but if the process models a *distributed* system, with programs running on distinct computers, the solution is less obvious. These issues are not directly handled in the π-calculus model, but only become apparent when we consider a more practical implementation of the set of names.

A radically different approach is taken in the Reflective Higher-Order (RHO or ρ) calculus proposed by Meredith and Radestock in [80]. These authors instead begin by positing that the set of names is built by a syntax, similar to the syntax for processes, and thus generated from a *finite* set of elements. One could imagine different possibilities for this syntax, but Meredith and Radestock here make the unusual choice of letting names be 'quoted' processes, written $\ulcorner P \urcorner$. Thus, if P is

[1] The material in this chapter first appeared in a short abstract presented at the 32nd Nordic Workshop on Programming Theory (NWPT) in 2021 [73]. A more expanded version then appeared in the proceedings of EXPRESS/SOS in 2022 [74]. Finally, I was invited to submit a journal version of the paper, which was published in *Information & Computation* in 2024 [76]. This revision included a correction of a fatal error in the separation result (see Section 3.8). The present chapter is equivalent to the journal version, except for some minor, typographical adjustments.

a process, then $\ulcorner P \urcorner$ is a name. This creates a mutually recursive definition, since processes also contain names.

In the ρ-calculus, runtime name generation is handled by the *lift* construct $x\langle\!|P|\!\rangle$, which quotes the process P, thereby creating the name $\ulcorner P \urcorner$, and outputs it on x; thus name generation is handled explicitly in the ρ-calculus, rather than implicitly by a π-calculus style ν-operator. The newly generated name can then be received by an input construct as in the π-calculus. Reductions in the ρ-calculus are thus of the form

$$x(y).P_1 \mid x\langle\!|P_2|\!\rangle \rightarrow P_1\{\ulcorner P_2 \urcorner/y\}$$

This is the second peculiarity of this calculus, since the newly generated name will be *free* in the continuation of the corresponding input and therefore also *observable* if substituted for the subject of an input or lift. As we shall see in section 3.8, this feature is crucial for showing a separation result w.r.t. the π-calculus.

Another special construct in this calculus is the *drop* construct, written $\urcorner x \ulcorner$, which removes the quotes of the name to run the process within them similar to a process variable X in e.g. HOπ [110; 109]. The combination of the lift and drop constructs, and the duality of processes and names, thus enable higher-order behaviour (i.e. process mobility), and it is also the reason for the 'reflective' epithet in the name of this calculus. It derives from Smith [117], who defined reflection as the ability of a program to turn code into data, compute with it, modify it, and turn it back into running code.

Although superficially quite similar to the π-calculus, these features suggest that the ρ-calculus is actually rather different. As argued above, the use of structured terms as names and explicit name generation seem more realistic from an implementation perspective, as it places the problems of choosing the next name and of ensuring freshness within the language itself, rather than simply assuming that these features just work behind the scenes. However, providing a *solution* to these problems is not trivial, as we shall see below. For example, in [80] Meredith and Radestock also propose an encoding of the asynchronous, choice-free fragment of the π-calculus into the ρ-calculus, reviewed in section 3.3, but as we shall show in section 3.4, this encoding contains two fatal errors invalidating their correctness result.

In what follows, we shall instead propose a different encoding of the π-calculus into the ρ-calculus (Section 3.6) and formally prove its correctness w.r.t. a number of encodability criteria closely related to those proposed by Gorla [48] (Section 3.5), but with some adaptations necessitated by the aforementioned peculiar features of the ρ-calculus (Propositions 1-5 and Corollary 5). Then, in section 3.7, we create a simple type system for the ρ-calculus to check that the names generated at runtime in the encoding of the π-calculus are used correctly. This serves as an example to illustrate the second peculiarity of the ρ-calculus, namely that the newly generated names are *free* in the continuation of an input. Lastly, we return to the matter of encodability in section 3.8, where we use the same encodability criteria to derive a

separation result, showing that there cannot be an encoding of the ρ-calculus into the π-calculus satisfying the same criteria (Theorem 4). This result, too, relies on the visibility of the generated names.

The separation result is quite surprising, and it suggests that we cannot always just reduce higher-order behaviour to the first-order paradigm, as Sangiorgi was able to do with HOπ in [109]. This is because higher-order behaviour in the ρ-calculus is not just an extension on top of an already computationally complete language, as it is the case with HOπ which extends the 'first-order' π-calculus, but rather appears as a special case of the more general phenomenon of reflection, where processes (code) are communicated without modification.

This chapter is an extended and revised version of a previous paper [74]. Additional explanations and examples have been added to the presentation of the ρ-calculus (section 3.2) and the definition of the encoding (section 3.6), as well as full proofs of correctness of the encoding (Propositions 1-5). Furthermore, the type system and associated proofs presented in section 3.7 are also new. Longer proofs are deferred to the end of the chapter; they can be found in Section 3.11.

3.2 The Reflective Higher-Order calculus

In this section, we first review the ρ-calculus, following Meredith and Radestock in [80], and then proceed to give some examples of its usage.

Definition 3 (ρ-calculus syntax)**.** The syntax of the ρ-calculus is:

$$
\begin{array}{rll}
P \in \mathcal{P}_\rho ::= & \mathbf{0} & \text{nil} \\
\mid & P_1 \mid P_2 & \text{parallel} \\
\mid & x\langle\!|P|\!\rangle & \text{lift} \\
\mid & x(y).P & \text{input} \\
\mid & \urcorner x \ulcorner & \text{drop} \\
x, y \in \mathcal{N} ::= & \ulcorner P \urcorner & \text{quote}
\end{array}
$$

■

The *lift* and *drop* constructs have already been described above. The remaining three constructs are as in the π-calculus: The *nil* process is the inactive process; the *parallel* construct is the parallel composition of processes P_1 and P_2; and the *input* construct is a blocking operation, awaiting a communication on the channel x of some name, which upon reception will be bound to y in the continuation P.

The semantics is given in terms of a reduction system. As usual, we shall firstly need a notion of *structural congruence* on processes, written $\equiv$. We shall postpone its precise definition slightly, but the intuition is that $P_1 \equiv P_2$ denotes that P_1 and P_2 are the same process, up to some insignificant structural change, such as reordering of components in parallel composition or a change of bound names (α-conversion).

Now, since names are quoted processes, this notion of structural congruence is extended to the set of names:

Definition 4 (Name equivalence)**.** The *name equivalence* relation, written $\equiv_{\mathcal{N}}$, is the least equivalence on names closed under the following rules:

$$[\text{N-STRUCT}]\frac{P_1 \equiv P_2}{\ulcorner P_1 \urcorner \equiv_{\mathcal{N}} \ulcorner P_2 \urcorner} \qquad [\text{N-DROP}]\frac{x_1 \equiv_{\mathcal{N}} x_2}{\ulcorner\urcorner x_1 \ulcorner\urcorner \equiv_{\mathcal{N}} x_2}$$ ■

The point of [N-STRUCT] is that if the processes within quotes have the same structure (up to structural congruence), then the quoted processes should also represent the same name. Furthermore, by [N-DROP], nested levels of quotes and drops are allowed to 'cancel out.'

Next, we shall need the notions of free and bound names, $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$, which are defined in the usual (syntactic) way, with input being the only formal binder in the language:

Definition 5 (Free and bound names)**.** The sets of free and bound names of a process, denoted $\mathrm{fn}(P)$ resp. $\mathrm{bn}(P)$, are defined thus:

$$\begin{aligned}
\mathrm{fn}(\mathbf{0}) &= \varnothing & \mathrm{bn}(\mathbf{0}) &= \varnothing \\
\mathrm{fn}(P_1 \mid P_2) &= \mathrm{fn}(P_1) \cup \mathrm{fn}(P_2) & \mathrm{bn}(P_1 \mid P_2) &= \mathrm{bn}(P_1) \cup \mathrm{bn}(P_2) \\
\mathrm{fn}(x\langle\!|P|\!\rangle) &= \{x\} \cup \mathrm{fn}(P) & \mathrm{bn}(x\langle\!|P|\!\rangle) &= \mathrm{bn}(P) \\
\mathrm{fn}(x(y).P) &= \{x\} \cup (\mathrm{fn}(P) \setminus \{y\}) & \mathrm{bn}(x(y).P) &= \{y\} \cup \mathrm{bn}(P) \\
\mathrm{fn}(\ulcorner x \urcorner) &= \{x\} & \mathrm{bn}(\ulcorner x \urcorner) &= \varnothing
\end{aligned}$$

We write $\mathrm{n}(P) \triangleq \mathrm{fn}(P) \cup \mathrm{bn}(P)$ for the set of *all names* of P. ■

The definition of $\mathrm{n}(P)$ can now be used to build the notion of a *fresh name*. In calculi with atomic names, such as the π-calculus, it would suffice to say that x is fresh for P if $x \notin \mathrm{n}(P)$. However, with *structured* names we need a more elaborate definition:

Definition 6 (Fresh name)**.** A name x is *fresh* for a process P, written $x\#P$, if x is not name-equivalent to any name in the set of names of P:

$$x_1\#P \triangleq \forall x_2 \in \mathrm{n}(P)\ .\ x_1 \not\equiv_{\mathcal{N}} x_2$$ ■

Lastly, we write $P\{x/y\}$ for the safe substitution of x for y within P. However, given our considerations about $\equiv_{\mathcal{N}}$ above, $P\{x/y\}$ will not only replace y, but also any name that is *name equivalent* to y. Note also, in particular, that substitution does *not* recur into processes under quotes. Thus:

$$\ulcorner P \urcorner\{x/y\} = \begin{cases} x & \text{if } \ulcorner P \urcorner \equiv_{\mathcal{N}} y \\ \ulcorner P \urcorner & \text{otherwise} \end{cases}$$

$$[\text{S-TRANS}]\frac{P_1 \equiv P_2 \quad P_2 \equiv P_3}{P_1 \equiv P_3} \qquad [\text{S-REFL}]\frac{}{P \equiv P}$$

$$[\text{S-COM}]\frac{}{P_1 \mid P_2 \equiv P_2 \mid P_1} \qquad [\text{S-SYM}]\frac{P_2 \equiv P_1}{P_1 \equiv P_2}$$

$$[\text{S-PAR}]\frac{P_1 \equiv P_2 \quad Q_1 \equiv Q_2}{P_1 \mid Q_1 \equiv P_2 \mid Q_2} \qquad [\text{S-NIL}]\frac{}{P \mid \mathbf{0} \equiv P}$$

$$[\text{S-OUT}]\frac{x_1 \equiv_{\mathcal{N}} x_2 \quad P_1 \equiv P_2}{x_1\langle P_1\rangle \equiv x_2\langle P_2\rangle} \qquad [\text{S-DROP}]\frac{x_1 \equiv_{\mathcal{N}} x_2}{\ulcorner x_1 \urcorner \equiv \ulcorner x_2 \urcorner}$$

$$[\text{S-ASS}]\frac{}{P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3}$$

$$[\text{S-IN}]\frac{x_1 \equiv_{\mathcal{N}} x_2 \quad P_1\{z/y_1\} \equiv P_2\{z/y_2\}}{x_1(y_1).P_1 \equiv x_2(y_2).P_2} \ (z \# P_1, P_2)$$

Figure 3.1: The rules and axioms for structural congruence in the ρ-calculus.

We shall now return to the definition of structural congruence: it is defined as the usual least congruence on processes, containing α-equivalence and the abelian monoid rules for parallel composition with $\mathbf{0}$ as the unit element. The rules are given in Figure 3.1. However, with structured terms as names, the congruence rules take on a slightly unusual form, since we now also need to *compare names* (rules [S-OUT], [S-IN] and [S-DROP]). This yields another mutual recursion, this time between structural congruence and name equivalence.

Definition 7 (ρ-calculus semantics)**.** The reduction relation is given by the following rules:

$$[\rho\text{-PAR}]\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2} \qquad [\rho\text{-STRUCT}]\frac{P_1 \equiv P_1' \quad P_1' \to P_2' \quad P_2' \equiv P_2}{P_1 \to P_2}$$

$$[\rho\text{-COM}]\frac{x_1 \equiv_{\mathcal{N}} x_2}{x_1(y).P_1 \mid x_2\langle P_2\rangle \to P_1\{\ulcorner P_2 \urcorner/y\}}$$

We write $\to^*$ for the reflexive and transitive closure of $\to$. ■

The [ρ-PAR] and [ρ-STRUCT] rules are standard (as in e.g. the π-calculus); the former lets us conclude a reduction of one component in a parallel composition, whilst the latter allows us to rewrite the process, using structural congruence $\equiv$, to build a redex that can match the conclusion of one of the other rules.

The [ρ-COM] rule is also *almost* standard: The process P_2 is quoted and sent out over x_2, and the matching input receives it as the name $\ulcorner P_2 \urcorner$ and substitutes it for y in the continuation P_1. However, since names in the ρ-calculus have structure, we must be able to explicitly conclude the equivalence $x_1 \equiv_{\mathcal{N}} x_2$ between the two subjects in a communication. This is thus different from calculi with atomic names where exact syntactic equality is (usually implicitly) required between subjects.

One last detail concerns substitution: In structural congruence, including α-equivalence, $P\{x/y\}$ is defined as the usual capture-avoiding substitution of names for names. In particular, the clause for drop is

$$\urcorner x_1 \ulcorner\{z/x_2\} = \begin{cases} \urcorner z \ulcorner & \text{if } x_1 \equiv_{\mathcal{N}} x_2 \\ \urcorner x_1 \ulcorner & \text{otherwise} \end{cases}$$

However, the substitution used in the *semantics* is slightly different, as it is also used to handle the $\urcorner x \ulcorner$ construct, which was not given a reduction rule above. The *semantic substitution* has the following clause for drop instead of the aforementioned:

$$\urcorner x_1 \ulcorner\{\ulcorner P \urcorner/x_2\} = \begin{cases} P & \text{if } x_1 \equiv_{\mathcal{N}} x_2 \\ \urcorner x_1 \ulcorner & \text{otherwise} \end{cases}$$

thus replacing the *process* $\urcorner x_1 \ulcorner$ with P. This is the only way in which a $\urcorner x \ulcorner$ is ever executed, and it implies that the drop of a *free* name is a deadlock, as it can never be touched by a substitution at runtime. Apart from the aforementioned difference, syntactic and semantic substitution behave in exactly the same way, and, notably, neither of them recur into processes under quotes.

3.2.1 Examples

Before proceeding to the technical details, we shall firstly give a few examples of the usage of the ρ-calculus to give the reader an intuition of its capabilities, and also show how we can encode some common operators found in other calculi. These notational short-hands will become useful later, when we give the encoding of the π-calculus into the ρ-calculus.

Example 1 (Reflection and name generation)**.** Consider the process

$$x(y).(y\langle\!|\mathbf{0}|\!\rangle \mid \urcorner y \ulcorner) \mid x\langle\!|P|\!\rangle$$

where P is some process. By rule [ρ-COM] we can conclude the reduction

$$x(y).(y\langle\!|\mathbf{0}|\!\rangle \mid \urcorner y \ulcorner) \mid x\langle\!|P|\!\rangle \to \ulcorner P \urcorner\langle\!|\mathbf{0}|\!\rangle \mid P$$

where the process P is quoted, sent out on x, and received by the matching input as the *name* $\ulcorner P \urcorner$. It is then substituted for the bound name y in the continuation,

which appears in two places: in subject position of $y\langle\!|\mathbf{0}|\!\rangle$, and inside the process $\ulcorner y \urcorner$ appearing in parallel composition with the lift. In the former case, the name y is substituted for the name $\ulcorner P \urcorner$, whilst in the latter case, the entire process $\ulcorner y \urcorner$ is replaced by P. As a result, we obtain the new process $\ulcorner P \urcorner\langle\!|\mathbf{0}|\!\rangle \mid P$. ■

The above example illustrates both the reflective and name-generating capabilities of the ρ-calculus. In one reduction step, we have both created a new name, $\ulcorner P \urcorner$, *and* composed a new process $\ulcorner P \urcorner\langle\!|\mathbf{0}|\!\rangle \mid P$ at runtime. This is essentially all we *can* do in the ρ-calculus.

Note also that

$$\ulcorner P \urcorner \notin \mathrm{fn}\,(x(y).(y\langle\!|\mathbf{0}|\!\rangle \mid \ulcorner y \urcorner) \mid x\langle\!|P|\!\rangle)$$

but we *do* have that $\ulcorner P \urcorner$ is free in the continuation after the reduction, i.e.

$$\ulcorner P \urcorner \in \mathrm{fn}\,(\ulcorner P \urcorner\langle\!|\mathbf{0}|\!\rangle \mid P)$$

because $\ulcorner P \urcorner$ now appears in subject position of the lift. Thus, this name will now also be *observable*, whereas it was not before the reduction. We shall formalise this notion of observability later, in Definition 9.

Example 2 (Output)**.** In calculi such as the π-calculus, we can only send *names* around, and not processes. The output of a name z on channel x is written $\overline{x}\langle z\rangle$. We can simulate this in the ρ-calculus as follows. Consider the process

$$x\langle z\rangle \triangleq x\langle\!|\ulcorner z \urcorner|\!\rangle$$

If z is a free name, then the resulting name sent out on x will be $\ulcorner\ulcorner z \urcorner\urcorner$. However, by the rule [N-DROP] we have that $z \equiv_{\mathcal{N}} \ulcorner\ulcorner z \urcorner\urcorner$, so using $\ulcorner\ulcorner z \urcorner\urcorner$ will be the same as using z itself.

Conversely, if z is a *bound* name, the extra level of quotes on the received name will be removed by the drop, so the level of quotes does not increase unboundedly. Consider for example the following process and reduction sequence:

$$\begin{aligned} & x(y).w\langle y\rangle \mid x\langle z\rangle \\ = {} & x(y).w\langle\!|\ulcorner y \urcorner|\!\rangle \mid x\langle\!|\ulcorner z \urcorner|\!\rangle \\ \to {} & w\langle\!|\ulcorner z \urcorner|\!\rangle \\ = {} & w\langle z\rangle \end{aligned}$$

The [N-DROP] rule thus lets us simulate output of names. ■

The ρ-calculus does not contain an operator for creating copies of processes; yet this can also be achieved by combining the lift, drop and input operators, as the following examples show:

Example 3 (Divergence)**.** Consider the process

$$D(x) \triangleq x(y).(\ulcorner y \urcorner \mid x\langle\!\langle \ulcorner y \urcorner \rangle\!\rangle)$$

This process will execute whatever it receives on x, whilst simultaneously making it available again on x. It is, in a sense, a copying machine. Now consider what happens, when we send the copying machine a copy of itself, i.e. $D(x) \mid x\langle\!\langle D(x) \rangle\!\rangle$. One reduction step gives us the following:

$$\begin{aligned} & D(x) \mid x\langle\!\langle D(x) \rangle\!\rangle \\ = {} & x(y).(\ulcorner y \urcorner \mid x\langle\!\langle \ulcorner y \urcorner \rangle\!\rangle) \mid x\langle\!\langle D(x) \rangle\!\rangle \\ \to {} & D(x) \mid x\langle\!\langle D(x) \rangle\!\rangle \end{aligned}$$

The process recreates itself after a single reduction step, thus creating a diverging process with an infinite reduction sequence:

$$D(x) \mid x\langle\!\langle D(x) \rangle\!\rangle \to D(x) \mid x\langle\!\langle D(x) \rangle\!\rangle \to D(x) \mid x\langle\!\langle D(x) \rangle\!\rangle \ldots \qquad \blacksquare$$

The process in Example 3 is reminiscent of the Ω-combinator $(\lambda x.xx)\lambda x.xx$ from the λ-calculus, which also recreates itself in a single step, similar to an infinite loop. Using this construction, we can also encode *replication* of a process P, written $!P$, which is usually taken as a first-class operator in first-order languages such as the π-calculus.

Example 4 (Unbounded replication)**.** Let $D(x)$ be defined as in Example 3. Consider now the process

$$!P \triangleq D(x) \mid x\langle\!\langle P \mid D(x) \rangle\!\rangle$$

where P is any process such that $x \notin \mathrm{fn}(P)$. Here is a reduction sequence (other possible sequences might involve reductions of P as well):

$$\begin{aligned} & D(x) \mid x\langle\!\langle P \mid D(x) \rangle\!\rangle \\ \to {} & P \mid D(x) \mid x\langle\!\langle P \mid D(x) \rangle\!\rangle \\ \to {} & P \mid P \mid D(x) \mid x\langle\!\langle P \mid D(x) \rangle\!\rangle \\ \to {} & \ldots \end{aligned} \qquad \blacksquare$$

In the above example, $!P$ creates an unbounded number of copies of P, using just a single name x, which must not appear elsewhere, lest it might interfere with the replication. As the encoding depends on the choice of this name, it might be more precise to write e.g. $!(x)P$, yet we shall omit this in the sequel and instead describe some techniques for obtaining a name that is ensured to be fresh.

Having unbounded replication is not always desirable, since the process diverges. Alternatively, we can also encode *input-guarded* replication, $!u(v).P$, which only creates a new copy of P when a communication is received on u. This can be achieved by prefixing the object of the lift with an input.

Example 5 (Input-guarded replication)**.** Consider the process:

$$!u(v).P \triangleq D(x) \mid x\langle u(v).(D(x) \mid P)\rangle$$

After a single reduction step

$$\begin{aligned} & !u(v).P \\ = {} & D(x) \mid x\langle u(v).(D(x) \mid P)\rangle \\ \to {} & u(v).(D(x) \mid P) \mid x\langle u(v).(D(x) \mid P)\rangle \end{aligned}$$

the process blocks, until it receives a communication on u. After receiving such a communication (say, the name z), we obtain the process

$$\begin{aligned} & D(x) \mid P\{z/v\} \mid x\langle u(v).(D(x) \mid P)\rangle \\ \equiv {} & P\{z/v\} \mid D(x) \mid x\langle u(v).(D(x) \mid P)\rangle \\ = {} & P\{z/v\} \mid !u(v).P \end{aligned}$$

which then is ready to perform one further replication step, before it again blocks. Thus, this encoding does not introduce divergence. ■

In the remainder of the chapter, we shall often make use of short-hands or 'syntactic sugar' such as $x\langle z\rangle$, $D(x)$ and $!u(v).P$ as defined in the preceding section.

3.3 The encoding of Meredith and Radestock

In [80], Meredith and Radestock proposed an encoding of the asynchronous, choice-free π-calculus, taking full abstraction w.r.t. weak, barbed bisimilarity as their correctness criterion. Unfortunately, that encoding is *not* correct, as we shall now show. The counter-examples are instructive, as they highlight some of the difficulties inherent in working with a calculus without the assumption of an infinite set of atomic names and explicit scoping operators.

First, we recall the syntax and semantics of the asynchronous choice-free π-calculus, as given e.g. in [96]. Note that some of the constructs and concepts are similar to those found in the ρ-calculus. We shall therefore reuse some of the symbols and rely on context to distinguish whether a π-calculus or ρ-calculus construct is meant. Assume a countably infinite set of atomic names $\mathcal{N}$, ranged over by x, y, z; then the syntax is as follows:

$$\begin{array}{lll} P \in \mathcal{P}_\pi ::= \mathbf{0} & & \text{nil} \\ \quad \mid P_1 \mid P_2 & & \text{parallel} \\ \quad \mid x(y).P & & \text{input} \\ \quad \mid \overline{x}\langle z\rangle & & \text{output} \\ \quad \mid (\boldsymbol{\nu} x)P & & \text{new} \\ \quad \mid !P & & \text{replication} \end{array}$$

The *nil, parallel* and *input* constructs are similar to their ρ-calculus counterparts. The *output* construct, $\overline{x}\langle z\rangle$, outputs the name z on the channel named x. The *new* construct, $(\boldsymbol{\nu} x)P$, declares a fresh name x, with scope limited to P. Lastly, the replication construct, $!P$, creates infinitely many copies of the process P.

The semantics is given in terms of a reduction system with the following rules:

$$[\pi\text{-COM}]\frac{}{x(y).P \mid \overline{x}\langle z\rangle \to P\{z/y\}} \qquad [\pi\text{-RES}]\frac{P \to P'}{(\boldsymbol{\nu} x)P \to (\boldsymbol{\nu} x)P'}$$

$$[\pi\text{-STRUCT}]\frac{P_1 \equiv P_1' \quad P_1' \to P_2' \quad P_2' \equiv P_2}{P_1 \to P_2} \qquad [\pi\text{-PAR}]\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2}$$

Structural congruence $\equiv$ over $\mathcal{P}_\pi$ contains the same rules as in the ρ-calculus, but with syntactic equality replacing name equivalence, and also the following rules for scoping and replication:

$$\begin{aligned}(\boldsymbol{\nu} x)\mathbf{0} &\equiv \mathbf{0}\\ (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P &\equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P\\ !P &\equiv P \mid !P\\ (\boldsymbol{\nu} x)P_1 \mid P_2 &\equiv (\boldsymbol{\nu} x)(P_1 \mid P_2) \text{ if } x \notin \mathrm{fn}(P_2)\end{aligned}$$

Now for the encoding, assume a function $\varphi : \mathcal{N} \to \mathcal{N}$ from π-calculus atomic names to ρ-calculus names. Since the set of π-calculus names is countably infinite, it can for example be mapped to the set of natural numbers. The function φ could then be regarded as an enumeration of names (or a successor function), starting e.g. from $\ulcorner\mathbf{0}\urcorner$ for the name x_0, and then letting the name x_{i+1} be defined in terms of the name x_i as for example $x_{i+1} \triangleq \ulcorner x_i\langle\!|\mathbf{0}|\!\rangle\urcorner$. In the sequel, we shall use the following definitions:

Definition 8 (Static name generation)**.** We let the one-step *left* (resp. *right*) increment of a name x, and the composition of two names x, y be defined thus:

$$\begin{aligned}+x &\triangleq \ulcorner x\langle\!|\mathbf{0}|\!\rangle\urcorner\\ x+ &\triangleq \ulcorner x(\ulcorner\mathbf{0}\urcorner).\mathbf{0}\urcorner\\ x \cdot y &\triangleq \ulcorner x\langle\!|\mathbf{0}|\!\rangle \mid y(\ulcorner\mathbf{0}\urcorner).\mathbf{0}\urcorner\end{aligned}$$ ■

Using left increments, we can generate a countably infinite sequence of names $x_0, x_1, x_2, \ldots$, starting from any name $x = x_0$, as

$$\begin{aligned}x_1 &= +x\\ x_2 &= +x_1 = ++x\\ &\vdots\end{aligned}$$

and so on. This shows that the set of π-calculus names can be implemented as ρ-names, as, by the definition of name equivalence and structural congruence, we have that $x \not\equiv_{\mathcal{N}} \ulcorner x\langle\mathbf{0}\rangle\urcorner$.

Alternatively, we can use right increments, which gives us another countably infinite sequence $x+ = x_1, x++ = x_1+ = x_2, \ldots$; or we can use name composition, which yields yet another sequence with $x^2 = x \cdot x, x^3 = x^2 \cdot x, x^4 = x^3 \cdot x, \ldots$ and so on.

These are all examples of *static quoting* techniques for consistent name generation, and each could be used to implement the function φ. Given such techniques, Meredith and Radestock then begin by assuming that all π-calculus names are already implemented as ρ-names. Their translation function $[\![P]\!]_{n_0,p_0}$ requires two names as parameters, which must be chosen such that they are distinct from all the names in P, and furthermore that no name *within* P can ever be *generated* from n_0 or p_0 by means of the aforementioned methods of static name generation. One way of ensuring this is by letting

$$n_0 = \ulcorner \prod_{x \in \mathrm{fn}(P)} x\langle\mathbf{0}\rangle\urcorner \qquad \text{and} \qquad p_0 = \ulcorner \prod_{x \in \mathrm{fn}(P)} x(\ulcorner\mathbf{0}\urcorner).\mathbf{0}\urcorner$$

where $\prod$ denotes generalised parallel composition. The translation $[\![P]\!] = [\![P]\!]_{n_0,p_0}$ [80, p. 13] is then given by the following recursive equations:[2]

$$\begin{aligned}
[\![\mathbf{0}]\!]_{n,p} &= \mathbf{0} \\
[\![\overline{x}\langle y\rangle]\!]_{n,p} &= x\langle y\rangle \\
[\![x(y).P]\!]_{n,p} &= x(y).[\![P]\!]_{n,p} \\
[\![P_1 \mid P_2]\!]_{n,p} &= [\![P_1]\!]_{+n,+p} \mid [\![P_2]\!]_{n+,p+} \\
[\![(\boldsymbol{\nu} x)P]\!]_{n,p} &= p(x).[\![P]\!]_{+n,+p} \mid p\langle n\rangle
\end{aligned}$$

$$\begin{aligned}
[\![!P]\!]_{n,p} = n \cdot p\big\langle\!\big\langle & n+(n).p+(p).\big([\![P]\!]_{n,p} \mid D(n \cdot p) \mid n+\langle\!\langle n\langle n\rangle\rangle\!\rangle \mid p+\langle\!\langle p\langle p\rangle\rangle\!\rangle\big) \\
& \mid D(n \cdot p) \mid n+\langle +\langle n\rangle \mid p+\langle +\langle p\rangle
\end{aligned}$$

where $x\langle y\rangle$ and $D(x)$ are notational short-hands as defined in Example 2 and Example 3.

A central element in this translation is the encoding of replication, $[\![!P]\!]_{n,p}$, so we shall give some further details about its underlying intuitions. Firstly, it uses the encoding of unguarded replication described in Example 4. As the reader may recall, this encoding required a single name, which must not be name equivalent to any name used by any other process, lest it might interfere with the replication. This is achieved in the above encoding by composing the two name parameters, n and p, to obtain a new name $n \cdot p$.

[2] The translation has been adapted to use our notation for name increments, which we find more intuitive than x^l and x^r, which is used in the original presentation. We also use $x\langle y\rangle$ rather than $x[y]$ for output, which is more in line with standard π-calculus notation.

Secondly, if $[\![P]\!]_{n,p}$ were simply copied as in the encoding in Example 4, any usage of the parameters n and p within the translation of P would also be copied, which thus could create a name clash. Therefore, the inner process is prefixed with two inputs that *bind* n and p within the continuation. In parallel, we then have two other processes, n+<+n> and p+<+p>, that output the new names +n and +p, which will be substituted for n and p. These processes are also copied, and in the next round of replication they will instead create the names ⌜+n<+n>⌝ and ⌜+p<+p>⌝, and so on, thereby implementing a *runtime* form of name generation, similar to our static quoting technique.

For the purpose of defining a notion of behavioural equivalence that is comparable to that of other calculi that do feature a ν-operator, Meredith and Radestock define a *name-restricted observation predicate* $\downarrow^{\mathcal{N}}$ for the ρ-calculus, parametrised with a set of names $\mathcal{N}$. The idea is to only allow observation of names in this set. We follow their definition, but also allow the observation predicate to distinguish between input x, and output $\overline{x}$:[3]

Definition 9 (Observation predicate)**.** The *$\mathcal{N}$-restricted observation predicate* is defined by the rules:

$$[\text{B-OUT}_\rho]\frac{x_1 \equiv_{\mathcal{N}} x_2 \quad x_1 \in \mathcal{N}}{x_1\langle\!|P|\!\rangle \downarrow^{\mathcal{N}} \overline{x_2}} \qquad [\text{B-IN}_\rho]\frac{x_1 \equiv_{\mathcal{N}} x_2 \quad x_1 \in \mathcal{N}}{x_1(y).P \downarrow^{\mathcal{N}} x_2}$$

$$[\text{B-PAR}_\rho]\frac{P_1 \downarrow^{\mathcal{N}} \hat{x} \quad \vee \quad P_2 \downarrow^{\mathcal{N}} \hat{x}}{P_1 \mid P_2 \downarrow^{\mathcal{N}} \hat{x}}$$

where $\hat{x}$ ranges over $x, \overline{x}$. ■

Definition 10 ($\mathcal{N}$-restricted barbed bisimulation)**.** An *$\mathcal{N}$-restricted barbed bisimulation* is a symmetric, binary relation $\mathcal{R}^{\mathcal{N}}$ on processes, parametrised with a set of names $\mathcal{N}$, such that $(P_1, P_2) \in \mathcal{R}^{\mathcal{N}}$ implies:

- If $P_1 \to P_1'$ then there exists a P_2' such that $P_2 \to P_2'$ and $(P_1', P_2') \in \mathcal{R}^{\mathcal{N}}$.
- If $P_1 \downarrow^{\mathcal{N}} \hat{x}$ then $P_2 \downarrow^{\mathcal{N}} \hat{x}$.

We say that P_1 is *$\mathcal{N}$-restricted barbed bisimilar* to P_2, written $P_1 \sim^{\mathcal{N}} P_2$, if there exists an $\mathcal{N}$-restricted barbed bisimulation $\mathcal{R}^{\mathcal{N}}$ such that $(P_1, P_2) \in \mathcal{R}^{\mathcal{N}}$. ■

Definition 11 (Weak observation predicate)**.** The 'weak' observation predicate is defined thus:

$$P \Downarrow^{\mathcal{N}} \hat{x} \triangleq \exists P' \,.\, P \to^* P' \wedge P' \downarrow^{\mathcal{N}} \hat{x}$$ ■

[3]The added distinction between input and output observations is only for use in our later development of a correct encoding, and does not invalidate our claim that the encoding by Meredith and Radestock is incorrect, since our counter-examples shall only rely on observing outputs.

By replacing $P_2 \downarrow^{\mathcal{N}} \hat{x}$ with $P_2 \Downarrow^{\mathcal{N}} \hat{x}$, and $P_2 \to P_2'$ with $P_2 \to^* P_2'$ in Definition 10, we obtain the corresponding notion of a *weak* $\mathcal{N}$-restricted barbed bisimulation:

Definition 12 (Weak $\mathcal{N}$-restricted barbed bisimulation)**.** A *weak $\mathcal{N}$-restricted barbed bisimulation* is a symmetric, binary relation $\mathcal{R}^{\mathcal{N}}$ on processes, parametrised with a set of names $\mathcal{N}$, such that $(P_1, P_2) \in \mathcal{R}^{\mathcal{N}}$ implies:

- If $P_1 \to P_1'$ then there exists a P_2' such that $P_2 \to^* P_2'$ and $(P_1', P_2') \in \mathcal{R}^{\mathcal{N}}$.
- If $P_1 \downarrow^{\mathcal{N}} \hat{x}$ then $P_2 \Downarrow^{\mathcal{N}} \hat{x}$.

We say that P_1 is *weakly $\mathcal{N}$-restricted barbed bisimilar* to P_2, written $P_1 \approx^{\mathcal{N}} P_2$, if there exists a weak $\mathcal{N}$-restricted barbed bisimulation $\mathcal{R}^{\mathcal{N}}$ that relates them. ■

The corresponding observation predicate for the π-calculus is built by the following rules for observation on output, restriction and replication

$$[\text{B-OUT}_\pi]\frac{x \in \mathcal{N}}{\overline{x}\langle y\rangle \downarrow^{\mathcal{N}} \overline{x}} \qquad [\text{B-RES}_\pi]\frac{P \downarrow^{\mathcal{N}} \hat{x}}{(\boldsymbol{\nu} z)P \downarrow^{\mathcal{N}} \hat{x}}\ (x \neq z) \qquad [\text{B-REP}_\pi]\frac{P \downarrow^{\mathcal{N}} \hat{x}}{!P \downarrow^{\mathcal{N}} \hat{x}}$$

and with rules similar to $[\text{B-PAR}_\rho]$ and $[\text{B-IN}_\rho]$ in the ρ-calculus for observation on parallel composition and input, with strict syntactic equality replacing name equivalence in the premise of the latter rule. The notions of a weak observation predicate, and (strong resp. weak) $\mathcal{N}$-restricted barbed bisimulation and bisimilarity for the π-calculus are then defined as in the ρ-calculus. We write $P \downarrow \hat{x}$, $P \Downarrow \hat{x}$, $P_1 \sim P_2$ and $P_1 \approx P_2$ when $\mathcal{N}$ is the set of all names, corresponding to no restriction on the names we can observe. This yields the familiar notions of (strong resp. weak) barbed bisimilarity in the π-calculus (as defined in e.g. [83]).

Given these notions of behavioural equivalence, Meredith and Radestock then state the following as a theorem [80, p. 14, Theorem 5.3], but without providing a proof:

$$P_1 \approx P_2 \iff [\![P_1]\!] \approx^{\text{fn}(P_1)\cup\text{fn}(P_2)} [\![P_2]\!] \tag{3.1}$$

with observation in the ρ-calculus restricted to $\text{fn}(P_1) \cup \text{fn}(P_2)$, i.e. the free names in P_1 and P_2, implemented as ρ-names.[4]

[4]Note that the original presentation [80, p. 14, Theorem 5.3] only has $P_1 \approx P_2 \iff [\![P_1]\!] \approx^{\text{fn}(P_1)} [\![P_2]\!]$, but we regard this as a simple omission, since it trivially would not hold for the implication from right to left: Take for example $P_1 \triangleq x\langle z\rangle$ and $P_2 \triangleq x\langle z\rangle \mid w\langle z\rangle$. Then we have that $\text{fn}(P_1) = \{x, z\}$, and indeed $[\![P_1]\!] \approx^{\{x,z\}} [\![P_2]\!]$ since for $i \in \{1, 2\}$ we have that $[\![P_i]\!] \not\to$ and $[\![P_i]\!] \downarrow^{\{x,z\}} \overline{x}$; but obviously $P_1 \not\approx P_2$, since $P_2 \downarrow \overline{w}$ but $P_1 \not\Downarrow \overline{w}$.

3.4 The errors

We shall now see why the claim stated in (3.1) does not hold. Firstly, consider the following π-calculus processes:

$$P_1 \triangleq !(\boldsymbol{\nu} z)\overline{u}\langle z\rangle \qquad \text{and} \qquad P_2 \triangleq (\boldsymbol{\nu} z)!\overline{u}\langle z\rangle$$

Clearly, they represent different behaviours: P_2 will continuously send out the *same* fresh name z on u, whilst P_1 will send out *different* fresh names, as we can see by applying α-conversion after unfolding the replication. We can also easily construct a testing context C, containing a single hole [ ], such that they can be distinguished by the (π-calculus) $\Downarrow \overline{x}$ predicate, for example

$$C \triangleq [\,] \mid u(n_1).u(n_2).\left(\overline{n_1} \mid n_2.\overline{x}\right)$$

where the objects for the input/output of $\overline{n_1}$, n_2 and $\overline{x}$ are ignored, as this only requires pure synchronisation. Clearly, if the two names received on u are the same, then n_1 and n_2 will be the same name, so they can synchronise and we will therefore be able to observe $\overline{x}$ after 3 reduction steps. And conversely, if the two names are distinct, then we will not observe $\overline{x}$. Thus $C\,[P_1] \not\Downarrow \overline{x}$ whilst $C\,[P_2] \Downarrow \overline{x}$ as argued above.

Now we make a slight adjustment to the two terms. By composing an arbitrary process Q with the inner output process $\overline{u}\langle z\rangle$ we obtain the following:

$$P_1' \triangleq !\left((\boldsymbol{\nu} z)\overline{u}\langle z\rangle \mid Q\right) \quad \text{and} \quad P_2' \triangleq (\boldsymbol{\nu} z)!\left(\overline{u}\langle z\rangle \mid Q\right)$$

The actual shape and behaviour of Q is irrelevant, as long as $u, x \notin \mathrm{fn}(Q)$; it is there solely to induce the parameter pair (n, p) to be split into a 'left pair' $(+n, +p)$ and a 'right pair' $(n+, p+)$ that are passed to the translations of the left (resp. right) parts of the parallel composition. The simplest choice would thus be to pick $Q = \mathbf{0}$. Note also that this composition changes nothing w.r.t. observability of $\overline{x}$: we still have that $C\,[P_1'] \not\Downarrow \overline{x}$ and $C\,[P_2'] \Downarrow \overline{x}$.

We shall now perform the actual translation. To make it more readable, we tabulate the names generated by static quoting during the translation and rename them as follows:

$$\begin{array}{cccccc} n \cdot p = a & p+ = c & +p = e & (+p)+ = g & ++n = i \\ n+ = b & +n = d & (+n)+ = f & (+n) \cdot (+p) = h & ++p = j \end{array}$$

Note that *none* of these names will be observable by the $\Downarrow^{\mathrm{fn}(P_1)\cup\mathrm{fn}(P_2)}$ predicate, because they are generated by the translation, and hence are not in the set $\mathrm{fn}(P_1) \cup \mathrm{fn}(P_2)$ of free names of P_1 and P_2. Now, here is the translation:

$$\begin{aligned} [\![P_1']\!]_{n,p} &= a\langle\!\langle b(n).c(p).\left(e(z).u\langle z\rangle \mid e\langle d\rangle \mid [\![Q]\!]_{f,g} \mid D(a) \mid b\langle\!\langle n\langle n\rangle\rangle\!\rangle \mid c\langle\!\langle p\langle p\rangle\rangle\!\rangle\right)\rangle\!\rangle \\ &\qquad \mid D(a) \mid b\langle d\rangle \mid c\langle e\rangle \\ [\![P_2']\!]_{n,p} &= p(z).h\langle\!\langle f(d).g(e).\left(u\langle z\rangle \mid [\![Q]\!]_{f,g} \mid D(h) \mid f\langle\!\langle d\langle d\rangle\rangle\!\rangle \mid g\langle\!\langle e\langle e\rangle\rangle\!\rangle\right)\rangle\!\rangle \\ &\qquad \mid D(h) \mid f\langle i\rangle \mid g\langle j\rangle \mid p\langle n\rangle \end{aligned}$$

By performing the reductions, we see (not surprisingly) that $[\![P_2']\!]_{n,p}$ firstly performs the communication on p, which causes z to be replaced by n, and the process afterwards expands into arbitrarily many instances of $u\langle n\rangle$ (see [75, p. 12] for a reduction sequence). On the other hand, the translated process $[\![P_1']\!]_{n,p}$ will immediately go through the replication steps, thereby creating arbitrarily many instances of the process $e(z).u\langle z\rangle \mid e\langle d\rangle$ corresponding to the translation of $(\boldsymbol{\nu} z)\overline{u}\langle z\rangle$. This process obviously reduces to $u\langle d\rangle$ in one step. However, precisely because of the aforementioned split of (n, p) over the translation of parallel composition, the name d will *not* be updated by the replication context. This process will therefore *also* repeatedly output the *same* name d on u, and the (translated) form of our testing context can therefore no longer distinguish the processes.

Both $[\![P_1']\!]$ and $[\![P_2']\!]$ thus reduce to arbitrarily many copies of either $u\langle d\rangle$ (for P_1') or $u\langle n\rangle$ (for P_2'), and u is the only name we can observe, as all the other names are created by the translation. This then gives us our desired counter-example: by also translating the testing context we obtain a pair of processes where

$$C\,[P_1'] \not\approx C\,[P_2'] \qquad \text{but} \qquad [\![C\,[P_1']]\!] \approx^{\mathrm{fn}(C[P_1'])\cup\mathrm{fn}(C[P_2'])} [\![C\,[P_2']]\!]$$

in contradiction of the implication from right to left in the claim stated in (3.1).

The detailed analysis above gives us a clear idea of the root cause of the problem: The translation of replication creates a context with the purpose of ensuring that the names (n, p) used within it will repeatedly be substituted with new, fresh names $(+n, +p)$ dynamically built from the previous names, and these act as sources of new names for any occurrence of $(\boldsymbol{\nu} z)P$ within a replicated process. The point is precisely to ensure that each instance of a replicated ν operator will generate a unique new name, and the parameters (n, p) on the translation function act as 'handles' to access this resource; they are the names that have *most recently* been replicated.

The problem arises because this property of being the 'most recently replicated names' is not preserved by the translation of parallel composition: It splits the pair into a left and a right pair, used in the translation of the left and right parallel components:

$$[\![P_1 \mid P_2]\!]_{n,p} = [\![P_1]\!]_{+n,+p} \mid [\![P_2]\!]_{n+,p+}$$

Thus, the access to the most recently replicated names is lost in the translation of the inner processes, because, as we noted above, substitution does *not* recur into processes under quotes. Therefore, when the replication context increments (n, p) at runtime, this update cannot touch the n and p embedded in the *statically* incremented names $(+n, +p)$ and $(n+, p+)$ which the translation function generates for the translation of parallel composition. This is why we added an arbitrary Q to create a parallel composition in our counter-example above.

However, the error above is not the only one in the claim by Meredith and Radestock: whilst its root cause was the splitting of names over the translation of

parallel composition, we can also create another example that is more directly related to the interplay between $(\boldsymbol{\nu} x)P$ and replication. Consider the following processes:

$$P_1 \triangleq\ !(\boldsymbol{\nu} z)\overline{u}\langle z\rangle \qquad \text{and} \qquad P_2 \triangleq\ !(\boldsymbol{\nu} q)(\boldsymbol{\nu} z)\overline{u}\langle z\rangle$$

Note that P_1 and P_2 are structurally congruent, since the new name q in P_2 is never used. Thus $P_1 \approx P_2$ also holds. Both processes should thus continuously generate fresh, distinct names. Yet when we translate those terms, the name incrementation in the translation of a term of the form $(\boldsymbol{\nu} x)P$ means that we again lose access to the most recently replicated names from the translation of replication:

$$[\![(\boldsymbol{\nu} x)P]\!]_{n,p} = p(x).[\![P]\!]_{+n,+p} \mid p\langle n\rangle$$

This creates a problem, which can be easily seen if we perform the translation stepwise, using the same tabulated list of names as before. For both processes, the translation of replication is the same:

$$\begin{aligned}[\![!P]\!]_{n,p} = a\langle\!\langle b(n).c(p).\big([\![P]\!]_{n,p} \mid D(a) \mid b\langle\!\langle n\langle n\rangle\rangle\!\rangle \mid c\langle\!\langle p\langle p\rangle\rangle\!\rangle\big)\rangle\!\rangle \\ \mid D(a) \mid b\langle d\rangle \mid c\langle e\rangle\end{aligned}$$

Now let $P_1' \triangleq (\boldsymbol{\nu} z)\overline{u}\langle z\rangle$ and $P_2' \triangleq (\boldsymbol{\nu} q)(\boldsymbol{\nu} z)\overline{u}\langle z\rangle$ and replace $[\![P]\!]_{n,p}$ above with $[\![P_1']\!]_{n,p}$ and $[\![P_2']\!]_{n,p}$ respectively. The translations of the inner processes yield:

$$\begin{aligned}[\![(\boldsymbol{\nu} z)\overline{u}\langle z\rangle]\!]_{n,p} &= p(z).u\langle z\rangle \mid p\langle n\rangle \\ [\![(\boldsymbol{\nu} q)(\boldsymbol{\nu} z)\overline{u}\langle z\rangle]\!]_{n,p} &= p(q).(e(z).u\langle z\rangle \mid e\langle d\rangle) \mid p\langle n\rangle\end{aligned}$$

which reduce to $u\langle n\rangle$ and $u\langle d\rangle$ respectively. The names n, p are bound in the replication context and will therefore be updated whenever the process replicates. However, in the case of P_2, these names are *statically* incremented in the translation of $(\boldsymbol{\nu} q)$ to yield the names $+n = d$ and $+p = e$, and these two names will therefore *not* be updated at runtime, just as in the previous counter-example. Consequently, in the case of P_2 the names sent out on u will *not* be distinct; they will all be the name $+n = d$. We can therefore use the same testing context C as in the previous example and proceed as before to generate another contradiction of the claim in (3.1); this time by *distinguishing* the translated terms, although we have $C[P_1] \approx C[P_2]$ in the π-calculus. In summary, neither of the implications in the claim in (3.1) hold.

3.5 Our criteria for encodability

Both of the previous examples illustrate the difficulties involved in reasoning about a parametrised translation. Usually, the parameters represent a property or invariant that is assumed to be preserved throughout the translation, and a proof of correctness

of the translation must therefore also include a proof that this invariant or property is indeed preserved. For example, in the present case, the invariant assumed to hold for the parameters is precisely that they always refer to the most recently replicated names. However, this assumption is never formally stated in the original ρ-calculus paper [80], and as the examples above show, it does not hold either. Thus, a naive attempt to show correctness of the translation by induction on the clauses of the translation function may therefore seemingly go through, if the parameters are not considered. This is doubly problematic in the present case, because the observation predicate used in the bisimulation relation over ρ-calculus terms is parametrised so that we precisely do *not* observe the names created by the translation function.

Full abstraction, of which the claim in 3.1 is an instance, may also not be the most informative correctness criterion, as argued by Gorla and Nestmann [49], since it says more about the process equivalences chosen for the source and target languages than about the encoding itself. For example, it does not necessarily prevent the translation from introducing divergence. Also, as we are here more interested in showing that the π-calculus is 'implementable' in the ρ-calculus than in transferring equations between the source and target language, we shall instead follow the approach of such authors as Gorla [48], Carbone and Maffeis [27] and others, and state a number of criteria for what we consider a valid encoding, where we also take the presence of parameters into account:

Definition 13 (Language)**.** A *language* $\mathcal{L}$ is a tuple $\mathcal{L} \triangleq (\mathcal{P}, \mathcal{N}, \rightarrow, \simeq)$, where $\mathcal{P}$ is a set of terms, $\mathcal{N}$ is a set of names, $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ is the reduction relation, with $\rightarrow^*$ denoting the reflexive and transitive closure of $\rightarrow$, and $\simeq \subseteq \mathcal{P} \times \mathcal{P}$ is a notion of behavioural equivalence. ■

We say a term $P \in \mathcal{P}$ *diverges*, written $P \rightarrow^{\omega}$, if P has an infinite reduction sequence. We use $\sigma : \mathcal{N} \rightarrow \mathcal{N}$ to denote a substitution in $\mathcal{L}$, and we write $P\sigma$ and $\sigma(x)$ to denote the application of a substitution to a process resp. a name. For encodings, we need the notion of a *source* and a *target* language, and we shall generally use the convention of subscripting s (for source) and t (for target) to a language $\mathcal{L}$ or its components, including substitutions, and we let $S \in \mathcal{P}_s$ and $T \in \mathcal{P}_t$.

Definition 14 (Encoding)**.** An *encoding* of $\mathcal{L}_s$ into $\mathcal{L}_t$ is a tuple $(\llbracket\ \rrbracket_N, \varphi, \delta)$, where $\llbracket\ \rrbracket_N : \mathcal{P}_s \rightarrow \mathcal{P}_t$ is a *translation function*, parametrised with a finite list of names $N \in \mathcal{N}_t^k$; and $\varphi : \mathcal{N}_s \rightarrow \mathcal{N}_t$ is a *renaming policy*, mapping names in the source language into names in the target language; and $\delta : \mathcal{N}_t^k \rightarrow \mathcal{N}_t^k$ is a *name derivation* function, mapping k-ary tuples of target names to tuples of equal arity for some k. ■

The name derivation function δ allows us to express that the list of name parameters N may evolve in some predictable way during the course of translation. This seems necessary in particular, when we are working with a language with structured

terms as names. In some cases we may also need to derive multiple tuples of names from the same input tuple; thus to comply with the requirement that δ is a single function, we could e.g. envision using an extra, designated name as an argument to control the derivation method used by δ. However, to abstract away from such details, we say that a tuple of names N_2 is *derivable* from some tuple of names N_1, written $N_1 \rightsquigarrow N_2$, if $\delta(N_1) = N_2$, and likewise that $N_1 \rightsquigarrow n$ if $n \in N_2$. Note that we abuse the notation slightly and treat the lists as sets when the position of each individual component does not matter.

Definition 15 (Valid encoding)**.** We shall regard an encoding as *valid*, if it satisfies at least the following criteria:

1. *Compositionality:*

$$[\![S_1 \mid \ldots \mid S_n]\!]_N = C \mid [\![S_1]\!]_{N_1} \mid \ldots \mid [\![S_n]\!]_{N_n}$$

   where C is an optional coordinating context and

$$\mathrm{fn}(C) \subseteq \varphi(\mathrm{fn}(S_1 \mid \ldots \mid S_n)) \cup N$$

   and for each $i \in \{1, \ldots, n\}$ we have that $N \rightsquigarrow N_i$.

2. *Substitution invariance:*

$$[\![S\sigma_s]\!]_N \simeq [\![S]\!]_N \sigma_t$$

   for each σ_s, where $\varphi(\sigma_s(x)) = \sigma_t(\varphi(x))$.

3. *Operational correspondence:*

$$S \to^* S' \iff \exists T' \,.\, [\![S]\!]_N \to^* T' \wedge T' \simeq [\![S']\!]_{N'}$$

   and $N \rightsquigarrow N'$

4. *Observational correspondence:*

$$P \downarrow^{\mathcal{N}} \hat{x} \iff [\![P]\!]_N \Downarrow^{\varphi(\mathcal{N})} \varphi(\hat{x})$$

   and we require that $N \cap \varphi(\mathcal{N}) = \varnothing$ for any set of observable names $\mathcal{N}$.

5. *Divergence reflection:*

$$[\![P]\!]_N \to^\omega \implies P \to^\omega$$

6. *Parameter independence:*

$$[\![P]\!]_{N_1} \simeq [\![P]\!]_{N_2}$$

   for each finite N_1, N_2. ■

These criteria are very close to those proposed by Gorla [48], except that we have chosen observational correspondence, rather than the less specific success testing; i.e. $P \rightarrow^* \downarrow \checkmark$ implies $[\![P]\!]_N \rightarrow^* \downarrow \checkmark$. This can easily be obtained, simply by choosing a specific name x and then defining $\checkmark$ as a process with x in subject position, as we did in our counter-examples above.

Furthermore, as we are here allowing parameters to appear on the translation, we have also added the criterion of parameter independence, which does not appear in [48]. This is just to ensure that the behaviour of the translated terms will not depend on the exact choice of the parameters. Likewise, we have also added name restriction to the observation predicate for observational correspondence $\Downarrow^{\mathcal{N}}$, and we require that $N \cap \varphi(\mathcal{N}) = \varnothing$; i.e. that the parameters should not be observable. This seems a natural requirement, since we also require that $N \subseteq \mathcal{N}_t$; i.e. that the parameters belong to the target language. They should therefore not be observable on the source terms.

3.6 A correct encoding

As the previous examples have illustrated, the main difficulty in creating an encoding of the π-calculus in the ρ-calculus is how to achieve a robust source of fresh names at runtime that are guaranteed never to cause a name clash. One way is to use a dedicated process for this purpose. Consider the following process, where $D(x)$ is defined as in Example 3:

$$\begin{aligned} !N(x,z,v,s) \triangleq\ & x\Big(\!\Big|z(a).v(r).\Big(D(x) \mid r\langle\!\langle \ulcorner a \urcorner \rangle\!\rangle \mid z\langle\!\langle a \langle\!\langle \mathbf{0} \rangle\!\rangle \rangle\!\rangle\Big)\Big|\!\Big\rangle \\ & \mid D(x) \mid z\langle\!\langle \ulcorner s \urcorner \rangle\!\rangle \end{aligned}$$

This process is a *name server*; it consistently generates names corresponding to consecutive left-increments of the initial name s and outputs them on the 'return address' r received on v. The parameter x is the replication name, used in the encoding of replication. We refer to the above form as the *initial state* of the name server, which we shall also write as $!N(x,z,v,s)^{\text{init}}$. After two reductions it evolves to the form

$$\begin{aligned} & v(r).\Big(D(x) \mid r\langle\!\langle \ulcorner s \urcorner \rangle\!\rangle \mid z\langle\!\langle \ulcorner \ulcorner s \urcorner \urcorner \langle\!\langle \mathbf{0} \rangle\!\rangle \rangle\!\rangle\Big) \\ \mid\ & x\Big(\!\Big|z(a).v(r).\Big(D(x) \mid r\langle\!\langle \ulcorner a \urcorner \rangle\!\rangle \mid z\langle\!\langle a \langle\!\langle \mathbf{0} \rangle\!\rangle \rangle\!\rangle\Big)\Big|\!\Big\rangle \end{aligned}$$

which we refer to as its *ready state*, written $!N(x,z,v,s)^{\text{ready}}$, where it blocks, awaiting a request for a new name on v. The first request will return $\ulcorner \ulcorner s \urcorner \urcorner$; a second request will return $\ulcorner \ulcorner \ulcorner s \urcorner \urcorner \langle\!\langle \mathbf{0} \rangle\!\rangle \urcorner = +s$, and so on. It takes two steps to complete a request, after which the name server returns to the initial state, but with the parameter s now left-incremented once. A sequence of transitions for the name server thus looks as

follows:

$$!N(x,z,v,s)^{\text{init}} \twoheadrightarrow\ !N(x,z,v,s)^{\text{ready}} \twoheadrightarrow\ !N(x,z,v,+s)^{\text{init}}$$

where we assume a request was received at the *ready* state. We shall refer to the 'internal' states between *init* and *ready* as *proc* states, written $!N(x,z,v,s)^{\text{proc}}$, where the name server is processing a request.

We can verify that the names generated by the name server will all be distinct, by considering the *quote depth* of a name (resp. process) defined thus:

Definition 16 (Quote depth)**.** The *quote depth* $\text{QD}(x)$ of a name x is computed by the recursive equations:

$$\text{QD}(\ulcorner P \urcorner) = \begin{cases} \text{QD}(x) & \text{if } P \equiv \ulcorner x \urcorner \\ 1 + \text{QD}(P) & \text{otherwise} \end{cases}$$

$$\text{QD}(P) = \begin{cases} \max\{\, \text{QD}(x) \mid x \in \text{fn}(P) \,\} & \text{if } \text{fn}(P) \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$$

■

The quote depth of a name x_1 corresponds to the maximum number of calls to [N-STRUCT] used to conclude name equivalence $x_1 \equiv_{\mathcal{N}} x_2$ for some name x_2. The leaves of the derivation tree of $x_1 \equiv_{\mathcal{N}} x_2$ will all be of the form $\mathbf{0} \equiv \mathbf{0}$, provided all processes are rewritten to a form with the smallest number of parallel $\mathbf{0}$ processes, since $\mathbf{0}$ is the only process constructor in the syntax (Definition 3) that does not contain names. Thus, a necessary (but not sufficient) condition for two names to be name equivalent is that they have the same quote depth. Names are therefore automatically stratified based on their quote depth:

Lemma 4 (Stratification)**.** $x_1 \equiv_{\mathcal{N}} x_2 \implies \text{QD}(x_1) = \text{QD}(x_2)$.

Proof. Assume for the sake of obtaining a contradiction that for some names x_1, x_2 we have that that $x_1 \equiv_{\mathcal{N}} x_2$ but $\text{QD}(x_1) = n$ and $\text{QD}(x_2) = k$, with $n < k$. Then in the derivation tree, there must exist a branch where the n'th application of [N-STRUCT] will have $\ulcorner 0 \urcorner \equiv_{\mathcal{N}} \ulcorner P \urcorner$ as its conclusion, for some P, with $\text{QD}(\ulcorner P \urcorner) = k - n$. The premise of this rule would then be of the form $\mathbf{0} \equiv P$ for some process P with $\text{QD}(P) = k - n$. But the only process that is structurally congruent to $\mathbf{0}$, is $\mathbf{0}$ itself, and $\text{QD}(\mathbf{0}) = 0$. Thus $k - n = 0$, thereby contradicting our assumption that $n < k$. □

We can also partition names into *namespaces* in the following way:

Definition 17 (Namespace)**.** let $\mathcal{N}_{[\ ]}$ be a collection of name contexts, ranged over by N, with one or more holes occurring in the position of free names. If s is a name, then so is $N\,[s]$ for some $N \in \mathcal{N}_{[\ ]}$. We write

$$\mathcal{N}_{[s]} \triangleq \left\{ N\,[s] \;\mid\; N \in \mathcal{N}_{[\ ]} \right\}$$

and we say that $\mathcal{N}_{[s]}$ is a *namespace rooted at s*. ■

Clearly, if $\mathrm{QD}\,(N) = n$ (counting $\mathrm{QD}\,([\ ]) = 0$), and $\mathrm{QD}\,(s) = i$ and $\mathrm{QD}\,(s') = j$, then $\mathrm{QD}\,(N\,[s]) = n + i$ and $\mathrm{QD}\,(N\,[s']) = n + j$. Thus, by the contrapositive of Lemma 4, we know that $N\,[s] \not\equiv_{\mathcal{N}} N\,[s']$ if $i \neq j$.

Using the concepts of name contexts, we can describe the aforementioned static quoting techniques of Definition 8 as three distinct name space 'templates,' built by the following grammars:

$$\begin{aligned}
{}^{+}N \in {}^{+}\mathcal{N}_{[\ ]} &::= [\ ] \;\mid\; \ulcorner {}^{+}N \langle \mathbf{0} \rangle \urcorner \\
N^{+} \in \mathcal{N}^{+}_{[\ ]} &::= [\ ] \;\mid\; \ulcorner N^{+} (\ulcorner \mathbf{0} \urcorner).\mathbf{0} \urcorner \\
N^{\circ} \in \mathcal{N}^{\circ}_{[\ ]} &::= [\ ] \;\mid\; \ulcorner N^{\circ} \langle \mathbf{0} \rangle \;\mid\; N^{\circ} (\ulcorner \mathbf{0} \urcorner).\mathbf{0} \urcorner
\end{aligned}$$

We shall use these namespace templates to implement the name derivation function δ. Thus, if we let $\widehat{N}$ denote any of the name contexts ${}^{+}N, N^{+}, N^{\circ}$ then $s \rightsquigarrow s'$ if there exists a name context $\widehat{N}$ such that $s' \equiv_{\mathcal{N}} \widehat{N}\,[s]$. This assures us that even if two namespaces use the same structure, e.g. ${}^{+}\mathcal{N}_{[\ ]}$, all their names will still be distinct *if* their roots are not name equivalent, and neither is derivable from the other.

In case of the name server, we see that it generates the namespace ${}^{+}\mathcal{N}_{[s]}$, i.e. the namespace of left-increments rooted at s, where s is a parameter. Thus if $s_1 \not\equiv_{\mathcal{N}} s_2$ and neither is derivable from the other, then $!N(x, z, v, s_1)$ and $!N(x, z, v, s_2)$ will generate similarly structured namespaces, ${}^{+}\mathcal{N}_{[s_1]}$ and ${}^{+}\mathcal{N}_{[s_2]}$, but consisting of different sets of names. Yet we can easily construct a mapping ${}^{+}\mathcal{N}_{[s_1]} \mapsto {}^{+}\mathcal{N}_{[s_2]}$ simply by replacing every occurrence of s_1 within ${}^{+}\mathcal{N}_{[s_1]}$ by s_2. This will be important in the proof for parameter independence below.

Definition 18 (Namespace mapping)**.** We write ${}^{+}\sigma_{s_1,s_2} : {}^{+}\mathcal{N}_{[s_1]} \to {}^{+}\mathcal{N}_{[s_2]}$ to denote the substitution mapping ${}^{+}N\,[s_1] \mapsto {}^{+}N\,[s_2]$ for every ${}^{+}N \in {}^{+}\mathcal{N}_{[\ ]}$. ■

Based on these considerations we can now construct our encoding. We let the encoding be defined as

$$[\![P]\!] \triangleq [\![P]\!]_{n,v} \;\mid\; !N(x, z, v, s)$$

where we assume we can choose the names n, v, x, z, s such that they are distinct from all free names in P and $n, v, x, z \notin {}^{+}\mathcal{N}_{[s]}$. As in the encoding by Meredith and Radestock, we shall assume that all π-calculus names are implemented as ρ-names,

and thus we shall generally omit explicit reference to the renaming policy φ in the following.

We shall also limit ourselves to the π-calculus fragment with only input-guarded replication, i.e. we shall only allow replicated π-calculus terms of the form $!x(y).P$, rather than $!P$. This ensures that the encoding does not introduce divergence, unlike the encoding by Meredith and Radestock which replicates eagerly and therefore always diverges.[5] As we saw in Example 5, input-guarded replication can be encoded in the ρ-calculus without introducing divergence.

Given these considerations, the translation function $[\![\ ]\!]_{n,v}$ is then given by the following equations:

$$\begin{aligned}
[\![\mathbf{0}]\!]_{n,v} &= \mathbf{0} \\
[\![P_1 \mid P_2]\!]_{n,v} &= [\![P_1]\!]_{+n,v} \mid [\![P_2]\!]_{n+,v} \\
[\![x(y).P]\!]_{n,v} &= x(y).[\![P]\!]_{n,v} \\
[\![\overline{x}\langle z\rangle]\!]_{n,v} &= x\langle z\rangle \\
[\![(\boldsymbol{\nu} x)P]\!]_{n,v} &= v\langle n\rangle \mid n(x).[\![P]\!]_{n^2,v} \\
[\![!x(y).P]\!]_{n,v} &= D(n) \mid n\langle\!\langle x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big)\rangle\!\rangle
\end{aligned}$$

The idea is that we simplify the 'bookkeeping' involved in runtime name generation by isolating it to a single, contextual process. This prevents errors of the first kind in the encoding by Meredith and Radestock, which resulted from processes losing access to the most recently replicated names. Here, the name v is used by all processes to contact the name server, and since it is never updated, this access can never be lost. Conversely, the name n, which is used for the 'return address,' as well as for replication, is *always* updated incrementally, during the translation. It is never bound or reused, unlike in the translation by Meredith and Radestock, where the replication context used $+n, +p$ but also bound n, p and passed them to the inner translation of P, which resulted in the second kind of error. We say that a name is *unique for the translation* if it is never generated more than once by the translation function, and this is the invariant that should hold for the parameter n:

Lemma 5 (Uniqueness). *For each clause $[\![C\,[P]]\!]_{n,v} = [\![C]\!]_{n,v}\big[[\![P]\!]_{n',v}\big]$, where $n \rightsquigarrow n'$, and $[\![C]\!]_{n,v}$ contains a set of names $N' = \{n_1, \ldots, n_k\}$ such that $n \rightsquigarrow N'$, it holds that if n is unique for the translation, then so are $n_1, \ldots, n_k$ and n'.*

Proof. If a name n' is unique for a translation, then it will never be generated twice. We prove this by examining the clauses of the translation function, assuming at each step that n is unique.

The cases for $[\![\mathbf{0}]\!]_{n,v}$ and $[\![x\langle z\rangle]\!]_{n,v}$ are immediate, as the translation does not recur. The remaining cases are more interesting:

[5] This is only a slight limitation, as we can use input-guarded replication to encode full replication. Note also that having only input-guarded replication would not have prevented any of the errors described in section 3.4.

- Case $[\![x(y).P]\!]_{n,v}$: Holds, as the translation does not use n, so $n = n'$.
- Case $[\![P_1 \mid P_2]\!]_{n,v}$: Here we have two recursive calls to the translation: $[\![P_1]\!]_{+n,v}$ and $[\![P_2]\!]_{n+,v}$. We have that $+n \not\equiv_{\mathcal{N}} n+$ and from the definition of name incrementation we see that $+n$ cannot be derived from $n+$, nor vice versa. By Lemma 4, n cannot be derived from either $+n$ or $n+$.
- Case $[\![(\boldsymbol{\nu} x)P]\!]_{n,v}$: We have a single, recursive call to $[\![P]\!]_{n^2,v}$ where we pass on n^2, and from the definition of name composition we see that n cannot be derived from n^2.
- Case $[\![!x(y).P]\!]_{n,v}$: We have a single, recursive call to $[\![P]\!]_{n^2,v}$ where we pass on n^2, and from the definition of name composition we see that n cannot be derived from n^2.

This concludes the proof. □

For every usage of n in a clause, we always either increase the quote depth of the parameter we pass to the inner call to the translation, or we shift the parameter into a new namespace by composition.

3.6.1 Committing and internal steps

The encoding presented here is *non-prompt*, in the sense that the ρ-calculus target term may be able to perform some reduction steps, which do not correspond to anything in the π-calculus source term, *before* actually performing the reduction matching a reduction in the source term. These steps are all internal, and may derive from the name server, as it progresses from its initial state to its ready state, or from a request for a new name by the encoding of $(\boldsymbol{\nu} x)P$, or from a replicated term.

Every reduction in the π-calculus must ultimately derive from a communication, since [π-COM] is the only axiom in the semantics. We say that the corresponding communication in the ρ-calculus is the *committing* step, and this step must therefore be a communication on a pair of names that came from the π-calculus. Any other step in the ρ-calculus is an *internal* (or non-committing) step.

We can formalise this intuitive notion of committing and internal steps as follows: By assumption, π-calculus names are implemented as ρ-names, which thus allows us to *syntactically* distinguish them from the names produced by the translation. As an example, we could let all π-calculus names be drawn from the ${}^{+}\mathcal{N}_{[t]}$ name space, which is similar to the name space ${}^{+}\mathcal{N}_{[s]}$ generated by the name server, but choosing a distinct root t, such that t is not derivable from s or vice versa. This would make the π-calculus names structurally similar to the names produced by the name server, whilst still not being name equivalent to any of them; i.e. both would be of the shape $\ulcorner {}^{+}N\,\langle\!|\mathbf{0}|\!\rangle \urcorner$, which immediately can be distinguished from names coming from any of

the other name spaces. We shall assume the π-calculus names are indeed generated thus in the following.

Definition 19 (Committing and internal steps)**.** Let T, T' be ρ-calculus terms, such that $[\![P]\!] \to^* T \to T'$ for some translated π-calculus term P; and let x_1, x_2 be the names used in the premise of the [ρ-COM] rule used in the derivation of $T \to T'$. We say

- $T \to T'$ is a *committing step*, if $x_1, x_2 \in {}^{+}\mathcal{N}_{[s]} \cup {}^{+}\mathcal{N}_{[t]}$
- $T \to T'$ is an *internal step*, if $x_1, x_2 \notin {}^{+}\mathcal{N}_{[s]} \cup {}^{+}\mathcal{N}_{[t]}$

where ${}^{+}\mathcal{N}_{[s]}$ is the namespace generated by the name server, and ${}^{+}\mathcal{N}_{[t]}$ is the namespace used to implement the renaming policy φ.

We write $\mapsto$ for an internal reduction step, and $\mapsto^k$ for a reduction sequence $P \mapsto \ldots \mapsto P'$ of *at most* k internal steps. We write $\Mapsto^k$ for a reduction sequence $P \mapsto^{k_1} \to \mapsto^{k_2} P'$ containing exactly one committing step, where $k = k_1 + k_2$. ■

A name generated by the name server corresponds to a name declared by $(\boldsymbol{\nu} x)$ in the π-calculus. A reduction on any of these names, or on any free name coming from the π-calculus, is therefore a committing step, whereas reductions on names of any other shape is an internal (non-committing) step.

For any translated term, the number of enabled internal steps is finite and determined by the structure of the source term. The name server initially requires two steps to evolve to the *ready* state; these steps are enabled for every translated process. Servicing a request for a new name requires two steps, and then two further steps to return to the *ready* state. Thus every unguarded subterm of the form $(\boldsymbol{\nu} x)Q$ will give rise to four steps, plus the number of internal steps enabled in Q. Lastly, every replicating term requires one step to expose the input guard. As these are all structural properties of the source term, we can therefore easily bound the maximum number of such non-committing steps by analysing the source term:

Definition 20 (Step-function)**.** Let $\mathcal{S} : \mathcal{P}_\pi \to \mathbb{N}$ be defined as

$$\mathcal{S}(P) = 2 + 4\mathcal{S}_\nu(P) + \mathcal{S}_!(P)$$

where $\mathcal{S}_!(!x(y).P) = 1$ and $\mathcal{S}_\nu((\boldsymbol{\nu} x)P) = 1 + \mathcal{S}_\nu(P)$, and for the remaining cases we have for $* \in \{\nu, !\}$ that $\mathcal{S}_*(P_1 \mid P_2) = \mathcal{S}_*(P_1) + \mathcal{S}_*(P_2)$ and $\mathcal{S}_*(P) = 0$ otherwise. ■

Lemma 6 (Maximum internal steps)**.** *For any π-calculus source term P, and for any ρ-calculus term T' such that $\mathcal{S}(P) = k$ and $[\![P]\!] \mapsto^k T'$, it holds that $T' \not\mapsto$.*

The proof is by induction on the structure of P. It can be found in Section 3.11.1.

Lemma 6 expresses that any translated target term $[\![P]\!]$ *at most* can perform a finite number of internal steps, before it either blocks or must perform a committing

step, corresponding to a reduction step $P \to P'$ by the source term. After the reduction, further non-committing steps may become enabled, if the reduction exposes a replication or a $(\boldsymbol{\nu} x)Q$ subterm, but the number of such steps is always finite and determined by the structure of the source term.

Lemma 7 (Sum internal steps)**.** *If $P \to P'$ and there exist terms T, T' and a number k such that $[\![P]\!] \Rightarrow^{k} T \not\mapsto$ and $[\![P']\!] \mapsto^{\mathcal{S}(P')} T' \not\mapsto$ then $k = \mathcal{S}(P) + \mathcal{S}(P') - 2$.*

Proof. By Definition 19, we have that $\Rightarrow^{k} = \mapsto^{k_1} \to \mapsto^{k_2}$ and $k = k_1 + k_2$. By Lemma 6, $k_1 \leq \mathcal{S}(P)$ since some enabled internal steps may not have been performed before the committing step. However, any such residual steps are then transferred to k_2, since we require that $T \not\mapsto$. Thus

$$\begin{aligned} k_2 &= \mathcal{S}(P) - k_1 + 4\mathcal{S}_{\nu}(P') + \mathcal{S}_{!}(P') && \text{by Definition 20} \\ &= \mathcal{S}(P) - k_1 + \mathcal{S}(P') - 2 \\ k_1 + k_2 &= \mathcal{S}(P) + \mathcal{S}(P') - 2 \end{aligned}$$

where we subtract the two extra steps from the name server, since it is not initialised twice. □

3.6.2 Correctness of the encoding

We can now proceed to show that our encoding in fact satisfies the validity criteria of Definition 15. Firstly, we show that the behaviour of the translated process does not depend on the structure of the name parameters n and s, as long as these names are unique for the translation:

Proposition 1 (Independence of parameters)**.** *If $n, n', s, s' \# \mathrm{n}(P)$ and all are unique for the translation, then*

$$[\![P]\!]_{n,v} \mid\ !N(x, z, v, s) \sim^{\mathrm{fn}(P)} [\![P]\!]_{n',v} \mid\ !N(x, z, v, s')$$

Proof. The translation only generates finitely many names, and this generation is deterministic (dependent only on the structure of the process). Changing the initial name from n to n' does not alter the number or structure of the generated names (up to the change from n to n'), and by Lemma 5, all the names are unique, regardless of the choice of the initial name.

Consider first the case where $s = s'$: By applying the namespace mapping ${}^{+}\sigma_{n,n'}$ to our translated process, we have that

$$([\![P]\!]_{n,v} \mid\ !N(x, z, v, s)) \, {}^{+}\sigma_{n,n'} = ([\![P]\!]_{n',v} \mid\ !N(x, z, v, s))$$

Consider now $\mathcal{I}$, the identity relation on $\mathcal{P}$, which is trivially a bisimulation. We construct a new relation $\mathcal{R}$ as follows:

$$\mathcal{R} \triangleq \left\{ (Q, Q \, {}^{+}\sigma_{n,n'}) \mid (Q, Q) \in \mathcal{I} \right\}$$

and as argued above, we have that

$$(\llbracket P \rrbracket_{n,v} \mid !N(x,z,v,s), \llbracket P \rrbracket_{n',v} \mid !N(x,z,v,s)) \in \mathcal{R}$$

We now have to show that $\mathcal{R}$ indeed is an $\mathrm{fn}(P)$-restricted bisimulation. Here it should suffice to note that for all pairs of processes $(Q, Q') \in \mathcal{R}$, Q and Q' are identical, except on the names replaced by the namespace mapping ${}^{+}\sigma_{n,n'}$. But as none of these names are in $\mathrm{fn}(P)$, they will therefore not be observable by the $\downarrow^{\mathrm{fn}(P)}$ predicate.

A similar argument can then be used for the case where $s \neq s'$, since by construction neither of these names are in the set $\mathrm{n}(P)$, nor are any of the names derivable from s or s'. Thus changing the root name of the namespace generated by the name server cannot lead to any difference in the set of observable names, and as the name generation method corresponds to a successor function, we can again create a namespace mapping ${}^{+}\sigma_{s_i,s'_i}$ for each s_i, s'_i. □

Next, we formulate a (mostly) standard result relating substitution in the two calculi:

Proposition 2 (Substitution)**.** *Let $\sigma_s \triangleq \{u/w\}$ denote substitution in the π-calculus, and let $\sigma_t \triangleq \{u/w\}$ denote substitution in the ρ-calculus. Then $\llbracket P\sigma_s \rrbracket_{n,v} = \llbracket P \rrbracket_{n,v}\sigma_t$ if $u, w \# P, n, v, \mathcal{N}_{[n]}$, where $\mathcal{N}_{[n]}$ is the finite subset of names in ${}^{+}\mathcal{N}_{[n]} \cup \mathcal{N}^{+}_{[n]} \cup \mathcal{N}^{\circ}_{[n]}$ generated by the translation.*

The proof is by induction on the clauses of the translation function. It can be found in Section 3.11.2.

The condition $u, w \# P, n, v, \mathcal{N}_{[n]}$ ensures that the substitution cannot touch any of the names created by the translation, which is reasonable, since the substitutions we care about should derive from communications in the π-calculus, and not from some of the internal reductions in the ρ-calculus that are used to simulate replication or requests for new names.

Our next result establishes that our translation preserves observability of subjects, as long as we restrict observations to the set of free names in P. However, as the encoding is non-prompt, we may need to perform some (finite) number of internal reductions first in the translated term, to ensure all subjects deriving from subjects in the π-calculus are unguarded. However, the number of such internal steps can at most be $\mathcal{S}(P)$, so we can actually show a stronger version than what is required in Property 4:

Proposition 3 (Observational correspondence)**.** *$P \downarrow \hat{x}$ iff there exists a T such that $\llbracket P \rrbracket \mapsto^{\mathcal{S}(P)} T \wedge T \not\mapsto$ and $T \downarrow^{\mathrm{fn}(P)} \hat{x}$.*

The proof is by induction on the rules of the two observation predicates; the π-calculus observation predicate for the forward direction, and the ρ-calculus observation predicate for the other direction. The proof is given in Section 3.11.3.

The next result we desire to have is that the encoding should satisfy the property of *operational correspondence* (Property 3), which ensures that the encoding in fact preserves the semantics of the source language. It can be stated in two parts: *operational completeness* for the forward direction, and *operational soundness* for the other direction. We shall break it up accordingly, since both proofs are quite lengthy.

However, just as with observational correspondence above, we can also show a stronger version of operational correspondence than what is required in Property 3, since the translation only introduces a finite number of internal steps for each term, some of which may occur either before or after the committing step. The committing step itself may then expose some further internal steps, but the maximum number remains finite and bounded by the structure of the source term.

Proposition 4 (Operational completeness)**.** *If $P \to P'$ then there exist terms T_1, T_2 such that $[\![P]\!] \mapsto\!\!\!\!\Rightarrow^{\mathcal{S}(P)+\mathcal{S}(P')-2} T_1$ and $[\![P']\!] \mapsto^{\mathcal{S}(P')} T_2$ and $T_1 \sim^{\mathrm{fn}(P)} T_2$.*

The first part of this statement is given by Lemma 7, so our task is to show that the two reducts T_1 and T_2 actually correspond. The proof then proceeds by induction on the reduction rules. It can be found in Section 3.11.4.

For the other direction of operational correspondence, we can also state a stronger version of the result than that required in Property 3, by using Lemma 6. Being able to distinguish the π-calculus names enables us to force the translated terms to perform exactly one committing step, and then reduce by $\mapsto$ until no further internal steps are possible. The result of operational soundness, which we can show for the encoding, is therefore as follows:

Proposition 5 (Operational soundness)**.** *If there exists a term T_1' such that $[\![P]\!] \mapsto\!\!\!\!\Rightarrow^{k} T_1' \not\mapsto$, then $P \to P'$ and there exists a term T_2' such that $[\![P']\!] \mapsto^{\mathcal{S}(P')} T_2' \not\mapsto$ and $T_1' \sim^{\mathrm{fn}(P)} T_2'$ and $k = \mathcal{S}(P) + \mathcal{S}(P') - 2$.*

The proof is by induction on the structure of translated terms. It can be found in Section 3.11.5.

Operational soundness assures us that a translated term can only perform finitely many non-committing steps before it must perform a committing step, matching one reduction in the π-calculus. Thus, if a translated target term can take infinitely many steps, it must also take infinitely many *committing* steps, corresponding to infinitely many steps in the π-calculus. Divergence reflection therefore follows as a simple corollary of Proposition 5:

Corollary 5 (Divergence reflection)**.** $[\![P]\!] \to^{\omega} \implies P \to^{\omega}$.

In summary, let us consider how the encoding conforms to the criteria given in Definition 15:

1. Compositionality is not proved directly, but it can be verified by inspecting the clauses of the translation function that the encoding does not reduce the degree of parallelism of the source term. The role of the coordinating context C is here played by the name server.
2. Substitution invariance is shown in Proposition 2.
3. Operational correspondence is shown in Proposition 4 (operational completeness) and Proposition 5 (operational soundness).
4. Observational correspondence is shown in Proposition 3.
5. Divergence reflection follows as a corollary of the stricter formulation of operational correspondence (Corollary 5).
6. Parameter independence is shown in Proposition 1.

Thus we conclude that the encoding indeed is valid according to the presented criteria.

3.7 Correct name-usage in the ρ-calculus

To write meaningful programs in the ρ-calculus, one needs a reliable source of fresh names, such as the name server $!N(x, z, v, s)$. A main point here is that only the *structure* of these names matters, not their *behaviour* if they were to be executed as processes. Their behaviour would likely be meaningless, and they are therefore never intended to be dropped and run. Thus we would like to ensure that this never happens.

One obvious way to ensure this would be by means of a type system along the lines of Milner's simple type system for the π-calculus [83]. However, this is actually not trivial to achieve, because of the reflective nature of the ρ-calculus, and specifically the fact that names are constructed at runtime rather than being defined syntactically with $(\boldsymbol{\nu} x)P$, since it leaves us no place in the syntax to write the types for the names. In this section, we shall see how it nevertheless may be done, by creating a very simple type system for the ρ-calculus. This is illustrative, because it highlights another peculiar feature of the ρ-calculus; namely that the generated names are *free* in the continuation of an input.

3.7.1 The type system

The purpose of our simple type system is to ensure correct name-usage; i.e. that names intended to be used as channels will never be dropped, and conversely that

processes, that have been quoted for the purpose of process mobility, will not be used as names. We can capture this with a simple language of types, consisting of three different 'tags'.

Definition 21 (Language of types)**.** The 'type tags' are:

$$\tau \in \mathcal{T} ::= \texttt{nch} \mid \texttt{pch} \mid \texttt{proc} \qquad \blacksquare$$

Here `nch` denotes that the name is used as a channel to carry a channel (name); `pch` denotes that the name is used as a channel to carry a process; and `proc` denotes that the name *is* a process, intended to be dropped.

Next, we shall need the notion of a *type environment* $\Gamma : \mathcal{N} \to \mathcal{T}$ to record the types of names. In most type systems, this is given as a finite, partial function from names to types, written as a list of pairs $x_1 : \tau_1, \ldots, x_n : \tau_n$. We shall do the same here, but also add a 'default' value to be returned for all names *not* found in Γ, thus creating a total function. We shall write $\varnothing$ for the empty list.

Definition 22 (Γ lookup)**.** Let the lookup of a name x in Γ be defined thus:

$$(x' : \tau, \Gamma)(x) = \begin{cases} \tau & \text{if } x = x' \\ \Gamma(x) & \text{otherwise} \end{cases}$$
$$\varnothing(x) = \texttt{nch} \qquad \blacksquare$$

The reason for this slightly unusual definition of Γ is precisely the aforementioned peculiarity that new names will be *free* in the continuation of an input, and hence will require a type to be found in the type environment.

Since Γ is now a full function, we shall also need a different way to refer to the names recorded in Γ, as opposed to the domain of Γ; hence, we shall use $\mathrm{n}(\Gamma)$, defined thus:

Definition 23 (Names in Γ)**.** Let $\mathrm{n}(\Gamma)$ denote the names recorded in Γ, with

$$\mathrm{n}(x : \tau, \Gamma) = \{ x \} \cup \mathrm{n}(\Gamma)$$

and $\mathrm{n}(\varnothing) = \varnothing$. We write

$$x_1 \# \Gamma \triangleq \forall x_2 \in \mathrm{n}(\Gamma) \ . \ x_1 \not\equiv_{\mathcal{N}} x_2 \qquad \blacksquare$$

Definition 24 (Well-formed Γ)**.** We say that Γ is *well-formed*, if it holds for all pairs of names $x_1, x_2 \in \mathrm{n}(\Gamma)$, that $x_1 \not\equiv_{\mathcal{N}} x_2$. $\blacksquare$

In the following, we shall always require, that Γ is well-formed; i.e. that Γ never contains name-equivalent names.

Type judgments are of the form $\Gamma \vdash P$ for processes and $\Gamma \vdash x : \tau$ for names, and are defined by the rules in Figure 3.2. Note the two different rules for lift: if

$$[\text{T-VAR}]\frac{}{\Gamma \vdash x_1 : \Gamma(x_2)}\ (x_1 \equiv_{\mathcal{N}} x_2) \qquad [\text{T-LIFT-N}]\frac{\Gamma \vdash x : \texttt{nch}}{\Gamma \vdash x\langle\!|P|\!\rangle}$$

$$[\text{T-NIL}]\frac{}{\Gamma \vdash \mathbf{0}} \qquad [\text{T-LIFT-P}]\frac{\Gamma \vdash x : \texttt{pch} \quad \Gamma \vdash P}{\Gamma \vdash x\langle\!|P|\!\rangle}$$

$$[\text{T-PAR}]\frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \qquad [\text{T-DROP}]\frac{\Gamma \vdash x : \texttt{proc}}{\Gamma \vdash \neg x \urcorner}$$

$$[\text{T-IN}]\frac{\Gamma \vdash x : \tau_1 \quad \Gamma, y : \tau_2 \vdash P}{\Gamma \vdash x(y).P}\ \left(\tau_2 = \begin{cases} \texttt{nch} & \text{if } \tau_1 = \texttt{nch} \\ \texttt{proc} & \text{if } \tau_1 = \texttt{pch} \end{cases}\right)$$

Figure 3.2: Rules for type judgments for names and processes.

x has type `nch`, then the lifted process is intended to be used as a name, and we type it with the rule [T-LIFT-N], which does not recur into the process P, since it is not intended to be run. Conversely, if the type of x is `pch`, then P may be dropped, and it must therefore also be checked, as is done in the premise of [T-LIFT-P]. This distinction appears again in the [T-IN] rule, where the type τ_2 of y, which is added as an assumption for the typing of the continuation P, will depend on the type of the subject, x. In all three cases, we require the subject to have a channel type, either `nch` or `pch`, whereas in [T-DROP] we require x to have a process type, `proc`. Thus, the rules capture our intended notion of correct name usage.

3.7.2 Properties of the type system

The property we wish to show for our type system is a classic subject-reduction property in the style of Wright and Felleisen [133], which says that well-typedness is preserved by the semantics; i.e.

$$\Gamma \vdash P \wedge P \to^* P' \implies \Gamma \vdash P'$$

To obtain this result, we shall need a few auxiliary lemmas, showing that well-typedness is preserved by weakening and strengthening of Γ, and by substitution and rewrites of P under structural congruence.

Lemma 8 (Weakening). *If* $\Gamma \vdash P$ *and* $x\#\Gamma, P$ *then* $\Gamma, x : \tau \vdash P$ *for some* τ.

Lemma 9 (Strengthening). *If* $\Gamma, x : \tau \vdash P$ *and* $x\#\,\mathrm{fn}(P)$ *then* $\Gamma \vdash P$.

Both results are easily shown by induction on the structure of P, so we omit the details. The substitution result is more interesting, since here we need to distinguish

between the syntactic and semantic substitution. Thus we break it up into two separate lemmas:

Lemma 10 (Syntactic substitution)**.** *Let* $\sigma = \{z/x\}$ *be a syntactic substitution. If* $\Gamma, x : \tau \vdash P$ *and* $z\#\Gamma, P$ *then* $\Gamma, z : \tau \vdash P\{z/x\}$.

Proof. Induction on the structure of P.

- Case $\mathbf{0}$: immediate.
- Case $P_1 \mid P_2$: Then $\Gamma, x : \tau \vdash P_1 \mid P_2$ was concluded by [T-PAR], with $\Gamma, x : \tau \vdash P_1$ and $\Gamma, x : \tau \vdash P_2$ as premises. By induction hypothesis, $\Gamma, z : \tau \vdash P_1\sigma$ and $\Gamma, z : \tau \vdash P_2\sigma$ then hold. As $P_1\sigma \mid P_2\sigma = (P_1 \mid P_2)\sigma$, we then conclude $\Gamma, z : \tau \vdash (P_1 \mid P_2)\sigma$.
- Case $x_1\langle\!|P|\!\rangle$: Then $\Gamma, x : \tau \vdash x_1\langle\!|P|\!\rangle$ was concluded by either [T-LIFT-N] or [T-LIFT-P]. We consider only the latter case, since the former is simpler. From the premise, we have that $\Gamma, x : \tau \vdash x_1 : \mathtt{nch}$ and $\Gamma, x : \tau \vdash P$, so $\Gamma, z : \tau \vdash \sigma(x_1) : \mathtt{nch}$ and $\Gamma, z : \tau \vdash P\sigma$ hold by the induction hypothesis. As $\sigma(x_1)\langle\!|P\sigma|\!\rangle = x_1\langle\!|P|\!\rangle\sigma$, we then conclude $\Gamma, z : \tau \vdash x_1\langle\!|P|\!\rangle\sigma$.
- Case $x_1(y).P$: Then $\Gamma, x : \tau \vdash x_1(y).P$ was concluded by [T-IN]. From the premise, we have that $\Gamma, x : \tau \vdash x_1 : \tau_1$ and $\Gamma, x : \tau, y : \tau_2 \vdash P$, with τ_2 determined by the side condition. As the substitution is required to be safe, we know that $y\#\sigma$. By the induction hypothesis, $\Gamma, z : \tau \vdash \sigma(x_1) : \tau_1$ and $\Gamma, z : \tau, y : \tau_2 \vdash P\sigma$ then hold. By factoring σ as before, we get that $\Gamma, z : \tau, y : \tau_2 \vdash x_1(y).P\sigma$, and by applying Lemma 9 we conclude $\Gamma, z : \tau \vdash x_1(y).P\sigma$.
- Case $\ulcorner x_1 \urcorner$: Then $\Gamma, x : \tau \vdash \ulcorner x_1 \urcorner$ was concluded by [T-DROP], with $\Gamma, x : \tau \vdash x_1 : \mathtt{proc}$ as premise. By the induction hypothesis, $\Gamma, z : \tau \vdash \sigma(x_1) : \mathtt{proc}$ then holds. As $\ulcorner\sigma(x_1)\urcorner = \ulcorner x_1 \urcorner\sigma$ (since this is syntactic substitution), we therefore conclude $\Gamma, z : \tau \vdash \ulcorner x_1 \urcorner\sigma$.
- Case x_1: Then $\Gamma, x : \tau \vdash x_1 : \tau'$ was concluded by [T-VAR]. If $x \not\equiv_{\mathcal{N}} x_1$, the result is immediate. Otherwise, $\tau = \tau'$ and $x_1\{z/x\} = z$, and thus $\Gamma, z : \tau \vdash z : \tau$ is seen to hold.

This concludes the proof. □

Lemma 11 (Semantic substitution)**.** *Let* $\sigma = \{\ulcorner Q \urcorner/x\}$ *be a semantic substitution. If* $\Gamma, x : \mathtt{proc} \vdash P$*,* $x\#Q$ *and* $\Gamma \vdash Q$ *then* $\Gamma, x : \mathtt{proc} \vdash P\{\ulcorner Q \urcorner/x\}$.

Proof. Induction on the structure of P. Most of the cases are as in Lemma 10, so we shall not repeat them. The interesting case is that of $\ulcorner x_1 \urcorner$:

- If $x \not\equiv_{\mathcal{N}} x_1$ the result is immediate, since no substitution is performed.
- Otherwise, we have that $\ulcorner x_1 \urcorner \{\ulcorner Q \urcorner / x\} = Q$, and $\Gamma, x : \texttt{proc} \vdash Q$ holds by assumption.

This concludes the proof. □

Next, we shall need a result showing that well-typedness is preserved by rewrites under structural congruence.

Lemma 12 (Structural congruence)**.** *If* $\Gamma \vdash P_1$ *and* $P_1 \equiv P_2$ *then* $\Gamma \vdash P_2$

Proof. Induction on the rules for concluding $P_1 \equiv P_2$ (Figure 3.1).

For [S-REFL], the result is immediate, and for [S-SYM] and [S-TRANS] by straightforward application of the induction hypothesis. For [S-COM], [S-NIL], [S-PAR] and [S-ASS], the result follows by using the type rule [T-PAR] either once or twice, and then by the induction hypothesis for the premises.

The interesting cases are the remaining, [S-DROP], [S-OUT] and [S-IN]:

- Case [S-DROP]: We know that $\ulcorner x_1 \urcorner \equiv \ulcorner x_2 \urcorner$ and $\Gamma \vdash \ulcorner x_1 \urcorner$, which was concluded by [T-DROP], and, from the premise, we therefore have that $\Gamma \vdash x_1 : \texttt{proc}$, which was concluded by [T-VAR]. By well-formedness of Γ and this rule, since $x_1 \equiv_{\mathcal{N}} x_2$, we therefore conclude that $\Gamma \vdash x_2 : \texttt{proc}$. Thus, by [T-DROP] we can conclude $\Gamma \vdash \ulcorner x_2 \urcorner$.
- Case [S-OUT]: We know that $x_1 \langle\!| P_1 |\!\rangle \equiv x_2 \langle\!| P_2 |\!\rangle$, and, from the premise, that $x_1 \equiv_{\mathcal{N}} x_2$ and $P_1 \equiv P_2$. We also know that $\Gamma \vdash x_1 \langle\!| P_2 |\!\rangle$, which must have been concluded by [T-LIFT-N] or [T-LIFT-P]. Assume the latter rule was used, since the other case is simpler. Then, from the premise, we have that $\Gamma \vdash x_1 : \texttt{pch}$ and $\Gamma \vdash P_1$. The former was concluded by [T-VAR], and by this rule, and by well-formedness of Γ, we can therefore conclude that $\Gamma \vdash x_2 : \texttt{pch}$. Since $P_1 \equiv P_2$, then by the induction hypothesis, $\Gamma \vdash P_2$. Thus we conclude by [T-LIFT-P] that $\Gamma \vdash x_2 \langle\!| P_2 |\!\rangle$.
- Case [S-OUT]: We know that $x_1(y_2).P_1 \equiv x_2(y_2).P_2$, and, from the premise, that $x_1 \equiv_{\mathcal{N}} x_2$ and $P_1\{z/y_1\} \equiv P_2\{z/y_2\}$, where z is chosen fresh for both P_1, P_2. We also know that $\Gamma \vdash x_1(y_1).P_1$, which was concluded by [T-IN]. From the premise of that rule, we then have that $\Gamma \vdash x_1 : \tau_1$ and $\Gamma, y_1 : \tau_2 \vdash P$. By a reasoning similar to above, we conclude that since $x_1 \equiv_{\mathcal{N}} x_2$, then also $\Gamma \vdash x_2 : \tau_1$.

  Assume $z \# \Gamma$, or otherwise apply Lemma 9 (Strengthening) until the statement holds. By Lemma 10 (Syntactic substitution) we can then conclude that $\Gamma, z : \tau_2 \vdash P_1\{z/y_1\}$, and we know from the premise of [S-IN] that $P_1\{z/y_1\} \equiv P_2\{z/y_2\}$. Then $\Gamma, z : \tau_2 \vdash P_2\{z/y_2\}$ by the induction hypothesis. By applying Lemma 10

again, we get that $\Gamma, y_2 : \tau_2 \vdash P$. Thus we can conclude $\Gamma \vdash x_2(y_2).P_2$ by [T-IN].

This concludes the proof. □

Note in the last case, for [S-OUT], that we used the lemma for *syntactic* substitution (Lemma 10), and not the lemma for semantic substitution (Lemma 11). This works, since the substitution used in the definition of structural congruence (Figure 3.1) is only syntactic substitution.

Before we can state and prove our main result of subject reduction, we need an assumption about the names that will be generated at runtime. For this purpose, we shall annotate the reduction relation with the object of a communication. Thus we shall write $P \xrightarrow{x} P'$. This annotation is introduced in the [ρ-COM] rule, where we conclude

$$x_1(y).P_1 \mid x_2\langle P_2 \rangle \xrightarrow{\ulcorner P_2 \urcorner} P_1\{\ulcorner P_2 \urcorner / y\}$$

and it is then straightforwardly extended to the [ρ-PAR] and [ρ-STRUCT] rules. For the reflexive and transitive closure of $\xrightarrow{x}$, we shall write $\rightarrow^*$ as before.

Definition 25 (Name generation safe)**.** We say that a process P is *name generation safe* for an environment Γ, written $\mathrm{NSafe}_\Gamma(P)$, if it holds that

$$P \rightarrow^* P'' \xrightarrow{x} P' \implies \Gamma(x) = \mathtt{nch}$$ ■

This requirement can manifest itself in two different ways:

1. If $P \xrightarrow{x_1} P'$ and there exists a name $x_2 \in \mathrm{n}(\Gamma)$ such that $x_1 \equiv_{\mathcal{N}} x_2$, then x_2 must be of type `nch`.
2. Conversely, if *no* such name x_2 exists in $\mathrm{n}(\Gamma)$, i.e. $x_2 \notin \mathrm{n}(\Gamma)$, then $\Gamma(x_1) = \mathtt{nch}$ by default.

Thus in effect, this requirement states that P must never, during the course of its reductions, create a name x_1 that is name equivalent to any name x_2 that is assumed to have type `pch` or `proc` in Γ. This limitation is necessitated by the fact that names are free in the continuation of an input. We return to this matter in the following section.

Theorem 3 (Subject reduction)**.** *If* $\mathrm{NSafe}_\Gamma(P)$ *and* $\Gamma \vdash P$ *and* $P \rightarrow P'$ *then* $\Gamma \vdash P'$.

Proof. Induction on the rules of $P \rightarrow P'$ (Definition 7). We have three cases to examine, but the interesting case is the rule [ρ-COM]:

- Case [ρ-PAR]: We have $\Gamma \vdash P_1 \mid P_2$ by [T-PAR], and, from the premise, that $\Gamma \vdash P_1$ and $\Gamma \vdash P_2$. Now $P_1 \mid P_2 \rightarrow P_1' \mid P_2$, and by the induction hypothesis $\Gamma \vdash P_1'$. Then by [T-PAR] we conclude that $\Gamma \vdash P_1' \mid P_2$.

- Case [ρ-STRUCT]: We have by assumption that $\Gamma \vdash P$, and $P \to P'$ was concluded by [ρ-STRUCT]. From the premise, $P \equiv P_1$, and by Lemma 12, we have that $\Gamma \vdash P_1$. Then $P_1 \to P_1'$ in the premise, and by the induction hypothesis $\Gamma \vdash P_1'$. Then $P_1' \equiv P'$, and we conclude $\Gamma \vdash P'$ by another application of Lemma 12.

- Case [ρ-COM]: We have by assumption that $\Gamma \vdash x_1(y).P_1 \mid x_2\langle\!\langle P_2 \rangle\!\rangle$, which was concluded by [T-PAR], and [T-IN] and either [T-LIFT-N] or [T-LIFT-P] for the premises. We examine each case separately:

  1. If [T-LIFT-N] was used, then we know $\Gamma \vdash x_1 : \mathtt{nch}$, and by well-formedness of Γ that also $\Gamma \vdash x_2 : \mathtt{nch}$ since $x_1 \equiv_{\mathcal{N}} x_2$. By the side condition of [T-IN], we then have that $\tau_2 = \mathtt{nch}$, so $\Gamma, y : \mathtt{nch} \vdash P_1$. Now

  $$x_1(y).P_1 \mid x_2\langle\!\langle P_2 \rangle\!\rangle \to P_1\{\ulcorner P_2 \urcorner / y\}$$

  by [ρ-COM]. We know that $\ulcorner P_2 \urcorner$ will be used as a *name* inside P, so there will only be a substitution of a name for a name. Thus by Lemma 10 (Syntactic substitution), we have that $\Gamma, \ulcorner P_2 \urcorner : \mathtt{nch} \vdash P_1\{\ulcorner P_2 \urcorner / y\}$.

  Now, if there exists a name $x \in \mathrm{n}(\Gamma)$ such that $x \equiv_{\mathcal{N}} \ulcorner P_2 \urcorner$, then we know that $\Gamma \vdash x : \mathtt{nch}$, since we require that $\mathsf{NSafe}_\Gamma(P)$. Thus the assumption $\ulcorner P_2 \urcorner : \mathtt{nch}$ can be removed, to comply with the well-formedness requirement. Conversely, if there does *not* exist such an x, then the assumption can still be omitted, since we then have $\Gamma(\ulcorner P_2 \urcorner) = \mathtt{nch}$ as default value by Definition 23. Thus we can conclude $\Gamma \vdash P_1\{\ulcorner P_2 \urcorner / y\}$.

  2. If [T-LIFT-P] was used, then we know that $\Gamma \vdash P_2$ and $\Gamma \vdash x_1 : \mathtt{pch}$, and by well-formedness of Γ that also $\Gamma \vdash x_2 : \mathtt{pch}$. By the side condition of [T-IN], we then have that $\tau_2 = \mathtt{proc}$, so from the premise we know that $\Gamma, y : \mathtt{proc} \vdash P_1$.

  As before, following the reduction, we have $P_1\{\ulcorner P_2 \urcorner / y\}$, and we know that $\ulcorner P_2 \urcorner$ will be used as a *process* inside P_1. Furthermore, we know that the substitution is safe (capture-avoiding), so $y \# P_2$. We can therefore apply Lemma 11 (Semantic substitution) to obtain $\Gamma, y : \mathtt{proc} \vdash P_1\{\ulcorner P_2 \urcorner / y\}$. Since $y \# P_2$, and all occurrences of y (and name equivalent names) within P_1 are replaced, we have that $y \# P_1\{\ulcorner P_2 \urcorner / y\}$, so we can apply Lemma 9 to obtain $\Gamma \vdash P_1\{\ulcorner P_2 \urcorner / y\}$.

This concludes the proof. □

By a simple, inductive argument and Theorem 3, we then obtain our desired safety result:

Corollary 6. $\mathsf{NSafe}_\Gamma(P) \wedge \Gamma \vdash P \wedge P \to^* P' \implies \Gamma \vdash P'$

3.7.3 Safety of the encoding

We wish to show that programs using the name server $!N(x, z, v, s)$ will use the generated names correctly, i.e. that they will not be dropped. One particularly important class of such programs are of course those created by the encoding of the π-calculus into the ρ-calculus.

We need to make a few, insignificant changes to the encoding, to make it fit with our type rules: we introduce an extra name parameter p to be used in the encoding of replication. Formerly, the encoding just used n for this, but now we want to give n the type `nch`, since it is used for sending names, and we therefore need a different name with type `pch` for sending processes. With these changes, the encoding is now as follows:

$$\begin{aligned}
[\![\mathbf{0}]\!]_{n,p,v} &= \mathbf{0} \\
[\![P_1 \mid P_2]\!]_{n,v,p} &= [\![P_1]\!]_{+n,p,v} \mid [\![P_2]\!]_{n+,p,v} \\
[\![x(y).P]\!]_{n,p,v} &= x(y).[\![P]\!]_{n,p,v} \\
[\![x\langle z\rangle]\!]_{n,p,v} &= x\langle z\rangle \\
[\![(\boldsymbol{\nu} x)P]\!]_{n,p,v} &= v\langle n\rangle \mid n(x).[\![P]\!]_{n^2,p,v} \\
[\![!x(y).P]\!]_{n,p,v} &= D(p) \mid p\langle\!\langle x(y).\big(D(p) \mid [\![P]\!]_{n,p^2,v}\big)\rangle\!\rangle
\end{aligned}$$

We now need to find an environment Γ_1, containing all the names of the translated process, to show that it is well-typed. This is quite easy:

- For a π-calculus process P, we assign all names $x \in \mathrm{fn}(P)$ the type $x : \mathtt{nch}$ and add them to Γ_1.
- Assign $v : \mathtt{nch}$ and add it to Γ_1.
- For all names n generated by the translation, we assign $n : \mathtt{nch}$ and add them to Γ_1.
- For all names p generated by the translation, we assign $p : \mathtt{pch}$ and add them to Γ_1.

Note that we can simply count all the names n (resp. p) generated by the translation. We could also build Γ_1 alongside the encoding, but we omit that to avoid clutter in the notation. It should now be clear that any process $[\![P]\!]_{n,p,v}$ generated by this translation is well-typed:

Proposition 6. *With Γ_1 built as described above, $\Gamma_1 \vdash [\![P]\!]_{n,p,v}$ for any π-calculus process P.*

Proof. By induction on the clauses of the translation:

- Case $\mathbf{0}$: Immediate from [T-NIL].
- Case $[\![P_1]\!]_{+n,p,v} \mid [\![P_2]\!]_{n+,p,v}$: By [T-PAR], and we apply the induction hypothesis for the premises.
- Case $x(y).[\![P]\!]_{n,p,v}$: By [T-IN]. By construction, $x : \texttt{nch}$ since this name came from the π-calculus. Thus we get $y : \texttt{nch}$ from the side condition. The premise holds by the induction hypothesis, and thus we get that $\Gamma_1, y : \texttt{nch} \vdash x(y).[\![P]\!]_{n,p,v}$. Applying Lemma 9 then allows us to conclude $\Gamma_1 \vdash x(y).[\![P]\!]_{n,p,v}$.
- Case $x\langle z\rangle$: By [T-LIFT-N]. By construction, $\Gamma_1(x) = \texttt{nch}$ since it came from the π-calculus.
- Case $v\langle n\rangle \mid n(x).[\![P]\!]_{n^2,p,v}$: Firstly by [T-PAR], and then [T-LIFT-N] and [T-IN] for the premises. By construction of Γ_1, we have that $\Gamma_1(v) = \texttt{nch}$ (and $\Gamma_1(n) = \texttt{nch}$), so $\Gamma_1 \vdash v\langle n\rangle$. Similarly for $n(x).[\![P]\!]_{n^2,p,v}$, since we know that $\Gamma_1(n) = \texttt{nch}$, and this case is thus similar to the case for input above.
- Case $D(p) \mid p\langle\!\langle x(y).(D(p) \mid [\![P]\!]_{n,p^2,v})\rangle\!\rangle$: Firstly by [T-PAR], and then by [T-IN] and [T-LIFT-P] for the premises. By construction, we have that $\Gamma_1(p) = \texttt{pch}$.

This concludes the proof. □

This result is hardly surprising: All names coming from the π-calculus are of course used as channels to communicate names, and so are the names introduced by the translation, except p and its derivates, which are used to encode replication. Furthermore, all occurrences of $\ulcorner x \urcorner$ are in the copying process $D(p)$, where the name is bound by an input; hence, there are no drop of free names, and therefore no names recorded with a type $\texttt{proc}$ in Γ_1.

Next, we consider the name server process $!N(x, z, v, s)$ itself, which we recall is defined as follows:

$$D(x) \mid x\langle\!\langle z(a).v(r).(D(x) \mid r\langle a\rangle \mid z\langle\!\langle a\langle\ulcorner \mathbf{0} \urcorner\rangle\rangle\!\rangle\rangle\!\rangle \mid z\langle s\rangle$$

The key part here is the process $z\langle\!\langle a\langle\ulcorner \mathbf{0} \urcorner\rangle\rangle\!\rangle$, which is the part that actually outputs the next new name to the requesting process.

Firstly, the free names in the name server must be recorded in a Γ_2 with the following type assignments: $v : \texttt{nch}$ (matching the type used in the translation function), $x : \texttt{pch}, z : \texttt{nch}, s : \texttt{nch}$. Now, by using the typing rules, it should be easy to see that we can conclude that

$$\Gamma_2 \vdash D(x) \mid x\langle\!\langle z(a).v(r).(D(x) \mid r\langle a\rangle \mid P)\rangle\!\rangle \mid z\langle s\rangle$$

relative to some Γ_2 containing the above assignments, *if* it is the case that

$$\Gamma_2, a : \texttt{nch}, r : \texttt{nch} \vdash P$$

for some P. Here P represents the name-generating part of the code, in this case $z\langle\!|a\texttt{<}\ulcorner\mathbf{0}\urcorner\texttt{>}|\!\rangle$. We have that a gets the type `nch` from z, and r gets the type `nch` from v, both by the side condition in the rule [T-IN]. Lastly,

$$\Gamma_2, a : \texttt{nch}, r : \texttt{nch} \vdash z\langle\!|a\texttt{<}\ulcorner\mathbf{0}\urcorner\texttt{>}|\!\rangle$$

can be concluded by [T-LIFT-N], with $\Gamma_2, a : \texttt{nch}, r : \texttt{nch} \vdash z : \texttt{nch}$ for the premise, which holds by construction of Γ_2. The inner process, $a\texttt{<}\ulcorner\mathbf{0}\urcorner\texttt{>}$, is precisely not checked by this rule, since it is assumed to be a name; hence, we are done. Combining this with Proposition 6 then immediately gives us the following:

Corollary 7. $\Gamma_1, \Gamma_2 \vdash [\![P]\!]_{n,p,v} \mid !N(x, z, v, s)$

Finally, we must argue that the encoding satisfies the requirement in Definition 25, i.e. that $\mathsf{NSafe}_{\Gamma_1,\Gamma_2}\big([\![P]\!]_{n,p,v} \mid !N(x, z, v, s)\big)$. For this, let $\Gamma = \Gamma_1, \Gamma_2$. We must argue that no name created at runtime by $[\![P]\!]$, including communicated processes, will be name equivalent to any name in Γ that has type `pch` or `proc`.

Firstly, notice that the translation produces no drop processes $\ulcorner x \urcorner$ where x is *not* bound, *except* inside lifts where the subject has type `nch`. These free drops occur in $[\![x\texttt{<}z\texttt{>}]\!]_{n,p,v} = x\texttt{<}z\texttt{>}$ in the translation function, where we recall that this notation is a short-hand for $x\langle\!|\ulcorner z \urcorner|\!\rangle$. However, since both x and z come from the π-calculus, they are given the type `nch` in the construction of Γ, if they are free. Thus, as the subject has type `nch`, these lifts are typed with the rule [T-LIFT-N], which does *not* check the lifted process in object position; hence it does not cause a type error that z in $\ulcorner z \urcorner$ is given the type `nch`, rather than `proc`.

The same applies to the lifts that derive from the translation of $(\boldsymbol{\nu} x)P$, i.e. $v\texttt{<}n\texttt{>}$, and the two subterms in the name server $z\langle\!|a\texttt{<}\ulcorner\mathbf{0}\urcorner\texttt{>}|\!\rangle$ and $z\texttt{<}s\texttt{>}$. $\ulcorner\mathbf{0}\urcorner$ is not given a type at all in Γ, and n and s (for all statically generated choices of these names) are given the type `nch` in the construction of Γ. Thus we conclude that there are *no* names of type `proc` in Γ.

Now consider the names $p, p^2, \ldots$ generated by the translation: These all belong to the namespace $\mathcal{N}^{\circ}_{[p_0]}$, generated from the root p_0. This is a parameter which, by requirement, must be chosen such that it does not overlap with the namespace ${}^{+}\mathcal{N}_{[s_0]}$ generated at runtime by the name server, and this is assumed in the definition of the encoding itself.

This leaves us with a single case to check: We must ensure that no lifted *process* ever coincides with any name in the namespace $\mathcal{N}^{\circ}_{[p_0]}$. Again, this can be verified by inspecting the definition of the encoding: Lifted processes, where the subject has type `pch`, only occur in two places:

- In the encoding of replication, where the lifted process is of the form

$$x(y).\big(D(p) \mid [\![P]\!]_{n,p^2,v}\big)$$

  But names in the namespace $\mathcal{N}^{\circ}_{[p_0]}$ are of the shape $\ulcorner N^{\circ}\langle\!\langle \mathbf{0} \rangle\!\rangle \mid N^{\circ}(\ulcorner \mathbf{0} \urcorner).\mathbf{0}\urcorner$. Thus, such a quoted process cannot be name equivalent to any name in $\mathcal{N}^{\circ}_{[p_0]}$, since the process within cannot be structurally congruent to the lifted process.

- In the name server, which also uses process mobility to replicate. Here, the lifted process is of the form

$$z(a).v(r).(D(x) \mid r\texttt{<}a\texttt{>} \mid z\langle\!\langle a\texttt{<}\ulcorner \mathbf{0} \urcorner\texttt{>} \rangle\!\rangle)$$

  The same argument as above therefore applies.

Thus we may conclude that $\mathsf{NSafe}_{\Gamma_1,\Gamma_2}([\![P]\!])$ indeed holds, so we can apply Corollary 6 to conclude that the translation indeed uses all names safely.

3.7.4 On typing the ρ-calculus

The type system works for the present purpose, but is, in general, quite limiting. However, it serves to illustrate some of the difficulties involved in reasoning about structured names, which may be generated at runtime. We had to impose two major restrictions in the type system:

1. We disallow sending of any names of type `pch`.
2. We only allow names to be created with the type `nch`.

These limitations are a consequence of our choice to make `nch` the 'default' type returned for any name not found in Γ. This solves a key problem in the ρ-calculus: If new, *free* names can be created at runtime, what type should they then be given? There are two obvious solutions to this problem:

1. Names are structured entities, so the type of a name should somehow be derived from the structure of the name. This would in a sense be similar to the approach we used in the proof of Proposition 5 to distinguish committing and non-committing steps. For example, one could say all names built from ${}^{+}\mathcal{N}_{[\,]}$ had type `nch`, and all names from $\mathcal{N}^{\circ}_{[\,]}$ had type `pch` and so on. We are not aware of any type system that employs this approach.
2. All new names are free. Thus, another possibility is to assign a type in advance to all names that ever *will* (or might) be created by the program at runtime. This is the solution employed in the type system by Hüttel et al. in [64] (see

Chapter 4). However, it then obviously necessitates that the programmer should know in advance all the names that eventually might be created by the program at runtime.

The present type system employs a simpler strategy than the second approach by always just assigning the same type to all created names; namely `nch`. In one sense, this is less flexible, but on the other hand it is easier, since we do not need to pre-assign types to all the names in advance. However, it then necessitates the $\mathrm{NSafe}_\Gamma(P)$ restriction, which must be checked 'outside' of the type system itself, since it would be unsound if it were possible to create the same name twice with two different types. This restriction also applies to the type system in [64], albeit indirectly, since it requires that name-equivalent names must have the same type.

In the specific case of the name server and the π-calculus encoding, this possibility is ruled out by construction, because the parameters p and s are chosen from distinct namespaces. And, as shown in section 3.6, a name from a namespace $\mathcal{N}^1_{[\,]}$ can never be used to construct a name from a namespace $\mathcal{N}^2_{[\,]}$, and vice versa, as long as they have distinct roots.

3.8 A separation result

The ρ-calculus can encode the π-calculus, as we saw in section 3.6. However, the converse does not hold. Under some general assumptions about the behavioural equivalence $\simeq$ used in the target language, we can show that there cannot be an encoding of the ρ-calculus into the π-calculus, that satisfies our validity criteria from Definition 15. This result relies on a few, simple facts about substitution and observability in the π-calculus:

Lemma 13. *Let $\sigma_t = \{x/n\}$ be a substitution in the π-calculus, with $n \in \mathrm{fn}(P)$ and $x\#P$. Then $P \to P' \implies P\sigma_t \to P'\sigma_t$.*

This can easily be shown by induction on the semantic rules, and then with an extra induction on structural congruence for the [π-STRUCT] rule.

Lemma 14. $P \downarrow \hat{x} \implies x \in \mathrm{fn}(P)$.

This is shown by induction on the rules of the observation predicate. It follows from the quite obvious fact that the only names we can observe are the free names in subject position of unguarded input and output operations.

Lemma 15. $P \Downarrow \hat{x} \implies x \in \mathrm{fn}(P)$.

This is shown by induction on the length of the reduction sequence, applying Lemma 14 at each step. Again, this is quite intuitive, as the only way to declare a new name in the π-calculus is by the $(\boldsymbol{\nu} z)P$ operator, but this name is precisely *bound* and

hence not observable. Thus, the set of free names in a process can only decrease or remain constant under reduction, but never increase.

Next, we consider our requirements for the notion of behavioural equivalence: First of all, $\simeq$ should obviously be an equivalence relation. Secondly, it should in some sense preserve the semantics of the processes it equates: as we are here working in a reduction system, it should at least preserve reductions and observability, and it should be preserved under substitution:

Definition 26 (Behavioural equivalence requirements)**.** We require that $\simeq$ be at least an equivalence relation over π-terms satisfying the following:

1. $P_1 \simeq P_2 \wedge P_1 \to^* P_1' \implies \exists P_2' \,.\, P_2 \to^* P_2' \wedge P_1' \simeq P_2'$
2. $P_1 \simeq P_2 \wedge P_1 \Downarrow \hat{x} \implies P_2 \Downarrow \hat{x}$
3. $P_1 \simeq P_2 \implies P_1\sigma_t \simeq P_2\sigma_t$ ■

The requirements suggest that $\simeq$ should be at least weak, barbed congruence, which does not seem too demanding. However, we prefer to keep the formulation general, without committing to one specific notion of behavioural equivalence, to emphasise that other, stronger choices are also possible. The following result will then hold for any such choice:

Theorem 4 (Separation)**.** *If $\simeq$ satisfies the requirements of Definition 26, then there is no encoding of the ρ-calculus into the π-calculus satisfying the criteria of Definition 15.*

Proof. Assume to the contrary that there exists a translation

$$[\![\,]\!]_N : \mathcal{P}_\rho \to \mathcal{P}_\pi$$

with a renaming policy φ and a name-derivation function δ, satisfying the criteria of Definition 15. We show that this leads to a contradiction. Firstly, let $u \triangleq \ulcorner\urcorner x_1 \urcorner \mid \urcorner x_2 \ulcorner\urcorner$, and consider the processes P and P' where

$$\begin{aligned} P &\triangleq P_1 \mid P_2 \\ P_1 &\triangleq a\langle\!|\, \urcorner x_1 \ulcorner \mid \urcorner x_2 \ulcorner \,|\!\rangle \\ P_2 &\triangleq a(n).n\langle\!|\, \mathbf{0} \,|\!\rangle \\ P' &\triangleq u\langle\!|\, \mathbf{0} \,|\!\rangle \end{aligned}$$

Thus

$$P = a\langle\!|\, \urcorner x_1 \ulcorner \mid \urcorner x_2 \ulcorner \,|\!\rangle \mid a(n).n\langle\!|\, \mathbf{0} \,|\!\rangle$$

and clearly $P \not\downarrow u$ and $u \notin \mathrm{fn}(P)$, but

$$P \to \ulcorner\urcorner x_1 \ulcorner \mid \urcorner x_2 \ulcorner\urcorner\langle\!|\, \mathbf{0} \,|\!\rangle = u\langle\!|\, \mathbf{0} \,|\!\rangle = P'$$

and $P' \downarrow u$.

Consider now the π-calculus substitution $\sigma_t \triangleq \{m/\varphi(u)\}$ for some fresh name m, i.e. $m \neq \varphi(u)$ and with $m \notin \mathrm{fn}(\llbracket P \rrbracket_N)$. By the contrapositive of Lemma 15, we have that $m \notin \mathrm{fn}(\llbracket P \rrbracket_N) \implies \llbracket P \rrbracket_N \not\Downarrow m$.

Now consider the term $\llbracket P \rrbracket_N \sigma_t$: Lemma 13 yields

$$\llbracket P \rrbracket_N \sigma_t \rightarrow^* T'\sigma_t \simeq \llbracket P' \rrbracket_{N'} \sigma_t$$

and by criterion 2 (substitution invariance) $\llbracket P' \rrbracket_{N'} \sigma_t \simeq \llbracket P' \sigma_s \rrbracket$. As we know that $P' \downarrow u$, this implies that $P'\sigma_s \downarrow \sigma_s(u)$, which again implies that $\llbracket P'\sigma_s \rrbracket_{N'} \Downarrow \sigma_t(\varphi(u))$, which implies $\llbracket P' \rrbracket_{N'} \sigma_t \Downarrow m$. This establishes that

$$\llbracket P \rrbracket_N \sigma_t \rightarrow^* T'\sigma_t \wedge T'\sigma_t \simeq \llbracket P' \rrbracket_{N'} \sigma_t \wedge \llbracket P' \rrbracket_{N'} \sigma_t \Downarrow m$$

as expected. By requirement 2 in Definition 26, since $\llbracket P' \rrbracket_{N'} \sigma_t \simeq T'\sigma_t$, it must therefore also be the case that $T'\sigma_t \Downarrow m$, and hence that $\llbracket P \rrbracket_N \sigma_t \Downarrow m$.

However, consider now the effect of applying the substitution $\llbracket P \rrbracket_N \sigma_t$. By criterion 1 (compositionality), we have that

$$\llbracket P \rrbracket_N \sigma_t = C\sigma_t \mid \llbracket P_1 \rrbracket_{N_1} \sigma_t \mid \llbracket P_2 \rrbracket_{N_2} \sigma_t = C \mid \llbracket P_1 \rrbracket_{N_1} \sigma_t \mid \llbracket P_2 \rrbracket_{N_2} \sigma_t$$

where we can eliminate the substitution from C, since $\varphi(u) \notin N \cup N_1 \cup N_2$, as this immediately would violate criterion 6 (parameter independence); and as we know that $u \notin \mathrm{fn}(P)$, we therefore also know that $\varphi(u) \notin \mathrm{fn}(C)$, since C at most can contain a subset of the (φ-translated) free names of the process and the parameters. Thus the substitution has no effect on C.

Now consider the two subterms $\llbracket P_1 \rrbracket_{N_1} \sigma_t$ and $\llbracket P_2 \rrbracket_{N_2} \sigma_t$. By criterion 2, $\llbracket P_1 \rrbracket_{N_1} \sigma_t \simeq \llbracket P_1 \sigma_s \rrbracket_{N_1}$ and $\llbracket P_2 \rrbracket_{N_2} \sigma_t \simeq \llbracket P_2 \sigma_s \rrbracket_{N_2}$, but when we apply the substitution, we get that

$$\begin{aligned} P_1\sigma_s &= (a\langle\!\langle \ulcorner x_1 \urcorner \mid \ulcorner x_2 \urcorner \rangle\!\rangle)\,\sigma_s = a\langle\!\langle \ulcorner x_1 \urcorner \mid \ulcorner x_2 \urcorner \rangle\!\rangle = P_1 \\ P_2\sigma_s &= (a(n).n\langle\!\langle \mathbf{0} \rangle\!\rangle)\,\sigma_s = a(n).n\langle\!\langle \mathbf{0} \rangle\!\rangle = P_2 \end{aligned}$$

since obviously $u \notin \mathrm{fn}(P_1)$ and $u \notin \mathrm{fn}(P_2)$, so the substitution has no effect on any of the subterms. This establishes that

$$C \mid \llbracket P_1 \sigma_s \rrbracket_{N_1} \mid \llbracket P_2 \sigma_s \rrbracket_{N_2} = C \mid \llbracket P_1 \rrbracket_{N_1} \mid \llbracket P_2 \rrbracket_{N_2}$$

and thus that $\llbracket P\sigma_s \rrbracket_N = \llbracket P \rrbracket_N$.

Criterion 2 (substitution invariance) requires that $\llbracket P\sigma_s \rrbracket_N \simeq \llbracket P \rrbracket_N \sigma_t$, but since we now know that $\llbracket P\sigma_s \rrbracket_N = \llbracket P \rrbracket_N$, it must therefore be the case that $\llbracket P \rrbracket_N \simeq \llbracket P \rrbracket_N \sigma_t$. This then yields the desired contradiction, since, as established above, $\llbracket P \rrbracket_N \sigma_t \Downarrow m$ but $\llbracket P \rrbracket_N \not\Downarrow m$, whilst by requirement 2 of Definition 26 it must then hold that

$$\llbracket P \rrbracket_N \sigma_t \simeq \llbracket P \rrbracket_N \wedge \llbracket P \rrbracket_N \sigma_t \Downarrow m \implies \llbracket P \rrbracket_N \Downarrow m$$

thus contradicting our conclusion that $\llbracket P \rrbracket_N \not\Downarrow m$. □

The above proof exploits the reflective capability of the ρ-calculus to create new, *free* names at runtime, which are therefore also observable and substitutable. Thus, a substitution can affect the reduct of a process, without affecting the process itself, if the reduction step creates a new name. Stated otherwise: in the ρ-calculus, the set of free names of a process *can* increase under reduction. This cannot be mimicked in the π-calculus, where names have no structure and cannot be composed at runtime. Any new *free* name appearing at runtime can therefore only come from the translation parameters, since it cannot come from the source term; but this would then violate the criterion of parameter independence, since we would then have to choose the parameters such that they correspond to the names that will be created at runtime.

This result does not directly depend on the higher-order characteristics of the ρ-calculus, and adding higher-order behaviour to the π-calculus would not suffice to enable it to encode the ρ-calculus. In [109], Sangiorgi gave an encoding of the Higher-Order π-calculus, HOπ, in the π-calculus. His encoding also satisfies our criteria from Definition 15, and we therefore also have the following result:

Corollary 8. *There is no encoding of the ρ-calculus into* HOπ *satisfying the criteria of Definition 15, when $\simeq$ satisfies the requirement in Definition 26.*

Indeed, if such an encoding existed, we could compose it with the encoding of HOπ into the π-calculus, to obtain an encoding of the ρ-calculus into the π-calculus, in contradiction of Theorem 4. This also indicates that the key feature of the ρ-calculus, which cannot be represented in the π-calculus, is not its higher-order characteristics per se, but rather its capability for reflection, which gives it higher-order characteristics as a by-product.

3.9 Related works

The issues of encodability and assessing the relative expressiveness of various process calculi has been considered by several authors. In particular, Gorla [48] proposed a framework for reasoning about encodability and separation w.r.t. a set of criteria that also served as inspiration for the criteria used in the present chapter. Towards the end of his paper, Gorla also discusses some of the difficulties involved in formulating a *general* framework for encodability in the presence of parameters, which particularly pertain to the question of which language the names belong to (the source or the target). In the present case, the answer is clearly the target language, which is further underscored by our restrictions on observability and compositionality; i.e. that the parameters should not be observable in the source term; and that, for each recursive call to the translation function, the parameters should be derivable from the initial set. Furthermore, we have added the criterion of parameter independence. We believe that such a criterion will generally be necessary for encodings that allow the set of parameters to 'evolve' or be updated in some structured way during the course of the

translation, which seems particularly likely when we are working with structured names or terms. More recently, van Glabbeek [46] has also proposed a definition of a valid encoding, which he derives from a notion of a semantic equivalence or preorder, rather than basing it on a list of commonly agreed-upon criteria (as we have done in the present chapter, following Gorla). However, this work also does not consider parametrised translations.

Also related is the work by Carbone and Maffeis [27] on expressivity of polyadic synchronisation. Their ${}^{e}\pi$-calculus substitutes names for names (as in the π-calculus), but allows n-ary vectors of names $x_1 \cdot \ldots \cdot x_n$ of arbitrary length $n \geq 0$ to appear in *subject* position of input/output prefixes, and subjects must then match on all n names to yield a reduction. Thus name vectors can be altered at runtime, but they cannot grow in length as in the ρ-calculus. However, we could conceive of a (purposefully ill-sorted) variant of ${}^{e}\pi$ that would allow entire vectors of names to be substituted for single names, thereby allowing new vectors of increasing length to be composed at runtime. We do not know if such a calculus could encode the ρ-calculus, but we suspect that it might, if equipped with an appropriate notion of name equivalence.

Another approach to using structured terms as names is given by Bengtson et al. [18; 19] and Parrow et al. [98] in their work on Ψ-calculi, which is based on the theory of nominal sets and datatypes by Gabbay and Pitts [43]. Ψ-calculi allow both subjects and objects to be terms from an arbitrary nominal datatype, and with substitution of terms for names. This enables runtime composition of terms, and, notably, the ρ-calculus *can* be instantiated as a (higher-order) Ψ-calculus, as the present author and others have shown in [64]. That paper also describes a generic type system for Ψ-calculi, which also can be instantiated to yield a type system for the ρ-calculus. Thus, it provides an alternative approach to creating a type system for the ρ-calculus, as we have also explored in the present chapter. However, as mentioned in section 3.7.4, it does not avoid the limitations and restrictions on name generation, which we also had to impose.

3.10 Conclusion

The original ρ-calculus paper [80] by Meredith and Radestock raises some interesting questions about the nature of names in process calculi. By including name generation in the language, it forces any process to give an explicit account of the source of any fresh names required during its execution, whilst this is entirely implicit in the π-calculus with the $(\boldsymbol{\nu} x)P$ operator. This adds a degree of realism to the ρ-calculus, which may be relevant from an implementation perspective, but also requires some extra care when we wish to reason about it formally. For example, Meredith and Radestock attempted to show that the π-calculus can be interpreted in the ρ-calculus, but their encoding did not properly account for the invariant that must hold for the names used as parameters in their encoding; i.e. that the parameters always refer

to the most recently replicated names, leading to two errors that invalidate their correctness result. The purpose of the present chapter has been to describe these errors and then give a new encoding of the π-calculus, for which we have shown correctness w.r.t. a set of criteria for encodability close to those proposed by Gorla [48]. The main difference is that we here use a parametrised translation, and we therefore had to take parameters into account in our criteria. This seems unavoidable when we are working with a calculus with structured names like the ρ-calculus, where all names are globally visible and can be generated at runtime.

Furthermore, we have created a simple type system for the ρ-calculus to ensure correct usage of names. This type system is in general rather limited, since it e.g. disallows the creation of new higher-order channels; yet it suffices to show that the encoding of the π-calculus is well-typed, because this encoding only requires finitely many higher-order channels to encode replication, and these can therefore be generated statically by the translation function. As such, it provides another perspective on the difficulties involved in reasoning about structured names with global visibility and runtime generation.

Our encoding works, modulo the criteria in Definition 15; yet it may not be an entirely satisfactory solution in at least one regard: the name server acts as a single, central source of fresh names. If we consider the scenarios one might wish to *model* in the π-calculus, having such a single central process might be acceptable for e.g. models of programs running on a single computer, or models of client-server systems with a star topology. However, for distributed systems with a different network topology, the translation would not yield an adequate representation. Thus, the encoding may preserve the semantics of a program, but not necessarily the intuitions underlying its structure.

We could instead conceive of a more elaborate encoding, where e.g. each replication also instantiates its own copy of a name server to service the replicated processes. This would be closer to the intention in the encoding by Meredith and Radestock; but as we have seen, one would then have to be careful to ensure that each replica of the name server will generate a distinct namespace to avoid the possibility of a name clash. This could be achieved by letting each replica first request fresh names for all its parameters, including the namespace root s which must then be composed or otherwise shifted into a new namespace. Yet this creates a scaffolding problem, where, in order to instantiate a new source of fresh names, one must first have a source of fresh names. It does not remove the need for an initial, 'top level' instance of the name server.

These considerations illustrate some of the difficulties involved in working with, and reasoning about, structured names with global visibility. None of these problems are present in the π-calculus, yet any implementation of a π-calculus program would need to include a solution to the problem of obtaining fresh names. In the words of Meredith and Radestock [80], the π-calculus does not provide a 'theory of names.'

Lastly, we have shown that the π-calculus cannot encode the ρ-calculus in a way

that satisfies the same encodability criteria, modulo some requirements on the notion of behavioural equivalence $\simeq$ used in Definition 26. The key to this separation result seems precisely to be the ability of the ρ-calculus to create new *free* names at runtime, which cannot be mimicked in the π-calculus. This ability is a consequence of *reflection* in the ρ-calculus, which also gives it higher-order characteristics as a by-product.

In process calculi, higher-order behavior is communication of processes. Yet, as the ρ-calculus illustrates, higher-order behaviour can alternatively be viewed as just a special case of reflection; i.e. the ability to compute with, modify and re-execute a piece of code. When process calculi model computation as communication, a process precisely *computes with* another process by communicating it, and higher-order behaviour is then simply reflection without modification of the communicated data. Thus, the separation result is also interesting in light of a remark by Sangiorgi regarding the encodability of HOπ into the π-calculus:

> *[...]* HOπ *is* representable *within the π-calculus. This proves that the first-order paradigm, being by far simpler, should be taken as* basic. *Such a conclusion takes away the interest in the opposite direction, namely the representability of the π-calculus within a language using purely communications of agents ... [110, p. 8].*

But as we have seen, this does not seem to hold in the more general case where higher-order characteristics derive from the capability of reflection. The ρ-calculus is indeed purely using communication of agents (processes), because, in the ρ-calculus, names and processes are the same thing.

3.11 Proofs

3.11.1 Proof of Lemma 6

Proof. By induction on the structure of P.

- Case $\mathbf{0}$: Then $\mathcal{S}(\mathbf{0}) = 2$, and

$$\begin{aligned}[\![\mathbf{0}]\!] &= \mathbf{0} \mid !N(x,z,v,s)^{\text{init}} \\ &\mapsto^2 !N(x,z,v,s)^{\text{ready}}\end{aligned}$$

  after which it blocks.

- The cases for $x(y).P$ and $\overline{x}\langle z\rangle$ are similar to the above, since both processes block.

- Case $P_1 \mid P_2$: Then

$$\begin{aligned}\mathcal{S}(P_1 \mid P_2) &= 2 + 4\mathcal{S}_\nu(P_1 \mid P_2) + \mathcal{S}_!(P_1 \mid P_2) \\ &= 2 + 4\mathcal{S}_\nu(P_1) + \mathcal{S}_!(P_1) + 4\mathcal{S}_\nu(P_2) + \mathcal{S}_!(P_2)\end{aligned}$$

  Say $4\mathcal{S}_\nu(P_1) + \mathcal{S}_!(P_1) = i$ and $4\mathcal{S}_\nu(P_2) + \mathcal{S}_!(P_2) = j$. Performing the translation, we obtain

$$[\![P_1 \mid P_2]\!] = [\![P_1]\!]_{+n,v} \mid [\![P_2]\!]_{n+,v} \mid !N(x,z,v,s)^{\text{init}}$$

  where the name server can perform 2 steps, and by hypothesis

$$\begin{aligned}[\![P_1]\!]_{+n,v} \mid !N(x,z,v,s)^{\text{ready}} &\mapsto^i T_1' \not\mapsto \\ [\![P_2]\!]_{n+,v} \mid !N(x,z,v,s)^{\text{ready}} &\mapsto^j T_2' \not\mapsto\end{aligned}$$

  which yields $2 + i + j$ steps in total.

- Case $(\boldsymbol{\nu} x)P$: Assume the statement holds for P, so $\mathcal{S}(P) = 2{+}4\mathcal{S}_\nu(P){+}\mathcal{S}_!(P) = k$. Adding the restriction and performing the translation and enabled reductions, we observe:

$$\begin{aligned}[\![(\boldsymbol{\nu} x)P]\!] &= v\langle n\rangle \mid n(x).[\![P]\!]_{n^2,v} \mid !N(x,z,v,s)^{\text{init}} \\ &\mapsto^2 v\langle n\rangle \mid n(x).[\![P]\!]_{n^2,v} \mid !N(x,z,v,s)^{\text{ready}} \\ &\mapsto^4 [\![P]\!]_{n^2,v}\{s/x\} \mid !N(x,z,v,s')^{\text{ready}}\end{aligned}$$

  which yields $2 + 4 + k$ steps. We omit showing that the number of steps cannot be increased by a substitution. In this case it should be clear, since s came from the name server, and hence cannot be used to perform an internal step, and the bound name x came from the π-calculus.

- Case $!x(y).P$: Perform the translation and internal reductions:

$$\begin{aligned}[\![!x(y).P]\!]_{n,v} &= D(n) \mid n\langle\!\langle x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big)\rangle\!\rangle \\ &\mapsto^1 x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big) \mid n\langle\!\langle x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big)\rangle\!\rangle\end{aligned}$$

  and with two steps from the name server. This matches $\mathcal{S}(!x(y).P) = 3$. □

3.11.2 Proof of Proposition 2

Proof. Note firstly that the names in $\mathcal{N}_{[n]}$ are only used internally: they are never *bound* in the translation, hence they cannot be affected by any substitution that occurs at runtime. Thus

$$\forall n' \in \mathcal{N}_{[n]} \,.\, \left(\begin{array}{lll} & (D(n'))\sigma_t & = D(n') \\ \wedge & (v\!<\!n'\!>)\sigma_t & = v\!<\!n'\!> \\ \wedge & (n'(x).P)\sigma_t & = n'(x).(P)\sigma_t \end{array}\right)$$

We can therefore ignore them in the proof.

Secondly, note that $[\![\sigma_s(x)]\!] = \sigma_t(x)$, since, by construction, the names in the π-calculus are assumed to be implemented as ρ-calculus names. Hence, the translation just directly maps each name to itself.

We then proceed by induction on the clauses of the translation function. Most of the cases are simple: the interesting ones are the cases for $(\boldsymbol{\nu} x)P$ and replication:

- Case $(\boldsymbol{\nu} x)P$:

$$\begin{aligned} & [\![((\boldsymbol{\nu} x)P)\sigma_s]\!]_{n,v} \\ = {} & [\![(\boldsymbol{\nu} x)P\sigma_s]\!]_{n,v} \\ = {} & v\!<\!n\!> \mid n(x).[\![P\sigma_s]\!]_{n^2,v} \end{aligned}$$

$$\begin{aligned} & [\![(\boldsymbol{\nu} x)P]\!]_{n,v}\sigma_t \\ = {} & (v\!<\!n\!> \mid n(x).[\![P]\!]_{n^2,v})\sigma_t \\ = {} & v\!<\!n\!> \mid n(x).([\![P]\!]_{n^2,v})\sigma_t \end{aligned}$$

  and by the induction hypothesis, $[\![P\sigma_s]\!]_{n^2,v} = [\![P]\!]_{n^2,v}\sigma_t$.

- Case $!x(y).P$:

$$\begin{aligned} & [\![(!x(y).P)\sigma_s]\!]_{n,v} \\ = {} & [\![!\sigma_s(x)(y).(P)\sigma_s)]\!]_{n,v} \\ = {} & D(n) \mid n\langle\!\langle [\![\sigma_s(x)]\!](y).\big(D(n) \mid [\![P\sigma_s]\!]_{n^2,v}\big)\rangle\!\rangle \end{aligned}$$

$$
\begin{aligned}
&[\![!x(y).P]\!]_{n,v}\sigma_t \\
&= (D(n) \mid n\langle\!|x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big)|\!\rangle)\sigma_t \\
&= D(n) \mid n\langle\!|\sigma_t(x)(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\sigma_t\big)|\!\rangle
\end{aligned}
$$

and by the induction hypothesis, $[\![P\sigma_s]\!]_{n^2,v} = [\![P]\!]_{n^2,v}\sigma_t$. And as argued above, $[\![\sigma_s(x)]\!] = \sigma_t(x)$.

In both cases we make use of the fact that the substitution cannot affect the names $n' \in \mathcal{N}_{[n]}$ created by the translation, nor the name v, and that names in the π-calculus are implemented as ρ-names. □

3.11.3 Proof of Proposition 3

Proof. By Lemma 6, we know there always exists a T such that $[\![P]\!] \mapsto^{\mathcal{S}(P)} T$ and $T \not\mapsto$, because the number of enabled internal steps has been exhausted, and this number is only finitely large. These steps must be performed to ensure that the name $\hat{x}$ is exposed, if it were inside the scope of some (other) restriction in the source term, or if it is the subject of an input-guard for replication.

For both directions, we let $\mathcal{N}$ on the ρ-calculus observation predicate denote the set of free names in the top-level process, to avoid confusion with the continuation P in the subterms.

For the forward direction we then proceed by induction on the observation predicate $\downarrow$. We wish to show that $P \downarrow \hat{x} \implies [\![P]\!] \mapsto^{\mathcal{S}(P)} T$ and $T \downarrow^{\mathrm{fn}(P)} \hat{x}$.

- The cases for $\mathbf{0}$, $x(y).P$ and $x\langle z\rangle$ are immediate. The greatest number of internal steps is here just 2, deriving from the name server going from the initial to the ready state.
- Case $P_1 \mid P_2 \downarrow \hat{x}$: If $P_1 \mid P_2 \downarrow \hat{x}$ then either $P_1 \downarrow \hat{x}$ or $P_2 \downarrow \hat{x}$ by [B-PAR$_\pi$]. Then by our induction hypothesis either

  $$[\![P_1]\!]_{+n,v} \mid !N(x,z,v,s) \mapsto^{\mathcal{S}(P_1)} T_1 \mid !N(x,z,v,s_1)$$

  or

  $$[\![P_2]\!]_{n+,v} \mid !N(x,z,v,s) \mapsto^{\mathcal{S}(P_2)} T_2 \mid !N(x,z,v,s_2)$$

  and $T_1 \mid T_2 \downarrow^{\mathcal{N}} \hat{x}$ by [B-PAR$_\rho$]. We can then take

  $$T = T_1 \mid T_2 \mid !N(x,z,v,s')$$

  where s' is derived from s by the sum of the increments used to derive s_1 and s_2 respectively, and then conclude $T \downarrow^{\mathcal{N}} \hat{x}$ by applying [B-PAR$_\rho$] twice.

- Case $!x(y).P \downarrow x$: Then this must have been concluded by [B-REP$_\pi$], and with the premise, in turn, concluded by [B-IN$_\pi$]. We have $\mathcal{S}(!x(y).P) = 3$ and performing the translation and reductions we get:

$$\begin{aligned}
&[\![!x(y).P]\!] \\
&= D(n) \mid n\langle\!\langle x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big)\rangle\!\rangle \mid !N(x,z,v,s)^{\text{init}} \\
&\mapsto^2 D(n) \mid n\langle\!\langle x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big)\rangle\!\rangle \mid !N(x,z,v,s)^{\text{ready}} \\
&\mapsto^1 x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big) \\
&\quad \mid D(n) \mid n\langle\!\langle x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big)\rangle\!\rangle \mid !N(x,z,v,s)^{\text{ready}}
\end{aligned}$$

  which has no internal steps enabled, and $x(y).\big(D(n) \mid [\![P]\!]_{n^2,v}\big) \downarrow^{\mathcal{N}} x$ by [B-IN$_\rho$], so we can conclude by (twice) applying [B-PAR$_\rho$].

- Case $(\boldsymbol{\nu} z)P$: Then $(\boldsymbol{\nu} z)P \downarrow \hat{x}$ must have been concluded by [B-RES$_\pi$] and $z \neq x$. We can also express this by α-converting z and choosing instead another name that is distinct from x and all the free names in P. We pick the name s; i.e. the next name that will be returned from the name server, and thus the premise becomes $P\{s/z\} \downarrow \hat{x}$. Our induction hypothesis is therefore that $[\![P\{s/z\}]\!] \mapsto^k T \wedge T \downarrow^{\mathcal{N}} \hat{x}$ with $k = \mathcal{S}(P\{s/z\}) = \mathcal{S}(P)$.

  We then perform the translation and reductions:

$$\begin{aligned}
&[\![(\boldsymbol{\nu} z)P]\!] \\
&= v\langle n\rangle \mid n(z).[\![P]\!]_{n^2,v} \mid !N(x,z,v,s)^{\text{init}} \\
&\mapsto^2 v\langle n\rangle \mid n(z).[\![P]\!]_{n^2,v} \mid !N(x,z,v,s)^{\text{ready}} \\
&\mapsto^4 [\![P]\!]_{n^2,v}\{s/z\} \mid !N(x,z,v,s')^{\text{ready}} \\
&= [\![P\{s/z\}]\!]_{n^2,v} \mid !N(x,z,v,s')^{\text{ready}} \qquad \text{by Proposition 2}
\end{aligned}$$

  By Proposition 1, the choice of parameters does not affect the observable names. Then by the induction hypothesis

$$[\![P\{s/z\}]\!]_{n^2,v} \mid !N(x,z,v,s')^{\text{ready}} \mapsto^{k-2} T \wedge T \downarrow^{\mathcal{N}} \hat{x}$$

  Note: we subtract two steps from k, since the name server is in the ready state, rather than the initial state.

For the other direction, we proceed by induction on the observation predicate $\downarrow^{\mathcal{N}}$ for the ρ-calculus and the clauses of the translation function. We wish to show that $[\![P]\!] \mapsto^{\mathcal{S}(P)} T \wedge T \downarrow^{\mathcal{N}} \hat{x} \implies P \downarrow \hat{x}$.

- If the source term P was $\mathbf{0}$, $x(y).P$ or $x\langle z\rangle$, the result is again immediate (vacuously; by [B-IN$_\rho$] and [B-INπ]; or by [B-OUT$_\rho$] and [B-OUT$_\pi$], respectively).

- If the source term was $P_1 \mid P_2$, then we obtain two terms T_1 and T_2 by the same steps as described above for the forward case. We know that $T_1 \mid T_2$ cannot interact, since only internal reductions are allowed, and an interaction would correspond to a committing step. Thus we know that $[\![P_1]\!]_{+n,v} \mid !N(x,z,v,s) \mapsto^{k_1} T_1 \mid !N(x,z,v,s_1)$ and $[\![P_2]\!]_{n+,v} \mid !N(x,z,v,s) \mapsto^{k_2} T_2 \mid !N(x,z,v,s_2)$, with $k_1 = \mathcal{S}(P_1)$ and $k_2 = \mathcal{S}(P_2)$ and $k_1 + k_2 - 2 = \mathcal{S}(P_1 \mid P_2)$.

  Now if $T_1 \mid T_2 \downarrow^{\mathcal{N}} \hat{x}$, then either $T_1 \downarrow^{\mathcal{N}} \hat{x}$ or $T_2 \downarrow^{\mathcal{N}} \hat{x}$ by [B-PAR$_\rho$]. Therefore, by the induction hypothesis, $P_1 \downarrow \hat{x}$ or $P_2 \downarrow \hat{x}$, so we conclude $P_1 \mid P_2 \downarrow \hat{x}$ by [B-PAR$_\pi$].

- If the source term was $(\boldsymbol{\nu} z)P$, then we know the target term is blocked by the input, requesting a name from the name server. This must first be reduced away, taking 6 steps.

  Assume then that

  $$[\![P]\!]_{n^2,v}\{s/z\} \mid !N(x,z,v,s')^{\text{ready}} \mapsto^{k} T \mid !N(x,z,v,s'')^{\text{ready}}$$

  and $T \downarrow^{\mathcal{N}} \hat{x}$ then holds. We can then move the substitution under the translation by Proposition 2 and obtain

  $$[\![P\{s/z\}]\!]_{n^2,v} \mid !N(x,z,v,s')^{\text{ready}} \mapsto^{k} T' \mid !N(x,z,v,s'')^{\text{ready}}$$

  and $T' \downarrow^{\mathcal{N}} \hat{x}$ then also holds, since $z \notin \mathcal{N}$ Then by the induction hypothesis $P\{s/z\} \downarrow \hat{x}$, and since we know that $s \neq x$, we can therefore also conclude $(\boldsymbol{\nu} s)P\{s/z\} \downarrow \hat{x}$ by [B-RES$_\pi$]. Therefore also $(\boldsymbol{\nu} z)P \downarrow \hat{x}$ by α-conversion.

This concludes the proof. □

3.11.4 Proof of Proposition 4

Proof. The first part of this statement is given by Lemma 7, so our task is here to show that the two reducts T_1 and T_2 actually correspond. We proceed by induction on the reduction rules. We shall use $\mathcal{N}$ on $\sim^{\mathcal{N}}$ to denote the set of free names in the top-level source term to avoid confusion with the free names of the subterm under consideration.

- Case [π-COM]: If this rule was used to conclude the committing step, then, from the conclusion, we have that $P = x(y).Q \mid x\langle z\rangle$ and $P \to Q\{z/y\} = P'$. We then have that

  $$\begin{aligned} &[\![x(y).Q \mid x\langle z\rangle]\!]_{n,v} \\ = &[\![x(y).Q]\!]_{+n,v} \mid [\![x\langle z\rangle]\!]_{n+,v} \end{aligned}$$

$$
\begin{aligned}
&= x(y).[\![Q]\!]_{+n,v} \mid x\langle z\rangle \\
&\to [\![Q]\!]_{+n,v}\{z/y\} && \text{by } [\rho\text{-com}] \\
&= [\![Q\{z/y\}]\!]_{+n,v} && \text{by Proposition 2} \\
&\sim^{\mathcal{N}} [\![Q\{z/y\}]\!]_{n,v} && \text{by Proposition 1}
\end{aligned}
$$

which is precisely the translation of the reduct P'. Note that we have here disregarded the name server, as it is not involved in this reduction. Thus, this reduction would actually be concluded by the rule $[\rho\text{-par}]$, and with $[\rho\text{-com}]$ used for the premise. The name server itself may at most be the source of two extra reductions, as argued in Lemma 6, and the total number of internal reductions will therefore be $\mathcal{S}(P) + \mathcal{S}(P') - 2$.

- Case $[\pi\text{-par}]$: By application of the induction hypothesis and Proposition 1 to change the parameter s' on the name server, which may have been incremented, if either of $P_1 \mid P_2$ contained any $(\boldsymbol{\nu} x)$-requests.

- Case $[\pi\text{-res}]$: If this rule was used to conclude the reduction, then, from the conclusion, we have that $P = (\boldsymbol{\nu} x)Q \to (\boldsymbol{\nu} x)Q' = P'$, and, from the premise, that $Q \to Q'$. We then have that

$$
\begin{aligned}
&[\![(\boldsymbol{\nu} x)Q]\!] \\
&= v\langle n\rangle \mid n(x).[\![Q]\!]_{n^2,v} \mid !N(x,z,v,s)^{\text{init}} \\
&\mapsto^6 [\![Q]\!]_{n^2,v}\{s/x\} \mid !N(x,z,v,s')^{\text{ready}} \\
&= [\![Q\{s/x\}]\!]_{n^2,v} \mid !N(x,z,v,s')^{\text{ready}} && \text{by Proposition 2}
\end{aligned}
$$

where the number 6 is the number of steps required to service the request for the new name. Now $Q\{s/x\} \to Q'\{s/x\}$, so by the induction hypothesis

$$[\![Q\{s/x\}]\!] \mid !N(x,z,v,s')^{\text{init}} \mapsto^k [\![Q'\{s/x\}]\!]_{n^2,v} \mid !N(x,z,v,s'')^{\text{ready}}$$

for some number of steps k and names s', s''.

Consider then the translation of the reduct $P' = (\boldsymbol{\nu} x)Q'$: We have that

$$
\begin{aligned}
&[\![(\boldsymbol{\nu} x)Q']\!] \\
&= v\langle n\rangle \mid n(x).[\![Q']\!]_{n^2,v} \mid !N(x,z,v,s)^{\text{init}} \\
&\mapsto^{6+k} [\![Q']\!]_{n^2,v}\{s/x\} \mid !N(x,z,v,s')^{\text{ready}} \\
&= [\![Q\{s/x\}]\!]_{n^2,v} \mid !N(x,z,v,s')^{\text{ready}} && \text{by Proposition 2}
\end{aligned}
$$

which is the same result as we obtained above, up to a change of the name parameters s', s'' by Proposition 1.

- Case [π-STRUCT]: By application of the induction hypothesis, and then with a case analysis of the possible reduction-enabling rewrites under structural congruence. Here, the interesting cases are the rules for scope intrusion/extrusion, unfolding replication, and removing unused restrictions, since these rewrites correspond to reductions in the ρ-calculus. We shall give some examples:

  - Case $P_1 = !x(y).Q \mid x\langle z\rangle$: In this case, the rewrite unfolds a replication that then communicates with some output process. Note that since we only allow input-guarded replication, this rewrite can precisely only give rise to a reduction, if there exists another process in composition, using the same name for output. We then have that

$$\begin{aligned} & !x(y).Q \mid x\langle z\rangle \\ \equiv\ & !x(y).Q \mid x(y).Q \mid x\langle z\rangle \\ \to\ & !x(y).Q \mid P\{z/y\} = P_2 \end{aligned}$$

    Performing the translation and reductions yields:

$$\begin{aligned} & [\![!x(y).Q \mid x\langle z\rangle]\!] \\ =\ & D(+n) \mid +n\langle\!|x(y).\big(D(+n) \mid [\![Q]\!]_{+n^2,\upsilon}\big)|\!\rangle \mid x\langle z\rangle \\ \mapsto\ & D(+n) \mid +n\langle\!|x(y).\big(D(+n) \mid [\![Q]\!]_{+n^2,\upsilon}\big)|\!\rangle \\ & \quad \mid x(y).\big(D(+n) \mid [\![Q]\!]_{+n^2,\upsilon}\big) \mid x\langle z\rangle \\ \to\ & D(+n) \mid +n\langle\!|x(y).\big(D(+n) \mid [\![Q]\!]_{+n^2,\upsilon}\big)|\!\rangle \mid [\![Q]\!]_{+n^2,\upsilon}\{z/y\} \\ =\ & [\![!x(y).Q]\!]_{+n,\upsilon} \mid [\![P]\!]_{+n^2,\upsilon}\{z/y\} \\ =\ & [\![!x(y).Q]\!]_{+n,\upsilon} \mid [\![Q\{z/y\}]\!]_{+n^2,\upsilon} \end{aligned}$$

    with the last step by Proposition 2, which gives us the form of the translation of our reduct P_2, up to a change of parameters by Proposition 1. The name server has been omitted, since it is not involved in the committing step, but it contributes two internal steps as usual. We then have $\mathcal{S}(!x(y).P \mid x\langle z\rangle) = 3$ as expected.

  - Case $P_1 = (\boldsymbol{\nu} x)Q_1 \mid Q_2 \equiv (\boldsymbol{\nu} x)(Q_1 \mid Q_2)$ where $x\#Q_2$: In this case, the rewrite extrudes the scope of $(\boldsymbol{\nu} x)$ to cover Q_2, which may require α-conversion, and this rewrite may then enable a communication between Q_1 and Q_2. Thus $(\boldsymbol{\nu} x)Q_1 \mid Q_2 \equiv (\boldsymbol{\nu} x)(Q_1 \mid Q_2) \to (\boldsymbol{\nu} x)(Q_1' \mid Q_2') = P_2$. We can then use α-conversion for the premise by choosing some fresh name s and thus obtain $(Q_1 \mid Q_2)\{s/x\} \to (Q_1' \mid Q_2')\{s/x\}$. Here we use the same trick as for the rule [π-RES] above, and pick the next name delivered by the name server as our fresh name s.

This rewrite does not correspond to anything in the ρ-calculus, as all names are globally visible. Performing the translation and reductions then yields

$$\begin{aligned} & v\langle +n\rangle \mid +n(x).[\![Q_1]\!]_{+n^2,v} \mid [\![Q_2]\!]_{n+,v} \mid !N(x,z,v,s) \\ \mapsto^6 & [\![Q_1]\!]_{+n^2,v}\{s/x\} \mid [\![Q_2]\!]_{n+,v} \mid !N(x,z,v,s')^{\text{ready}} \\ = & ([\![Q_1]\!]_{+n^2,v} \mid [\![Q_2]\!]_{n+,v})\{s/x\} \mid !N(x,z,v,s')^{\text{ready}} \end{aligned}$$

with the last step by $x\#Q_2$. We can now use Proposition 1 to change the parameters on the translations to $+n$ and $n+$:

$$\begin{aligned} & \sim^{\mathcal{N}} ([\![Q_1]\!]_{+n,v} \mid [\![Q_2]\!]_{n+,v})\{s/x\} \mid !N(x,z,v,s') && \text{by Prop. 1} \\ & = ([\![Q_1 \mid Q_2]\!]_{n,v})\{s/x\} \mid !N(x,z,v,s') && \text{by trans.} \\ & = [\![(Q_1 \mid Q_2)\{s/x\}]\!]_{n,v} \mid !N(x,z,v,s') && \text{by Prop. 2} \\ & \Rightarrow^k [\![(Q_1' \mid Q_2')\{s/x\}]\!]_{n,v} \mid !N(x,z,v,s') && \text{by i.h.} \\ & = [\![Q_1'\{s/x\} \mid Q_2'\{s/x\}]\!]_{n,v} \mid !N(x,z,v,s') \end{aligned}$$

Consider now the reduct P_2: Performing the translation and reduction yields:

$$\begin{aligned} & [\![(\boldsymbol{\nu} x)(Q_1' \mid Q_2')]\!]_{n,v} \mid !N(x,z,v,s)^{\text{init}} \\ = & \, v\langle n\rangle \mid n(x).[\![Q_1' \mid Q_2']\!]_{n^2,v} \mid !N(x,z,v,s)^{\text{init}} \\ \mapsto^6 & \, ([\![Q_1' \mid Q_2']\!]_{n^2,v})\{s/x\} \mid !N(x,z,v,s')^{\text{ready}} \\ = & \, [\![(Q_1' \mid Q_2')\{s/x\}]\!]_{n^2,v} \mid \mid !N(x,z,v,s')^{\text{ready}} && \text{by Prop. 2} \end{aligned}$$

which is of the same form as the reduct we obtained above, up to a change of parameters by Proposition 1.

– The remaining cases proceed in a similar fashion, by translating the initial source term and the reduct, and performing the reductions in the ρ-calculus, using Proposition 1 to alter the parameters so that they match, and Proposition 2 to move a substitution under the translation. The greatest number of steps required is then determined by the structure of the source term as in Lemma 6.

This concludes the proof. □

3.11.5 Proof of Proposition 5

Proof. We proceed by induction on the structure of translated terms. The statement holds trivially for terms of the form $[\![\mathbf{0}]\!]$, $[\![x(y).P]\!]$ and $[\![x\langle z\rangle]\!]$, since they cannot

perform any committing steps, but only two non-committing steps deriving from the name server. Likewise it trivially holds for $[\![!x(y).P]\!]$, which can perform one non-committing step because of the replication, and two deriving from the name server, after which it blocks. Thus $[\![(\boldsymbol{\nu} x)P]\!]$ and $[\![P_1 \mid P_2]\!]$ are the only interesting cases. Here we shall only consider the case for $(\boldsymbol{\nu} x)P$, since it also covers the case for when $P = P_1 \mid P_2$, but is slightly more complicated, due to the presence of the restriction.

Consider the term $P = [\![(\boldsymbol{\nu} x)Q]\!]$: It must perform 6 non-committing steps, deriving from the restriction and the name server. This yields the term

$$\begin{aligned} & [\![Q]\!]_{n^2,v}\{s/x\} \mid !N(x,z,v,s')^{\text{ready}} \\ = \; & [\![Q\{s/x\}]\!]_{n^2,v} \mid !N(x,z,v,s')^{\text{ready}} \qquad \text{by Proposition 2} \end{aligned}$$

Now assume

$$[\![Q\{s/x\}]\!]_{n^2,v} \mid !N(x,z,v,s')^{\text{ready}} \mapsto^{k_1} \to \mapsto^{k_2} T_1' \mid !N(x,z,v,s'')$$

for some numbers of non-committing steps k_1, k_2, where $k_1 \leq \mathcal{S}(Q)$ as argued above, and where $T_1' \not\mapsto$. Hence $k_1 + k_2 = \mathcal{S}(Q) + \mathcal{S}(Q') - 2$.

This term corresponds to $[\![Q\{s/x\}]\!]$, up to a change of parameters by Proposition 1. By construction, $s\#Q$, so we have by the induction hypothesis that $Q\{s/x\} \to Q'\{s/x\}$. By α-conversion and rule [π-RES], we can therefore also conclude $(\boldsymbol{\nu} x)Q \to (\boldsymbol{\nu} x)Q'$. Performing the translation and all non-committing steps then gives us

$$[\![(\boldsymbol{\nu} x)Q']\!] \mapsto^6 [\![Q']\!]_{n^2,v}\{s/x\} \mid !N(x,z,v,s') \mapsto^{\mathcal{S}(Q')} T_2'$$

We must now argue that $T_1' \sim^{\mathcal{N}} T_2'$, where $\mathcal{N}$ is the set of free names in the top-level source term. We proceed by case analysis of the source of the committing step. From the syntax of the ρ-calculus, we see that every process is of the form

$$P_1 \mid \ldots \mid P_n$$

and from the semantics that every reduction is concluded with [ρ-COM] as axiom; i.e. the committing step must be a communication with a redex of the form

$$x_1(y).P_1 \mid x_2\langle\!|P_2|\!\rangle$$

with $x_1 \equiv_{\mathcal{N}} x_2$. This redex is built by applying some combination of [ρ-STRUCT] and [ρ-PAR]. As this is a committing step, we know x_1, x_2 came from the π-calculus or the name server, so in fact $x_1 = x_2$, and we shall therefore just write x_1.

By the translation, there are only two ways in which such a redex could have been built from the subterms of $[\![Q]\!]$: Q must have contained an output $x_1\langle u\rangle$ for some name u, and then either an input $x_1(y).R$ or an input-guarded replication $!x_1(y).R$.

- If it was a plain input, then the translation must be of the form

$$[\![Q]\!]_{n^2,v} = [\![Q_1]\!]_{n_1,v} \mid \ldots \mid x_1\langle u\rangle \mid \ldots \mid x_1(y).[\![R]\!]_{n_i,v} \mid \ldots \mid [\![Q_n]\!]_{n_k,v}$$

  up to a reordering of the terms, and $n_1, \ldots, n_k$ derived from n^2. Thus we conclude $x_1(y).[\![R]\!]_{n_i,v} \mid x_1\langle u\rangle \to [\![R]\!]_{n_i,v}\{u/y\} = [\![R\{u/y\}]\!]_{n_i,v}$ by Proposition 2, after which $[\![R\{u/y\}]\!]_{n_i,v}$ may perform at most k_2 non-committing steps. This gives us the shape of our T_1':

$$T_1' = \left([\![Q_1]\!]_{n_1,v} \mid \ldots \mid [\![R'\{u/y\}]\!]_{n_i,v} \mid \ldots \mid [\![Q_k]\!]_{n_k,v}\right)\{s/x\}$$

  Likewise, we can conclude $x_1\langle u\rangle \mid x_1(y).R \to R\{u/y\}$ in the π-calculus. Adding back the restriction gives us the form of Q':

$$Q' = (\nu x)(Q_1 \mid \ldots \mid R\{u/y\} \mid Q_k)$$

  and by performing the translation and the non-committing reductions we obtain:

$$\begin{aligned} & [\![(\nu x)(Q_1 \mid \ldots \mid R\{u/y\} \mid Q_k)]\!] \\ = {} & v\langle n\rangle \mid n(x).\left([\![Q_1]\!]_{n_1,v} \mid \ldots \mid [\![R\{u/y\}]\!]_{n_i,v} \mid [\![Q_k]\!]_{n_k,v}\right) \\ \mapsto^4 {} & \left([\![Q_1]\!]_{n_1,v} \mid \ldots \mid [\![R\{u/y\}]\!]_{n_i,v} \mid [\![Q_k]\!]_{n_k,v}\right)\{s/x\} \end{aligned}$$

  Then $[\![R\{u/y\}]\!]_{n_i,v}$ may perform at most k_2 non-committing steps to become $[\![R'\{u/y\}]\!]_{n_i,v}$, which then gives us our T_2'. Thus we conclude that T_1 and T_2 are not only equivalent, but actually equal, up to a reordering of terms and a change of parameters by Proposition 1.

- If it was a *restricted* communication, with $x_1 = x$, i.e. the subject was the restricted name from the outer (νx), then we proceed exactly as above, but apply the substitution $\{s/x\}$ in the ρ-calculus first, such that the committing step is concluded with s as subject. Then in the π-calculus, we use the same trick as before, by using α-conversion and picking s as the fresh name.

- If it was an input-guarded replication, then the case is almost exactly similar to above, since an input-guarded replication reduces to a plain input plus a blocked lift after one non-committing step. In this case, we know that the reduct $[\![R\{u/y\}]\!]_{n_i,v}$ will be of the form

$$D(n_i) \mid n_i\left\langle\!\left| x_1(y).\left(D(n_i) \mid [\![R]\!]_{n_i^2,v}\right)\right|\!\right\rangle \mid [\![R\{u/y\}]\!]_{n_i,v}$$

  which will perform one non-committing step to ready the replication for another input, plus the i_2 steps performed by $[\![R\{u/y\}]\!]_{n_i,v}$ to yield $[\![R'\{u/y\}]\!]_{n_i,v}$.

In the π-calculus, we likewise get $!x_1(y).R \mid R\{u/y\}$, and when we translate this term and perform the non-committing steps, we again obtain the same result, up to a reordering of terms and a change of parameters by Proposition 1.

We omit the case analysis for $P_1 \mid P_2$, since it proceeds exactly as above, with the input and output terms found either in P_1 or P_2, or one in each. □

4 A Generic Type System for Higher-Order Ψ-calculi[1]

4.1 Introduction

Process calculi are formalisms for modelling and reasoning about concurrent and distributed computations; a prominent example being the π-calculus of Milner et al. [83; 113], which models computation as communication between processes, by passing messages on named channels.

Since its inception, a multitude of variants of the π-calculus have appeared; e.g. Dπ [53], the calculus of explicit fusions [44], the spi-calculus with correspondence assertions [1] and the ${}^{e}\pi$-calculus [27]. These calculi are all *first-order*, in the sense that only atomic channel names can be passed around, not processes themselves. Moreover, the calculi have semantics that share characteristics such as the treatment of scope extension and name passing but also differ in specific aspects that pertain to the calculus under consideration.

Bengtson et al. [18; 19] created Ψ*-calculi* as a generalisation of the first-order variants and extensions of the π-calculus, allowing them to be expressed as *instances* of the Ψ-calculus framework through appropriate settings of a small number of parameters. One of these parameters is the set of messages or *terms*, which can be passed around and used as channels, and the choice of this set is subject only to some very mild constraints. Thus, even in the 'first-order' Ψ-calculus, the set of terms may be chosen such that it contains the set of processes, thereby allowing processes to be passed around as messages.

However, the Ψ-calculus lacks the one feature that is common to process calculi with higher-order communication, namely, the ability to execute a received message as a process. This feature was added as an extension to the framework by Parrow et al. [98], who thus created the *Higher-Order* Ψ-calculus, HOΨ. This framework

[1]The material presented in this chapter is joint work with Hans Hüttel, Bjarke Bredow Bojesen and Alex Rønning Bendixen and originates from ideas developed during the writing of our Master's thesis [16]. The material was first published in the proceedings of EXPRESS/SOS in 2022 [15]. We were invited to submit a journal version of the paper, which was published in *Information & Computation* in 2024 [64], which includes a full redoing of all major proofs, as well as a rewrite of much of the material. The present chapter is identical to the journal version, except for some minor, typographical adjustments.

can be parametrised to yield instances corresponding to higher-order calculi such as HOπ and CHOCS, as well as every calculus that can be expressed in the first-order Ψ-calculus. One advantage of this generalised framework is the ability to reason about these many extensions and variants of the π-calculus collectively. For example, in [19; 98], the authors define a notion of strong bisimilarity for the Ψ-calculus, which then is automatically inherited by every instance of the calculus.

An important approach for reasoning about processes is that of *type systems*. For process calculi, this approach originates with Milner [83] whose first type system deals with the notion of correct usage of channels in the π-calculus: In a well-typed process, only names of the correct type can be communicated. Pierce and Sangiorgi [100] later described a type system that uses subtyping and capability tags to control the use of names as input or output channels. Many of the aforementioned first-order extensions of the π-calculus have also been given type systems that capture properties such as secrecy, authenticity and safe migration.

In [61], Hüttel noted that these type systems, despite arising in different settings, share certain characteristics: The type judgments for processes P are all of the form $\Gamma \vdash P$ where Γ is a type environment recording the types of the free names in P, so processes are only classified as being either well-typed or not. On the other hand, terms M are given a type T, so type judgements for terms are of the form $\Gamma \vdash M : T$. Based on these shared characteristics, Hüttel then created a generic type system for the first-order Ψ-calculus framework, that generalises several of the type systems for the π-calculus and its variants. This generic type system can similarly be instantiated through parameter settings to yield both well-known and new type systems for the calculi that are representable as first-order Ψ-calculi. An important advantage of this approach is that a general result of type system soundness can be formulated, which is then inherited by all instances of the type system. The soundness result is one of subject reduction along with a generalized version of the channel usage property from Milner's original system.

There has been other work on generic type systems, notably work by König [70], Caires [23] and Igarashi and Kobayashi [65]. As these contributions demonstrate, the notion of genericity can understood in different ways. The type system by Caires [23] is formulated only for the first-order monadic π-calculus, but is parametrised with a subtyping relation, which allows different typing disciplines to be expressed. These include a simple, sorting-like system for correct channel, input/output-capabilities and certain behavioural properties. Seen in this light, the type system by Caires generalises more typing disciplines, but it is only formulated for a specific calculus, whereas other generic type systems can only express a more narrow collection of properties directly, but can be instantiated for a broad range of calculi.

However, all generic type systems mentioned in the above are formulated for variants of the first-order π-calculus. This means that they cannot be instantiated to yield type system for *higher-order* calculi, such as HOπ or CHOCS. Both of these higher-order calculi can be encoded into the first-order π-calculus, as shown by

Sangiorgi in [109], and may therefore also be represented in just the first-order Ψ-calculus. Not surprisingly, there is therefore little work on type systems for higher-order calculi, since these encodings allow us to disregard the higher-order behaviour and instead just type the first-order translations.

One exception is the type system for termination in variants of HOπ, due to Demangeon et al. [36]. As the authors argue, it may not always be desirable (or even possible) to type a higher-order language through a first-order representation, if the language contains features that are difficult (or impossible) to encode. Another example of this is the Reflective Higher-Order (RHO or ρ) calculus of Meredith and Radestock [80], which is a name-passing process calculus in which names are generated by "quoting" process expressions and higher-order communication is obtained by passing around such names which can then be "unquoted" (or "dropped"). The ρ-calculus cannot be uniformly encoded in the π-calculus, as shown in [76] (see Chapter 3) so we here have an example of a language that cannot easily be represented in the first-order paradigm. This makes it difficult (even impossible) to adapt any of the existing first-order type systems to this language.

In the present chapter, we create a new, generic type system for higher-order Ψ-calculi, which extends the generic type system of Hüttel [61] for the first-order Ψ-calculus, and we show both how existing type systems for higher-order calculi can be expressed as instances of this generic type system, and how new type systems can be obtained. Like its predecessor, our generic type system satisfies a general subject reduction property that is inherited by all instances.

Our generic type system is able to handle subtyping at the level of terms, as the type rules for terms are specific for each instance of it. It is therefore simple to incorporate a subsumption rule. Other forms of polymorphism can be handled by means of the channel compatibility predicate. On the other hand, the type rule for parallel composition reveals that we cannot handle linear type systems since the premise says that the type environment to be used for typing each parallel component is the same; this will also be clear by our requirement that every instance must satisfy the notions of weakening and strengthening.

We use our generic type system to formulate simple type systems for HOπ, and show that the type system for termination by Demangeon et al. can also be captured as an instance of our type system. Moreover, we present new type systems. First, we demonstrate that our generic type system can be instantiated to yield a type system for the ρ-calculus, and this establishes that our type system is richer than the first-order type systems. Finally we use the generic type system to present a new type system for non-interference for mobile code.

The chapter is structured as follows: In Section 4.2 we introduce the syntax and semantics of the higher-order Ψ-calculus. Then, in Section 4.3, we present the generic type system, and in Section 4.4 we establish important properties of the type system. In particular, we prove a subject reduction property (Theorem 5) for the type system and introduce a general notion of safety. Finally, in Section 4.5 we give some examples

of instantiations to show how a number of existing type systems for higher-order calculi can be seen as instances of the generic type system; and also how the generic type system can be instantiated to yield new type systems.

The present chapter is an extension of [15]. Compared to the previous version, the review of the HOΨ-calculus and the presentation of the generic type system and the instance assumptions have been substantially rewritten and expanded to provide a more thorough and detailed presentation. We also provide full proofs for the subject reduction property (Theorem 5) and most of the associated lemmas (Lemmas 16–23), and for the property of operational correspondence for the ρ-calculus instance (Theorem 7). Lastly, the type system instance for non-interference presented in Section 4.5.4 is also new.

4.2 The Higher-Order Ψ-calculus

The first-order Ψ-calculus [18; 19] generalises the common characteristics of variants of the π-calculus that allow for transmission of structured message terms M, including, in principle, also processes. The Higher-Order Ψ-calculus then extends the original Ψ-calculus with an invocation construct, **run** M, which allows a received message to be executed as a running process. In this section we first review the syntax of HOΨ as given in [98], and then proceed to give a reduction semantics for the calculus.

4.2.1 Parameters

The Higher-Order Ψ-calculus is a general framework, which is intended to allow many different calculi to be obtained as instances, by setting a small number of parameters in the form of definitions of three (not necessarily disjoint) sets of *terms, conditions* and *assertions*, and four operations on these sets. Elements of these sets appear in the syntax of the Ψ-calculus, and thus different calculi can be obtained as instances by making different choices for the definitions of these sets.

To allow the framework to be as general and flexible as possible, the authors of [18; 98] identify only a few restrictions that must be imposed on the sets of terms, conditions and assertions: they must be *nominal datatypes*. Informally, a *nominal set*, in the sense of Gabbay and Pitts [43], is a set whose members can be affected by names being bound or swapped. Firstly, assume a countably infinite set of names $\mathcal{N}$, ranged over by $a, \ldots, z$. If a, b are names and X is an element of a nominal set, then the *transposition* of a and b on X, written $(a, b)\cdot X$, swaps all occurrences of a for b in X and vice versa. A function f on a nominal set is *equivariant*, if it is unaffected by name swapping; i.e. if $(a, b)\cdot f(X) = f((a, b)\cdot X)$, and a *nominal datatype* is a nominal set together with a set of equivariant functions on it. This requirement is very mild and allows e.g. non-well-founded sets to be used in an instantiation; for example, the set of processes can itself be included in the sets of terms, conditions or assertions.

Two other, important notions are those of support and freshness: if X is an element of a nominal set, a name a is said to *occur* in X, if it can be affected by transposition, and the *support* of X, written $\mathrm{n}(X)$, is the set of names that occur in X. Conversely, a name a is *fresh for* X, written $a\#X$, if $a \notin \mathrm{n}(X)$. This notion extends to sets of names A, such that $A\#X$ if it is the case that $\forall a \in A.a \notin \mathrm{n}(X)$, and this is pointwise extended to lists of elements $X_1, \ldots, X_n$, so we write $A\#X_1, \ldots, X_n$ for $A\#X_1 \wedge \ldots \wedge A\#X_n$.

As mentioned above, any Ψ-calculus instance requires a specification of three nominal datatypes: the terms, conditions and assertions. Unlike in the original presentations of Ψ-calculi [18; 19; 98], we shall be using a *typed* syntax, with types appearing in the name binders, so we therefore add a fourth parameter to be specified in an instantiation, namely the nominal datatype of *types*. This is further described in section 4.3.1. The sets to be specified are thus:

$$\begin{array}{rl} M, N \in \mathbb{T} & \text{Terms} \\ \varphi \in \mathbb{C} & \text{Conditions} \\ \Psi \in \mathbb{A} & \text{Assertions} \\ T \in \mathcal{T} & \text{Types} \end{array}$$

Terms M, N are used as subjects and objects in communication; they could be e.g. single names, as in the monadic π-calculus, vectors of names as in ${}^e\pi$ and the polyadic π-calculus; or elements of a composite datatype. Conditions φ are used in conditional expressions; e.g. matching and mismatch checks. Lastly, assertions Ψ can appear in the syntax and become enabled during the execution of a process, where they can affect the evaluation of conditions.

Each of the nominal datatypes must include the definition of an equivariant *substitution function* σ, written $(\cdot)[\tilde{a} := \tilde{M}]$ for the non-trivial part, which is the substitution of tuples of terms $\tilde{M}$ for tuples of names $\tilde{a}$ of equal arity. It must be defined such that it satisfies the following *substitution laws:*

- If $\tilde{a} \subseteq \mathrm{n}(X)$ and $b \in \mathrm{n}(\tilde{Y})$ then $b \in \mathrm{n}(X[\tilde{a} := \tilde{Y}])$
- If $\tilde{u}\#X, \tilde{v}$ then $X[\tilde{v} := \tilde{Y}] = ((\tilde{u}, \tilde{v}) \cdot X)[\tilde{u} := \tilde{Y}]$

These requirements are quite general and should be satisfied by any ordinary definition of substitution: The first law states that names cannot be lost in substitution, i.e. the names present in $\tilde{Y}$ must also be present when the substitution has been performed. The second law states that substitution cannot be affected by transposition.

For the purpose of defining the type system, we need to impose two further restrictions on the definition of the nominal sets, which are not present in the original formulation of Ψ-calculi: We require that the nominal sets of terms, conditions, assertions and types can be described as term algebras; i.e. that they are generated by a signature of constructors. This follows the presentation of abstract syntax given in [52].

Since we allow term constructors with binders, we define capture-avoiding substitutions as follows, also following [52]. For any n-ary term constructor f with arity $(\widetilde{s_1}.s_1, \ldots, \widetilde{s_n}.s_n)$ and substitution $\sigma = [\tilde{x} \mapsto \widetilde{M}]$ we require that σ satisfies the following conditions.

1. $x\sigma = M$ if $\sigma(x) = M$ and $y\sigma = y$ if $y \notin \mathrm{dom}(\sigma)$
2. $(f(\widetilde{x_1}.M_1, \ldots, \widetilde{x_n}.M_n))\sigma = f(\widetilde{x_1}.M_1', \ldots, \widetilde{x_n}.M_n')$ where for each $1 \leq i \leq n$ we require that $\widetilde{x_i} \# M_i$ and we have $M_i' = M_i\sigma$ if $\tilde{x} \# \widetilde{x_i}$ and $M_i' = M_i$ otherwise.

In other words, term substitution distributes over function symbols. These requirements are is also imposed in the type system proposed in [61], and they will ensure that a standard substitution lemma for type judgments will hold in our generic type system introduced later.

The Ψ-calculus allows arbitrary terms to be used as channels. Any Ψ-calculus instance therefore requires a definition of two equivariant operators, *channel equivalence* $\dot{\leftrightarrow}$ and *assertion composition* $\otimes$, a *unit element* $\mathbf{1}$ of assertions, and an *entailment relation* $\Vdash$, defined on the respective nominal datatypes and with the following signatures:

$$\begin{array}{rl} \dot{\leftrightarrow} : \mathbb{T} \times \mathbb{T} \to \mathbb{C} & \text{channel equivalence} \\ \otimes : \mathbb{A} \times \mathbb{A} \to \mathbb{A} & \text{assertion composition} \\ \mathbf{1} \in \mathbb{A} & \text{assertion unit} \\ \Vdash \subseteq \mathbb{A} \times \mathbb{C} & \text{entailment relation} \end{array}$$

We write the entailment relation as $\Psi \Vdash \varphi$ to denote that the condition φ holds, given the assertions Ψ. Our notation differs from e.g. that of [18], since we use the $\vdash$ symbol in the type judgements introduced later. Note that comparison by channel equivalence $M_1 \dot{\leftrightarrow} M_2$ is itself a condition, which may or may not be entailed by some assertions Ψ, according to the definition of the entailment relation.

In HOΨ, higher-order communication is handled by declaring *handles* for processes, written $M \Leftarrow P$, which denotes that the term M is a handle for the process P. We assume the existence of conditions of the form $M \Leftarrow P$ in the definition of the set of conditions $\mathbb{C}$, such that the entailment $\Psi \Vdash M \Leftarrow P$ is always well-defined. Note that the set of processes may itself be included in the set of terms, thus allowing conditions of the form $P \Leftarrow P$ whereby a process becomes a handle for itself.

The authors of [98] impose two restrictions for the parameter settings to be *valid:* Channel equivalence must be symmetric and transitive, and $\otimes$ must satisfy the commutative monoidal laws, with $\mathbf{1}$ as the unit element.

Finally, for the purpose of defining the type system, we need to impose a constraint on the entailment of some conditions, specifically channel equivalence and handles. Both must have a weakening property:

- If $\Psi \Vdash M_1 \dot{\leftrightarrow} M_2$ then $\Psi \otimes \Psi' \Vdash M_1 \dot{\leftrightarrow} M_2$.

- If $\Psi \Vdash M \Leftarrow P$ then $\Psi \otimes \Psi' \Vdash M \Leftarrow P$.

These requirements ensure that an extension of the assertions Ψ cannot invalidate channel equivalence, nor delete a handle for a process P. This could of course be generalised by requiring the weakening property to hold for all conditions, but since this requirement is not present in the original presentation of Ψ-calculi [18; 19; 98], we prefer this less constraining formulation.

In summary, we say that a Ψ-calculus instance is *valid*, if its parameter settings satisfy the following requirements:

Definition 27 (Valid instance)**.** A Ψ-calculus instance is *valid*, if the parameter settings satisfy the following:

1. Substitution for each nominal datatype satisfies the substitution laws:

$$\begin{aligned} \tilde{a} \subseteq \mathrm{n}(X) \wedge b \in \mathrm{n}(\widetilde{Y}) &\implies b \in \mathrm{n}\big(X[\tilde{a} := \widetilde{Y}]\big) \\ \tilde{u}\#X, \tilde{v} &\implies X[\tilde{v} := \widetilde{Y}] = \big((\tilde{u}, \tilde{v})\cdot X\big)[\tilde{u} := \widetilde{Y}] \end{aligned}$$

2. $\mathbb{T}$, $\mathbb{C}$, $\mathbb{A}$ and $\mathcal{T}$ are generated by a signature of constructors, and term substitutions σ distribute over function symbols f:

$$f(M_1, \ldots, M_n)\sigma = f(M_1\sigma, \ldots, M_n\sigma)$$

3. Handles are included in the set of conditions: $M \Leftarrow P \in \mathbb{C}$.

4. Channel equivalence $\leftrightarrow$ is symmetric and transitive.

5. $(\mathbb{A}, \otimes, \mathbf{1})$ is a commutative monoid w.r.t. an equivalence on assertions, written $\Psi_1 \simeq \Psi_2$. This equivalence is defined such that $\Psi_1 \simeq \Psi_2$ if for all φ it holds that that $\Psi_1 \Vdash \varphi \iff \Psi_2 \Vdash \varphi$.

6. Entailment satisfies a weakening property for channel equivalence and handles:

$$\begin{aligned} \Psi \Vdash M_1 \leftrightarrow M_2 &\implies \Psi \otimes \Psi' \Vdash M_1 \leftrightarrow M_2 \\ \Psi \Vdash M \Leftarrow P &\implies \Psi \otimes \Psi' \Vdash M \Leftarrow P \end{aligned}$$

7. Entailment is compositional for assertion equivalence:

$$\Psi \simeq \Psi' \implies \Psi \otimes \Psi'' \simeq \Psi' \otimes \Psi''. \qquad \blacksquare$$

We shall only consider valid instances in the following.

4.2.2 Syntax

The set of HOΨ-calculus *processes* $\mathcal{R}_\Psi$ is built by the formation rules

$$
\begin{array}{rll}
P \in \mathcal{R}_\Psi ::= & \mathbf{0} & \text{Nil} \\
\mid & P_1 \mid P_2 & \text{Parallel} \\
\mid & \overline{M}N.P & \text{Output} \\
\mid & \underline{M}(\lambda \tilde{x} : \widetilde{T})N.P & \text{Input} \\
\mid & \mathbf{run}\ M & \text{Invocation} \\
\mid & \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} & \text{Selection} \\
\mid & (\boldsymbol{\nu} x : T)P & \text{Restriction} \\
\mid & !P & \text{Replication} \\
\mid & (\!|\Psi|\!) & \text{Assertion}
\end{array}
$$

where the shorthand $\widetilde{\varphi} : \widetilde{P}$ denotes a list of cases $\varphi_1 : P_1 \ \square \ \ldots \ \square \ \varphi_n : P_n$.

The nil, parallel, restriction and replication constructs are those of the π-calculus.

The input and output prefixes generalise those of the π-calculus, since here both subject and object are *terms* rather than just names. Thus $\overline{M}N.P$ outputs the term N on M and continues as P, whilst $\underline{M}(\lambda \tilde{x} : \widetilde{T})N.P$ receives a term (e.g. K) on M that must match the pattern N. Here, $\tilde{x}$ is a list of pattern variables, binding into N and P: they are used to extract subterms from K that will then be substituted for the occurrences of $\tilde{x}$ within the continuation P. Any term K received on M must match this pattern for the communication to succeed; this means that it must be possible to obtain K from N by instantiating the variables $\tilde{x} = x_1, \ldots, x_k$ in N with terms $N_1, \ldots N_k$, such that $N[x_1, \ldots, x_k := N_1, \ldots, N_k] = K$.

Unlike the presentation in [98], we here use an explicitly typed version of the language: the types of the pattern variables are found in the list $\widetilde{T}$ where $|\tilde{x}| = |\widetilde{T}|$. Likewise, in the restriction $(\boldsymbol{\nu} x : T)P$, we annotate the name x bound in P with its type T.

The selection construct $\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}$ is a shorthand for a list of cases and is to be understood as saying: If condition φ_i is entailed by the assertions Ψ, we continue as P_i. If more than one condition is entailed, the process is chosen non-deterministically among those where the condition holds. This construct thus generalises the choice and matching operators of the π-calculus.

Higher-order communication is achieved by declaring terms as handles for processes; this may typically be done in an assertion. If the condition $M \Leftarrow P$ is entailed, we can then send P by sending its handle M, and P may thence be executed by using the invocation construct $\mathbf{run}\ M$.

Finally, the authors in [98] impose the restriction that processes must be *well-formed*, which is defined as follows:

Definition 28 (Assertion-guarded processes)**.** An assertion $(\!|\Psi|\!)$ is said to be *guarded*, if it occurs as a sub-process within an input or output, and *unguarded* otherwise. A process is *assertion guarded* if all its assertions are guarded. ■

Definition 29 (Well-formed processes)**.** Let G be an assertion-guarded process. A process P is *well-formed* if it satisfies the following:

- In every sub-process of P, every process in replication, case expressions and handles are assertion-guarded: $!G$, $\mathbf{case}\ \widetilde{\varphi} : \widetilde{G}$ and $M \Leftarrow G$.
- In input prefixes $\underline{M}(\lambda\tilde{x})N$, $\tilde{x} \subseteq \mathrm{n}(N)$ is a sequence without duplicates.
- If $M \Leftarrow P$ then $\mathrm{n}(P) \subseteq \mathrm{n}(M)$. ■

The third criterion states that if M is a handle for P, then M must contain *at least* all the names of P. This is imposed in [98] to ensure that a bound name cannot be sent out of its scope and become exposed by means of the invocation construct.

4.2.3 Reduction semantics

Previous presentations of Ψ-calculi [18; 19; 98] give the semantics in terms of a labelled transition system, and the type system by Hüttel [61] also used the labelled semantics. In this presentation, we shall instead give a reduction semantics for HOΨ, to simplify some of the proofs for the type system.

There already exists a reduction semantics for the first-order Ψ-calculus, given by Åman Pohjola in [136], but this has not been extended to the higher-order Ψ-calculus, and also uses reduction contexts, rather than inference rules and structural congruence, to handle the unfolding of case expressions. In the Higher-Order Ψ-calculus, we also need to handle the unfolding of **run** M terms, but this seems difficult to achieve with reduction contexts, since the process P, for which M is a handle, does not have be to explicitly mentioned. For example, it could be defined as part of the entailment relation that $M \Leftarrow P$. Thus, we shall instead give a more conventional reduction semantics, where processes may be rewritten to build redexes.

Firstly, we define a notion of structural congruence, $\equiv$, containing α-equivalence $\equiv_\alpha$, nil-absorption, the commutative monoidal rules for parallel composition, and the rule for scope extrusion:

Definition 30 (Structural congruence)**.** Structural congruence is the least congruence on process terms containing the following rules:

$$
\begin{aligned}
&[\textsc{s-alpha}]\ P_1 \equiv_\alpha P_2 \implies P_1 \equiv P_2 \\
&[\textsc{s-scope}]\ (\boldsymbol{\nu} x : T)P_1 \mid P_2 \equiv (\boldsymbol{\nu} x : T)(P_1 \mid P_2) \qquad \text{if } x\#P_2 \\
&[\textsc{s-res}_1]\ (\boldsymbol{\nu} x : T)\mathbf{0} \equiv \mathbf{0} \\
&[\textsc{s-res}_2]\ (\boldsymbol{\nu} x_1 : T_1)(\boldsymbol{\nu} x_2 : T_2)P \equiv (\boldsymbol{\nu} x_2 : T_2)(\boldsymbol{\nu} x_1 : T_1)P
\end{aligned}
$$

$$[\text{E-REP}]\frac{}{\Psi \rhd\ !P \underline{\ggg} P \mid !P} \qquad [\text{E-STRUCT}]\frac{P \equiv P'}{\Psi \rhd P \underline{\ggg} P'}$$

$$[\text{E-CASE}]\frac{\Psi \Vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \underline{\ggg} P_i} \qquad [\text{E-RUN}]\frac{\Psi \Vdash M \Leftarrow P}{\Psi \rhd \mathbf{run}\ M \underline{\ggg} P}$$

$$[\text{E-RES}]\frac{\Psi \rhd P \underline{\ggg} P'}{\Psi \rhd (\boldsymbol{\nu} x : T)P \underline{\ggg} (\boldsymbol{\nu} x : T)P'}\ (x \# \Psi)$$

$$[\text{E-PAR}]\frac{\Psi \otimes \mathcal{F}_\Psi(P_2) \rhd P_1 \underline{\ggg} P_1'}{\Psi \rhd P_1 \mid P_2 \underline{\ggg} P_1' \mid P_2}\ (\mathcal{F}_\nu(P_2) \# \Psi, \mathcal{F}_\nu(P_1), P_1)$$

Figure 4.1: The rules defining the evaluation relation.

$[\text{S-IDENT}]\ P \mid \mathbf{0} \equiv P$
$[\text{S-ASSOC}]\ P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$
$[\text{S-COMM}]\ P_1 \mid P_2 \equiv P_2 \mid P_1$ ■

Next we shall need the notion of a *frame* of a process. In Ψ-calculi, new assertions $(\!|\Psi|\!)$ may appear in the syntax and therefore become enabled during the evolution of the program. These are collected by the functions, $\mathcal{F}_\Psi(P)$ and $\mathcal{F}_\nu(P)$, which extract respectively the free assertions and locally declared names from P (the latter of which may bind into the assertions). We call the pair $(\mathcal{F}_\nu(P), \mathcal{F}_\Psi(P))$ the *frame* of P.

Definition 31 (Frame of a process)**.** The relevant clauses for $\mathcal{F}_\Psi(P)$ and $\mathcal{F}_\nu(P)$ are:

$$\begin{aligned} \mathcal{F}_\Psi(P_1 \mid P_2) &\triangleq \mathcal{F}_\Psi(P_1) \otimes \mathcal{F}_\Psi(P_2) & \mathcal{F}_\nu(P_1 \mid P_2) &\triangleq \mathcal{F}_\nu(P_1) \cup \mathcal{F}_\nu(P_2) \\ \mathcal{F}_\Psi((\boldsymbol{\nu} x : T)P) &\triangleq \mathcal{F}_\Psi(P) & \mathcal{F}_\nu((\boldsymbol{\nu} x : T)P) &\triangleq \{x\} \cup \mathcal{F}_\nu(P) \\ \mathcal{F}_\Psi(\!|\Psi|\!) &\triangleq \Psi \end{aligned}$$

and with all remaining clauses of the forms $\mathcal{F}_\Psi(P) \triangleq \mathbf{1}$ and $\mathcal{F}_\nu(P) \triangleq \varnothing$ respectively. ■

Structural congruence will be used for rewriting terms, but as this relation is symmetric, we cannot use it to unfold case expressions and **run** M terms, since that would also allow a reverse reading of the rules. Instead, we introduce a parametrised, asymmetric *evaluation relation* $\cdot \rhd \cdot \underline{\ggg} \cdot$ to properly handle unfolding, which may depend on the assertions currently in effect. It is defined as the least preorder closed under the rules given in Figure 4.1. It replaces the usual structural congruence rule in the reduction semantics to ensure that neither of these operations may be reversed by a reverse reading of the rules, whilst including $\equiv$ for the other kinds of process rewrites where symmetry is unproblematic.

$$[\text{R-COM}]\frac{\Psi \Vdash M_1 \leftrightarrow M_2}{\Psi \rhd \overline{M_1} N[\tilde{x} := \widetilde{K}].P_1 \mid \underline{M_2}(\lambda \tilde{x} : \widetilde{T})N.P_2 \to P_1 \mid P_2[\tilde{x} := \widetilde{K}]}$$

$$[\text{R-EVAL}]\frac{\Psi \rhd P_1 \ggg P_1' \quad \Psi \rhd P_1' \to P_2}{\Psi \rhd P_1 \to P_2}$$

$$[\text{R-RES}]\frac{\Psi \rhd P \to P'}{\Psi \rhd (\boldsymbol{\nu} x : T)P \to (\boldsymbol{\nu} x : T)P'} \ (x \# \Psi)$$

$$[\text{R-PAR}]\frac{\Psi \otimes \mathcal{F}_\Psi(P_2) \rhd P_1 \to P_1'}{\Psi \rhd P_1 \mid P_2 \to P_1' \mid P_2} \ (\mathcal{F}_\nu(P_2) \# \Psi, \mathcal{F}_\nu(P_1), P_1)$$

Figure 4.2: The rules defining the reduction relation.

Lastly, the *reduction relation* $\cdot \rhd \cdot \to \cdot$ is given by the rules in Figure 4.2. Reductions are thus on the form $\Psi \rhd P \to P'$, i.e. relative to a global Ψ containing the assertions currently in effect.

The frame of a process is used in the [R-PAR] and [E-PAR] rules, where the free assertions $(\!|\Psi|\!)$ occurring in the second component P_2 of a parallel composition are composed with the global Ψ, since they may affect the evaluation (resp. reduction) of the first component, P_1. However, P_2 may also contain local, new names $(\boldsymbol{\nu} x : T)$, which may bind into $(\!|\Psi|\!)$, and it must therefore be ensured that these are fresh w.r.t. both the free and locally declared names in P_1, as well as any names found in the global Ψ. This is stated in the side conditions of these rules, where the local names are collected by $\mathcal{F}_\nu(P_2)$.

The reduction relation defined above does not coincide exactly with the τ-labelled transition relation $\xrightarrow{\tau}$ from the original presentation in [98]. It is slightly larger, since, in the 'unfolding rules' [E-CASE], [E-REP] and [E-RUN], we allow the respective constructs to unfold to a process P, regardless of whether P itself can perform a reduction step or not. In contrast, the labelled semantics requires that a transition $P \xrightarrow{\alpha} P'$ can be concluded for the unfolded process. However, the following result can be shown:

Proposition 7. $\Psi \rhd P \xrightarrow{\tau} P' \implies \Psi \rhd P \to P'$ *where* $\xrightarrow{\tau}$ *is the* τ*-labelled transition relation given in [98].*

The proof in itself is not difficult: it proceeds by induction in the derivation of $\Psi \rhd P \xrightarrow{\tau} P'$, where we show that for each possible transition of this form, we can derive a corresponding conclusion of the form $\Psi \rhd P \to P'$ by using the rules of the reduction semantics. The main importance of this is that even though $\to$

only approximates the relation $\xrightarrow{\tau}$, it will suffice for the purpose of showing subject reduction.

4.3 The generic type system

As in other type systems, we need to describe when processes are well-typed, but since we in the HOΨ-calculus can have arbitrary terms, conditions and assertions, we shall also need a way to decide when *they* are well-typed. As they are parameters to the HOΨ-calculus, we cannot specify a set of type rules for them, as we can with processes. Instead, such rules must likewise be provided as parameters to create an instance of the generic type system. The parameters to be specified consist of *rules* for concluding type judgments of the form

$$\Gamma, \Psi \vdash M : T \qquad \Gamma, \Psi \vdash \varphi \qquad \Gamma, \Psi \vdash \Psi'$$

and two *predicates* $T_1 \leftrightarrows T_2$ and $T \curvearrowleft \Gamma$. These rules must then satisfy a number of requirements, here denoted *instance assumptions*, which we shall need in the proof for subject reduction. These are described in the following sections.

4.3.1 Types and environments

In type systems for process calculi, types can contain names, as exemplified by the instances considered in the work on first-order typed psi-calculi [61]. We therefore assume that the set of types $\mathcal{T}$ is a nominal datatype ranged over by T and that substitutions can affect types. Furthermore, we need the concept of a type environment Γ to record the types of free names; thus Γ is a partial function with finite support $\Gamma : \mathcal{N} \rightharpoonup \mathcal{T}$. Sometimes it can be convenient to think of Γ as a set of tuples $\Gamma \subseteq \mathcal{N} \times \mathcal{T}$ where $(x, T) \in \Gamma$ if $\Gamma(x) = T$. We write $\Gamma, x : T$ to denote the type environment Γ extended by the name x with type T.

As usual, our type judgments will be relative to a type environment Γ. However, due to the presence of assertions which may affect the well-typedness of a process, term, condition, or indeed an assertion, our type judgments must also be relative to a global assertion Ψ. Thus, type judgements for processes will be of the form $\Gamma, \Psi \vdash P$. As the global assertion may be composed with assertions appearing in a process, we shall therefore also need the notion of a extension ordering on assertions, and we also define a corresponding notion for type environments. To this end, we identify assertions up to commutativity and associativity of composition. The resulting notion of equivalence is denoted $=_\Psi$.

Definition 32 (Extension ordering)**.** We say that $\Psi_1 \leq \Psi_2$ if $\mathrm{n}(\Psi_1) \subseteq \mathrm{n}(\Psi_2)$ and there exists a Ψ such that $\Psi_1 \otimes \Psi =_\Psi \Psi_2$. Likewise, we say that $\Gamma_1 \leq \Gamma_2$ if $\mathrm{dom}(\Gamma_1) \subseteq \mathrm{dom}(\Gamma_2)$ and there exists a Γ such that $\Gamma_1, \Gamma = \Gamma_2$. ■

The extension ordering is superficially similar to the notion of static implication, but differs from it as no notion of assertion entailment is involved. We deliberately use the notation $\Psi_1 \leq \Psi_2$ in order to be reminiscent of the extension ordering for type environments.

Lastly, the environments Γ, Ψ appearing in type judgments must be *well-formed*. As $\mathcal{T}$ is a nominal datatype, names can also appear inside a type T, and we must therefore require that if Γ contains a type annotation $x : T$, with names with type annotations appearing inside T, then these must coincide with those found in Γ.

Example 6. Consider a size-dependent list type $\mathsf{List}(x : \mathsf{Int})$ where the size x has been annotated with a type. Then for a type environment $\Gamma = x : T_x, y : \mathsf{List}(x : \mathsf{Int})$ we require that $T_x : \mathsf{Int}$. ■

Definition 33 (Well-formed environments)**.** Let Γ_T denote the type annotations extracted from a type T and let $\mathrm{an}(T)$ denote the set of annotated names occurring in T. We say the environments Γ, Ψ are *well-formed* if $\mathrm{fn}(\Psi) \subseteq \mathrm{dom}(\Gamma)$, and for all names $x \in \mathrm{dom}(\Gamma)$ the following holds:

$$\Gamma(x) = T \implies \mathrm{fn}(T) \subseteq \mathrm{dom}(\Gamma)$$
$$\Gamma(x) = T \implies \forall y \in \mathrm{an}(T) \,.\, \Gamma_T(y) = \Gamma(y)$$ ■

4.3.2 Channel compatibility

When we type an input or output prefix term, the type of the subject M and the type of the object (the term transmitted on channel M) must be compatible w.r.t. a *compatibility predicate* $\nleftrightarrow$ that describes which types of values can be carried by channels of a given type. Thus, $T_1 \nleftrightarrow T_2$ denotes that channels of type T_1 can carry terms of type T_2, and we require that the set of types be defined such that this holds. Furthermore, we distinguish between output compatibility $\nleftrightarrow^+$, and input compatibility $\nleftrightarrow^-$, and we write $T_1 \nleftrightarrow T_2$ if both $T_1 \nleftrightarrow^+ T_2$ and $T_1 \nleftrightarrow^- T_2$.

Example 7. Consider the channel types in the sorting system by Milner [83]. Here, a name has type $\mathrm{ch}(T)$, if it is a channel that can be used to transmit names of type T, so in that case we would therefore require that $\nleftrightarrow$ be defined such that $\mathrm{ch}(T) \nleftrightarrow T$. ■

In our definition of compatibility, we assume given a subtype ordering $\leq$ on types. If $T_1 \leq T_2$, a term of type T_1 can be used wherever a term of type T_2 is needed. Thus we require the usual subsumption rule for types, namely that a term of a given type T_1 can also be typed with a supertype T_2:

$$[\textsc{t-subs}]\frac{\Gamma, \Psi \vdash M : T_1}{\Gamma, \Psi \vdash M : T_2} \ (T_1 \leq T_2)$$

The compatibility predicate for a type T must further satisfy the following requirements w.r.t. the subtyping relation:

1. If a channel type can carry two distinct types, then the types have to be related by the subtype ordering. That is, if $* \in \{+, -\}$, $T \leftrightarrow\!\!\!\!\rho^{*}\ T_1$ and $T \leftrightarrow\!\!\!\!\rho^{*}\ T_2$ with $T_1 \neq T_2$, then $T_1 \leq T_2$ or $T_2 \leq T_1$.

2. Output compatibility is *contravariant*. That is, if $T \leftrightarrow\!\!\!\!\rho^{+}\ T_2$ and $T_1 \leq T_2$, then also $T \leftrightarrow\!\!\!\!\rho^{+}\ T_1$. This requirement mirrors that of [100]. If $T_1 \leq T_2$, then a term of type T_1 can be used wherever a term of type T_2 is needed, and a channel that outputs terms of the more general type T_2 can therefore be used, wherever a channel of the specialized type T_1 is required.

3. Input compatibility is *covariant*. That is, if $T \leftrightarrow\!\!\!\!\rho^{-}\ T_1$ and $T_1 \leq T_2$, then also $T \leftrightarrow\!\!\!\!\rho^{-}\ T_2$. This requirement, too, mirrors that of [100]. Here, if $T_1 \leq T_2$, a channel that accepts terms of type T_1 can also be used to accept terms of type T_2.

Like the other assumptions, the compatibility assumptions are needed to ensure soundness. Consider, for instance, the requirement that channels cannot carry values that are not related by subtyping. Here is an example that illustrates the need for this particular requirement:

Example 8. Assume that the constant $+$ has its type given by $+ : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$, that $\mathsf{Int} \not\leq \mathsf{Bool}$ and $\mathsf{Bool} \not\leq \mathsf{Int}$. Finally, assume that $c : T$ where $T \leftrightarrow\!\!\!\!\rho\ \mathsf{Int}$ and $T \leftrightarrow\!\!\!\!\rho\ \mathsf{Bool}$. The process

$$\overline{c}17 \mid \overline{c}\mathsf{true} \mid \underline{c}(\lambda x).\overline{c}x + x$$

can reduce to an ill-typed process, namely

$$\overline{c}17 \mid \overline{c}\mathsf{true} + \mathsf{true}$$

If we require that all types carried by a channel must be related by the subtype ordering, the problem disappears, as we can then type the subject x of the input prefix in the receiving subprocess with the least supertype of the message terms that can be bound to x. ■

4.3.3 Instance assumptions

The type rules for terms, assertions and conditions will depend on how these parameters are defined for a specific instance of the HOΨ-calculus, and they must therefore be provided as part of the instantiation of the generic type system.

However, like type judgments for processes, they must also be relative to a type environment Γ and a global Ψ, so we require that they be of the form $\Gamma, \Psi \vdash \mathcal{J}$, where $\mathcal{J}$ is defined by the formation rules:

$$\mathcal{J} ::= M : T \mid \varphi \mid \Psi$$

We introduce a collection of assumptions that must hold for an instance of the generic type system to be valid. They pertain to the rules for type judgments $\Gamma, \Psi \vdash \mathcal{J}$ for terms, conditions and assertions, on which we must impose certain restrictions to allow us to prove subject reduction for the generic type system. Whether or not the following requirements are minimal is an open problem at the time of writing.

Definition 34 (Qualified judgement)**.** We say a judgement $\Gamma, \Psi \vdash \mathcal{J}$ is *qualified*, if Γ, Ψ are well-formed, and if $\mathrm{fn}(\mathcal{J}) \subseteq \mathrm{dom}(\Gamma)$. ■

Type rules have zero or more *premises* and a *conclusion* that are qualified judgements of the form $\Gamma, \Psi \vdash \mathcal{J}$. They may also include a *side condition*, which is a predicate that is not a qualified type judgement, but which can depend on the judgements in the rule. Thus, they are of the following form:

$$[\textsc{Name}]\frac{\Gamma, \Psi \vdash \mathcal{J}_1 \quad \cdots \quad \Gamma, \Psi \vdash \mathcal{J}_n}{\Gamma, \Psi \vdash \mathcal{J}} \ (\textsc{Pred})$$

where Pred is a (possibly empty) predicate, and $\Gamma, \Psi \vdash \mathcal{J}_i$ is a (possibly empty) list of premises. We assume the type rules for terms, conditions and assertions are of this form.

Furthermore, we assume that the type rules for terms, conditions and assertions are defined such that the usual properties of weakening and strengthening of the type environment hold for all type judgments $\mathcal{J}$. These properties are summarised in the following definition:

Definition 35 (Weakening and strengthening)**.** Judgements of the form $\Gamma, \Psi \vdash \mathcal{J}$ satisfy

1. The *environment weakening property*, if whenever $\Gamma, \Psi \vdash \mathcal{J}$, then for any $x \notin \mathrm{dom}(\Gamma)$ and type T we have $\Gamma, x : T, \Psi \vdash \mathcal{J}$
2. The *environment strengthening property*, if whenever $\Gamma, x : T, \Psi \vdash \mathcal{J}$ and $x \# \mathcal{J}$ we have $\Gamma, \Psi \vdash \mathcal{J}$.
3. The *assertion weakening property*, if whenever $\Gamma, \Psi \vdash \mathcal{J}$ and $\Psi \leq \Psi'$ and $\mathrm{n}(\Psi') \subseteq \mathrm{dom}(\Gamma)$, then also $\Gamma, \Psi' \vdash \mathcal{J}$. ■

The weakening assumptions in Definition 35 tell us that judgements are *monotonic* w.r.t. the extension ordering: If $\Gamma, \Psi \vdash \mathcal{J}$ and $\Gamma \leq \Gamma_1$ then also $\Gamma_1, \Psi \vdash \mathcal{J}$ and if $\Psi \leq \Psi_1$ then also $\Gamma, \Psi_1 \vdash \mathcal{J}$. Likewise, the strengthening assumption tells us that the extension ordering $\leq$ for type environments is *well-founded*: If $\Gamma, \Psi \vdash \mathcal{J}$ then there is a least $\Gamma_0 \leq \Gamma$ such that $\Gamma_0, \Psi \vdash \mathcal{J}$. We do not need to require a similar well-foundedness property to hold for the assertion environment, but it would also be quite natural to assume that there exists a least $\Psi_0 \leq \Psi$ such that $\Gamma, \Psi_0 \vdash \mathcal{J}$.

Weakening and strengthening properties tell us that we can introduce additional hypotheses and remove superfluous hypotheses and preserve typability. In type systems for process calculi that have a notion of communication between parallel components P and Q as well as the notion of restriction known from the π-calculus, properties of strengthening and weakening are necessary for soundness because of scope extrusion. In this setting, the need for these properties mirrors the need for open/close rules in a labelled semantics for name-passing calculi [85].

Example 9. Consider a process $((\boldsymbol{\nu} x)P) \mid Q$ where the bound name x is sent by P and received by Q. Weakening is necessary here if we are to establish that P and Q can be typed within the same context Γ, which includes x, and strengthening is needed in order to deal with the fact that the resulting process $(\boldsymbol{\nu} x)(P \mid Q)$ must be typable in the same Γ, which assumes that the x that was local to one component is still not a free name. ■

Next, we must require type judgments for terms to respect channel equivalence w.r.t. the type annotations in Γ:

$$[\text{T-EQUAL}]\frac{\Gamma, \Psi \vdash M : T}{\Gamma, \Psi \vdash N : T} \ \left(\Psi \Vdash M \dot{\leftrightarrow} N\right)$$

Thus, if two terms are channel equivalent, then they must also have the same type.

Another group of assumptions concern *compositionality:* for each of the three nominal datatypes of terms, conditions and assertions, we require that there is a type rule, that allows us to type a composite element by typing each of its constituents. These assumptions are similar to those found in [61]. For assertions, the rule must be of the following form:

$$[\text{T-COMP-ASS}]\frac{\Gamma, \Psi \vdash \Psi_1 \quad \Gamma, \Psi \vdash \Psi_2}{\Gamma, \Psi \vdash \Psi_1 \otimes \Psi_2}$$

For conditions and terms, we make use of the assumption that these nominal datatypes must be describable as term algebras with constructor symbols f. If a type rule types a composite condition $f(\varphi_1, \ldots, \varphi_n)$, it must be of the following form:

$$[\text{T-COMP-COND}]\frac{\Gamma, \Psi \vdash \varphi_i \quad 1 \leq i \leq n}{\Gamma, \Psi \vdash f(\varphi_1, \ldots, \varphi_n)}$$

Likewise for terms, the rule must be of the form

$$[\text{T-COMP-TERM}]\frac{\Gamma, \Psi \vdash M_i : T_i \quad 1 \leq i \leq n}{\Gamma, \Psi \vdash f(M_1, \ldots, M_n) : F(T_1, \ldots, T_n)}$$

where $F(T_1, \ldots, T_n)$ is the type of the composite term and F is a function over types. This assumption will guarantee that the type of a composite term $f(M_1, \ldots, M_n)$ can be determined from the types T_i (for $1 \leq i \leq n$) using F.

4.3.4 Higher-order assumptions

In the higher-order Ψ-calculus, terms M serve a dual purpose: They can be used as simple terms or as handles, that is, terms for which the **run** M primitive can be used to obtain a process P. As an example, in the ρ-calculus a name can be used both as the name of a channel and as the handle of a process.

Processes do not have types in our type system; only terms do. However, we can view the 'type' of a process P as a type environment Γ that make P well-typed. Thus we require that the type T of a term M, and the 'type' (environment) of a process P must be compatible w.r.t. a *higher-order compatibility predicate* $\curvearrowleft$, if M is a handle for P. Thus $T \curvearrowleft \Gamma$ denotes that terms of type T can be used to obtain processes that are well-typed w.r.t. Γ, if the term is executed with the invocation construct. $\curvearrowleft$ must therefore be defined such that the following holds:

$$[\text{T-HANDLE}] \quad \Gamma, \Psi \vdash M : T \wedge \Psi \Vdash M \Leftarrow P \implies \exists \Gamma'. T \curvearrowleft \Gamma' \wedge \Gamma', \Psi \vdash P$$

This requirement is necessitated by the formulation of the HOΨ-calculus itself. The HOΨ-calculus does not require that a process P, for which M is a handle, must be defined within the syntax of a program where M may be invoked. It is entirely possible to define it directly as part of the entailment relation.

Example 10. Assume for some HOΨ-instance that entailment contains the following definition:

$$\Vdash \triangleq \ldots \cup \{ (\mathbf{1}, M \Leftarrow \mathbf{0}) \mid M \in \mathbb{T} \}$$

This would make *every* term M be a handle for the $\mathbf{0}$ process, which would be entailed by the unit assertion $\mathbf{1}$. Because of the weakening requirement of Definition 27 (Valid parameters, requirement 6), handles are not allowed to be altered by composing new assertions onto this minimal assertion environment; hence *no* other handles can be declared by this setting. ■

In the above example, $\mathbf{0}$ was chosen for simplicity, but indeed *any* assertion-guarded process P could have been used. As our type system is syntactic (i.e. it analyses the syntax of programs), we have no means to check that such processes are well-typed, and we therefore have no other option than to simply require them to be well-typed. If handles are defined in this way, well-typedness of the handled process must therefore be shown manually.

Usually, however, handles would be defined in assertions appearing in the syntax of a program, e.g. as $(\!|\{ M \Leftarrow P \}|\!)$, and with the definition of entailment containing e.g. the following rule:

$$[\text{H-ENTAIL}] \frac{\{ M \Leftarrow P \} \in \Psi}{\Psi \Vdash M \Leftarrow P}$$

If that is the case, then [T-HANDLE] can be satisfied by including a type rule for typing handles in the type rules for assertions:

$$[\text{T-ASS-HANDLE}]\frac{\Gamma,\Psi\vdash M:T \quad \Gamma',\Psi\vdash P}{\Gamma,\Psi\vdash\{M\Leftarrow P\}}\begin{pmatrix} T\curvearrowleft\Gamma' \\ \Gamma'\leq\Gamma \end{pmatrix}$$

Example 11. A simple way to define types of handles, such that $\curvearrowleft$ holds, is to let $\mathcal{T}$ be defined as

$$T\in\mathcal{T}::=\ldots \mid (T,\Gamma)$$

and then let $\curvearrowleft$ be defined such that $(T,\Gamma)\curvearrowleft\Gamma$. Indeed, we shall do that in our type system instances in Section 4.5. ■

We shall also need to impose a requirement on the definition of $\curvearrowleft$, when handles are subjected to a term substitution σ:

$$[\text{T-SUBS-HANDLE}]\qquad \begin{array}{ll} & \Gamma,\Psi\vdash M:T_1 \\ \wedge & \Gamma,\Psi\vdash M\sigma:T_2 \\ \wedge & T_1\curvearrowleft\Gamma_1 \\ \Longrightarrow & \exists\Gamma_2.T_2\curvearrowleft\Gamma_2\wedge\Gamma_2\leq\Gamma_1 \end{array}$$

This requirement essentially states that if M has a higher-order type T_1, i.e. such that $T_1\curvearrowleft\Gamma_1$, and M is subjected to a term substitution σ, such that the resulting term $M\sigma$ has type T_2, then T_2 too must be a higher-order type. Moreover, the Γ_2 we obtain from the type of $M\sigma$ by means of $T_2\curvearrowleft\Gamma_2$ must be such that $\Gamma_2\leq\Gamma_1$; i.e. it can *at most* contain the same names as Γ_1. In other words, a substitution is not allowed to alter the fact that a term has a higher-order type; and the new type must not allow the substituted term to be used as handle for a process containing more free names than the type of the original term would allow, prior to substitution. This is easily satisfied by any ordinary definition of substitution.

Example 12. Consider a HOΨ-instance where terms are just atomic names: $\mathbb{T}\triangleq\mathcal{N}$, and with simple channel types as in Example 7. Assume that $\Gamma,\Psi\vdash x:\mathrm{ch}(T)$ and $\Gamma,\Psi\vdash z:T$, where $T=(T',\Gamma')$ for some T',Γ', and with $\curvearrowleft$ defined such that $(T',\Gamma')\curvearrowleft\Gamma'$. Thus z has a higher-order type, and it may therefore be used as a handle, so we want to be able to pass it around.

Consider now the reduction:

$$\underline{x}(\lambda y:T)y.\mathbf{run}\ y\mid\overline{x}z.\mathbf{0}\rightarrow\mathbf{run}\ y[y:=z]$$

[T-SUBS-HANDLE] amounts to saying that $\Gamma,\Psi\vdash y[y:=z]:T_1$ and $T_1\curvearrowleft\Gamma''$ such that $\Gamma''\leq\Gamma'$, which indeed holds, since $T=T_1$ and therefore $\Gamma'=\Gamma''$. ■

4.3.5 Type rules for processes

Unlike the type rules for terms, conditions and assertions, the type rules for processes are common to every instance. These are of the form $\Gamma,\Psi\vdash P$. As before, we only

$$[\text{T-CASE}]\frac{\Gamma,\Psi\vdash\varphi_i \quad \Gamma,\Psi\vdash P_i}{\Gamma,\Psi\vdash \mathbf{case}\ \widetilde{\varphi}:\widetilde{P}} \qquad [\text{T-REP}]\frac{\Gamma,\Psi\vdash P}{\Gamma,\Psi\vdash !P}$$

$$[\text{T-RES}]\frac{\Gamma,x:T,\Psi\vdash P}{\Gamma,\Psi\vdash(\boldsymbol{\nu}x:T)P}\ (x\#\Psi) \qquad [\text{T-ASS}]\frac{\Gamma,\Psi\vdash\Psi'}{\Gamma,\Psi\vdash(\!|\Psi'|\!)}$$

$$[\text{T-RUN}]\frac{\Gamma,\Psi\vdash M:T}{\Gamma,\Psi\vdash \mathbf{run}\ M}\begin{pmatrix} T\curvearrowleft\Gamma' \\ \Gamma'\leq\Gamma\end{pmatrix} \qquad [\text{T-NIL}]\frac{}{\Gamma,\Psi\vdash\mathbf{0}}$$

$$[\text{T-IN}]\frac{\Gamma,\Psi\vdash M:T \quad \Gamma,\tilde{x}:\widetilde{T},\Psi\vdash N:T' \quad \Gamma,\tilde{x}:\widetilde{T},\Psi\vdash P}{\Gamma,\Psi\vdash \underline{M}(\lambda\tilde{x}:\widetilde{T})N.P}\ (T\leftarrow\!\!\rho^{-}T')$$

$$[\text{T-OUT}]\frac{\Gamma,\Psi\vdash M:T \quad \Gamma,\Psi\vdash N:T' \quad \Gamma,\Psi\vdash P}{\Gamma,\Psi\vdash \overline{M}N.P}\ (T\leftarrow\!\!\rho^{+}T')$$

$$[\text{T-PAR}]\frac{\begin{array}{c}\Gamma,\mathcal{F}_\Gamma(P_2),\Psi\otimes\mathcal{F}_\Psi(P_2)\vdash P_1\\ \Gamma,\mathcal{F}_\Gamma(P_1),\Psi\otimes\mathcal{F}_\Psi(P_1)\vdash P_2\end{array}}{\Gamma,\Psi\vdash P_1\mid P_2}\begin{pmatrix}\mathcal{F}_\nu(P_1)\#\Psi,\mathcal{F}_\nu(P_2),P_2\\ \mathcal{F}_\nu(P_2)\#\Psi,\mathcal{F}_\nu(P_1),P_1\end{pmatrix}$$

Figure 4.3: Type judgements for processes.

consider qualified judgements; i.e. where Γ,Ψ are well-formed and $\text{fn}(P)\subseteq\text{dom}(\Gamma)$, so every name mentioned in the process in the judgement is bound in the type environment.

We shall need a way to extract the type annotations of scoped names from a process P. This will be needed in the rule for typing a parallel composition $P_1\mid P_2$, since the free assertions in P_2 must be composed onto the global assertion environment for the typing of P_1, and vice versa. Such a free assertion $(\!|\Psi|\!)$ may, however, be declared inside the scope of a local name $(\boldsymbol{\nu}x:T)$, with x binding into $(\!|\Psi|\!)$, so the type information must therefore be added to the type environment Γ. Thus we shall write $\mathcal{F}_\Gamma(\cdot)$ to mean $\mathcal{F}_\Gamma((\boldsymbol{\nu}x:T)P)\triangleq x:T,\mathcal{F}_\Gamma(P)$, and for all other clauses this function is defined similar to $\mathcal{F}_\nu(\cdot)$.

The type rules for processes are given in Figure 4.3; they are mostly similar to those of [61], except for the rule [T-RUN] used to type the **run** M construct, which is the only construct that is new to the higher-order setting. Here we require that M must have a higher-order type, containing a Γ matching the Γ it is typed relative to, which ensures that no new names are introduced by running the process associated with the handle M.

In the rule [T-PAR] we require that for a parallel composition $P_1\mid P_2$ to be typable, P_1 and P_2 must both be typable within type environments and assertions that add

information extracted from the other component. This is a natural requirement, since P_1 can, among other things, mention handles for processes established in P_2, and vice versa. The side condition then asserts that all new names declared in P_1, using the $(\boldsymbol{\nu} x : T)$ construct, must be fresh for Ψ *and* both the free and new names occurring in P_2, and vice versa for P_2, similar to the side conditions for the [E-PAR] and [R-PAR] rules in the semantics.

4.4 Properties of the generic type system

Type systems normally ensure two properties of well-typed programs: a *subject reduction* property guarantees that a well-typed program remains well-typed under reduction; and a *safety* property ensures that if a program is well-typed then a certain safety predicate holds. The latter may depend on the particular instance of the type system and must therefore be shown individually, for each instance, but subject reduction can be shown for the generic type system.

We establish these properties through a series of lemmas, building up to the main result. Here we shall see the various instance assumptions and parameter requirements (distributivity and weakening) from section 4.3.3, come into play. We begin with the usual results of weakening and strengthening of the type environment Γ:

Lemma 16 (Weakening)**.** *If* $\Gamma, \Psi \vdash P$ *and* $x \notin \mathrm{dom}(\Gamma)$ *then* $\Gamma, x : T, \Psi \vdash P$.

Proof. By induction in the rules of the type judgment $\Gamma, \Psi \vdash P$.

The case for $\mathbf{0}$ is immediate, and the cases for $P_1 \mid P_2$, $(\boldsymbol{\nu} x : T)P$ and $!P$ follow from straightforward application of the induction hypothesis. The remaining cases contain either terms M, conditions φ, or assertions $(\!|\Psi|\!)$, so here the environment weakening assumption (Definition 35, item 1) is required.

Consider the case for output: By [T-OUT], we know that $\Gamma, \Psi \vdash \overline{M}N.P$, and from the premises that $\Gamma, \Psi \vdash M : T_1$, $\Gamma, \Psi \vdash N : T_2$ and $\Gamma, \Psi \vdash P$. Then $\Gamma, x : T, \Psi \vdash P$ holds by induction hypothesis, and $\Gamma, x : T, \Psi \vdash M : T_1$ and $\Gamma, x : T, \Psi \vdash N : T_2$ both hold by the environment weakening assumption (Definition 35, item 1). Thus we can conclude $\Gamma, x : T, \Psi \vdash \overline{M}N.P$ by [T-OUT].

The cases for [T-IN] and [T-RUN] are similar. In [T-CASE], the premise (for each φ_i in $\widetilde{\varphi}$) that $\Gamma, x : T, \Psi \vdash \varphi_i$ is also satisfied because of the environment weakening assumption (Definition 35, item 1), and likewise for the premise $\Gamma, x : T, \Psi \vdash \Psi'$ in [T-ASS]. □

Lemma 17 (Strengthening)**.** *If* $\Gamma, x : T, \Psi \vdash P$ *and* $x \# P, \Psi$ *then* $\Gamma, \Psi \vdash P$.

Proof. By induction in the rules of the type judgment $\Gamma, \Psi \vdash P$.

This proof proceeds in the same way as the proof for Lemma 16 (Weakening), but this time using the environment strengthening assumption (Definition 35, item 2) for the cases where terms, conditions or assertions appear in the premise. □

We shall also need a weakening result for the global assertion environment Ψ. This result is necessitated by the syntax of the HOΨ-calculus itself, which allows guarded assertions in continuations to become unguarded after a reduction, and hence they may become composed onto the global Ψ. The result we wish to have is thus that any well-typed process remains well-typed after a composition of any assertion in the assertion environment, as long as all names in the new assertion environment are in the support of the type environment:

Lemma 18 (Assertion environment weakening). *If* $\Gamma, \Psi \vdash P$ *and* $\mathrm{n}(\Psi') \subseteq \mathrm{dom}(\Gamma)$ *and* $\Psi \leq \Psi'$ *then* $\Gamma, \Psi' \vdash P$.

Proof. By induction in the rules of the type judgment $\Gamma, \Psi \vdash P$.

- For [T-NIL], [T-IN], [T-OUT], [T-CASE], [T-REP], [T-RUN] and [T-ASS] the proof proceeds in the same way as the proof for Lemma 16 (Weakening), but this time using the assertion weakening assumption (Definition 35, item 3) for the cases where terms, conditions or assertions appear in the premise.

- In the cases of [T-RES] and [T-PAR] we must ensure that the freshness requirements in the side conditions are satisfied. Thus, consider the case for [T-PAR], since the case for [T-RES] is simpler: We know that $\Gamma, \Psi \vdash P_1 \mid P_2$, and from the premise that $\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1$, and from the side condition that $\mathcal{F}_\nu(P_2) \#\Psi, \mathcal{F}_\nu(P_1), P_1$ (and conversely for P_2). We then use α-conversion as necessary to ensure that the condition $\mathcal{F}_\nu(P_2) \#\Psi'$ holds, such that $\mathcal{F}_\Gamma(P_2)$ will not capture any free names in Ψ' that are not in Ψ. We then do the same for the other premise, and then they both hold by induction hypothesis. Then we conclude $\Gamma, \Psi' \vdash P_1 \mid P_2$ by [T-PAR]. □

Next, we need a result of *substitution*, which must be shown for both the 'parameter' type judgements $\Gamma, \Psi \vdash \mathcal{J}$ and type judgments for processes $\Gamma, \Psi \vdash P$:

Lemma 19 (Substitution). *Let* $\mathcal{G} ::= \mathcal{J} \mid P$*, and let* σ *be a term substitution. If* $\Gamma, \Psi \vdash \mathcal{G}$, $\mathrm{dom}(\Gamma) = \mathrm{dom}(\sigma)$*, and for all* $x \in \mathrm{dom}(\sigma)$ *it holds that* $\Gamma, \Psi \vdash \sigma(x) : \Gamma(x)$*, then* $\Gamma, \Psi \vdash \mathcal{G}\sigma$.

Proof. There are four different kinds of type judgments to consider, since $\mathcal{G}$ can be a term, a condition, an assertion or a process:

If the judgment is of the form $\Gamma, \Psi \vdash M : T$, we proceed by induction in the structure of M. If M is a name, the result is immediate. Else, M must be a

composite term with n immediate constituents; i.e. $M = f(M_1, \dots, M_n)$. Thus, this judgment must have been concluded by the (assumed) compositionality rule for terms, [T-COMP-TERM], so we must have that

$$\frac{\Gamma, \Psi \vdash M_i : T_i \qquad 1 \leq i \leq n}{\Gamma, \Psi \vdash f(M_1, \dots, M_n) : F(T_1, \dots, T_n)}$$

By induction hypothesis, we have that $\Gamma, \Psi \vdash M_i\sigma : T_i\sigma$ for all the constituents, and by [T-COMP-TERM] we can conclude that

$$\Gamma, \Psi \vdash f(M_1\sigma, \dots, M_n\sigma) : F(T_1\sigma, \dots, T_n\sigma)$$

By the distributivity assumption, we have that

$$f(M_1\sigma, \dots, M_n\sigma) : F(T_1\sigma, \dots, T_n\sigma) = f(M_1, \dots, M_n)\sigma : F(T_1, \dots, T_n)\sigma$$

which thus lets us conclude $\Gamma, \Psi \vdash f(M_1, \dots, M_n)\sigma : F(T_1, \dots, T_n)\sigma$ as desired.

If the judgment is of the form $\Gamma, \Psi \vdash \varphi$ or $\Gamma, \Psi \vdash \Psi'$, the same reasoning applies, using the respective compositionality assumption [T-COMP-COND] or [T-COMP-ASS]. If the judgment is of the form $\Gamma, \Psi \vdash P$, we proceed by induction in the structure of P:

- For $\mathbf{0}$, the result is immediate. For assertions $(\!|\Psi|\!)$, it follows from the result above, and likewise for case constructs $\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}$, with $\Gamma, \Psi \vdash P_i\sigma$ from the induction hypothesis. For replication $!P$, it follows directly from the induction hypothesis, and also for $(\boldsymbol{\nu} x : T)P$, since we know by requirement that $x \notin \mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Gamma) = \mathrm{dom}(\sigma)$; hence σ cannot affect the name x.

- For input and output, $\underline{M}(\lambda \tilde{x} : \widetilde{T})N.P$ and $\overline{M}N.P$, the result follows from the induction hypothesis for the continuation P, and from the above result regarding substitution on terms for the subject and object M, N, where again we know that the substitution cannot affect any of the bound names $\tilde{x}$.

- For parallel composition we have that $\Gamma, \Psi \vdash P_1 \mid P_2$, and we wish to conclude that $\Gamma, \Psi \vdash (P_1 \mid P_2)\sigma$. From the premise we know that $\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1$, and conversely for P_2. Applying the substitution gives us, that we must be able to conclude

  $$\Gamma, \mathcal{F}_\Gamma(P_2\sigma), \Psi \otimes \mathcal{F}_\Psi(P_2\sigma) \vdash P_1\sigma$$

  The substitution cannot introduce new, bound names, which would be collected by $\mathcal{F}_\Gamma(\cdot)$, hence $\mathcal{F}_\Gamma(P_2) = \mathcal{F}_\Gamma(P_2\sigma)$. Now since we know that $\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1$, then by Lemma 18 we can conclude that $\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2\sigma) \vdash P_1$. We then apply the induction hypothesis and obtain

  $$\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2\sigma) \vdash P_1\sigma$$

  A similar reasoning then applies for P_2, which allows us to conclude.

- Finally for **run** M, we know that $\Gamma, \Psi \vdash \mathbf{run}\, M$, which must have been concluded by [T-RUN], and we wish to conclude $\Gamma, \Psi \vdash \mathbf{run}\, M\sigma$. From the premise and side condition we have that $\Gamma, \Psi \vdash M : T$ and $T \curvearrowleft \Gamma'$ and $\Gamma' \leq \Gamma$. By the case for terms above, we therefore have that $\Gamma, \Psi \vdash M\sigma : T\sigma$. Then by requirement [T-SUBS-HANDLE] we get that $T\sigma \curvearrowleft \Gamma''$ and $\Gamma'' \leq \Gamma'$, and by transitivity of $\leq$ therefore also $\Gamma'' \leq \Gamma$. We can then conclude $\Gamma, \Psi \vdash \mathbf{run}\, M\sigma$ by [T-RUN]. □

As we here use reduction semantics with an asymmetric evaluation relation to handle unfolding of **case** and **run** expressions, we shall also need two results that describe how frames can evolve during evaluation. The first result establishes that the property of being assertion-guarded is preserved by the evaluation relation $\underline{\ggg}$.

Lemma 20 (Frame post evaluation)**.** *If* $\Psi \rhd P \underline{\ggg} P'$ *and* $\mathcal{F}_\nu(P) \# \Psi$ *and* $\mathcal{F}_\nu(P') \# \Psi$*, then* $\Psi \otimes \mathcal{F}_\Psi(P) =_\Psi \Psi \otimes \mathcal{F}_\Psi(P')$.

Proof. The proof proceeds by induction in the rules for concluding the evaluation $\Psi \rhd P \underline{\ggg} P'$ (Figure 4.1).

- The cases for [E-RES] and [E-PAR] are straightforward by application of the induction hypothesis.
- For [E-STRUCT] we need an extra induction in the rules of structural congruence (Definition 30), but it is immediately clear from the definition that a rewrite by structural congruence cannot expose any new assertions which were not free before.
- For the cases of [E-REP], [E-CASE] and [E-RUN], we know from the criterion of well-formedness for processes that neither of these processes may contain unguarded assertions. Thus, unfolding $!P$ or **case** $\widetilde{\varphi} : \widetilde{P}$ or **run** M cannot expose any new assertions. □

The second result states that the free assertions in a process at most can increase after a reduction:

Lemma 21 (Frame post reduction)**.** *If* $\Psi \rhd P \to P'$ *then* $\mathcal{F}_\Psi(P) \leq \mathcal{F}_\Psi(P')$.

Proof. By induction in the rules for concluding the reduction $\Psi \rhd P \to P'$ (Figure 4.2).

- Case [R-PAR] and [R-RES]: by the induction hypothesis.
- Case [R-COM]: Here the initial assertions are $\mathbf{1}$ (the unit assertion), since neither of the processes contain free assertions. Then after the reduction, we have $\mathbf{1} \leq \mathcal{F}_\Psi\big(P_1 \mid P_2[\tilde{x} := \widetilde{K}]\big)$, which is immediately seen to hold.

- Case [R-EVAL]: We have $\Psi \rhd P_1 \to P_2$, and from the premise that $\Psi \rhd P_1 \ggg P_1'$ and $\Psi \rhd P_1' \to P_2$. By induction hypothesis $\mathcal{F}_\Psi(P_1') \leq \mathcal{F}_\Psi(P_2)$, and by Lemma 20 we have that $\mathcal{F}_\Psi(P_1) =_\Psi \mathcal{F}_\Psi(P_1')$. Thus we conclude that $\mathcal{F}_\Psi(P_1) \leq \mathcal{F}_\Psi(P_2)$. □

The above lemmas can now be used to prove the subject reduction result, which states that well-typed processes remains well-typed after reduction. We shall break it into three parts and first show that well-typedness is preserved by structural congruence, the evaluation relation, and lastly the reduction relation:

Lemma 22 (Structural congruence)**.** *If* $\Gamma, \Psi \vdash P_1$ *and* $P_1 \equiv P_2$ *then* $\Gamma, \Psi \vdash P_2$.

Proof. By induction in the rules for concluding $P_1 \equiv P_2$ (Definition 30).

- The cases for rules for congruence are straightforward by induction hypothesis.
- The case for [S-ALPHA] is also straightforward, since we only change bound names, and none of these are found in Γ.
- The cases for the monoidal rules [S-IDENT], [S-ASSOC] and [S-COMM] simply require applying the rule [T-PAR] in different order, or applying it an extra time.
- Case [S-RES$_1$]: For the forward direction, we know $\Gamma, \Psi \vdash (\boldsymbol{\nu} x : T)\mathbf{0}$ by [T-RES], and $\Gamma, x : T, \Psi \vdash \mathbf{0}$ holds by the premise. By Lemma 16 (Weakening) therefore also $\Gamma, \Psi \vdash \mathbf{0}$. For the other direction, we weaken Γ first and then conclude by rule [T-RES].
- Case [S-RES$_2$]: By twice application of [T-RES].
- Case [S-SCOPE]: For the forward direction, we know that

  $$\Gamma, \Psi \vdash (\boldsymbol{\nu} x : T)P_1 \mid P_2$$

  and $x \# P_2$. This judgement must have been concluded by [T-PAR] with premises

  $$\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash (\boldsymbol{\nu} x : T)P_1$$
  $$\Gamma, \mathcal{F}_\Gamma(P_1), \Psi \otimes \mathcal{F}_\Psi(P_1) \vdash P_2$$

  The premise $\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash (\boldsymbol{\nu} x : T)P_1$ was concluded using [T-RES] with

  $$\Gamma, x : T, \Psi \otimes \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1$$

  as premise. Since $x \# P_2$, we can then also conclude

  $$\Gamma, x : T, \Psi \vdash P_2$$

by Lemma 16 (Weakening). By [T-PAR] we can then conclude

$$\Gamma, x : T, \Psi \vdash P_1 \mid P_2$$

and then by [T-RES] we conclude

$$\Gamma, \Psi \vdash (\nu x : T)(P_1 \mid P_2)$$

For the other direction, apply Lemma 17 to conclude $\Gamma, \Psi \vdash P_2$ from $\Gamma, x : T, \Psi \vdash P_2$, and then [T-RES] for P_1 and conclude by [T-PAR]. □

Lemma 23 (Subject evaluation). *If* $\Gamma, \Psi \vdash P \wedge \Psi \rhd P \underline{\ggg} P'$ *then* $\Gamma, \Psi \vdash P'$.

Proof. By induction in the rules for concluding $\Psi \rhd P \underline{\ggg} P'$ (Figure 4.1).

- Case [E-RES]: By [T-RES] and the induction hypothesis.
- Case [E-STRUCT]: By Lemma 22.
- Case [E-REP]: We know that $\Gamma, \Psi \vdash\ !P$, which was concluded by [T-REP]. From the premise we get $\Gamma, \Psi \vdash P$, and by well-formedness requirement (Definition 29) we know that P cannot contain free assertions. Thus we can conclude $\Gamma, \Psi \vdash P \mid\ !P$ by [T-PAR].
- Case [E-CASE]: Straightforward from the premise of [T-CASE].
- Case [E-PAR]: We know that $\Psi \rhd P_1 \mid P_2 \underline{\ggg} P_1' \mid P_2$. From the premise and side condition, we have that

$$\begin{gathered} \Psi \otimes \mathcal{F}_\Psi(P_2) \rhd P_1 \underline{\ggg} P_1' \\ \mathcal{F}_\nu(P_2) \# \Psi, \mathcal{F}_\nu(P_1), P_1 \end{gathered}$$

  We also know that $\Gamma, \Psi \vdash P_1 \mid P_2$, which was concluded by [T-PAR]. From the premise and side condition, we have that

$$\begin{gathered} \Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1 \\ \Gamma, \mathcal{F}_\Gamma(P_1), \Psi \otimes \mathcal{F}_\Psi(P_1) \vdash P_2 \\ \mathcal{F}_\nu(P_1) \# \Psi, \mathcal{F}_\nu(P_2), P_2 \\ \mathcal{F}_\nu(P_2) \# \Psi, \mathcal{F}_\nu(P_1), P_1 \end{gathered}$$

  By the induction hypothesis, the statement holds for P_1, i.e.:

$$\Gamma, \Psi \vdash P_1 \wedge \Psi \rhd P_1 \underline{\ggg} P_1' \implies \Gamma, \Psi \vdash P_1'$$

Combining the above with Lemma 16 (weakening) and Lemma 18 (assertion environment weakening), we can then conclude that

$$\begin{aligned} &\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1 \\ &\wedge \Psi \otimes \mathcal{F}_\Psi(P_2) \rhd P_1 \underline{\ggg} P_1' \\ \Longrightarrow\ &\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1' \end{aligned}$$

This satisfies the first premise of [T-PAR].

We must then show that $\Gamma, \mathcal{F}_\Gamma(P_1'), \Psi \otimes \mathcal{F}_\Psi(P_1') \vdash P_2$ also holds: By Lemma 20 we know that $\mathcal{F}_\Psi(P_1) =_\Psi \mathcal{F}_\Psi(P_1')$, since the evaluation cannot expose any new assertions. We therefore immediately have that $\Gamma, \mathcal{F}_\Gamma(P_1), \Psi \otimes \mathcal{F}_\Psi(P_1') \vdash P_2$. We now have three cases to consider:

1. If $\mathcal{F}_\Gamma(P_1) =_\Psi \mathcal{F}_\Gamma(P_1')$ then $\Gamma, \mathcal{F}_\Gamma(P_1'), \Psi \otimes \mathcal{F}_\Psi(P_1') \vdash P_2$ follows from the definition of $=_\Psi$ and associativity and commutativity of $\otimes$.
2. If $\mathcal{F}_\Gamma(P_1) \leq \mathcal{F}_\Gamma(P_1')$, then we use Lemma 16 (weakening) to conclude that $\Gamma, \mathcal{F}_\Gamma(P_1'), \Psi \otimes \mathcal{F}_\Psi(P_1') \vdash P_2$ holds.
3. Otherwise, if $\mathcal{F}_\Gamma(P_1) \not\leq \mathcal{F}_\Gamma(P_1')$, we instead use Lemma 17 (strengthening).

If new bound names are exposed in P_1', such that they would be collected by $\mathcal{F}_\Gamma(P_1')$ and $\mathcal{F}_\nu(P_1')$, we use α-conversion such that

$$\begin{aligned} &\mathcal{F}_\nu(P_1') \# \Psi, \mathcal{F}_\nu(P_2), P_2 \\ &\mathcal{F}_\nu(P_2) \# \Psi, \mathcal{F}_\nu(P_1'), P_1 \end{aligned}$$

both hold. As both premises and side conditions now are satisfied, we can conclude $\Gamma, \Psi \vdash P_1' \mid P_2$ by [T-PAR].

- Case [E-RUN]: We know that $\Psi \rhd \mathbf{run}\, M \underline{\ggg} P$ and $\Psi \Vdash M \Leftarrow P$, and we wish to conclude $\Gamma, \Psi \vdash P$. $\Gamma, \Psi \vdash \mathbf{run}\, M$ was concluded by [T-RUN] with the premise $\Gamma, \Psi \vdash M : T$ and side condition $T \curvearrowleft \Gamma'$ and $\Gamma' \leq \Gamma$. By requirement [T-HANDLE] we get that $\Gamma', \Psi \vdash P$. We can then conclude $\Gamma, \Psi \vdash P$ by Lemma 16 (Weakening). □

Theorem 5 (Subject reduction). *If* $\Gamma, \Psi \vdash P$ *and* $\Psi \rhd P \to P'$ *then* $\Gamma, \Psi \vdash P'$.

Proof. By induction in the rules for concluding $\Psi \rhd P \to P'$ (Figure 4.2).

- Case [R-EVAL]: By Lemma 23 (Subject evaluation) and the induction hypothesis.
- Case [R-RES]: By [T-RES] and the induction hypothesis.

- Case [R-PAR]: We know that $\Gamma, \Psi \vdash P_1 \mid P_2$, which was concluded by [T-PAR], and $\Psi \rhd P_1 \mid P_2 \to P_1' \mid P_2$ which was concluded by [R-PAR]. Our goal is to show that $\Gamma, \Psi \vdash P_1' \mid P_2$. From the premises and side condition of [T-PAR] we have

$$\begin{aligned}\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) &\vdash P_1 \\ \Gamma, \mathcal{F}_\Gamma(P_1), \Psi \otimes \mathcal{F}_\Psi(P_1) &\vdash P_2 \\ \mathcal{F}_\nu(P_1) \# \Psi, \mathcal{F}_\nu(P_2), P_2 \\ \mathcal{F}_\nu(P_2) \# \Psi, \mathcal{F}_\nu(P_1), P_1\end{aligned}$$

  and from the premises and side conditions of [R-PAR] we have:

$$\begin{aligned}\mathcal{F}_\nu(P_2) \# \Psi, \mathcal{F}_\nu(P_1), P_1 \\ \Psi \otimes \mathcal{F}_\Psi(P_2) \rhd P_1 \to P_1'\end{aligned}$$

  By the induction hypothesis, the statement holds for P_1, i.e.:

$$\Gamma, \Psi \vdash P_1 \wedge \Psi \rhd P_1 \to P_1' \implies \Gamma, \Psi \vdash P_1'$$

  By the above, and Lemma 16 (weakening) and Lemma 18 (assertion environment weakening) we can then conclude that

$$\begin{aligned}&\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1 \\ &\wedge \Psi \otimes \mathcal{F}_\Psi(P_2) \rhd P_1 \to P_1' \\ \implies &\Gamma, \mathcal{F}_\Gamma(P_2), \Psi \otimes \mathcal{F}_\Psi(P_2) \vdash P_1'\end{aligned}$$

  This satisfies the first premise of [T-PAR].

  We must then show that $\Gamma, \mathcal{F}_\Gamma(P_1'), \Psi \otimes \mathcal{F}_\Psi(P_1') \vdash P_2$ also holds: By Lemma 21, we have that $\mathcal{F}_\Psi(P_1) \leq \mathcal{F}_\Psi(P_1')$. From $\Gamma, \mathcal{F}_\Gamma(P_1), \Psi \otimes \mathcal{F}_\Psi(P_1) \vdash P_2$ and Lemma 18 we can then conclude that $\Gamma, \mathcal{F}_\Gamma(P_1), \Psi \otimes \mathcal{F}_\Psi(P_1') \vdash P_2$. We now have three cases to consider:

  1. If $\mathcal{F}_\Gamma(P_1) =_\Psi \mathcal{F}_\Gamma(P_1')$ then $\Gamma, \mathcal{F}_\Gamma(P_1'), \Psi \otimes \mathcal{F}_\Psi(P_1') \vdash P_2$ follows from the definition of $=_\Psi$ and associativity and commutativity of $\otimes$.
  2. If $\mathcal{F}_\Gamma(P_1) \leq \mathcal{F}_\Gamma(P_1')$, then we use Lemma 16 (weakening) to conclude that $\Gamma, \mathcal{F}_\Gamma(P_1'), \Psi \otimes \mathcal{F}_\Psi(P_1') \vdash P_2$ holds.
  3. Otherwise, if $\mathcal{F}_\Gamma(P_1) \not\leq \mathcal{F}_\Gamma(P_1')$, we instead use Lemma 17 (strengthening).

  If new bound names are exposed in P_1', such that they would be collected by $\mathcal{F}_\Gamma(P_1')$ and $\mathcal{F}_\nu(P_1')$, we use α-conversion such that

$$\begin{aligned}\mathcal{F}_\nu(P_1') \# \Psi, \mathcal{F}_\nu(P_2), P_2 \\ \mathcal{F}_\nu(P_2) \# \Psi, \mathcal{F}_\nu(P_1'), P_1\end{aligned}$$

both hold. As both premises and side conditions now are satisfied, we can conclude $\Gamma, \Psi \vdash P_1' \mid P_2$ by [T-PAR].

- Case [R-COM]: We know that

$$\Psi \rhd \overline{M_1} N[\tilde{x} := \tilde{K}].P_1 \mid \underline{M_1}(\lambda \tilde{x} : \tilde{T}) N.P_2 \to P_1 \mid P_2[\tilde{x} := \tilde{K}]$$

which was concluded by [R-COM], and $\Psi \Vdash M_1 \leftrightarrow M_2$ from the premise. We also know that

$$\Gamma, \Psi \vdash \overline{M_1} N[\tilde{x} := \tilde{K}].P_1 \mid \underline{M_1}(\lambda \tilde{x} : \tilde{T}) N.P_2$$

which must have been concluded by [T-PAR]. As there are no free assertions or $(\boldsymbol{\nu} x : T)$ in either of the processes, the side condition and extensions of Γ and Ψ in [T-PAR] simplify to just

$$[\text{T-PAR}] \frac{\Gamma, \Psi \vdash \overline{M_1} N[\tilde{x} := \tilde{K}].P_1 \quad \Gamma, \Psi \vdash \underline{M_2}(\lambda \tilde{y} : \tilde{T}) N.P_2}{\Gamma, \Psi \vdash \overline{M_1} N[\tilde{x} := \tilde{K}].P_1 \mid \underline{M_2}(\lambda \tilde{x} : \tilde{T}) N.P_2}$$

with [T-OUT] and [T-IN] used for the premises. From the premises of these rules we get the following:

$$\begin{aligned} \Gamma, \Psi &\vdash M_1 : T_1 \\ \Gamma, \Psi &\vdash N[\tilde{x} := \tilde{K}] : T_3 \\ \Gamma, \Psi &\vdash P_1 \\ \Gamma, \Psi &\vdash M_2 : T_2 \\ \Gamma, \tilde{x} : \tilde{T}, \Psi &\vdash N : T_4 \\ \Gamma, \tilde{x} : \tilde{T}, \Psi &\vdash P_2 \end{aligned}$$

where $M_1 \leftarrow T_3$ and $M_2 \leftarrow T_4$ from the side conditions.

Now by the requirement [T-EQUAL], since $\Psi \Vdash M_1 \leftrightarrow M_2$ then $T_1 = T_2$. By the assumptions on compatibility, either $T_3 \leq T_4$ or $T_4 \leq T_3$. Assume that the common supertype of T_3 and T_4 is T_3. By Lemma 19 (Substitution) we have that $T_3 = F(\tilde{x})[\tilde{x} := \tilde{K}]$ for some function F over types. By the compositionality requirement we therefore have that $\Gamma, \Psi \vdash \tilde{K} : \tilde{T}$. Thus $\Gamma, \Psi \vdash P_2[\tilde{x} := \tilde{K}]$ by Lemma 19 and Lemma 17 (Strengthening).

Since we now know that $\Gamma, \Psi \vdash P_1$ and $\Gamma, \Psi \vdash P_2[\tilde{x} := \tilde{K}]$, we can conclude

$$\begin{aligned} \Gamma, \mathcal{F}_\Gamma\left(P_2[\tilde{x} := \tilde{K}]\right), \Psi &\vdash P_1 \\ \Gamma, \mathcal{F}_\Gamma\left(P_1\right), \Psi &\vdash P_2[\tilde{x} := \tilde{K}] \end{aligned}$$

by Lemma 16 (Weakening), and then

$$\Gamma, \mathcal{F}_\Gamma\left(P_2[\tilde{x} := \widetilde{K}]\right), \Psi \otimes \mathcal{F}_\Psi\left(P_2[\tilde{x} := \widetilde{K}]\right) \vdash P_1$$
$$\Gamma, \mathcal{F}_\Gamma\left(P_1\right), \Psi \otimes \mathcal{F}_\Psi\left(P_1\right) \vdash P_2[\tilde{x} := \widetilde{K}]$$

by Lemma 18 (Assertion environment weakening). Lastly, we use α-conversion as described in the case for $P_1 \mid P_2$ above to ensure that the side condition holds, and thus we can conclude $\Gamma, \Psi \vdash P_1 \mid P_2[\tilde{x} := \widetilde{K}]$ as desired. □

We now have that the subject reduction result also holds for the τ-labelled transitions of the labelled semantics for HOΨ as given in [98]. This follows as a simple corollary:

Corollary 9. *If* $\Gamma, \Psi \vdash P$ *and* $\Psi \rhd P \xrightarrow{\tau} P'$ *then* $\Gamma, \Psi \vdash P'$ *where* $\xrightarrow{\tau}$ *is the* τ*-labelled transition relation given in [98].*

Proof. By Proposition 7 we have that $\Psi, \rhd P \xrightarrow{\tau} P' \implies \Psi \rhd P \to P'$. Then apply Theorem 5 to conclude $\Gamma, \Psi \vdash P'$. □

4.4.1 Safety in the generic type system

A type system normally guarantees two properties: A subject reduction property and a *safety* property, which must be implied by well-typedness. The notion of safety will depend on the particular instantiation of the type system, so, unlike with the result of subject reduction, we cannot prove a general result of safety for the generic type system. This will instead have to be shown individually for each instance. Such a result requires a definition of a *now-safe* predicate $\mathsf{NSafe}_\Gamma(P)$, which must ensure that P is safe to perform *at least one* reduction step (if it can reduce at all). Showing safety then amounts to showing that well-typedness implies now-safety:

$$\Gamma, \Psi \vdash P \implies \mathsf{NSafe}_\Gamma(P)$$

If this holds, then by subject reduction, $\mathsf{NSafe}_\Gamma(P')$ also holds for every reduct $P \to^* P'$ after any number of reductions, so now-safety is invariant under reduction of well-typed processes. Hence, well-typed processes are invariantly now-safe, so we say they are *safe*.

Although this notion of safety will depend on the instance, the generic type system does guarantee a basic notion of *channel safety*. This property ensures that channels are always used to transmit messages whose type is compatible with that of the channel. It follows as a consequence of the side conditions in the type rules [T-IN] and [T-OUT]. To see this, we use the definition of a simple now-safety predicate for channel-safety, given by the rules in Figure 4.4.

$$[\textsc{chns-nil}]\frac{}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(\mathbf{0})} \qquad [\textsc{chns-rep}]\frac{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(P)}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(!P)}$$

$$[\textsc{chns-case}]\frac{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(P_i)}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(\mathbf{case}\ \widetilde{\varphi}\ :\ \widetilde{P})} \qquad [\textsc{chns-ass}]\frac{}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(\langle\!|\Psi'|\!\rangle)}$$

$$[\textsc{chns-par}]\frac{\begin{array}{c}\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\mathcal{F}_\Gamma(P_2),\Psi\otimes\mathcal{F}_\Psi(P_2)}(P_1)\\ \mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\mathcal{F}_\Gamma(P_1),\Psi\otimes\mathcal{F}_\Psi(P_1)}(P_2)\end{array}}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(P_1 \mid P_2)}\ \begin{pmatrix}\mathcal{F}_\nu(P_1)\,\#\Psi, \mathcal{F}_\nu(P_2), P_2\\ \mathcal{F}_\nu(P_2)\,\#\Psi, \mathcal{F}_\nu(P_1), P_1\end{pmatrix}$$

$$[\textsc{chns-in}]\frac{\Gamma,\Psi\vdash M\ :\ T \quad \Gamma,\tilde{x}\ :\ \widetilde{T},\Psi\vdash N\ :\ T'}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(\underline{M}(\lambda\tilde{x}\ :\ \widetilde{T})N.P)}\ (T \nleftrightarrow T')$$

$$[\textsc{chns-out}]\frac{\Gamma,\Psi\vdash M\ :\ T \quad \Gamma,\Psi\vdash N\ :\ T'}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(\overline{M}N.P)}\ (T \nleftrightarrow T')$$

$$[\textsc{chns-run}]\frac{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(P)}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(\mathbf{run}\ M)}\ (\Psi \Vdash M \Leftarrow P)$$

$$[\textsc{chns-res}]\frac{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,x:T,\Psi}(P)}{\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}((\boldsymbol{\nu} x\ :\ T)P)}\ (x\#\Psi)$$

Figure 4.4: The $\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(P)$ predicate.

As can be seen, the rules closely mirror the type rules of Figure 4.3, except that we only examine the *prefixes* of input and output, without recursing into the continuations. The side conditions in [CHNS-IN] and [CHNS-OUT] ensure compatibility between the types of subject and object, matching those of the corresponding type rules.

For **run** M, we must also examine the process for which M is a handle, since it can be released immediately (i.e. *before* the next reduction step). Note that this imposes the restriction that a **run** M cannot be unguarded, if M is not a handle for a process. This is only a slight limitation, which could also be lifted by adding another rule to conclude $\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(\mathbf{run}\ M)$ whenever M is *not* a handle for any process; i.e. $\Psi \Vdash \not\exists P.\, M \Leftarrow P$ for any P. However, we have omitted this for simplicity.

Given this predicate, we can then show the following result:

Lemma 24 (Channel safety). $\Gamma, \Psi \vdash P \implies \mathsf{NSafe}^{ch}_{\Gamma,\Psi}(P)$.

This is shown by induction in the type rules (Figure 4.3). We omit the proof, since it should be clear from the close similarity between the type rules and the rules of the $\mathsf{NSafe}^{\mathsf{ch}}_{\Gamma,\Psi}(P)$-predicate that it indeed holds. By the result of subject reduction above (Theorem 5) it then follows that channel now-safety is preserved under reduction of well-typed processes. Thus well-typed processes are *channel-safe*, and this property is then also inherited by every instance of the generic type system.

4.5 Instances of the generic type system

We now show how our generic type system can be applied to provide sound type systems for higher-order process calculi. We first consider type systems for a version of the HOπ-calculus [109], and then a type system for the ρ-calculus [80] introduced by Meredith and Radestock.

4.5.1 The Higher-Order π-calculus

Parrow et al. [98] give several examples of HOΨ-instances with process mobility: for example, by including the set of processes $\mathcal{R}_\Psi$ in $\mathbb{T}$, a process P may appear as the object of an output. If for all $P \in \mathcal{R}_\Psi . P \Leftarrow P$ is entailed by all assertions, a language similar to Thomsen's Plain CHOCS [122] is obtained, and by further allowing both names and processes to appear as objects of an output, we get a simplified version of Sangiorgi's HOπ-calculus, similar to the one described in [96]. We set the parameters for $\mathbb{T}, \mathbb{C}$ and entailment thus:

$$\begin{aligned}
\mathbb{T} &\triangleq \mathcal{N} \cup \mathcal{R}_\Psi \\
\mathbb{C} &\triangleq \left\{ a \leftrightarrow b \mid a, b \in \mathcal{N} \right\} \cup \left\{ P \Leftarrow Q \mid P, Q \in \mathcal{R}_\Psi \right\} \cup \{\mathsf{T}\} \\
\Vdash &\triangleq \left\{ (\mathbf{1}, a \leftrightarrow a) \mid a \in \mathcal{N} \right\} \cup \{ (\mathbf{1}, P \Leftarrow P) \mid P \in \mathcal{R}_\Psi \} \cup \{ (\mathbf{1}, \mathsf{T}) \}
\end{aligned}$$

and (initially) with $\mathbb{A} \triangleq \{\varnothing\}$, $\otimes \triangleq \cup$ and $\mathbf{1} \triangleq \varnothing$. We also include the symbol $\top$ in $\mathbb{C}$ to represent a condition that is entailed by all assertions, and use that for every condition in a **case** $\widetilde{\varphi} : \widetilde{P}$ construct to obtain a representation of non-deterministic choice. This parameter setting obviously allows unwanted processes such as

$$\overline{a}P.\mathbf{0} \mid \underline{a}(\lambda x)x.\overline{x}b.\mathbf{0} \to \overline{P}b.\mathbf{0}$$

where the process P is substituted for the *subject* x in the output construct $\overline{x}b.\mathbf{0}$ after a reduction step. However, we can now use our generic type system to create an instantiation that will disallow such possibilities. We define the types of terms as:

$$T ::= (T_S, \Gamma) \qquad\qquad T_S ::= \mathrm{ch}(T) \mid \bullet$$

The intention is that higher-order types of the form $(\bullet, \Gamma)$ are the types of names that correspond to processes, whereas higher-order types of the form $(\mathrm{ch}(T), \Gamma)$ would be the types of names that could be used both as channels and as references to processes. In HOπ, we disallow the latter. The behaviour of channels and first-order variables is captured in the same manner as the simple sorting system for the π-calculus [83]. In particular, the compatibility predicate is defined by $\mathrm{ch}(T) \not\leftrightarrow T$.

Run-time errors can be expressed by means of a simple error predicate of the form $\mathsf{Wrong}_\Gamma(P)$, which is just the converse of a now-safe predicate; i.e. we can define $\mathsf{NSafe}_\Gamma(P) \triangleq \neg\mathsf{Wrong}_\Gamma(P)$. The most relevant rules in the definition of the error predicate are the following, which describe that only terms of simple channel types can be used as channels and only terms of higher order types can be used as arguments of a **run** M operator.

$$\frac{\Gamma, \Psi \vdash M : (\mathrm{ch}(T'), \Gamma)}{\mathsf{Wrong}_\Gamma(\underline{M}(\lambda x)x.P)} \qquad\qquad \frac{\Gamma, \Psi \vdash M : \mathrm{ch}(T)}{\mathsf{Wrong}_\Gamma(\mathbf{run}\ M)}$$

We now define the instance parameters as follows:

$$[\textsc{t-con}]\frac{}{\Gamma, \Psi \vdash \top} \qquad [\textsc{t-ass}]\frac{P : (\bullet, \Gamma) \in \Psi'}{\Gamma, \Psi \vdash (\!|\Psi'|\!)} \qquad [\textsc{Term}_1]\frac{\Gamma(x) = \mathrm{ch}(T)}{\Gamma, \Psi \vdash x : \mathrm{ch}(T)}$$

We can now show safety for the type system instance:

Theorem 6. *If* $\Gamma, \Psi \vdash P$ *then* $\neg\mathsf{Wrong}_\Gamma(P)$.

The proof is by induction in the rules of $\Gamma, \Psi \vdash P$. Details are given in [17].

4.5.2 A type system for termination

We now turn our attention to an instance of the generic type system that captures a liveness property. Demangeon et al. [36] present a type system for checking termination in variants of the HOπ-calculus: for any well-typed process P we have

$$[\text{IN}]\frac{\Gamma, X : (k-1) \vdash P : n \quad \Gamma(a) = \text{ch}^k(\diamond)}{\Gamma \vdash a(X).P : n} \qquad [\text{OUT}]\frac{\Gamma \vdash Q : m \quad \Gamma \vdash P : n \quad \Gamma(a) = \text{ch}^k(\diamond) \quad m < k}{\Gamma \vdash \overline{a}\langle Q\rangle.P : \max(k, n)}$$

$$[\text{VAR}]\frac{\Gamma(X) = n}{\Gamma \vdash X : n} \qquad [\text{NEW}]\frac{\Gamma, a : \text{ch}^k(\diamond) \vdash P : n}{\Gamma \vdash (\boldsymbol{\nu} a : T)P : n}$$

$$[\text{NIL}]\frac{}{\Gamma \vdash \mathbf{0} : 0} \qquad [\text{PAR}]\frac{\Gamma \vdash P : m \quad \Gamma \vdash Q : n}{\Gamma \vdash P \mid Q : \max(m, n)}$$

Figure 4.5: The type rules of Demangeon et al. [36] for HOpi_2

that $P \to^* P' \not\to$. The version of the HOπ-calculus, HOpi_2, is a higher-order process calculus in which only processes can be communicated and replication is absent. Its syntax is given by the formation rules

$$P ::= \mathbf{0} \mid a(X).P \mid \overline{a}\langle Q\rangle.P \mid P_1 \mid P_2 \mid (\boldsymbol{\nu} a : T)P \mid X$$

Type judgements in the type system for HOpi_2 are of the form $\Gamma \vdash P : n$, where n is a natural number called the *level* of P. Names a have types of the form $\text{ch}^k(\diamond)$, where $\diamond$ denotes the type of processes and k is a natural number, the level of a. The intention is that names with type $\text{ch}^k(\diamond)$ can only be used to carry processes whose level n is less than k.

The type rules, shown in Figure 4.5, ensure that the level of processes that are sent on any channel a will be strictly smaller than that of a. Because of the absence of replication, this will ensure that well-typed processes will always terminate.

It is straightforward to represent the HOpi_2 calculus as an instance of the Higher-Order Ψ-calculus, using a variant of the parameter setting described in section 4.5.1. In order to represent the type system, we introduce assertions of the form

$$\Psi ::= n \mid n^- \mid n^+$$

We use assertions to indicate in which way a channel is to be used; an input use can only be typed in the presence of an assertion n^- and output use must be used with an assertion n^+. We have that $n \otimes n^- = n^- \otimes n = n$; that $n \otimes n^+ = n^+ \otimes n = n$; and that $n_1 \otimes n_2 = \max(n_1, n_2)$. We distinguish explicitly between input uses ($\text{ch}^k_-(\diamond)$) and output uses ($\text{ch}^k_+(\diamond)$) of channels.

$$T \in \mathcal{T} ::= n \mid \text{ch}^k_-(\diamond) \mid \text{ch}^k_+(\diamond)$$

and we let $\text{ch}^n(\diamond) \not\hookleftarrow (n-1)$ and $\text{ch}^n(\diamond) \not\hookleftarrow k$ whenever $k < n$. Type judgements are of the form $\Gamma, m \vdash M : T$ for terms and $\Gamma, m \vdash P$ for processes. We represent the judgement $\Gamma \vdash P : n$ as $\Gamma, n \vdash P$. The type rules for channels are the following.

$$[\textsc{Ch-in}]\frac{\Gamma(a) = \mathrm{ch}^k_-(\diamond)}{\Gamma, n^- \vdash a : \mathrm{ch}^k_-(\diamond)} \qquad [\textsc{Ch-out}]\frac{\Gamma(a) = \mathrm{ch}^k_+(\diamond)}{\Gamma, n^+ \vdash a : \mathrm{ch}^k_+(\diamond)}$$

4.5.3 The ρ-calculus

The Reflective Higher-Order calculus of Meredith and Radestock [80] is less well-known than e.g. CHOCS and HOπ, so we recall it in some detail. Unlike other calculi, the ρ-calculus does not assume an infinite set of names: instead, names and processes are both built from the same syntax, so names are structured terms, rather than atomic entities. The syntax for both processes and names is given by the formation rules:

$$\begin{aligned} P &::= \mathbf{0} \mid P \mid P \mid x\langle\!|P|\!\rangle \mid x(y).P \mid \urcorner x \ulcorner \\ x, y &::= \ulcorner P \urcorner \end{aligned}$$

where the syntax for names is $\ulcorner P \urcorner$, pronounced *quote* P. Names can be passed around as in the π-calculus, as well as un-quoted, written $\urcorner x \ulcorner$ (pronounced *drop* x), and thus higher-order behaviour becomes an inherent property of the calculus, rather than just an extension on top of an already computationally complete language.

The parallel, and the input construct $x(y).P$, are similar to their π-calculus counterparts. The *lift* operation, $x\langle\!|P|\!\rangle$ is an output construct that quotes the process P, thereby creating the *name* $\ulcorner P \urcorner$, and sends it out on x; thus the calculus can generate new names at runtime without the need of a ν-operator. The converse of lift is the *drop* operation, $\urcorner x \ulcorner$: it is a request to run the process within a name, by removing the quotes around it. This is not performed by a reduction, but rather by a form of substitution

$$\urcorner x \ulcorner\{\ulcorner P \urcorner / x'\} = P \qquad \text{if } x \equiv_{\mathcal{N}} x'$$

where $\equiv_{\mathcal{N}}$ is the *name equivalence* relation, defined further down. Here, the entire *process* $\urcorner x \ulcorner$ is replaced with the process P found within the substituted name, similar to how process variables are replaced by processes in e.g. HOπ. Notably, this means that if x is a *free* name, then $\urcorner x \ulcorner$ will be a *deadlock*, since x can never be touched by a substitution at runtime. Otherwise, substitution is the standard, capture-avoiding substitution of names for names, and note in particular that substitution does *not* recur into processes under quotes; i.e. $\ulcorner P \urcorner\{x/y\} = \ulcorner P \urcorner$ if $y \not\equiv_{\mathcal{N}} \ulcorner P \urcorner$ regardless of whether the name y exists somewhere within $\ulcorner P \urcorner$.

The reduction semantics is given by the standard rules for parallel composition and structural congruence (as in e.g. the π-calculus) plus a rule for communication:

$$[\rho\text{-par}]\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2} \qquad [\rho\text{-struct}]\frac{P_1 \equiv P_1' \quad P_1' \to P_2' \quad P_2' \equiv P_2}{P_1 \to P_2}$$

$$[\rho\text{-com}]\frac{x_1 \equiv_{\mathcal{N}} x_2}{x_1(y).P_1 \mid x_2\langle\!|P_2|\!\rangle \to P_1\{\ulcorner P_2 \urcorner / y\}}$$

One subtlety of this calculus concerns the notion of structural congruence, $\equiv$. It is the usual least congruence on processes, containing α-equivalence, $\equiv_\alpha$, and the abelian monoid rules for parallel composition with $\mathbf{0}$ as the unit element. However, with *structured* terms as names, $\equiv_\alpha$ in turn requires a notion of *name equivalence*, written $\equiv_{\mathcal{N}}$, that is also used for comparing subjects in the [ρ-COM] rule above. It is defined as the smallest equivalence relation on *quoted* processes, closed forward under the rules:

$$[\rho\text{-NAMEEQ}_1]\frac{P \equiv Q}{\ulcorner P \urcorner \equiv_{\mathcal{N}} \ulcorner Q \urcorner} \qquad [\rho\text{-NAMEEQ}_2]\frac{}{\ulcorner\urcorner x \ulcorner\urcorner \equiv_{\mathcal{N}} x}$$

This yields a mutual recursion between name equivalence, structural congruence and α-equivalence, albeit one that always terminates as proved in [80], because both the sets of names and processes are well-founded; their smallest elements being $\mathbf{0}$ (the inactive process) and $\ulcorner \mathbf{0} \urcorner$ respectively.

4.5.3.1 Instantiation as a Ψ-calculus

The ρ-calculus is interesting in the present setting, because it cannot be encoded in the π-calculus in a way that satisfies a number of generally accepted criteria of encodability, similar to those of [48].[2] This has been established by one of the authors in [76] (see Chapter 3).

The key reason for this impossibility lies in the ability of the ρ-calculus to generate new, *free*, and hence observable, names at runtime, whilst this is not possible in the π-calculus; and, dually, its use of name equivalence, which will equate more names than strict syntactic equality. However, the ρ-calculus *can* be represented in the HOΨ-framework as follows. We define

$$\begin{aligned}\mathbb{T} &\triangleq \mathcal{N} \cup \{\, \ulcorner P \urcorner \mid P \in \mathcal{R}_\Psi \,\} \cup \{\, \langle\!| \ulcorner P \urcorner |\!\rangle \mid P \in \mathcal{R}_\Psi \,\} \\ \mathbb{C} &\triangleq \left\{ M \mathrel{\dot{\leftrightarrow}} N \mid M, N \in \mathbb{T} \right\} \cup \{\, P_1 \equiv P_2 \mid P_1, P_2 \in \mathcal{R}_\Psi \,\} \\ &\quad \cup \{\, M \Leftarrow P \mid M \in \mathbb{T} \wedge P \in \mathcal{R}_\Psi \,\}\end{aligned}$$

and (initially) with $\mathbb{A} \triangleq \{\varnothing\}$, $\otimes \triangleq \cup$ and $\mathbf{1} \triangleq \varnothing$ as before. Note the two different kinds of terms: we use terms of the form $\ulcorner P \urcorner$ to represent a *statically* quoted name in the ρ-calculus, which can never be dropped and never substituted into. Conversely, we use $\langle\!| \ulcorner P \urcorner |\!\rangle$ for the equivalent of the object of a $x\langle\!|P|\!\rangle$, which in the ρ-calculus is a *process* that therefore *can* be substituted into, and which later may be dropped. Furthermore, we shall assume that all *bound names* are implemented as distinct atomic names $x \in \mathcal{N}$; this is a trivial conversion, since their structure has no semantic meaning in the ρ-calculus. The encoding is then given by the translation:

[2] The criteria are: Preserving the degree of parallelism, correspondence of substitutions (the ability to move a substitution into/out of the translation, up to a notion of behavioural equivalence), observational correspondence (the same names must be observable in the source and translated target terms), operational correspondence, and divergence reflection (if a translated target term diverges, then the source term must diverge).

$$\begin{aligned}
[\![\mathbf{0}]\!] &= \mathbf{0} & [\![\ulcorner x \urcorner]\!] &= \mathbf{run}\ x \\
[\![P_1 \mid P_2]\!] &= [\![P_1]\!] \mid [\![P_2]\!] & [\![\ulcorner \ulcorner P \urcorner \urcorner]\!] &= \mathbf{0} \\
[\![n\langle\!|P|\!\rangle]\!] &= \overline{[\![n]\!]}\langle\!|\ulcorner [\![P]\!] \urcorner|\!\rangle.\mathbf{0} & [\![\ulcorner P \urcorner]\!] &= \ulcorner \mathcal{N}[\![P]\!] \urcorner \\
[\![n(x).P]\!] &= \underline{[\![n]\!]}(\lambda x)\langle\!|x|\!\rangle.[\![P]\!] & [\![x]\!] &= x
\end{aligned}$$

where $\mathcal{N}[\![P]\!]$ is identical to $[\![P]\!]$ *except* that $\mathcal{N}[\![\ulcorner \ulcorner P \urcorner \urcorner]\!] = \mathbf{run}\ \ulcorner \mathcal{N}[\![P]\!] \urcorner$.

Note the two translations of drop for processes: the process $\ulcorner \ulcorner P \urcorner \urcorner$ has no reduction in the ρ-calculus and is therefore behaviourally equivalent to $\mathbf{0}$; but its counterpart $\mathbf{run}\ \ulcorner P \urcorner$ *might* have a reduction, since $\mathbf{run}\ M$ is not evaluated eagerly in the HOΨ-calculus. For the purpose of preserving operational correspondence, we therefore translate the drop of a *free name* $\ulcorner P \urcorner$ as $\mathbf{0}$, and the drop of an atomic name x as $\mathbf{run}\ x$, since atomic names are bound by construction. However, we cannot do this within *names*, since name equivalence is determined by the structure, rather than the behaviour of the process within quotes. Thus we use the second level translation $\mathcal{N}[\![P]\!]$ for statically quoted names, since these can never be dropped.

Lastly, we shall define entailment such that it contains the rule $\Psi \Vdash \ulcorner P \urcorner \Leftarrow P$, making every term $\ulcorner P \urcorner$ a handle for the process P within, to mirror the duality of names and processes in the ρ-calculus. We furthermore include the following rules for entailment of *channel equivalence* $\dot{\leftrightarrow}$, mirroring the rules [ρ-NameEq$_1$] and [ρ-NameEq$_2$] for concluding name equivalence:

$$[\textsc{ChanEq}_1]\frac{\Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash \ulcorner P_1 \urcorner \dot{\leftrightarrow} \ulcorner P_2 \urcorner} \qquad [\textsc{ChanEq}_2]\frac{}{\Psi \Vdash \ulcorner \mathbf{run}\ M \urcorner \dot{\leftrightarrow} M}$$

including the symmetric and transitive closure of $\dot{\leftrightarrow}$. We then let the entailment relation for conditions of *structural congruence* $\equiv$ be defined such that $\equiv$ contains α-equivalence; that $(\mathcal{P}/_{\equiv}, \mid, \mathbf{0})$ is an abelian monoid; and containing the four congruence rules derived from the above translation:

$$[\textsc{Par}]\frac{\Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash P_1 \mid R \equiv P_2 \mid R}$$

$$[\textsc{Run}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2}{\Psi \Vdash \mathbf{run}\ M_1 \equiv \mathbf{run}\ M_2}$$

$$[\textsc{Out}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2 \qquad \Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash \overline{M_1}\langle\!|\ulcorner P_1 \urcorner|\!\rangle \equiv \overline{M_2}\langle\!|\ulcorner P_2 \urcorner|\!\rangle}$$

$$[\textsc{In}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2 \qquad \Psi \Vdash P_1[x_1 := z] \equiv P_2[x_2 := z]}{\Psi \Vdash \underline{M_1}(\lambda x_1)\langle\!|x_1|\!\rangle.P_1 \equiv \underline{M_2}(\lambda x_2)\langle\!|x_2|\!\rangle.P_2}\ (z \# P_1, P_2)$$

Given that this instantiation is a little more involved (i.e. requires a little more encoding) than the HOπ-instantiation, we shall now formally prove that it indeed

preserves the semantics of the ρ-calculus. In other words, we shall prove (strict) operational correspondence between the two semantics, up to a reasonable notion of behavioural equivalence $\simeq$. We need not choose any particular notion, but we require that it contains at least structural congruence and the axiom $\mathbf{run}\ulcorner P\urcorner \simeq P$. This is reasonable to include, since if P cannot perform any reductions, then unfolding $\mathbf{run}\ulcorner P\urcorner$ cannot enable any reductions either; and conversely, if $P \to P'$ then $\mathbf{run}\ulcorner P\urcorner \to P'$ after unfolding by the evaluation relation.

As a first step, we need a lemma relating name equivalence in the ρ-calculus with the entailment of channel equivalence in our HOΨ-instance:

Lemma 25. $x_1 \equiv_{\mathcal{N}} x_2 \iff \varnothing \Vdash [\![x_1]\!] \leftrightarrow [\![x_2]\!]$

Proof sketch. By induction in the rules of name equivalence and structural congruence, and the entailment of channel equivalence. This is straightforward, as the latter is defined as a one-to-one match of the rules of name equivalence and structural congruence. □

Secondly, we shall need a lemma relating substitution in the two calculi:

Lemma 26. *Let $\sigma_s \triangleq \{\ulcorner Q\urcorner/x\}$ be a substitution in the ρ-calculus (source language), and let $\sigma_t \triangleq [x := [\![\ulcorner Q\urcorner]\!]]$ denote the corresponding substitution in the* HOΨ*-calculus (target language). Then $[\![P\sigma_s]\!] \simeq [\![P]\!]\sigma_t$ and $\varnothing \Vdash [\![n\sigma_s]\!] \leftrightarrow \sigma_t([\![n]\!])$.*

Proof. By induction in the clauses of the translation function:

The cases for $\mathbf{0}$, $P_1 \mid P_2$ and $x\langle\!|P|\!\rangle$ are simple: For the left-hand side apply the substitution first and then perform the translation, and conversely for the right-hand side. Then apply the induction hypothesis (where necessary) to conclude.

The other cases are more interesting:

- Case input: We show that $[\![(n(y).P)\sigma_s]\!] = [\![n(y).P]\!]\sigma_t$. For the left-hand side we apply the substitution first, and then the translation:

  $$\begin{aligned} & [\![(n(y).P)\sigma_s]\!] \\ = {} & [\![\sigma_s(n)(y).(P)\sigma_s]\!] \\ = {} & \underline{[\![\sigma_s(n)]\!]}(\lambda y)\langle\!|y|\!\rangle.[\![P\sigma_s]\!] \end{aligned}$$

  Assuming all bound names are distinct, we omit the extra step of α-converting y.

  For the right-hand side we perform the translation first and then the substitution and obtain:

  $$\begin{aligned} & [\![n(y).P]\!]\sigma_t \\ = {} & ([\![n]\!](\lambda y)\langle\!|y|\!\rangle.[\![P]\!])\sigma_t \\ = {} & \underline{\sigma_t([\![n]\!])}(\lambda y)\langle\!|y|\!\rangle.([\![P]\!])\sigma_t \end{aligned}$$

Then apply the induction hypothesis to conclude that $\varnothing \Vdash [\![\sigma_s(n)]\!] \leftrightarrow \sigma_t([\![n]\!])$ and $[\![P\sigma_s]\!] \simeq [\![P]\!]\sigma_t$.

- Case bound drop: We show that $[\![\ulcorner x \urcorner \sigma_s]\!] \simeq [\![\ulcorner x \urcorner]\!]\sigma_t$. Note firstly that of $x \notin \mathrm{dom}(\sigma_s)$, then we trivially obtain $[\![\ulcorner x \urcorner]\!] = \mathbf{run}\ x$ in both cases, so we shall focus on the other case. For the left-hand side we apply the substitution first, and then the translation:

$$[\![\ulcorner x \urcorner \{\ulcorner Q \urcorner / x\}]\!] = [\![Q]\!]$$

  For the right-hand side we perform the translation first and then the substitution and obtain:

$$\begin{aligned} & [\![\ulcorner x \urcorner]\!][x := [\![\ulcorner Q \urcorner]\!]] \\ = & \mathbf{run}\ x[x := \ulcorner [\![Q]\!] \urcorner] \\ = & \mathbf{run}\ \ulcorner [\![Q]\!] \urcorner \end{aligned}$$

  and by our requirement to the behavioural equivalence relation, we precisely get that $Q \simeq \mathbf{run}\ \ulcorner Q \urcorner$.

- Case free drop: We show that $[\![\ulcorner\!\ulcorner P \urcorner\!\urcorner \sigma_s]\!] \simeq [\![\ulcorner\!\ulcorner P \urcorner\!\urcorner]\!]\sigma_t$. Note that $\ulcorner P \urcorner$ here is a *free* name, so by construction *no* atomic names can appear within it. Thus, in either case, the substitution will have no effect; and by applying our translation function, we then in both cases get that $[\![\ulcorner\!\ulcorner P \urcorner\!\urcorner]\!] = \mathbf{0}$.

- Case quote: We show that $\varnothing \Vdash [\![\sigma_s(\ulcorner P \urcorner)]\!] \leftrightarrow \sigma_t([\![\ulcorner P \urcorner]\!])$. Like above, $\ulcorner P \urcorner$ is a free name which by construction cannot contain any atomic names, so in either case the substitution has no effect. By applying the translation, we then trivially obtain that $\varnothing \Vdash \ulcorner \mathcal{N}[\![P]\!] \urcorner \leftrightarrow \ulcorner \mathcal{N}[\![P]\!] \urcorner$.

- Case atomic: We show that $\varnothing \Vdash [\![\sigma_s(x)]\!] \leftrightarrow \sigma_t([\![x]\!])$. For the left-hand side we apply the substitution first, and then the translation and obtain:

$$\begin{aligned} & [\![x\{\ulcorner Q \urcorner / x\}]\!] \\ = & [\![\ulcorner Q \urcorner]\!] \\ = & \ulcorner [\![Q]\!] \urcorner \end{aligned}$$

  For the right-hand side we perform the translation first and then the substitution and obtain:

$$\begin{aligned} & [\![x]\!][x := [\![\ulcorner Q \urcorner]\!]] \\ = & x[x := \ulcorner [\![Q]\!] \urcorner] \\ = & \ulcorner [\![Q]\!] \urcorner \end{aligned}$$

  and we then trivially obtain that $\varnothing \Vdash \ulcorner [\![Q]\!] \urcorner \leftrightarrow \ulcorner [\![Q]\!] \urcorner$.

This concludes the proof. □

We can now state our desired result. The translation is sound and complete w.r.t. operational correspondence up to a reasonable notion of behavioural equivalence $\simeq$:

Theorem 7 (Operational correspondence)**.** *Let $\simeq$ be a notion of behavioural equivalence for processes of the* HOΨ*-instance of the ρ-calculus, that includes at least structural congruence and the axiom* **run** $\ulcorner[\![P]\!]\urcorner \simeq [\![P]\!]$*. Then*

$$P \to P' \iff [\![P]\!] \to\!\simeq [\![P']\!]$$

Proof. For the forward direction (completeness), we use induction in the reduction rules of the ρ-calculus semantics. The case for [ρ-PAR] and [ρ-STRUCT] are given by a straightforward application of the induction hypothesis. Thus, the only interesting case is for the communication rule, [ρ-COM]: We have that

$$n_1\langle\!|P_1|\!\rangle \mid n_2(y).P_2 \to P_2\{\ulcorner P_1 \urcorner/y\}$$

with the premise that $n_1 \equiv_{\mathcal{N}} n_2$. We perform the translation on both sides and obtain

$$\varnothing \rhd \overline{[\![n_1]\!]}\langle\!|\ulcorner[\![P_1]\!]\urcorner|\!\rangle.\mathbf{0} \mid \underline{[\![n_2]\!]}(\lambda y)\langle\!|y|\!\rangle.[\![P_2]\!] \to \mathbf{0} \mid [\![P_2]\!][y := \ulcorner[\![P_1]\!]\urcorner]$$

This can be concluded by the HOΨ-calculus reduction rule [R-COM]. The premise requires that $\varnothing \Vdash [\![n_1]\!] \leftrightarrow [\![n_2]\!]$, which holds, since by Lemma 25 we have that

$$n_1 \equiv_{\mathcal{N}} n_2 \implies \varnothing \Vdash [\![n_1]\!] \leftrightarrow [\![n_2]\!]$$

Then by Lemma 26, we have that

$$[\![P_2]\!][y := \ulcorner[\![P_1]\!]\urcorner] \simeq [\![P_2\{\ulcorner P_1 \urcorner/y\}]\!]$$

and by our requirements to the behavioural equivalence relation, that

$$\mathbf{0} \mid [\![P_2\{\ulcorner P_1 \urcorner/y\}]\!] \simeq [\![P_2\{\ulcorner P_1 \urcorner/y\}]\!]$$

which is precisely the translation of our reduct. Thus completeness is proved.

For the other direction (soundness), we proceed similarly by examining the reduction rules of the HOΨ-calculus semantics. Note that as restriction $(\boldsymbol{\nu} x)$ does not appear in our translation, we can disregard the [R-RES] rule. The case for [R-PAR] and [R-EVAL] are again given by a straightforward application of the induction hypothesis. Thus again, the only interesting case is the communication rule, [R-COM]: We have that

$$\varnothing \rhd \overline{[\![n_1]\!]}\langle\!|\ulcorner[\![P_1]\!]\urcorner|\!\rangle.\mathbf{0} \mid \underline{[\![n_2]\!]}(\lambda y)\langle\!|y|\!\rangle.[\![P_2]\!] \to \mathbf{0} \mid [\![P_2]\!][y := \ulcorner[\![P_1]\!]\urcorner]$$

where we know from the premise that $\varnothing \Vdash [\![n_1]\!] \leftrightarrow [\![n_2]\!]$. Let P be the redex, and T' be the reduct. Just like above, we note that

$$\mathbf{0} \mid [\![P_2]\!][y := \ulcorner[\![P_1]\!]\urcorner] \simeq [\![P_2]\!][y := \ulcorner[\![P_1]\!]\urcorner] \simeq [\![P_2\{\ulcorner P_1 \urcorner/y\}]\!]$$

where the last rewrite is concluded by Lemma 26. Let this form be our $[\![P']\!]$; thus $P' = P_2\{\ulcorner P_1 \urcorner/y\}$. Then take the following ρ-calculus reduction:

$$n_1\langle\!|P_1|\!\rangle \mid n_2(y).P_2 \to P_2\{\ulcorner P_1 \urcorner/y\}$$

The redex correspond exactly to the HOΨ redex, so let it be our P. This reduction can be concluded by the [ρ-COM] rule. The premise requires that $n_1 \equiv_{\mathcal{N}} n_2$, but by Lemma 25 we have that

$$\varnothing \Vdash [\![n_1]\!] \leftrightarrow [\![n_2]\!] \implies n_1 \equiv_{\mathcal{N}} n_2$$

so the premise holds. Thus soundness is proved. □

4.5.3.2 A type system for reflection

Other higher-order calculi such as CHOCS and HOπ can be encoded in the π-calculus and may thus be typable through translation, but as we noted above there cannot be such an encoding of the ρ-calculus into the π-calculus. Thus, we cannot hope to create a type system for the ρ-calculus by adapting an existing first-order type system. In fact, we are not aware of any general type system for the ρ-calculus, so we shall now create one by instantiating our generic type system.[3] We let types for names be of the form

$$T \in \mathcal{T} ::= \langle T, \Gamma\rangle \mid \langle B, \Gamma\rangle$$

where B is a base type, and Γ is a type environment representing the possibility of executing the process within the name. Furthermore we shall use assertions as type environments for processes as we previously did with HOπ, so we update the definition accordingly.

$$\mathbb{A} \triangleq 2^{\begin{array}{c}\{\ulcorner P\urcorner : T \mid P \in \mathcal{R}_\Psi \wedge T \in \mathcal{T}\} \\ \cup \{\langle\!|\ulcorner P\urcorner|\!\rangle : T \mid P \in \mathcal{R}_\Psi \wedge T \in \mathcal{T}\}\end{array}}$$

with assertion unit and composition as $\mathbf{1} \triangleq \varnothing$ and $\otimes \triangleq \cup$ respectively. Note that by construction $\forall x \in \mathcal{N}.x\#\ulcorner P\urcorner$, so substitution can only occur in terms of the form $\langle\!|\ulcorner P\urcorner|\!\rangle : T$. We then append an assertion to the encoding of input and output:

$$[\![\ulcorner R\urcorner\langle\!|P|\!\rangle]\!] \triangleq \overline{\ulcorner \mathcal{N}[\![R]\!]\urcorner}\langle\!|\ulcorner [\![P]\!]\urcorner|\!\rangle.\mathbf{0} \mid (\!|\{\ulcorner \mathcal{N}[\![R]\!]\urcorner : T, \langle\!|\ulcorner [\![P]\!]\urcorner|\!\rangle : T'\}|\!)$$
$$[\![\ulcorner R\urcorner(x).P]\!] \triangleq \underline{\ulcorner \mathcal{N}[\![R]\!]\urcorner}(\lambda x)\langle\!|x|\!\rangle.[\![P]\!] \mid (\!|\{\ulcorner \mathcal{N}[\![R]\!]\urcorner : T\}|\!)$$

[3] A recent paper, [76] (see Chapter 3), describes a form of type system for checking channel safety in the ρ-calculus. However, it has certain limitations that only make it suitable for checking encoded π-calculus processes, so we would not describe it as a *general* type system for the ρ-calculus. We discuss this further at the end of this section.

Lastly, we also need to take the type information into account when concluding channel equivalence, to ensure that two terms with initially dissimilar types cannot become channel equivalent after a substitution. Thus we redefine the entailment rule [ChanEq$_1$] as follows:

$$[\textsc{ChanEq}_1]\frac{\Gamma, \Psi \Vdash P_1 \equiv P_2 \qquad \Gamma, \Psi \vdash \ulcorner P_1 \urcorner : T \iff \Gamma, \Psi \vdash \ulcorner P_2 \urcorner : T}{\Gamma, \Psi \Vdash \ulcorner P_1 \urcorner \leftrightarrow \ulcorner P_2 \urcorner}$$

Now we can instantiate the generic type system by defining the instance parameters:

$$[\textsc{Term-1}]\frac{\ulcorner P \urcorner : \langle T, \Gamma' \rangle \in \Psi \qquad \Gamma', \Psi \vdash P}{\Gamma, \Psi \vdash \ulcorner P \urcorner : \langle T, \Gamma' \rangle}$$

$$[\textsc{Term-2}]\frac{(\!|\ulcorner P \urcorner|\!) : \langle T, \Gamma' \rangle \in \Psi \qquad \Gamma', \Psi \vdash P}{\Gamma, \Psi \vdash (\!|\ulcorner P \urcorner|\!) : \langle T, \Gamma' \rangle}$$

The rules [Term-1] and [Term-2] tell us that the process found within a term must be well-typed w.r.t. the type environment in the second component of its type, and that the process-type pair must be represented in the assertion. The other instance rules ensure that names, that are quoted processes according to an assertion Ψ, correspond to processes that are typable within the same environment (this is expressed by [T-Ass]) and can carry processes. These are as follows:

$$[\text{T-Ass}]\frac{P : T \in \Psi' \implies T \curvearrowleft \Gamma}{\Gamma, \Psi \vdash (\!|\Psi'|\!)} \qquad [\text{T-End}]\frac{}{\langle T, \Gamma \rangle \curvearrowleft \Gamma}$$

$$[\text{T-Chan}]\frac{}{\langle T, \Gamma \rangle \nleftarrow T} \qquad [\text{T-Term-3}]\frac{\Gamma(x) = T}{\Gamma, \Psi \vdash x : T}$$

Example 13. Consider the ρ-calculus process:

$$P \triangleq x(\!|Q|\!) \mid x(y).(y(\!|\mathbf{0}|\!) \mid \urcorner y \ulcorner)$$

for some process Q and some free name $x \triangleq \ulcorner X \urcorner$. A single reduction step looks as follows:

$$x(\!|Q|\!) \mid x(y).(y(\!|\mathbf{0}|\!) \mid \urcorner y \ulcorner) \to \ulcorner Q \urcorner(\!|\mathbf{0}|\!) \mid Q$$

so in this simple example, we have both created a new name $\ulcorner Q \urcorner$ and released a process, Q. The HOπ-encoding of P yields the following:

$$[\![P]\!] = \overline{\ulcorner \mathcal{N}[\![R]\!] \urcorner}(\!|\ulcorner [\![Q]\!] \urcorner|\!).\mathbf{0} \mid \underline{\ulcorner \mathcal{N}[\![R]\!] \urcorner}(\lambda y)(\!|y|\!).\left(\overline{y}(\!|\ulcorner \mathbf{0} \urcorner|\!).\mathbf{0} \mid \mathbf{run}\ y\right)$$

We wish to allow both usages of the newly created name $\ulcorner Q \urcorner$, so we append the following two assertions to the input- and output processes:

$$(\!|\{\!| \ulcorner \mathcal{N}[\![R]\!] \urcorner : \langle T, \varnothing \rangle |\!\}|\!) \qquad \text{and} \qquad (\!|\{\!| (\!|\ulcorner [\![Q]\!] \urcorner|\!) : \langle T, \Gamma \rangle |\!\}|\!)$$

where Γ is a type environment that makes Q well-typed. The type T must describe the usage of the newly created name $\ulcorner Q \urcorner$, when it is used *as a name*. Here, it is used to send the process $\mathbf{0}$; hence we can choose T to be a higher-order type with both the name component and the higher-order component being empty, i.e. $T = \langle \varnothing, \varnothing \rangle$, since the process $\mathbf{0}$ contains no names and has no behaviour. ■

The type system works, but it does have certain limitations: Since we include [Chaneq$_1$] in order to properly simulate the ρ-calculus, all names, that eventually become equivalent during reduction, *must* have the same type. This amounts to requiring that the programmer must know in advance *all* the names that will be generated by the program during execution. This is not entirely satisfactory, since the ρ-calculus can be used to create processes that may generate infinitely many new names, e.g. a 'name generator' process. Such a process is for example needed in the encoding of the π-calculus into the ρ-calculus given in [76] (see Chapter 3). However, this encoding would not be typable using the present type system.

The aforementioned paper also offers a different approach to creating a type system for the ρ-calculus, by assuming a default type for names *not* found in the type environment Γ. This allows it to be used to type encoded π-calculus processes. However, this assumption, in turn, leads to other limitations, since it necessitates a 'manual' proof that the typed process will never generate new names of the non-default type at runtime (unless they are listed in Γ). Hence, that type system is not really usable in the general case of typing arbitrary ρ-calculus processes. We have yet to find a type system for the ρ-calculus that does not impose such limiting restrictions on the runtime-generated names.

4.5.4 A type system for non-interference

As our final example, we present a type system for security properties of mobile code in the version of the higher-order π-calculus that we studied in Section 4.5.1. This system makes use of the model of non-interference proposed by Goguen and Meseguer [47]. Our type system will classify processes according to their security levels, which can be high (hi) or low (lo), and the system will guarantee that actions of high-security users do not affect observations of low-security users in that a process can be high-security or low-security.

In the following we assume that the set of messages is that of the set of names but this is not essential to the underlying ideas of the type system. Security levels are described by means of extending the set of assertions such that assertions are now defined thus:

$$\Psi \in \mathbb{A} ::= x \Leftarrow P \mid \mathsf{hi} \mid \mathsf{lo}$$

We define an ordering of the levels by $\mathsf{lo} \leq \mathsf{hi}$. Assertion composition now corresponds to finding a lower bound of security levels. Thus we define

$$\mathsf{lo} \otimes \mathsf{lo} = \mathsf{lo} \qquad \mathsf{lo} \otimes \mathsf{hi} = \mathsf{lo} = \mathsf{hi} \otimes \mathsf{lo} \qquad \mathsf{hi} \otimes \mathsf{hi} = \mathsf{hi}$$

Simple types T_S are defined as

$$T_S ::= B^{\Psi} \mid \mathrm{drop}(\Gamma)^{\Psi} \mid \mathrm{ch}^{\Psi}(T)$$

where B ranges over an unspecified set of base types and $\mathrm{drop}(\Gamma)$ is the type of process handles. The set of types is then defined thus:

$$T ::= T_S \mid (T_S, \Gamma)$$

Next, we define the *level* of a type T, written $\mathsf{level}(T)$, as follows:

$$\begin{aligned} \mathsf{level}(B^{\Psi}) &= \Psi \\ \mathsf{level}(\mathrm{drop}(\Gamma)^{\Psi}) &= \Psi \\ \mathsf{level}(\mathrm{ch}^{\Psi}(T)) &= \Psi \\ \mathsf{level}(T_S, \Gamma) &= \mathsf{level}(T_S) \end{aligned}$$

In this type system, the idea is that $\Gamma, \Psi \vdash M : T$ holds if M has type T at security level Ψ. Likewise, we have that $\Gamma, \Psi \vdash P$ if all communication in P communicates data at security level Ψ.

Channels of type $\mathrm{ch}^{\mathsf{hi}}(T)$ can transmit content with high security level but can also safely be used to transmit content whose security level is low. On the other hand, channels of type $\mathrm{ch}^{\mathsf{lo}}(T)$ can only transmit content with low security level. The definition of the compatibility relation is therefore that $T \leftarrow\!\!\!\rho\ \mathrm{ch}^{\Psi}(T)$ if $\mathsf{level}(T) \leq \Psi$. This allows us to send low-security data on high-security channels but prevents us from sending high-security data on low-security channels.

$$[\textsc{low}]\frac{\Gamma, \mathsf{hi} \vdash M : T^{\mathsf{lo}}}{\Gamma, \mathsf{lo} \vdash M : T^{\mathsf{lo}}} \qquad [\textsc{high}]\frac{\Gamma, \mathsf{lo} \vdash M : T^{\mathsf{lo}}}{\Gamma, \mathsf{hi} \vdash M : T^{\mathsf{lo}}}$$

Moreover, we must ensure that high-security processes cannot be unleashed in low-security settings. This is handled by the rule [T-RUN].

We again express run-time errors by means of an error predicate for which the central rules reflecting the aforementioned requirements are given below:

$$\frac{\Gamma, \mathsf{lo} \vdash M : T \quad \Gamma, \mathsf{hi} \vdash N : T}{\mathsf{Wrong}_{\Gamma}(M\langle N\rangle.x.P)} \qquad \frac{\Gamma, \mathsf{lo} \vdash M : (T_S, \Gamma)^{\mathsf{hi}}}{\mathsf{Wrong}_{\Gamma}(\mathbf{run}\ M)}$$

We have that

Theorem 8. *If* $\Gamma, \Psi \vdash P$ *then* $\neg\mathsf{Wrong}_{\Gamma}(P)$.

The proof of this proceeds by induction in the type rules.

4.6 Conclusions and future work

We have presented a generic type system for higher-order Ψ-calculi, which extends a previous type system for first-order Ψ-calculi. Like its predecessor [61], type judgements for processes are of the form $\Gamma \vdash P$ and are given by a fixed set of rules. Terms, assertions and conditions are assumed to form nominal datatypes, and only a few requirements on type rules are imposed.

The generic type system allows us to identify what should be required of type systems for higher-order process calculi that are instances of the Ψ-calculus; these requirements take the form of instance assumptions. Thus it may also yield important insights into the general structure of type systems for higher-order calculi, and it may therefore also be taken as a starting point for developing more advanced type systems for any language that can be shown to be an instance of higher-order Ψ-calculi. Note, however, that we do not know whether the instance assumptions are *minimal* in any sense. It is possible that some of the requirements can be relaxed or further generalised to yield a smaller set of necessary assumptions. This is an open problem at the time of writing.

Our type system satisfies a general subject-reduction property and can be instantiated to yield type systems with a notion of channel safety for higher-order calculi such as CHOCS, HOπ and also the ρ-calculus. The latter in particular is interesting, as there is no valid encoding of the ρ-calculus into the π-calculus, and thus we cannot capture higher-order typability in a purely first-order setting. This establishes that our generic type system is richer than first-order type systems. However, typability in the ρ-calculus comes at the cost of necessitating that we include type information directly in the definition of channel equivalence. This amounts to saying that the programmer must know (and specify) in advance the type of all names that will be generated during the course of program evaluation. We do not know whether it is possible to create other (non-trivial) type systems for the ρ-calculus without such a restriction.

There are two important lines of future work in this direction: In [62], Hüttel extends the generic type system to consider more general notions of subtyping and resource awareness, and in [63] also considers *session types* for psi-calculi.

Both of these extensions are formulated for first-order Ψ-calculi only, and would therefore both relevant to also consider in the higher-order setting. One example is [135] in which Yoshida defines a type system for a higher-order π-calculus that is used in security analyses for higher-order code mobility. In this type system, the notion of linearity is central. Another is the work of Hepburn and Wright [54] which considers a type system for non-interference in which different parallel components are allowed to have different security levels – unlike the one described in the present chapter, for which all parallel components must agree on a security level. In both of these type systems, the context splitting rule of linear type systems is essential.

5 Typing Composite Subjects[1]

5.1 Introduction

Process calculi are formalisms for modelling and reasoning about concurrent and distributed programs, and one way of reasoning about the safety of such programs is by means of *type systems*. In that approach, types are usually assigned to atomic identifiers, such as variables, function names and, in the case of process calculi with communication primitives, to *channel names*. For example, the original π-calculus [83; 86] features communication primitives for transmitting single, atomic names on named channels: $x\langle z\rangle.P$ outputs the name z on the channel x and continues as P, whilst $x(y).P$ receives a name on the channel x and binds it to y within the continuation P. In standard π-calculus terminology, x is referred to as the *subject*, and y (resp. z) as the *object* of the input (resp. output) construct. This can be compared to a procedure x in an imperative-style language, taking a single parameter, with y and z as the formal and actual parameters, respectively.

Further extensions of the π-calculus also allow polyadicity for objects, such that multiple values can be transmitted in a single communication, written $x(\tilde{y}).P$ and $x\langle\tilde{z}\rangle.P$ for input and output, respectively, where $\tilde{y}$ and $\tilde{z}$ represent vectors of names. This is known as the polyadic π-calculus [113], and although it is well-known that polyadicity can be encoded in the monadic π-calculus, this extension makes the correspondence with imperative procedure calls even clearer, since procedures usually are able to take multiple arguments. This correspondence between synchronisation in the π-calculus, and classic imperative features such as procedures, references and variables is well-known and has already been explored in a number of papers, e.g. [57; 69; 89; 107; 111; 129; 131].

However, not all process calculi use only *atomic* identifiers as channels. For example, the language ${}^{e}\pi$ [27] extends the π-calculus with polyadic *synchronisation*. Thus, in ${}^{e}\pi$, subjects can be *vectors* of names of arbitrary length, e.g. $x_1 \cdot x_2\langle z\rangle.P$ or $x_1 \cdot x_2 \cdot x_3(y).P$; and since names can be passed around, such name vectors can

[1] The material presented in this chapter is joint work with Luca Aceto and Daniele Gorla. The material is unpublished, but a version is made available online in [3]. The chapter is equivalent to the online version, except for minor typographical changes.

also be composed at runtime, although they cannot grow in length. Every π-calculus process is then also an ${}^e\pi$-process, with the length of subjects restricted to a single name; hence, the π-calculus is also referred to as π^1, and every fixed length n of subjects then yields a language π^n, with ${}^e\pi$ being the union of all these languages.

Polyadic synchronisation and runtime composition of channels provide a more realistic way of expressing communication in a distributed setting, compared to the standard π-calculus. Consider for example an IP address in CIDR notation: it is not an *atomic* identifier, but rather a *composition* of elements, which also may be built at runtime. Unlike the case for polyadic objects, such name compositions cannot be expressed in the standard π-calculus without introducing divergence, as was shown by Carbone and Maffeis in [27]. Admitting polyadic subjects into the language thus yields increased expressivity.

The aforementioned correspondence with imperative features then poses a new question: namely, how to view polyadic synchronisation in an imperative setting. Our proposal in the present chapter is that subject vectors are comparable to *name-spaced* procedures and variables, such as in class-based and object-oriented programming, where methods and fields are grouped under a shared class name. A natural way to represent a method call

$$\texttt{call}\ X.f(v_1, \ldots, v_n)$$

to a method f, in a class or object X, with actual parameters $v_1, \ldots, v_n$, would therefore simply be as an output

$$X \cdot f\langle v_1, \ldots, v_n \rangle$$

where we assume X, f are names.[2] Likewise, an input

$$X \cdot f(\tilde{y}).P$$

would correspond to a *declaration* of a method f with body P in a class or object named X. We shall explore this correspondence in detail in the following.

This encoding, in itself, is quite trivial as indicated above. However, we can then use the intuitions afforded by this encoding to guide the development of a type system for ${}^e\pi$, which again should be such that it corresponds to an "expectable" type system for a simple, class-based or object-oriented, imperative language. This is non-trivial, since types are normally given to individual names, whereas subjects in ${}^e\pi$ are composites. Hence, the type of a subject $x_1 \cdot \ldots \cdot x_n$ must somehow be derived from the types of the individual names $x_1, \ldots, x_n$. This can be done in several possible ways, each with some benefits and drawbacks, depending on what subject vectors are intended to represent.

[2] This representation technically only makes use of the π^2 sublanguage of ${}^e\pi$, but one could easily imagine a language with e.g. nested classes, which would require arbitrarily long subject vectors to represent an arbitrary degree of nesting.

5.2 On typing polyadic subjects

Structured channel names pose some interesting problems for the development of type systems, because types are normally given to *atomic* identifiers. This is explored in some detail in [76], which describes the ρ-calculus: a language with structured channels, but without any atomic identifiers. The type system therein is consequently very limited, and in practice only suitable for encodings of languages that *do* feature atomic channel names, such as the π-calculus.

In ${}^{e}\pi$, the situation is somewhat better, since even though channels are composable, they are at least composed of atomic names. However, composing channels at runtime still means that we cannot assign a *single* type to each channel, as is otherwise standard in type systems for the π-calculus (e.g. [83; 100; 113; 126]).

A third language, that allows polyadic subjects is the Ψ-calculus [18; 19]. Briefly, the Ψ-calculus is a generalisation of many of the variants and extensions of the π-calculus. It is a generic framework based on nominal sets [43], and the idea is that one defines three nominal sets (called *terms*, *conditions* and *assertions*) and three operations on them (called *channel equivalence*, *assertion composition* and *entailment*), which are supplied to the Ψ-calculus framework to yield an *instance* of the Ψ-calculus. In [18; 19], Bengtson et al. then show how the π-calculus and several of its variants, including ${}^{e}\pi$, can be obtained as instances, by choosing different settings for these parameters. For example, both subjects and objects of communication are drawn from the set of terms, so an instance with both polyadic subjects and objects can be obtained by defining terms as lists of names. In [98], Parrow et al. further extend the Ψ-calculus with a construct for higher-order communication, and in [64] it is shown that this variant of the Ψ-calculus can even encode the ρ-calculus.

A key point of the Ψ-calculus approach is that certain results (e.g. on bisimulation equivalences) can be formulated and proved for the abstract Ψ-calculus, and they are then automatically inherited by every instance. This idea was then taken up by Hüttel in a series of papers [61; 62; 63], wherein he created generic type systems (of different typing disciplines) for the Ψ-calculus and proved a general subject-reduction result. The generic type systems take a number of parameters, including the parameters for a Ψ-calculus instance, and yield type systems for that instance which then inherit the subject-reduction property, thereby reducing the amount of work needed in order to show soundness of the type system.

However, the aforementioned approach also has some disadvantages. Being a *static* analysis technique, type systems are closely tied to the *syntax* of the language, but in the Ψ-calculus key elements of the syntax are left unspecified (namely, terms, conditions and assertions). This, in turn, means that the generic type systems must require type rules for these syntactic categories to be provided as parameters for the instantiation. The aforementioned papers do give some examples of instances, including type rules for these syntactic categories; however, all except one of these instances are for variants of the π-calculus which only have *single* names as subjects.

Thus, these papers do not actually provide much insight on the question of how to type composite subjects: they merely push the question of how to assign a single type to a composite subject onto the instance parameters.

The sole exception is an instance described in [61], which is a type system for the (Ψ-calculus instance of) Dπ [53]. In this language, a process P executes at a specific *location* l, written $l[P]$, and can then migrate between locations with the construct go $l'.P'$. The located processes are organised in a single-level network, i.e. without nested networks occurring inside locations. As was shown in [27], this language can actually be encoded in ${}^{e}\pi$, specifically in the sub-language π^2 (where subject vectors all have length 2). The relevant clauses of the encoding are as follows:

$$\begin{array}{ll} [\![l[P]]\!] = [\![P]\!]_l & [\![\text{go } k.P]\!]_l = [\![P]\!]_k \\ [\![x\langle z\rangle.P]\!]_l = l \cdot x\langle z\rangle.[\![P]\!]_l & [\![x(y).P]\!]_l = l \cdot x(y).[\![P]\!]_l \end{array}$$

The type language (another parameter) given for the type system in [61] is then the following:

$$T ::= \mathsf{ch}(T) \mid \mathsf{loc}\{x_i : \mathsf{ch}(T_i)\}_{i \in I} \mid B$$

where I is a finite index set, x is a channel name and B denotes base types (integers, booleans etc.). The type rule for assigning a single type to a subject vector is then given as

$$[\text{T-LOC}]\frac{E \vdash l : \mathsf{loc}\{x_i : \mathsf{ch}(T_i)\}_{i \in I} \quad \exists j \in I \,.\, E \vdash x_j : \mathsf{ch}(T_j)}{E \vdash l \cdot x_j : \mathsf{ch}(T_j)}$$

where E is a type environment. As can be seen, the choice here is to repeat the type for x_j, so it must occur both on its own and inside the type of the location. In effect, this means that there is no difference w.r.t. channel capabilities between using the composite subject $l \cdot x_j$ and just using the name x_j. Furthermore, as channel names x_i's appear directly inside the location types $\mathsf{loc}\{x_i : \mathsf{ch}(T_i)\}_{i \in I}$, this setting also encounters problems with α-conversion, in case some of the x_i's are restricted. In general, these choices may suffice for this *specific* instance, since it is limited to *encoded* Dπ-processes, with the encoding ensuring that no processes would lead to problems with the chosen type language and type rules. However, as a *general* type system for π^2, this choice does not seem sufficient, nor does it offer much insight into how it might be generalised to the full ${}^{e}\pi$ language, where subject vectors may have an arbitrary length.

Another approach is described in [26, Chapter 6.5], where Carbone creates two type systems for ${}^{e}\pi$: a 'structural' one and a 'nominal' one. Of these, to our mind, only the latter provides a satisfactory solution to the problem of deriving a composite type T for a subject vector such as $x_1 \cdot x_2$ from the types of the constituents x_1 and x_2. This is done by using *named types* with type names I, and two separate type environments, Δ and Γ. The first maps single names to type names, e.g. $\Delta(x_1) = I_1$ and $\Delta(x_2) = I_2$, and the second maps composite type names to types, e.g. $\Gamma(I_1 \cdot I_2) = T$. This avoids

the problem with α-conversion, in case a name is restricted, since names do not appear directly in the types.

The downside of the aforementioned approach is that this way of structuring the types bears little semblance to familiar concepts from object-oriented languages, which is what we are aiming for in the present work. As argued in the preceding section, the correspondence between methods grouped under a shared class name, and inputs with a common prefix in their subjects, is straightforward. In OO-terminology, the type of a class or object is its interface, consisting of the signatures of the fields and methods it declares, but the correspondence between the type-name vectors and such interfaces is less obvious.

In the present chapter we shall focus on developing a simple type system for ${}^{e}\pi$ for ensuring correct channel usage, whilst ensuring that the intuition from the correspondence with the imperative, object-oriented paradigm remains clear. Our proposal stems from the observation that the capability of a name to be used as a channel and its capability to be used in compositions are orthogonal, and, furthermore, that the order in a composition matters; i.e. $x_1 \cdot x_2$ is different from $x_2 \cdot x_1$, and different compositions may have distinct behaviours. For example, x_1 alone can deliver integers, $x_1 \cdot x_2$ can deliver pairs of integers and $x_1 \cdot x_3$ can deliver booleans. We use a tree-structure to represent both capabilities in our types as follows: Each node in such a 'tree type' is labelled with a type name I and a communication capability C, describing whether the vector can be used as a channel or not (and, in that case, what it can communicate). The root node describes the type of a single name (e.g. x_1); each sub-tree below the root describes the type of a vector composed with x_1, and so also which compositions are allowed. Thus, as ${}^{e}\pi$ allows for name vectors of arbitrary lengths, types for names are trees of arbitrary height.

The text is structured as follows: We start in Section 5.3 by presenting the syntax and labelled operational semantics of ${}^{e}\pi$. We then move to the focus of this chapter and discuss in Section 5.4 the question of how to type subject vectors. The type system we propose satisfies the usual properties of subject reduction and safety, which alone assure us of the quality of our proposal. In fact, it is equivalent to a type system proposed by Carbone in [26], although his types have a different structure, as discussed above. However, our proposal of 'tree-shaped' types also resembles the concept of interfaces, known from type systems for object-oriented languages.

We spell out this correspondence in detail, following the approach of [69; 89; 107; 111; 129; 131]. In Section 5.5, we define the simple class-based language `WC`, (**W**hile with **C**lasses), an imperative language obtained by extending `While` [92] with classes, fields and methods, together with an "expectable" type system for `WC` programs. Then, in Section 5.6, we propose an encoding of `WC` into ${}^{e}\pi$ and show that the type system presented here for ${}^{e}\pi$ exactly corresponds to the "expectable" one for `WC`. This comparison contributes to understanding the relationship between our types and conventional types of OO languages and provides ${}^{e}\pi$ with a type system that makes it a good metalanguage for the semantics of typed OO languages.

5.3 The $^e\pi$-calculus

The version of $^e\pi$ presented in [27] extends the *monadic* π-calculus only with polyadic synchronisation; a natural further extension is to also allow polyadicity for *objects*, as in the polyadic π-calculus [83]. Furthermore, we extend $^e\pi$ with: data values v, consisting of names, integers and booleans; integer and boolean expressions $\tilde{e}$ in outputs $\tilde{x}\langle\tilde{e}\rangle$; and a guarded choice operator $\sum_{i=1}^{n}[e_i]P_i$, where the e_i are boolean expressions. Hence, the syntax of $^e\pi$ processes that we consider in this chapter is:

$$\begin{aligned} P \in \mathcal{P} &::= \mathbf{0} \mid \tilde{x}(\tilde{y}).P \mid \tilde{x}\langle\tilde{e}\rangle.P \mid P_1 \mid P_2 \mid (\boldsymbol{\nu}\tilde{x})P \mid\ !P \mid \sum_{i=1}^{n}[e_i]P_i \\ e \in \mathsf{Exp} &::= \mathrm{op}(\tilde{e}) \mid v \\ v \in \mathsf{Val} &::= \mathcal{N} \cup \mathbb{Z} \cup \mathbb{B} \end{aligned}$$

where $\mathcal{N}$ is the set of names, ranged over by x, y; $\mathbb{Z}$ is the set of integers; $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$ is the set of boolean values, and op denotes a collection of standard operators on these values. We use a tilde $\tilde{\cdot}$ to denote an ordered sequence of elements, including for example name vectors (that can occur both as subjects and as objects of prefixes).

Most constructs are as in the π-calculus: $\mathbf{0}$ is the stopped process (we usually omit trailing $\mathbf{0}$ after prefixes); $P_1 \mid P_2$ is the parallel composition of processes P_1 and P_2; $(\boldsymbol{\nu}\tilde{x})P$ creates new names $\tilde{x}$ with visibility limited to P; and the replication operator $!P$ creates arbitrarily many copies of P. The input and output operators will synchronise on the *vector* of names $\tilde{x}$, and we likewise allow the data transmitted to be a list of values $\tilde{v}$. In the output operator $\tilde{x}\langle\tilde{e}\rangle$, we allow expressions to appear in object position of the output, which must evaluate to a list of values $\tilde{v}$ to be transmitted. The guarded choice operator $\sum_{i=1}^{n}[e_i]P_i$, which will be often abbreviated to $\sum[\tilde{e}]\widetilde{P}$, allows us to choose between the processes P_i for which the associated guards e_i evaluate to the boolean value T (true). Henceforth, we shall write **if** e **then** P_T **else** P_F in place of $[e]P_\mathsf{T} + [\neg e]P_\mathsf{F}$.

For the operators op appearing in expressions e, we shall assume that they, and their arguments, can be evaluated to a single value by some semantics $\rightarrow_e$, which we shall not detail further. We shall write $\tilde{e} \rightarrow_e \tilde{v}$ as an abbreviation of $e_1 \rightarrow_e v_1, \ldots, e_n \rightarrow_e v_n$. Note that we assume that no operation may yield a name as a value. Thus, names may appear as arguments to an operator op, e.g. for equality testing, but an operator may not be used to *create* a name.

We can give the semantics for processes as a (mostly) standard, early labelled semantics. First, we define the labels:

$$\alpha ::= \tau \mid \tilde{x}!(\boldsymbol{\nu}\tilde{y})\tilde{v} \mid \tilde{x}?\tilde{v}$$

The τ label is the unobservable action, and the label $\tilde{x}?\tilde{v}$ is for inputting values $\tilde{v}$ from the name vector $\tilde{x}$. The label $\tilde{x}!(\boldsymbol{\nu}\tilde{y})\tilde{v}$ is for (bound) output: here, $\tilde{y}$ is a list of bound names, such that $\tilde{y} \subseteq \tilde{v}$, i.e. they bind into the object vector, and $\tilde{y}\#\tilde{x}$ (meaning $\tilde{y} \cap \tilde{x} = \emptyset$), i.e. none of the bound names can appear in the subject vector. Hence, we

$$[\text{E-Out}]\frac{}{\tilde{x}\langle\tilde{e}\rangle.P \xrightarrow{\tilde{x}!\tilde{v}} P}\ (\tilde{e} \to_e \tilde{v}) \qquad [\text{E-In}]\frac{}{\tilde{x}(\tilde{y}).P \xrightarrow{\tilde{x}?\tilde{v}} P\{\tilde{v}/\tilde{y}\}}$$

$$[\text{E-Open}]\frac{P \xrightarrow{\tilde{x}!(\boldsymbol{\nu}\tilde{y})\tilde{v}} P'}{(\boldsymbol{\nu}\tilde{z})P \xrightarrow{\tilde{x}!(\boldsymbol{\nu}\tilde{y},\tilde{z})\tilde{v}} P'}\ \begin{pmatrix}\tilde{z} \subseteq \tilde{v}\\ \tilde{z}\ \#\ \tilde{x},\tilde{y}\end{pmatrix} \qquad [\text{E-Res}]\frac{P \xrightarrow{\alpha} P'}{(\boldsymbol{\nu}\tilde{x})P \xrightarrow{\alpha} (\boldsymbol{\nu}\tilde{x})P'}\ (\tilde{x}\#\alpha)$$

$$[\text{E-Sum}]\frac{P_i \xrightarrow{\alpha} P_i'}{\sum[\tilde{e}]\tilde{P} \xrightarrow{\alpha} P_i'}\ (e_i \to_e \mathsf{T}) \qquad [\text{E-Rep}]\frac{!P \mid P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

$$[\text{E-Com}_1]\frac{P_1 \xrightarrow{\tilde{x}!(\boldsymbol{\nu}\tilde{y})\tilde{v}} P_1' \quad P_2 \xrightarrow{\tilde{x}?\tilde{v}} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{y})(P_1' \mid P_2')}\ (\tilde{y}\#P_2)$$

$$[\text{E-Par}_1]\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2}\ (\mathrm{bn}(\alpha)\,\#P_2)$$

Figure 5.1: Early labelled semantics for the ${}^e\pi$-calculus.

define $\mathrm{bn}(\tilde{x}!(\boldsymbol{\nu}\tilde{y})\tilde{v}) = \tilde{y}$, and $\mathrm{bn}(\alpha) = \varnothing$ for all other labels α. If there are no bound names, we shall just write $\tilde{x}!\tilde{v}$ instead of $\tilde{x}!(\boldsymbol{\nu}\epsilon)\tilde{v}$, which is the label for *free* output.

The early, labelled semantics is given by the transition relation $\xrightarrow{\alpha}$ defined by the rules in Figure 5.1. In the semantic rules, a substitution, written $P\{\tilde{v}/\tilde{x}\}$, is the pointwise (capture avoiding) replacement of each name x_i by the corresponding value v_i; thus we require that $\tilde{x}$ and $\tilde{v}$ have the same arity, and we therefore assume that terms are well-sorted. Note that we omit the symmetric versions of the rules [E-Par$_1$] and [E-Com$_1$]. Furthermore, we assume all rules are defined up to α-equivalence, and that bound names within $(\boldsymbol{\nu}\tilde{x})P$ can be reordered. Lastly, we use the notation $\tilde{x}\#P$ to say that the names $\tilde{x}$ are *fresh* for P, analogously to the notation used for labels.

The [E-Par$_1$], [E-Res], [E-Open] and [E-In] are standard, similar to e.g. the labelled semantics for the π-calculus given in [96], but extended in the obvious way to allow both polyadic objects, as in the polyadic π-calculus [83], and polyadic subjects, as in [27]. As usual, polyadicity (i.e. polyadic objects) leads to have a bound output label both in the premise and in the conclusion of the [E-Open] rule. In the rule [E-Com$_1$], the list of bound names $\tilde{y}$ in $\tilde{x}!(\boldsymbol{\nu}\tilde{y})\tilde{v}$ can be empty; thus, it can act both as an ordinary communication rule and as a close rule.

Finally, as usual, we write $\xrightarrow{\tau}{}^*$ to denote the reflexive and transitive closure of $\xrightarrow{\tau}$.

5.4 A simple type system for ${}^{e}\pi$

The type system we wish to create is a standard type system for ensuring correct channel usage, similar to the simple type system for the polyadic π-calculus [113; 126]. The types are intended to describe *data* (i.e. channel usage and values), so they will be assigned once and do not change. In the type system for the polyadic π-calculus, the types T are of the form $\mathsf{ch}(\widetilde{T})$, for names that can be used as channels (subjects) to send/receive values of types $\widetilde{T}$, and nil, for names that cannot be used as channels. Other data values (e.g. integers and booleans) have a base type B as usual (e.g. int, bool). Thus, for example, $\mathsf{ch}(\mathsf{int}, \mathsf{ch}(\mathsf{nil}))$ is the type of a channel that can be used to communicate two values: an integer and a name of type $\mathsf{ch}(\mathsf{nil})$, which in turn can be used as a channel to send/receive names that cannot be used as channels.

A key problem in devising a type system for ${}^{e}\pi$ is to decide how type judgments for subject vectors $\tilde{x}$ should be concluded, if types are given to individual names. Of course, this does not have necessarily to be the case: types could be given directly to vectors of names. However, that is an inflexible solution. Indeed, since subject vectors can be composed at runtime in ${}^{e}\pi$, using the above-mentioned approach would necessitate that the type environment Γ must contain entries for all name vectors that may potentially be created during the execution of a process. This might be sufficient in certain scenarios, e.g. with translations from other languages, where the form of all subject vectors may be known (and known not to change at runtime), but as a general solution it does not seem satisfactory. Hence, we maintain that types should be given to individual names. A number of possible solutions then come to mind:

- We can require that all names in a name vector must have the same type to be well-typed. Thus, if x_1 has type T and x_2 has type T, then $x_1 \cdot x_2$ would have type T as well, and so would $x_2 \cdot x_1$. This solution does not seem satisfactory, since it does not distinguish between using the names individually, and using the composition.

- We can let types be *sets* of capabilities. The type of a composition of names could then be the intersection of the sets. For example, if

$$\Gamma \vdash x_1 : \{\mathsf{ch}(T_1), \mathsf{ch}(T_2)\} \qquad \text{and} \qquad \Gamma \vdash x_2 : \{\mathsf{ch}(T_2), \mathsf{ch}(T_3)\}$$

  then $\Gamma \vdash x_1 \cdot x_2 : \{\mathsf{ch}(T_2)\}$. However, that would lead to issues of non-well-foundedness, if we want to allow names to be sent on the vector, which are also *in* the vector. For example, to type the process $x_1 \cdot x_2\langle x_1\rangle.\mathbf{0}$, we would need the type of $x_1 \cdot x_2$ to contain $\mathsf{ch}(T)$, which implies that both the types (sets) of x_1 and x_2 must contain $\mathsf{ch}(T)$. But since T is the type of x_1, then we have some non well-foundedness issues.

- We can allow names to appear in types to express which compositions should be allowed. This is the method used in the type system in [61] for the distributed

π-calculus Dπ [53], as described above in Section 5.1. Suppose, for example, that we have $\Gamma \vdash x_1 : \{x_2, x_3\}$, to signify that x_1 can be followed by either x_2 or x_3 in a composition. This avoids the problem of non-well-foundedness, since we could then have e.g. that $\Gamma \vdash x_2 : \mathsf{ch}(\{x_2, x_3\})$, which would make it possible to type the process $x_1 \cdot x_2\langle x_1\rangle.\mathbf{0}$. However, having names appear directly in types creates other problems, if the names are bound. Consider the process

$$P \triangleq (\boldsymbol{\nu} x_2 : \mathsf{ch}(\{x_2, x_3\}))(x_1 \cdot x_2\langle x_1\rangle.\mathbf{0} \mid x_1 \cdot x_2(y).\mathbf{0})$$

with the type annotation added in the restriction. If $\Gamma \vdash x_1 : \{x_2, x_3\}$ as before, then P is well-typed. However, since x_2 is bound, then P is α-equivalent to another process P' where the name x_2 has been converted to some other, fresh name, e.g. z. But z does not appear in the type of x_1 in Γ, so α-conversion must also be applied to Γ, lest well-typedness would not be preserved by α-equivalence. This is doable, but at least inelegant, since it would mean that all restrictions, at least implicitly, must be assumed to have scope over Γ. Furthermore, it also does not allow us to distinguish between the use of x_2 alone and in the composition $x_1 \cdot x_2$.

5.4.1 The language of types

In light of the considerations above, we shall choose our language of types as follows. Firstly, to avoid using channel names directly in types, we shall instead use *named types* as was done in [26]. Thus, we introduce a separate, countably infinite set of atomic *type names* $\mathcal{I}$, ranged over by I, which is distinct from the set of channel names $\mathcal{N}$. To avoid the issues of non-well-foundedness, we shall give every type a name from this set, i.e. we shall be using *named types*.

Secondly, the capability of a name to be used as a channel and its capability to be used in compositions appear to be more or less orthogonal. Hence, we shall distinguish between the 'channel capabilities' of a name and its 'composition capabilities.' We shall therefore use a pair for the type of a name: the first component describes whether the name can be used as a channel or not, and, in the former case, what it can be used to communicate; the second component describes whether the name can be composed with other names and, if so, which ones.

Lastly, as $x_1 \cdot x_2$ is distinct from $x_2 \cdot x_1$, we may want to allow a name to have different meanings, depending on *where* it occurs in a composition. The type of a name is therefore not a single tuple, but a *tree* of capabilities. One way to view this is to think of each type name I as a declaration of a name space, which in turn may contain other name space declarations.

Definition 36 (Types)**.** We use the following language of types and type environments:

$$
\begin{array}{rll}
B ::= I \mid \mathsf{int} \mid \mathsf{bool} & & \text{base types} \\
C ::= \mathsf{ch}(\widetilde{B}) \mid \mathsf{nil} & & \text{capability types} \\
T \in \mathcal{T} ::= B \mid (C, \Delta) & & \text{types} \\
\Gamma, \Delta \in \mathcal{E} ::= \mathcal{N} \cup \mathcal{I} \rightharpoonup \mathcal{T} & & \text{type environments}
\end{array}
$$

We assume that Γ can be written as an unrodered sequence of pairs $x : T, \Gamma'$, with $\varnothing$ denoting the empty environment; we use the same notation to denote that the environment Γ' is extended with the pair assigning type T to x. ■

Γ stores assumptions about the types of names, as well as about type names. The idea in this type language is that a name x should only have one *base type* B, as recorded in Γ. The type of a name should then almost always be a type name I, except when we type the continuation after an input, since the bound names here may also be substituted for the other data types. Furthermore, for a type name I to be meaningful, it must also have its own entry in Γ, which must be a tuple (C, Δ), where C is a capability type and Δ is a type environment that records the meaning of type names under the present name. We shall use the notation $I \mapsto (C, \Delta)$ to refer to a type name plus its entry, and we say a type environment Γ is *well-formed* if all type names have entries. We shall only consider well-formed type environments in the following.

We shall henceforth use a *typed* syntax for processes, with explicit (base) types added to the declaration of new names, i.e. $(\boldsymbol{\nu} \tilde{x} : \widetilde{B})P$, where $|\tilde{x}| = |\widetilde{B}|$. Likewise, types must also now appear in the list of bound names in labels, so we write $\tilde{x}!(\boldsymbol{\nu} \tilde{z} : \widetilde{B})\tilde{v}$.

The type language above is richer than the usual one for the π-calculus, although most of the complexities will not appear directly in the syntax of processes, since the idea is that names should only be given base types B. However, with this more complex type language, we can also achieve the same effect as some of the simpler solutions mentioned above, such as for example requiring that all names in a vector must have the same type. This can easily be achieved, since types are referred to by their type name I, so multiple names can have the *same* type (and not just structurally similar types). As an added benefit, named types also allow us to type processes such as $x\langle x \rangle.\mathbf{0}$ without requiring explicit recursive types to be added to the language, since x can now simply be given the type $I \mapsto (\mathsf{ch}(I), \varnothing)$.

5.4.2 The type system

The type judgement $\Gamma \vdash P$ ensures that P uses channels as prescribed by Γ, ensures that it only contains well-typed expressions (e.g., by rejecting $\mathsf{T} + 2$), and also rules out obviously malformed processes such as $x \cdot 2(y).P'$, since such processes would lead to the semantics being stuck. As we assume well-sortedness of terms, we do not check for arity-mismatch in output objects, although this can also be included if desired.

$$[\text{T-NIL}]\frac{}{\Gamma \vdash \mathbf{0}} \qquad [\text{T-SUM}]\frac{\forall i.(\Gamma \vdash e_i : \text{bool} \quad \Gamma \vdash P_i)}{\Gamma \vdash \sum[\tilde{e}]\widetilde{P}}$$

$$[\text{T-RES}]\frac{\Gamma, \tilde{x} : \widetilde{B} \vdash P}{\Gamma \vdash (\boldsymbol{\nu}\tilde{x} : \widetilde{B})P} \qquad [\text{T-OP}]\frac{\Gamma \vdash \tilde{e} : \widetilde{B} \quad \text{op} : \widetilde{B} \to B}{\Gamma \vdash \text{op}(\tilde{e}) : B}$$

$$[\text{T-PAR}]\frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \qquad [\text{T-VEC}_2]\frac{(\text{fst} \circ \Delta \circ \Gamma)(x) = C}{\Gamma; \Delta \vdash x : C}$$

$$[\text{T-REP}]\frac{\Gamma \vdash P}{\Gamma \vdash\ !P} \qquad [\text{T-IN}]\frac{\Gamma; \Gamma \vdash \tilde{x} : \text{ch}(\widetilde{B}) \quad \Gamma, \tilde{y} : \widetilde{B} \vdash P}{\Gamma \vdash \tilde{x}(\tilde{y}).P}$$

$$[\text{T-VAL}]\frac{}{\Gamma \vdash v : B}\left(B = \begin{cases} \text{int} & \text{if } v \in \mathbb{Z} \\ \text{bool} & \text{if } v \in \mathbb{B} \\ \Gamma(v) & \text{if } v \in \mathcal{N} \end{cases}\right)$$

$$[\text{T-VEC}_1]\frac{(\text{snd} \circ \Delta \circ \Gamma)(x) = \Delta_x \quad \Gamma; \Delta_x \vdash \tilde{x} : C}{\Gamma; \Delta \vdash x \cdot \tilde{x} : C}$$

$$[\text{T-OUT}]\frac{\Gamma; \Gamma \vdash \tilde{x} : \text{ch}(\widetilde{B}) \quad \Gamma \vdash \tilde{e} : \widetilde{B} \quad \Gamma \vdash P}{\Gamma \vdash \tilde{x}\langle\tilde{e}\rangle.P}$$

Figure 5.2: Type rules for values, subject vectors, and processes.

The rules for values, subject vectors and processes are given in Figure 5.2. For expressions, if e is a value, we use rule [T-VAL]; if e contains an operator op, we use rule [T-OP], that relies on the signature $\widetilde{B} \to B$, obtained by considering op as a function $\text{op} : \text{Val}^* \to \text{Val}$ (e.g. $+ : \texttt{int}^2 \to \texttt{int}$ and $\wedge : \texttt{bool}^2 \to \texttt{bool}$). As expectable, the main peculiarity of these rules is the typing for subject vectors $\tilde{x}$ (required in the premise of rules [T-IN] and [T-OUT]). This judgment is of the form $\Gamma; \Gamma \vdash \tilde{x} : C$, and is concluded by the rules [T-VEC$_1$] and [T-VEC$_2$]. To conclude that $x_1 \cdot x_2 \cdot \ldots \cdot x_n$ (for $n > 1$) has type C, we proceed as follows:

1. we start with rule [T-VEC$_2$], that takes x_1 and looks it up in Γ; since Γ is well-formed, this lookup must yield $\Gamma(x_1) = I_{x_1}$, with $\Gamma(I_{x_1}) = (C_{x_1}, \Delta_{x_1})$;

2. we then use function snd to extract the second component of this tuple, i.e. the environment Δ_{x_1}, that will be used to give the meaning of a selection of type names below I_{x_1} (the names that may be composed with x_1 are the names that occur in the domain of Δ_{x_1});

$$[\text{W-PAR}]\frac{\mathsf{Wrong}_\Gamma(P_1) \vee \mathsf{Wrong}_\Gamma(P_2)}{\mathsf{Wrong}_\Gamma(P_1 \mid P_2)} \qquad [\text{W-IN}]\frac{\Gamma \vdash \tilde{x} : \mathsf{ch}(\widetilde{B}) \implies |\widetilde{B}| \neq |\tilde{y}|}{\mathsf{Wrong}_\Gamma(\tilde{x}(\tilde{y}).P)}$$

$$[\text{W-RES}]\frac{\mathsf{Wrong}_{\Gamma,\tilde{x}:\widetilde{B}}(P)}{\mathsf{Wrong}_\Gamma((\boldsymbol{\nu}\tilde{x} : \widetilde{B})P)} \qquad [\text{W-OUT}]\frac{\Gamma \vdash \tilde{x} : \mathsf{ch}(\widetilde{B}) \implies \Gamma \not\vdash \tilde{e} : \widetilde{B}}{\mathsf{Wrong}_\Gamma(\tilde{x}\langle\tilde{e}\rangle.P)}$$

$$[\text{W-REP}]\frac{\mathsf{Wrong}_\Gamma(P)}{\mathsf{Wrong}_\Gamma(!P)} \qquad [\text{W-SUM}]\frac{\exists i.(\Gamma \not\vdash e_i : \mathsf{bool} \vee \mathsf{Wrong}_\Gamma(P_i))}{\mathsf{Wrong}_\Gamma(\sum[\tilde{e}]\widetilde{P})}$$

Figure 5.3: Error-predicate for processes.

3. we recur on $x_2 \cdot \ldots \cdot x_n$, but we now use Δ_{x_1} for giving a type to I_{x_2} ($= \Gamma(x_2)$) in point 1 above.

For vectors of length 1, we use [T-VEC$_1$], which consist of the same first two steps above, except that it uses the function fst to extract the returned channel capability C.

As an example, imagine that x_1 is a name that: (1) when used alone, it can carry integers; (2) when followed by x_2, it can carry pairs of integers; (3) when followed by x_2 and x_3, it can carry booleans; and (4) when preceeded by x_2, it can carry pairs of booleans. To accommodate this situation, we need to use a typing environment Γ such that: $\Gamma(x_1) = I_1$, $\Gamma(x_2) = I_2$, $\Gamma(x_3) = I_3$, $\Gamma(I_1) = \langle \mathsf{ch}(\mathsf{int}), \Delta_1 \rangle$, and $\Gamma(I_2) = \langle \mathsf{nil}, \Delta_2 \rangle$, where $\Delta_1(I_2) = \langle \mathsf{ch}(\mathsf{int}, \mathsf{int}), \Delta_3 \rangle$, $\Delta_2(I_1) = \langle \mathsf{ch}(\mathsf{bool}, \mathsf{bool}), \emptyset \rangle$, and $\Delta_3(I_3) = \langle \mathsf{ch}(\mathsf{bool}), \emptyset \rangle$. Then, the processes $x_1\langle 3\rangle$, $x_1 \cdot x_2\langle 3, 5\rangle$, $x_1 \cdot x_2 \cdot x_3\langle \mathsf{T}\rangle$, and $x_2 \cdot x_1\langle \mathsf{T}, \mathsf{F}\rangle$ are all typeable under Γ. We only depict a part of the third inference, since the other ones are similar or simpler:

$$\dfrac{\dfrac{(\mathsf{snd} \circ \Gamma \circ \Gamma)(x_1) = \Delta_1 \qquad \dfrac{(\mathsf{snd} \circ \Delta_1 \circ \Gamma)(x_2) = \Delta_3 \qquad \dfrac{(\mathsf{fst} \circ \Delta_3 \circ \Gamma)(x_3) = \mathsf{ch}(\mathsf{bool})}{\Gamma; \Delta_3 \vdash x_3 : \mathsf{ch}(\mathsf{bool})}}{\Gamma; \Delta_1 \vdash x_2 \cdot x_3 : \mathsf{ch}(\mathsf{bool})}}{\Gamma; \Gamma \vdash x_1 \cdot x_2 \cdot x_3 : \mathsf{ch}(\mathsf{bool})}}{\Gamma \vdash x_1 \cdot x_2 \cdot x_3\langle \mathsf{T}\rangle.\mathbf{0}}$$

5.4.3 Run-time errors, Safety, and Soundness results

We formalise the notion of safety as an error-predicate, written $\mathsf{Wrong}_\Gamma(P)$, which checks for mismatched types in input/output, and also whether each expression guard in a sum has a boolean type. It is given by the rules in Figure 5.3. We say that a process P is *now-safe*, written $\mathsf{NSafe}_\Gamma(P)$, if P does not satisfy the error-predicate; and likewise, we say that a process is *safe*, written $\mathsf{Safe}_\Gamma(P)$, if it is now-safe for all

its τ-labelled transitions:

$$\mathsf{NSafe}_\Gamma(P) \triangleq \neg\mathsf{Wrong}_\Gamma(P)$$
$$\mathsf{Safe}_\Gamma(P) \triangleq \forall P' \,.\, (P \xrightarrow{\tau}{}^{*} P' \implies \mathsf{NSafe}_\Gamma(P'))$$

Note that the concepts of now-safety and safety only make sense if all the free names of P appear in Γ; hence, in what follows, we shall implicitly assume that $\mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)$. Then, our first result is that well-typed processes are now-safe:

Theorem 9 (Safety)**.** *If* $\Gamma \vdash P$ *then* $\mathsf{NSafe}_\Gamma(P)$.

The proof is by induction on (the height of) the judgement $\Gamma \vdash P$; it can be found in Section 5.8.1

The second result states that well-typedness is preserved by the (labelled) semantics. To express this, we first need the concept of well-typed labels:

Definition 37 (Well-typed label)**.** We say that a label α is well-typed for a given Γ if it can be concluded using one of the following rules:

$$[\text{T-TAU}]\frac{}{\Gamma \vdash \tau}$$

$$[\text{T-SND}]\frac{\Gamma; \Gamma \vdash \tilde{x} : \mathsf{ch}(\widetilde{B}_2) \quad \Gamma, \tilde{z} : \widetilde{B}_1 \vdash \tilde{v} : \widetilde{B}_2}{\Gamma \vdash \tilde{x}!(\boldsymbol{\nu}\tilde{z} : \widetilde{B}_1)\tilde{v}}$$

$$[\text{T-RCV}]\frac{\Gamma; \Gamma \vdash \tilde{x} : \mathsf{ch}(\widetilde{B}) \quad \Gamma \vdash \tilde{v} : \widetilde{B}}{\Gamma \vdash \tilde{x}?\tilde{v}}$$ ■

The second result we desire to have is then that well-typedness is preserved by the transition relation:

Theorem 10 (Subject reduction)**.** *Let* $\Gamma \vdash P$*. If* $P \xrightarrow{\alpha} P'$ *and* $\Gamma \vdash \alpha$*, then* $\Gamma' \vdash P'$*, where* $\Gamma' = \Gamma, \tilde{z} : \widetilde{B}$ *if* $\alpha = \tilde{x}!(\boldsymbol{\nu}\tilde{z} : \widetilde{B})\tilde{v}$*, and* $\Gamma' = \Gamma$ *otherwise.*

The proof is by induction on the inference for $P \xrightarrow{\alpha} P'$; it can be found in Section 5.8.3. Finally, by combining Theorems 9 and 10, we obtain that a well-typed process remains now-safe after any number of τ-transitions:

Corollary 10 (Soundness)**.** *If* $\Gamma \vdash P$*, then* $\mathsf{Safe}_\Gamma(P)$.

Conjecture 1. *We conjecture that our type system is equivalent to the nominal type system of [26, Chapter 6.5].*

$$
\begin{aligned}
DC &::= \epsilon \mid \texttt{class}\ A\ \texttt{\{}\ DF\ DM\ \texttt{\}}\ DC \\
DF &::= \epsilon \mid \texttt{field}\ p\ \texttt{:=}\ v\texttt{;}\ DF \\
DM &::= \epsilon \mid \texttt{method}\ f(\tilde{x})\ \texttt{\{}\ S\ \texttt{\}}\ DM \\
e &::= v \mid x \mid \texttt{this} \mid e.p \mid \mathrm{op}(\tilde{e}) \\
S &::= \texttt{skip} \mid \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S \mid x\ \texttt{:=}\ e \mid \texttt{this}.p\ \texttt{:=}\ e \mid S_1\texttt{;}S_2 \\
&\quad \mid \texttt{if}\ e\ \texttt{then}\ S_{\mathsf{T}}\ \texttt{else}\ S_{\mathsf{F}} \mid \texttt{while}\ e\ \texttt{do}\ S \mid \texttt{call}\ e.f(\tilde{e})
\end{aligned}
$$

where $x, y \in \mathsf{VNames}$ (variable names), $p, q \in \mathsf{FNames}$ (field names), $A, B \in \mathsf{CNames}$ (class names), $f, g \in \mathsf{MNames}$ (method names), and $v \in \mathbb{Z} \cup \mathbb{B} \cup \mathsf{CNames}$ (values). Throughout the chapter, we assume, without loss of generality, that all field names are distinct from each other (within a class), and likewise for all method names.

Figure 5.4: The syntax of WC.

We shall not attempt to formally prove the above claim, since it is beside the point of the present chapter. However, it seems clear that it should be the case, since the type system in [26, Chapter 6.5] associates a type T to each permitted combination of type names, paralleling the permitted combinations of names in subject vectors. In our type system, this would correspond to a path from root to leaf in one of the tree types stored in Γ. Thus, if the combinations $I_1 \cdot I_2$ and $I_1 \cdot I_2$ exist in the type environment of [26, Chapter 6.5], this would correspond to a tree (rooted at I_1), with branches for I_2 and I_3 in our type system. Hence, it should be possible to create a bidirectional mapping between the type environments of the two type systems.

5.5 The WC language

The intuition that a single-name input $x(\tilde{y}).P$ corresponds to a (non-persistent) function definition, and that an output $x\langle\tilde{e}\rangle$ corresponds to a function call, is well-known from Milner's π-calculus encoding of the λ-calculus [82], and also from Pierce and Turner's work on the PICT language [102; 126]. Also, the encoding of object-oriented (OO) languages into (variants of the) π-calculus is by now well established [69; 111; 131]; hence, it is not surprising that polyadic synchronisations would make the latter task even easier. What is less obvious is what kind of typing discipline corresponds to the type system we have presented in Section 5.4. To answer this question, we shall define WC (**W**hile with **C**lasses), an object-oriented, imperative language similar to While [92], but extended with classes, fields and methods, and propose an encoding into ${}^{e}\pi$. We will prove that the type system presented here for ${}^{e}\pi$ exactly corresponds to the "expectable" type system one would devise for WC.

5.5.1 Syntax and (big-step) operational semantics

The syntax of WC is given in Figure 5.4, where, for simplicity, we assume that all name sets are pairwise disjoint, and that field and methods names are unique within each class declaration.

To provide a suitable operational semantics, we shall need some environments to record the bindings of classes, methods, fields, and local variables, including the 'magic variable' `this`. We define them as sets of partial functions as follows:

Definition 38 (Binding model)**.** We define the following sets of partial functions:

$$\begin{aligned}
\mathsf{env}_V \in \mathsf{Env}_V &::= \mathsf{VNames} \cup \{\,\texttt{this}\,\} \rightharpoonup \mathsf{Val} \\
\mathsf{env}_F \in \mathsf{Env}_F &::= \mathsf{FNames} \rightharpoonup \mathsf{Val} \\
\mathsf{env}_M \in \mathsf{Env}_M &::= \mathsf{MNames} \rightharpoonup \mathsf{VNames}^* \times \mathsf{Stm} \\
\mathsf{env}_S \in \mathsf{Env}_S &::= \mathsf{CNames} \rightharpoonup \mathsf{Env}_F \\
\mathsf{env}_T \in \mathsf{Env}_T &::= \mathsf{CNames} \rightharpoonup \mathsf{Env}_M
\end{aligned}$$

We regard each environment env_X, for any $X \in \{V, F, M, S, T\}$, as a list of tuples $(d, c), \mathsf{env}'_X$ for some $d \in \mathrm{dom}(\mathsf{env}_X)$ and $c \in \mathrm{codom}(\mathsf{env}_X)$. The notation $\mathsf{env}_X[d \mapsto c]$ denotes an update of env_X. We write $\mathsf{env}_X^{\emptyset}$ for the empty environment. ■

When two or more environments appear together, we shall use the convention of writing e.g. env_{ST} instead of $\mathsf{env}_S, \mathsf{env}_T$ to simplify the notation.

Our binding model then consists of two environments: A *method table* env_T, which maps class names to method environments, such that we for each class can retrieve the list of methods it declares; and a *state* env_S, which maps class names to lists of fields and their values. The method table will be constant, once all declarations are performed, but the state will change during the evaluation of a program. The semantics of declarations concern the initial construction of the field and method environments, env_F and env_M, and the state and method table env_S and env_T. We give the semantics in classic big-step style. Transitions are thus on the form $\langle DX, \mathsf{env}_X\rangle \to_{DX} \mathsf{env}'_X$ for $X \in \{F, M, C, ST\}$. The rules are given in Figure 5.5.

In practice, we shall assume that class declarations are simply in terms of the two environments env_{ST}, representing the methods and fields declared for each class. Thus, for every declared class A, we have that $\mathsf{env}_T(A)(f) = (\tilde{x}, S)$ and $\mathsf{env}_S(A)(p) = v$, for every method f (with formal parameters $\tilde{x}$ and body S) and field p (with current value v) declared in A. Furthermore, we shall use another environment, env_V, to store the bindings of local variables x, i.e. such that $\mathsf{env}_V(x) = v$ for every local variable x with current value v.

Figure 5.6 gives the semantics of expressions e. Expressions have no side effects, so they cannot contain method calls, but they can access both local variables and

$$[\text{WC-DCLF}_1]\frac{\langle DF, \text{env}_F\rangle \to_{DF} \text{env}'_F}{\langle \texttt{field}\ p\ \texttt{:=}\ v\texttt{;}DF, \text{env}_F\rangle \to_{DF} (p, v), \text{env}'_F}$$

$$[\text{WC-DCLF}_2]\frac{}{\langle \epsilon, \text{env}_F\rangle \to_{DF} \text{env}_F}$$

$$[\text{WC-DCLM}_1]\frac{\langle DM, \text{env}_M\rangle \to_{DM} \text{env}'_M}{\langle \texttt{method}\ f\texttt{(}\tilde{x}\texttt{)}\ \texttt{\{}\ S\ \texttt{\}}\ DM, \text{env}_M\rangle \to_{DM} (f, (\tilde{x}, S)), \text{env}'_M}$$

$$[\text{WC-DCLM}_2]\frac{}{\langle \epsilon, \text{env}_M\rangle \to_{DM} \text{env}_M}$$

$$[\text{WC-DCLC}_1]\frac{\begin{array}{c}\left\langle DF, \text{env}_F^{\varnothing}\right\rangle \to_{DF} \text{env}_F \\ \left\langle DM, \text{env}_M^{\varnothing}\right\rangle \to_{DM} \text{env}_M \\ \langle DC, \text{env}_{ST}\rangle \to_{DC} \text{env}'_{ST}\end{array}}{\begin{array}{r}\langle \texttt{class}\ A\ \texttt{\{}\ DF\ DM\ \texttt{\}}\ DC, \text{env}_{ST}\rangle \\ \to_{DC} ((A, \text{env}_F), \text{env}'_S), ((A, \text{env}_M), \text{env}'_T)\end{array}}$$

$$[\text{WC-DCLC}_2]\frac{}{\langle \epsilon, \text{env}_{ST}\rangle \to_{DC} \text{env}_{ST}}$$

Figure 5.5: Semantics of WC declarations.

$$[\text{WC-VAL}]\frac{}{\text{env}_{SV} \vdash v \to_e v}$$

$$[\text{WC-VAR}]\frac{}{\text{env}_{SV} \vdash x \to_e v}\ (\text{env}_V(x) = v)$$

$$[\text{WC-OP}]\frac{\text{env}_{SV} \vdash \tilde{e} \to_e \tilde{v} \quad \text{op}(\tilde{v}) \to_{\text{op}} v}{\text{env}_{SV} \vdash \text{op}(\tilde{e}) \to_e v}$$

$$[\text{WC-FIELD}]\frac{\text{env}_{SV} \vdash e \to_e A}{\text{env}_{SV} \vdash e.p \to_e v}\ (\text{env}_S(A)(p) = v)$$

Figure 5.6: Semantics of WC expressions.

fields of any class. Thus transitions are of the form $\text{env}_{SV} \vdash e \to_e v$, i.e. transitions must be concluded relative to the state and variable environments. We do not give explicit rules for the boolean and integer operators subsumed under op, but simply

assume that they can be evaluated to a single value by some semantics $\mathrm{op}(\vec{v}) \to_{\mathrm{op}} v$. Note that we assume that *no* operations can *yield* a class name A, so we disallow any form of pointer arithmetic.

A program in WC consists of the class definitions, recorded in the environments env_{ST}, plus a single method call. The semantics of statements describes the actual execution steps of a program. In Figure 5.7 we give the semantics in big-step style, where a step describes the execution of a statement in its entirety. Statements can read from the method table, and they can modify the variable and property bindings, and hence the state. The result of executing a statement is a new state, so transitions must here be of the form $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$ with $\mathsf{env}'_{SV} = \mathsf{env}'_S, \mathsf{env}'_V$ denoting that both the property values in env_S and the values of the local variables in env_V *may* have been modified.

5.5.2 Typed WC

There are also textbook examples of type systems for While-like languages; for example the language BUMP given in [60, pp. 185-198], which also includes procedure calls. We can define a similar type system for WC as follows:

Definition 39 (WC-types)**.** We use the following language of types:

$$\begin{array}{llll} IC ::= \epsilon \mid \texttt{interface}\ I\ \{\ IF\ \ IM\ \}\ IC & & B ::= I \mid \mathsf{int} \mid \mathsf{bool} \\ IF ::= \epsilon \mid \texttt{field}\ p\ :\ B;\ IF & & T ::= B \mid \mathrm{proc}(\widetilde{B}) \mid \Delta \\ IM ::= \epsilon \mid \texttt{method}\ f\ :\ \mathrm{proc}(\widetilde{B});\ IM & & \Gamma, \Delta ::= \mathcal{N} \rightharpoonup T \end{array}$$

where $I \in \mathcal{I}$ is a type name and $\mathcal{N} ::= \mathsf{CNames} \cup \mathsf{FNames} \cup \mathsf{VNames} \cup \mathsf{MNames} \cup \mathcal{I}$. ■

We use a simple 'interface definition language' analogous to the class definitions, to specify the signatures of fields and methods. We shall henceforth use a typed version of the syntax, where each class is annotated with a type name I (viz., `class` $A{:}I$ `{` $DF\ \ DM$ `}`); a type environment Γ then is such that $\Gamma(A) = I$ and $\Gamma(I) = \Delta$, where Δ is again a type environment containing the signatures of fields and methods listed in the interface definition for I; thus, different classes can implement the same interface. Finally, also local variables are now declared with a type, i.e. `var` B `x :=` e `in` S.

The type rules for WC are given in Figure 5.8 (environment agreement), Figure 5.9 (expressions), and Figure 5.10. Note that we continue to use the abbreviated notation for environments, and thus write e.g. $\Gamma \vdash \mathsf{env}_{SV}$ for $\Gamma \vdash \mathsf{env}_S \wedge \Gamma \vdash \mathsf{env}_V$.

Well-typedness of statements ensures that: types are preserved in assignments (see rules [T-DECV] and [T-ASSF]), methods are called with the correct number and types of arguments (rule [T-CALL]), and classes contain the accessed members (rules [T-FIELD] and [T-CALL]). We shall forgo the definition of an explicit $\mathrm{Safe}_\Gamma(\cdot)$ predicate,

$$[\textsc{wc-skip}]\ \frac{}{\mathsf{env}_T \vdash \langle \texttt{skip}, \mathsf{env}_{SV}\rangle \to \mathsf{env}_{SV}}$$

$$[\textsc{wc-seq}]\ \frac{\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \to \mathsf{env}''_{SV} \quad \mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \to \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle S_1\,;S_2, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}}$$

$$[\textsc{wc-if}]\ \frac{\mathsf{env}_{SV} \vdash e \to_e b \in \mathbb{B} \quad \mathsf{env}_T \vdash \langle S_b, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_{\mathsf{T}}\ \texttt{else}\ S_{\mathsf{F}}\ , \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}}$$

$$[\textsc{wc-while}_{\mathsf{T}}]\ \frac{\begin{array}{c}\mathsf{env}_{SV} \vdash e \to_e \mathsf{T} \\ \mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to \mathsf{env}''_{SV} \\ \mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}''_{SV}\rangle \to \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}}$$

$$[\textsc{wc-while}_{\mathsf{F}}]\ \frac{\mathsf{env}_{SV} \vdash e \to_e \mathsf{F}}{\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}_{SV}\rangle \to \mathsf{env}_{SV}}$$

$$[\textsc{wc-decv}]\ \frac{\begin{array}{c}\mathsf{env}_{SV} \vdash e \to_e v \\ \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, ((x, v), \mathsf{env}_V)\rangle \to \mathsf{env}'_S, ((x, v'), \mathsf{env}'_V)\end{array}}{\mathsf{env}_T \vdash \langle \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}}$$

where:

$$x \notin \mathrm{dom}\,(\mathsf{env}_V)$$

$$[\textsc{wc-assv}]\ \frac{\mathsf{env}_{SV} \vdash e \to_e v}{\mathsf{env}_T \vdash \langle x\ \texttt{:=}\ e, \mathsf{env}_{SV}\rangle \to \mathsf{env}_S, \mathsf{env}_V[x \mapsto v]}\ (x \in \mathrm{dom}\,(\mathsf{env}_V))$$

$$[\textsc{wc-assf}]\ \frac{\mathsf{env}_{SV} \vdash e \to_e v}{\mathsf{env}_T \vdash \langle \texttt{this}.p\ \texttt{:=}\ e, \mathsf{env}_{SV}\rangle \to \mathsf{env}_S[A \mapsto \mathsf{env}_F[p \mapsto v]]\,, \mathsf{env}_V}$$

where:

$$p \in \mathrm{dom}\,(\mathsf{env}_F)$$
$$A = \mathsf{env}_V(\texttt{this})$$
$$\mathsf{env}_F\ \mathsf{env}_S(A)$$

$$[\textsc{wc-call}]\ \frac{\mathsf{env}_{SV} \vdash e \to_e A \quad \mathsf{env}_{SV} \vdash \tilde{e} \to_e \tilde{v} \quad \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, \mathsf{env}'_V\rangle \to \mathsf{env}''_{SV}}{\mathsf{env}_T \vdash \langle \texttt{call}\ e.f(\tilde{e}), \mathsf{env}_{SV}\rangle \to \mathsf{env}''_S, \mathsf{env}_V}$$

where:

$$(\tilde{x}, S) = \mathsf{env}_T(A)(f)$$
$$k = |\tilde{x}| = |\tilde{v}|$$
$$\mathsf{env}'_V = (\texttt{this}, A), (x_1, v_1), \ldots, (x_k, v_k)$$

Figure 5.7: Semantics of WC statements.

$$[\text{T-ENV}^{\emptyset}]\frac{X \in \{T, S, V, M, F\}}{\Gamma \vdash \text{env}_X^{\emptyset}} \qquad [\text{T-ENV}_V]\frac{\Gamma(x) = B \quad \Gamma \vdash v : B \quad \Gamma \vdash \text{env}_V}{\Gamma \vdash (x, v), \text{env}_V}$$

$$[\text{T-ENV}_T]\frac{\Gamma \vdash_A \text{env}_M \quad \Gamma \vdash \text{env}_T}{\Gamma \vdash (A, \text{env}_M), \text{env}_T} \qquad [\text{T-ENV}_M]\frac{\begin{array}{c}\Gamma(A) = I \qquad \Gamma(I)(f) = \text{proc}(\widetilde{B}) \\ \Gamma, \tilde{x} : \widetilde{B} \vdash S \quad \Gamma \vdash_A \text{env}_M\end{array}}{\Gamma \vdash_A (f, (\tilde{x}, S)), \text{env}_M}$$

$$[\text{T-ENV}_S]\frac{\Gamma \vdash_A \text{env}_F \quad \Gamma \vdash \text{env}_S}{\Gamma \vdash (A, \text{env}_F), \text{env}_S} \qquad [\text{T-ENV}_F]\frac{\begin{array}{c}\Gamma(A) = I \quad \Gamma(I)(p) = B \\ \Gamma \vdash v : B \quad \Gamma \vdash_A \text{env}_F\end{array}}{\Gamma \vdash_A (p, v), \text{env}_F}$$

Figure 5.8: Type rules for WC-environment agreement.

$$[\text{T-VAR}]\frac{\Gamma(x) = B}{\Gamma \vdash x : B} \qquad [\text{T-OP}]\frac{\Gamma \vdash \tilde{e} : \widetilde{B} \quad \text{op} : \widetilde{B} \to B}{\Gamma \vdash \text{op}(\tilde{e}) : B}$$

$$[\text{T-FIELD}]\frac{\Gamma \vdash e : I \quad \Gamma(I)(p) = B}{\Gamma \vdash e.p : B} \qquad [\text{T-VAL}]\frac{}{\Gamma \vdash v : B}\left(B = \begin{cases}\text{int} & \text{if } v \in \mathbb{Z} \\ \text{bool} & \text{if } v \in \mathbb{B} \\ \Gamma(v) & \text{if } v \in \mathcal{N}\end{cases}\right)$$

Figure 5.9: Type rules for WC-expressions

$$[\text{T-SKIP}]\frac{}{\Gamma \vdash \texttt{skip}}$$

$$[\text{T-SEQ}]\frac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1 ; S_2}$$

$$[\text{T-CALL}]\frac{\Gamma \vdash e : I \quad \Gamma \vdash \tilde{e} : \widetilde{B}}{\Gamma \vdash \texttt{call}\ e.f(\tilde{e})}$$
where:
$\Gamma(I)(f) = \text{proc}(\widetilde{B})$

$$[\text{T-DECV}]\frac{\Gamma \vdash e : B \quad \Gamma, x : B \vdash S}{\Gamma \vdash \texttt{var}\ B\ x\ \texttt{:=}\ e\ \texttt{in}\ S}$$

$$[\text{T-ASSF}]\frac{\Gamma \vdash \texttt{this}.p : B \quad \Gamma \vdash e : B}{\Gamma \vdash \texttt{this}.p\ \texttt{:=}\ e}$$

$$[\text{T-IF}]\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash S_\text{T} \quad \Gamma \vdash S_\text{F}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ S_\text{T}\ \texttt{else}\ S_\text{F}}$$

$$[\text{T-ASSV}]\frac{\Gamma \vdash x : B \quad \Gamma \vdash e : B}{\Gamma \vdash x\ \texttt{:=}\ e}$$

$$[\text{T-WHILE}]\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash S}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ S}$$

Figure 5.10: Type rules for WC-statements.

since it is obvious. We can then show the following theorem, which assures us that the type system is sound:

Theorem 11 (Subject reduction). *Let* $\Gamma \vdash \text{env}_{TSV}$. *If* $\Gamma \vdash S$ *and* $\text{env}_T \vdash \langle S, \text{env}_{SV}\rangle \to \text{env}'_{SV}$, *then* $\Gamma \vdash \text{env}'_{SV}$.

The proof is by induction on the inference for $\text{env}_T \vdash \langle S, \text{env}_{SV}\rangle \to \text{env}'_{SV}$. It can be found in Section 5.8.5. This is the 'expectable' result for a simple type system for a language with a big-step semantics, i.e. that a well-typed program, if it terminates when given a well-typed 'input' (in the form of the environments env_{SV}), will produce a well-typed output (environments env'_{SV}).

5.6 Encoding WC in $^e\pi$

We shall now see how to represent the WC-language in the $^e\pi$-calculus. In WC, we already assume that all field names are distinct from each other (within a class), and likewise for all method names. Our strategy is to let all class names A, field names p, variable names x (including this), and method names f, be names drawn from the set of $^e\pi$ names $\mathcal{N}$, and then create composite names to scope the methods and fields, such that a field p in a class A will get the composite name $A \cdot p$; and likewise, a method f will now get the composite name $A \cdot f$.

To encode imperative variables (and fields) and the memory store (and state) represented by the environments env_{SV}, we shall use a technique proposed in [57], where they show how a reference ℓ (i.e., a location/address) can be represented in

the π-calculus simply by means of an asynchronous output:

$$(\boldsymbol{\nu}\ell\ :\ B\ :=\ e)P \triangleq (\boldsymbol{\nu}\ell\ :\ B)\,(\ell\texttt{<}e\texttt{>}\ \mid\ P)$$
$$\ell \rhd (y).P \triangleq \ell(y).(\ell\texttt{<}y\texttt{>}\ \mid\ P)$$
$$\ell \lhd \texttt{<}e\texttt{>}.P \triangleq \ell(x).(\ell\texttt{<}e\texttt{>}\ \mid\ P)$$

where $x \notin fn(P, e)$. Here, $(\boldsymbol{\nu}\ell\ :\ B\ :=\ e)P$ declares a new reference ℓ of type B in P, which is initialised with the expression e; then, $\ell \rhd (y).P$ reads the contents of ℓ (non-destructively) and binds it to y within P; and $\ell \lhd \texttt{<}e\texttt{>}.P$ writes the expression e into ℓ, overwriting the previous value, and continues as P.

In [57], ℓ is just a single π-calculus name, but as we are using this representation in ${}^{e}\pi$, the reference will be a *vector* of names. Thus, we can encode both local variables and fields, where, in the latter case, ℓ will be the vector $A \cdot p$, representing the field named p in the class A. Note however that, unlike local variables (which are private), fields are publicly visible; therefore, their declaration must be represented as a plain output $A \cdot p\texttt{<}e\texttt{>}$, i.e. without restricting them. This is safe, since we assume that the field names and method names are unique within each class, so the encoding of fields cannot clash with the encoding of method calls.

In the type system for WC, there is no distinction between a value of type B and a variable (or field) capable of *containing* such a value; however, given our representation of variables as outputs in ${}^{e}\pi$, we must now explicitly distinguish between the two. A value v will have a base type B as in WC, but variables (and fields) *storing* such a value will now be given a 'container type' $I^{B} \mapsto (\mathsf{ch}(B), \varnothing)$. Thus, a variable storing e.g. a boolean value in WC can now be given the type $I^{\mathsf{bool}} \mapsto (\mathsf{ch}(\mathsf{bool}), \varnothing)$, matching the encoding in [57] of a channel capable of communicating a single boolean value. We shall assume one such type is defined for each base type B used in the WC-program to be translated. Furthermore, we shall need one other type, $I^{\mathsf{ret}} \mapsto (\mathsf{ch}(), \varnothing)$, called the *return type*. As shown below, it denotes a name that can only be used for pure synchronisation signals; we use such names in the encoding to control the execution of sequential compositions and to signal the return from method calls.

The encoding must also be parameterised with a type environment Γ of WC-types, containing the interface definitions for the program; this is necessary because we are using a typed syntax, and the encoding will introduce some auxiliary names, which must therefore be given a type. Note, however, that the encoding itself will not depend on Γ: had we used an untyped syntax, it could have been omitted.

Finally, recall that the semantics of WC is given in terms of transitions of the form $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$, since env_T is never modified during transitions. However, for the purpose of the encoding, we shall slightly change this format and instead write $\langle S, \mathsf{env}_{TSV}\rangle \to \mathsf{env}'_{TSV}$, since all elements of the configuration, including env_T, must be encoded. Likewise, we shall write transitions for expressions as $\langle e, \mathsf{env}_{SV}\rangle \to_e \langle v, \mathsf{env}_{SV}\rangle$. The top-level call to the translation function for initial

and final configurations is then:

$$[\![\langle S, \mathsf{env}_{TSV}\rangle]\!]^\Gamma = [\![\mathsf{env}_T]\!]^\Gamma \mid [\![\mathsf{env}_S]\!] \mid (\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![\mathsf{env}_V]\!]^\Gamma\big[[\![S]\!]^\Gamma_r\big] \mid r().\mathbf{0}\big)$$
$$[\![\mathsf{env}_{TSV}]\!]^\Gamma = [\![\mathsf{env}_T]\!]^\Gamma \mid [\![\mathsf{env}_S]\!] \mid [\![\mathsf{env}_V]\!]^\Gamma[\mathbf{0}]$$

where the notation $[\![\mathsf{env}_V]\!]^\Gamma\big[[\![S]\!]^\Gamma_r\big]$ (resp. $[\![\mathsf{env}_V]\!]^\Gamma[\mathbf{0}]$) indicates that the variable environment env_V is translated as a *process context* containing a single hole, written [ ], into which the translation of the statement $[\![S]\!]^\Gamma_r$ (resp. the stopped process $\mathbf{0}$) is inserted. We also declare a new name r, i.e. the *return signal*, which is passed as a parameter to the encoding of S, and await an input on this name, which signals that the program has finished. The two other environments are translated as processes running in parallel; their encoding is given below:

$$[\![\mathsf{env}_V^\varnothing]\!]^\Gamma = [\,] \qquad [\![(p, v), \mathsf{env}_F]\!]_A = [\![\mathsf{env}_F]\!]_A \mid A \cdot p\langle v\rangle$$
$$[\![\mathsf{env}_F^\varnothing]\!]_A = \mathbf{0} \qquad [\![(f, (\tilde{x}, S)), \mathsf{env}_M]\!]^\Gamma_A = [\![\mathsf{env}_M]\!]^\Gamma_A \mid\, !A \cdot f(r, \tilde{a})$$
$$.(\boldsymbol{\nu}\tilde{x} : I^{\widetilde{B}} := \tilde{a})(\boldsymbol{\nu}\mathtt{this} : I^{\Gamma(A)} := A)\big([\![S]\!]^\Gamma_r\big)$$
$$\text{where } \Gamma(A) = I_A \text{ and } \Gamma(I_A)(f) = \mathrm{proc}(\widetilde{B})$$
$$[\![\mathsf{env}_M^\varnothing]\!]^\Gamma_A = \mathbf{0} \qquad [\![(A, \mathsf{env}_M), \mathsf{env}_T]\!]^\Gamma = [\![\mathsf{env}_T]\!]^\Gamma \mid [\![\mathsf{env}_M]\!]^\Gamma_A$$
$$[\![\mathsf{env}_T^\varnothing]\!]^\Gamma = \mathbf{0} \qquad [\![(A, \mathsf{env}_F), \mathsf{env}_S]\!] = [\![\mathsf{env}_S]\!] \mid [\![\mathsf{env}_F]\!]_A$$
$$[\![\mathsf{env}_S^\varnothing]\!] = \mathbf{0} \qquad [\![(x, v), \mathsf{env}_V]\!]^\Gamma = (\boldsymbol{\nu} x : I^{\Gamma(x)} := v)\big([\![\mathsf{env}_V]\!]^\Gamma\big)$$

There are a few points to note: First, the local variable bindings in env_V are translated as a process context consisting of a series of declarations and with the empty environment translated as the hole [ ]; thus, the declarations bind the translated statement $[\![S]\!]^\Gamma_r$, which is inserted into the hole. Second, the encoding of the environments containing the field and method declarations (i.e., env_F and env_M) are both parameterised with the name of the current class, A, which is used to compose the subject vector of the outputs. Third, a method declaration $(f, (\tilde{x}, S))$ is translated as an input-guarded replication $!A \cdot f(r, \tilde{a})$ to make it persistent. The actual parameters will be received and bound to $\tilde{a}$, and these, in turn, are then bound to the formal parameters $\tilde{x}$ in a list of declarations $(\boldsymbol{\nu}\tilde{x} : \widetilde{B} := \tilde{a})$. This extra re-binding step is necessary, since the formal parameters should act as *variables* (rather than values) within the body, so they too must follow the protocol of the encoding in [57]. Likewise, we also add a binding for the special variable `this`, which is available within the body S, so it can be used to access the fields of the current class. Finally, besides the formal parameters, we also add an extra name r to be received by the input $A \cdot f(r, \tilde{a})$; this is the return signal, on which the method will signal when it finishes.

Statements are encoded as follows:

$$
\begin{aligned}
[\![\texttt{skip}]\!]^{\Gamma}_{r} &= r\texttt{<>} \\
[\![\texttt{var } B\ x\texttt{:=}e\ \texttt{in } S]\!]^{\Gamma}_{r} &= (\boldsymbol{\nu} z : I^{B})\big([\![e]\!]^{\Gamma}_{z} \mid z(y).(\boldsymbol{\nu} x : I^{B} := y)\big([\![S]\!]^{\Gamma, x:B}_{r}\big)\big) \\
[\![x := e]\!]^{\Gamma}_{r} &= (\boldsymbol{\nu} z : I^{B})\big([\![e]\!]^{\Gamma}_{z} \mid z(y).x \lhd \texttt{<}y\texttt{>}.r\texttt{<>}\big) \\
&\quad \text{where } \Gamma \vdash e : B \\
[\![\texttt{this}.p := e]\!]^{\Gamma}_{r} &= (\boldsymbol{\nu} z : I^{B})\big([\![e]\!]^{\Gamma}_{z} \mid z(y).\texttt{this} \rhd (Y).Y \cdot p \lhd \texttt{<}y\texttt{>}.r\texttt{<>}\big) \\
&\quad \text{where } \Gamma \vdash e : B \\
[\![S_1 ; S_2]\!]^{\Gamma}_{r_2} &= (\boldsymbol{\nu} r_1 : I^{\mathrm{ret}})\big([\![S_1]\!]^{\Gamma}_{r_1} \mid r_1().[\![S_2]\!]^{\Gamma}_{r_2}\big) \\
[\![\texttt{if } e \texttt{ then } S_{\mathsf{T}} \texttt{ else } S_{\mathsf{F}}]\!]^{\Gamma}_{r} &= (\boldsymbol{\nu} z : I^{\mathrm{bool}})\big([\![e]\!]^{\Gamma}_{z} \mid z(y).\textbf{if } y \textbf{ then } [\![S_{\mathsf{T}}]\!]^{\Gamma}_{r} \textbf{ else } [\![S_{\mathsf{F}}]\!]^{\Gamma}_{r}\big) \\
[\![\texttt{while } e \texttt{ do } S]\!]^{\Gamma}_{r} &= (\boldsymbol{\nu} r' : I^{\mathrm{ret}})\Big(r'\texttt{<>} \mid \texttt{!}r'().(\boldsymbol{\nu} z : I^{\mathrm{bool}}) \\
&\qquad \big([\![e]\!]^{\Gamma}_{z} \mid z(y).\textbf{if } y \textbf{ then } [\![S]\!]^{\Gamma}_{r'} \textbf{ else } r\texttt{<>}\big)\Big) \\
[\![\texttt{call } e.f(\tilde{e})]\!]^{\Gamma}_{r} &= (\boldsymbol{\nu} a : I^{I_A}, \tilde{z} : I^{\widetilde{B}})\Big([\![e]\!]^{\Gamma}_{a} \mid [\![\tilde{e}]\!]^{\Gamma}_{\tilde{z}} \\
&\qquad \mid a(Y).z_1(y_1) \ldots z_n(y_n).Y \cdot f\texttt{<}r, y_1, \ldots, y_n\texttt{>}\Big) \\
&\quad \text{where } \Gamma \vdash e : I_A \text{ and } \Gamma \vdash \tilde{e} : \widetilde{B} \text{ and } |\tilde{e}| = |\tilde{z}| = n \\
&\quad \text{and } [\![\tilde{e}]\!]^{\Gamma}_{\tilde{z}} = [\![e_1]\!]^{\Gamma}_{z_1} \mid \ldots \mid [\![e_n]\!]^{\Gamma}_{z_n}
\end{aligned}
$$

Most of this encoding is straightforward: `skip` does nothing, so it simply emits the return signal. In a sequential composition $S_1 ; S_2$, we first declare a new return signal r', which is passed to the first component; then, we wait a synchronisation on this name before executing the second component, which then will emit a signal on r. The `if-then-else` construct maps directly to our syntactic sugar for guarded choice, with both branches receiving r, since only one of them is chosen. We also use this in the encoding of `while`, which is just a guarded replication of an `if-then-else`: the true-branch will emit a return signal on a fresh name r', which will trigger another replication; the false-branch will emit a return signal on the outer return name r. Lastly, in method calls, r is passed as the first parameter.

Note that, in each construct containing an expression e, we translate e as a separate process, rather than mapping it directly to an expression in ${}^{e}\pi$. This, again, is a consequence of our use of the encoding from [57]: since expressions in `WC` can contain variables, these must be read in accordance with the protocol. The translation function for expressions is parametrised with a name z, on which the value resulting from evaluating the expression will be delivered. It is defined as follows:

$$
\begin{aligned}
[\![v]\!]^{\Gamma}_{z} &= z\texttt{<}v\texttt{>} \\
[\![x]\!]^{\Gamma}_{z} &= x \rhd (y).z\texttt{<}y\texttt{>}
\end{aligned}
$$

$$
\begin{aligned}
[\![e.p]\!]^{\Gamma}_{z} &= (\boldsymbol{\nu} z' : I^{I_A})\big([\![e]\!]^{\Gamma}_{z'} \mid z'(Y).Y \cdot p \triangleright (y).z\langle y\rangle\big) \\
&\quad \text{where } \Gamma \vdash e : I_A \\
[\![\mathrm{op}(e_1, \ldots, e_n)]\!]^{\Gamma}_{z} &= (\boldsymbol{\nu} z_1 : I^{B_1}, \ldots, z_n : I^{B_n})\big([\![e_1]\!]^{\Gamma}_{z_1} \mid \ldots \mid [\![e_n]\!]^{\Gamma}_{z_n} \\
&\qquad \mid z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle\big) \\
&\quad \text{where } \Gamma \vdash e_1 : B_1, \ldots, \Gamma \vdash e_n : B_n
\end{aligned}
$$

Finally we can give the translation of the types and the type environment Γ. This is complicated by the fact that we now need to add a container type I^B for each variable and field of type B, and for each method of type $\mathrm{proc}(\widetilde{B})$. We use a multi-level encoding, and we assume for the sake of simplicity that duplicate entries in the output are ignored:

$$
\begin{aligned}
[\![x : B, \Gamma]\!] &= x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma]\!] \\
[\![A : I, \Gamma]\!] &= A : I, [\![\Gamma]\!] \\
[\![I : \Delta, \Gamma]\!] &= I : (\mathsf{nil}, ([\![\Delta]\!]^2)), [\![\Delta]\!]^3, [\![\Gamma]\!] \\
[\![\epsilon]\!] &= \epsilon
\end{aligned}
$$

$$
\begin{aligned}
[\![p : B, \Delta]\!]^2 &= p : I^B, [\![\Delta]\!]^2 \\
[\![f : \mathrm{proc}(\widetilde{B}), \Delta]\!]^2 &= f : I^{\widetilde{B}}, [\![\Delta]\!]^2 \\
[\![\epsilon]\!]^2 &= \epsilon
\end{aligned}
$$

$$
\begin{aligned}
[\![p : B, \Delta]\!]^3 &= I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3 \\
[\![f : \mathrm{proc}(\widetilde{B}), \Delta]\!]^3 &= I^{\widetilde{B}} : (\mathsf{ch}(I^{\mathrm{ret}}, \widetilde{B}), \varnothing), [\![\Delta]\!]^3 \\
[\![\epsilon]\!]^3 &= \epsilon
\end{aligned}
$$

Our main result now states that the two type systems exactly correspond: well-typed WC-programs are mapped to well-typed ${}^{e}\pi$-processes, and vice versa for ill-typed programs. Thus we know that the ${}^{e}\pi$-type system exactly captures the notion of well-typedness in WC.

Theorem 12 (Type correspondence)**.** $\Gamma \vdash \mathsf{env}_{TSV}$ *and* $\Gamma \vdash S$ *iff* $[\![\Gamma]\!] \vdash [\![\langle S, \mathsf{env}_{TSV}\rangle]\!]^{\Gamma}$.

The proof of this theorem relies on a collection of lemmas (Lemmas 40–47), which shows the type correspondence between each of the individual elements in a WC configuration and its corresponding encoding. These lemmas, and their proofs, can be found in Section 5.8.6. Using these lemmas, we can then prove the type correspondence theorem:

Proof of Theorem 12. We have two directions to prove. For the forward direction, we must show that $\Gamma \vdash \mathsf{env}_T$ and $\Gamma \vdash \mathsf{env}_S$ and $\Gamma \vdash \mathsf{env}_V$ and $\Gamma \vdash S$ together imply that

$[\![\Gamma]\!] \vdash [\![\langle S, \mathsf{env}_{TSV}\rangle]\!]^{\Gamma}$. The translation for initial configurations yields:

$$[\![\langle S, \mathsf{env}_{TSV}\rangle]\!]^{\Gamma} = [\![\mathsf{env}_T]\!]^{\Gamma} \mid [\![\mathsf{env}_S]\!] \mid (\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![\mathsf{env}_V]\!]^{\Gamma}\,[\![S]\!]^{\Gamma}_r\big] \mid r().\mathbf{0}\big)$$

Assume $[\![\Gamma]\!] \vdash P$ for some process P. We then have the following:

$$\begin{aligned}
\Gamma \vdash S &\implies [\![\Gamma]\!], r : I^{\mathrm{ret}} \vdash [\![S]\!]^{\Gamma}_r && \text{by Lemma 42}\\
\Gamma \vdash \mathsf{env}_T &\implies [\![\Gamma]\!] \vdash [\![\mathsf{env}_T]\!]^{\Gamma} && \text{by Lemma 44}\\
\Gamma \vdash \mathsf{env}_S &\implies [\![\Gamma]\!] \vdash [\![\mathsf{env}_S]\!] && \text{by Lemma 46}\\
\Gamma \vdash \mathsf{env}_V &\implies [\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^{\Gamma}\,[P] && \text{by Lemma 47}
\end{aligned}$$

We can then conclude the following:

$$\begin{aligned}
&[\![\Gamma]\!] \vdash (\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![S]\!]^{\Gamma}_r \mid r().\mathbf{0}\big) && \text{by [T-IN], [T-PAR], [T-RES]}\\
&[\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^{\Gamma}\big[(\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![S]\!]^{\Gamma}_r \mid r().\mathbf{0}\big)\big]\\
&[\![\Gamma]\!] \vdash (\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![\mathsf{env}_V]\!]^{\Gamma}\,[\![S]\!]^{\Gamma}_r\big] \mid r().\mathbf{0}\big)
\end{aligned}$$

where the last step follows, because we know that $r\#[\![\mathsf{env}_V]\!]^{\Gamma}$, so the scope can be extruded. Then we can conclude that

$$[\![\Gamma]\!] \vdash [\![\mathsf{env}_T]\!]^{\Gamma} \mid [\![\mathsf{env}_S]\!] \mid (\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![\mathsf{env}_V]\!]^{\Gamma}\,[\![S]\!]^{\Gamma}_r\big] \mid r().\mathbf{0}\big)$$

by [T-RES] and [T-PAR], as desired.

For the other direction, we must show that $[\![\Gamma]\!] \vdash [\![\langle S, \mathsf{env}_{TSV}\rangle]\!]^{\Gamma}$ implies that $\Gamma \vdash \mathsf{env}_T$ and $\Gamma \vdash \mathsf{env}_S$ and $\Gamma \vdash \mathsf{env}_V$ and $\Gamma \vdash S$. First, we unfold the translation of the initial configuration, which yields:

$$[\![\langle S, \mathsf{env}_{TSV}\rangle]\!]^{\Gamma} = [\![\mathsf{env}_T]\!]^{\Gamma} \mid [\![\mathsf{env}_S]\!] \mid (\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![\mathsf{env}_V]\!]^{\Gamma}\,[\![S]\!]^{\Gamma}_r\big] \mid r().\mathbf{0}\big)$$

and thus we know that

$$[\![\Gamma]\!] \vdash [\![\mathsf{env}_T]\!]^{\Gamma} \mid [\![\mathsf{env}_S]\!] \mid (\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![\mathsf{env}_V]\!]^{\Gamma}\,[\![S]\!]^{\Gamma}_r\big] \mid r().\mathbf{0}\big)$$

which must have been concluded by the rules [T-IN], [T-PAR] and [T-RES]. From their premises, we get

$$\begin{aligned}
&[\![\Gamma]\!] \vdash [\![\mathsf{env}_T]\!]^{\Gamma}\\
&[\![\Gamma]\!] \vdash [\![\mathsf{env}_S]\!]\\
&[\![\Gamma]\!] \vdash (\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![\mathsf{env}_V]\!]^{\Gamma}\,[\![S]\!]^{\Gamma}_r\big] \mid r().\mathbf{0}\big)
\end{aligned}$$

As we know that the name r does not occur in $[\![\mathsf{env}_V]\!]^{\Gamma}$, since the name is introduced by the translation, we can also conclude that $[\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^{\Gamma}\big[(\boldsymbol{\nu} r : I^{\mathrm{ret}})\big([\![S]\!]^{\Gamma}_r \mid r().\mathbf{0}\big)\big]$ by intruding the scope.

By the encoding of env_V, we have that it is translated as a process context consisting of a list of scopes and bound outputs

$$(\boldsymbol{\nu} x_1 : I^{\Gamma(x_1)}, \ldots, x_n : I^{\Gamma(x_n)})\,(x_1\langle v_1\rangle \mid \ldots \mid x_n\langle v_n\rangle \mid [\,])$$

so $[\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^\Gamma \left[(\boldsymbol{\nu} r : I^{\mathrm{ret}})\left([\![S]\!]^\Gamma_r \mid r().\mathbf{0}\right)\right]$ is concluded by [T-RES] and [T-PAR], and from the premise of the latter, we get that

$$[\![\Gamma]\!], x_1 : I^{\Gamma(x_1)}, \ldots, x_n : I^{\Gamma(x_n)} \vdash (\boldsymbol{\nu} r : I^{\mathrm{ret}})\left([\![S]\!]^\Gamma_r \mid r().\mathbf{0}\right)$$

However, by Lemma 41, applied to $[\![\Gamma]\!], x : I^{\Gamma(x)} \vdash x$, we also know that

$$\Gamma, x : \Gamma(x) \vdash x : \Gamma(x),$$

hence Γ must already contain a type entry B for x, which is used in the encoding of env_V given above. Then by the encoding of Γ

$$[\![x : B, \Gamma']\!] = x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!]$$

so the type entries $x : I^B$ are already in $[\![\Gamma]\!]$. Thus, we can simplify the statement above as

$$[\![\Gamma]\!] \vdash (\boldsymbol{\nu} r : I^{\mathrm{ret}})\left([\![S]\!]^\Gamma_r \mid r().\mathbf{0}\right)$$

which must be inferred by [T-RES], and from its premise, we have that

$$[\![\Gamma]\!], r : I^{\mathrm{ret}} \vdash [\![S]\!]^\Gamma_r \mid r().\mathbf{0}$$

Now, let $P \triangleq (\boldsymbol{\nu} r : I^{\mathrm{ret}})\left([\![S]\!]^\Gamma_r \mid r().\mathbf{0}\right)$ and note that we know that $[\![\Gamma]\!] \vdash P$. Finally, we have that

$$\begin{aligned}
[\![\Gamma]\!] \vdash [\![\mathsf{env}_T]\!]^\Gamma &\implies \Gamma \vdash \mathsf{env}_T && \text{by Lemma 44}\\
[\![\Gamma]\!] \vdash [\![\mathsf{env}_S]\!] &\implies \Gamma \vdash \mathsf{env}_S && \text{by Lemma 46}\\
[\![\Gamma]\!], r : I^{\mathrm{ret}} \vdash [\![S]\!]^\Gamma_r &\implies \Gamma \vdash S && \text{by Lemma 42}\\
[\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^\Gamma\,[P] &\implies \Gamma \vdash \mathsf{env}_V && \text{by Lemma 47}
\end{aligned}$$

which allows us to conclude that $\Gamma \vdash \mathsf{env}_T$, $\Gamma \vdash \mathsf{env}_S$, $\Gamma \vdash \mathsf{env}_V$ and $\Gamma \vdash S$, as desired. □

5.6.1 On the quality of the encoding

Theorem 12 above only speaks of the correspondence between the two type systems, not of the correctness of the encoding itself. However, once we define an encoding, it is natural to also inquire about its quality, since there is by now a well-established

literature on the topic (see [127; 128; 48; 49; 97; 99], just to mention the most methodological papers). We conjecture that the encoding presented in this chapter satisfies most of the properties a 'good encoding' should have, in particular operational correspondence and divergence sensitiveness. However, proving these properties for an encoding between languages that rely on different styles of semantics (small-step for ${}^{e}\pi$ vs. big-step for WC) is a non-trivial task. For example, the definition of when a statement diverges under a big-step operational semantics is in general not straightforward (see, e.g., [30; 71; 105] on this topic). We realise that for WC, divergence coincides with not having any (big-step) evolution; however, even with this simplifying feature, proving divergence reflection turned out to be technically challenging. Since the quality of our encoding is an orthogonal issue to the main aim of this chapter (that is the study of typing composite names), we prefer to leave this investigation for future work. One possibility is to keep the big-step semantics for WC and follow the path put forward in [34]; alternatively, we can abandon the (more straightforward) big-step semantics in favour of a small-step one, which would simplify the proofs for the quality of the encoding at the price of complicating those for subject reduction in WC.

5.7 Conclusion and future works

In the present chapter we have explored the question of how to create a type system with subtyping for a language with composite channels, when such channels can be composed at runtime. We conjecture that the core of our type system is equivalent to the nominal type system given by Carbone in [26, Chapter 6.5], but the structure of our composite types is different, which yields a more intuitive correspondence with the concept of interfaces known from OO languages, as is made clear through our encoding of a small, class-based language. In that sense, our work is in line with the remark by Milner in [82, p. 120], where he notes the connexion between the π-calculus and object-oriented programming: both are characterised by transmitting *access* to agents (resp. objects), rather than the agents themselves. The polyadic synchronisation of ${}^{e}\pi$ makes this connexion even stronger, as evidenced by the simplicity of our encoding, since it yields a natural way to represent fields and methods, scoped under a common class name, as synchronisation vectors with a common prefix. Our 'tree-shaped' types then follow naturally from this similarity.

As mentioned in Section 5.2, our work relates to that of Hüttel [61; 62; 63], since these papers did not provide a general solution to the question of how to type composite subjects. It is therefore also worth emphasising here that our type system can be used to type the encoding of Dπ, even in the cases where the location name or the located subject name might be bound. This can be done by creating types for the locations l of the form

$$I_l \mapsto (\mathsf{nil}, (I_1 \mapsto (\mathsf{ch}(B_1), \varnothing), \dots, I_n \mapsto (\mathsf{ch}(B_n), \varnothing)))$$

and with the types I_i assigned to the subjects x_i occurring at location l. This closely corresponds to our encoding of the WC-types, with location names l corresponding to class names A. The WC-language could in principle also have been encoded directly in Dπ, since the encoding only makes use of the π^2 fragment of ${}^e\pi$. However, WC could easily be extended to allow nested class declarations, as is found in some object-oriented languages, i.e. with declarations of the form

$$DC ::= \epsilon \mid \texttt{class}\ A\ \{\ DC\ \ DF\ \ DM\ \}$$

which would then necessitate subject vectors of arbitrary length to encode an arbitrary nesting depth. In the present chapter, we have forgone this possibility for the sake of simplicity. Nevertheless, our correspondence results contribute to further the understanding of how a typed, object-oriented language may be represented in the setting of typed process calculi, whilst still preserving some of the structure afforded by interface types in such languages.

A natural further step in this direction would be to consider subtyping, as was done for the π-calculus by Pierce and Sangiorgi [100], who distinguish between the input and output capabilities of a channel. Indeed, such a subtyping relation is also assumed (to be provided as a parameter) in the type systems by Hüttel for the Ψ-calculus, but none of the instance examples provide any hints as to how it might be defined. The type system for ${}^e\pi$ in [26] also does not have a subtyping relation, so the question of how it might be defined is entirely open.

Input/output capabilities can of course easily be introduced for the channel capability component C of our types, so the interesting question in this regard is rather what a subtyping relation might look like for the *composition capability* component Δ. Again, the correspondence with object-oriented languages may provide some intuitions. For example, it would seem natural to require that the composition capabilities of the subtype Δ_1 should be *at least* the same as the supertype Δ_2; or that each type name I in Δ_1 again should be a subtype of an entry in Δ_2. This would correspond to the usual understanding from object-oriented languages, where an object O_1 is a subtype of another object O_2, if O_1 contains at least the same public fields and methods as O_2, and the type of each such field, resp. method, in O_1 again is a subtype of the type of the corresponding field, resp. method, in O_2.

5.8 Proofs

5.8.1 Proof of Theorem 9

Proof. By induction on (the height of) the judgement $\Gamma \vdash P$. The base case is when [T-NIL] has been used and it is trivial, since the error predicate has no rule for $\mathbf{0}$. For the inductive step, we reason by case analysis on the last type rule used for concluding $\Gamma \vdash P$.

- [T-PAR]: From the premise, we know that $\Gamma \vdash P_1$ and $\Gamma \vdash P_2$. By the induction hypothesis, this implies that $\mathsf{NSafe}_\Gamma(P_1)$ and $\mathsf{NSafe}_\Gamma(P_2)$, hence $\neg\mathsf{Wrong}_\Gamma(P_1)$ and $\neg\mathsf{Wrong}_\Gamma(P_2)$. The only rule, that can conclude an error in a parallel composition is [W-PAR], but by the above, none of the premises can hold. Thus we conclude that $\mathsf{NSafe}_\Gamma(P_1 \mid P_2)$.

- [T-REP]: From the premise, we have that $\Gamma \vdash P$. By the induction hypothesis, this implies $\mathsf{NSafe}_\Gamma(P)$, hence $\neg\mathsf{Wrong}_\Gamma(P)$. To conclude that $!P$ is not safe, we can only use [W-REP], but by the above, the premise cannot hold. Thus we conclude that $\mathsf{NSafe}_\Gamma(!P)$.

- [T-RES]: From the premise, we have that $\Gamma, \tilde{x} : \widetilde{B} \vdash P$, which by the induction hypothesis implies that $\mathsf{NSafe}_{\Gamma,\tilde{x}:\widetilde{B}}(P)$, hence $\neg\mathsf{Wrong}_{\Gamma,\tilde{x}:\widetilde{B}}(P)$. The only rule that can be used to conclude that $\mathsf{Wrong}_\Gamma\big((\boldsymbol{\nu}\tilde{x} : \widetilde{B})P\big)$ is [W-RES], but by the above, the premise cannot hold. Thus we conclude that $\mathsf{NSafe}_\Gamma\big((\boldsymbol{\nu}\tilde{x} : \widetilde{B})P\big)$.

- [T-SUM]: From the premise, we have that every guard e_i must have a boolean type, and every P_i is such that $\Gamma \vdash P_i$. By the induction hypothesis, this implies $\mathsf{NSafe}_\Gamma(P_i)$, hence $\neg\mathsf{Wrong}_\Gamma(P_i)$. The only rule that can be used to conclude $\mathsf{Wrong}_\Gamma\big(\sum[\tilde{e}]\widetilde{P}\big)$ is [W-SUM], but by the above the premise cannot hold for every i. Thus we conclude that $\mathsf{NSafe}_\Gamma\big(\sum[\tilde{e}]\widetilde{P}\big)$.

- [T-IN]: From the premise, we know that $\Gamma;\Gamma \vdash \tilde{x} : \mathsf{ch}(\widetilde{B})$, and $\Gamma, \tilde{y} : \widetilde{B} \vdash P$, where the latter implies that $|\tilde{y}| = |\widetilde{B}|$, since every name must be given a type. The only rule that can be used to conclude an error for input is [W-IN], but by the above, the premises cannot hold. Thus we conclude that $\mathsf{NSafe}_\Gamma(\tilde{x}(\tilde{y}).P)$.

- [T-OUT]: From the premise we have that $\Gamma \vdash \tilde{x} : \mathsf{ch}(\widetilde{B})$ and $\Gamma \vdash \tilde{e} : \widetilde{B}$. The only rule that can be used to conclude an error for an output is [W-OUT], but by the above, none of the premises can hold. Thus we conclude that $\mathsf{NSafe}_\Gamma(\tilde{x}\langle\tilde{e}\rangle.P)$. □

5.8.2 Auxiliary lemmas for the proof of Theorem 10

The proof of Theorem 10 relies on some standard lemmas, that are all easily shown by induction on the typing judgement. Weakening and strengthening say that we can add and remove unused typing assumptions. Substitution says that we can replace a list of names by a list of values (which may include names), as long as the two lists have pointwise matching types; hence, no well-typed substitution can alter the type of (a vector of) names.

Lemma 27 (Weakening). *If* $\Gamma \vdash P$ *and* $x \notin \mathrm{fn}(P)$*, then* $\Gamma, x : B \vdash P$.

Lemma 28 (Strengthening). *If* $\Gamma, x : B \vdash P$ *and* $x \notin \mathrm{fn}(P)$*, then* $\Gamma \vdash P$.

Definition 40 (Well-typed substitution). We say a substitution σ is well-typed, written $\Gamma \vdash \sigma$, if $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\Gamma)$ and $\forall x \in \mathrm{dom}(\sigma)\ .\ \Gamma \vdash \sigma(x) : \Gamma(x)$. ■

Lemma 29 (Substitution). *If* $\Gamma \vdash P$ *and* $\Gamma \vdash \sigma$*, then* $\Gamma \vdash P\sigma$.

The next lemma says that expression evaluation respects typing; this is shown by induction on the rules of $\rightarrow_e$, which we have omitted since they depend on the operators op that we choose to admit in the language.

Lemma 30 (Safety for expressions). *If* $\Gamma \vdash \tilde{e} : \widetilde{B}$ *and* $\tilde{e} \rightarrow_e \tilde{v}$ *then* $\Gamma \vdash \tilde{v} : \widetilde{B}$

Lastly, we show that a well-typed process can only originate well-typed (output/τ) labels:

Lemma 31 (Well-typed labels). *If* $\Gamma \vdash P$ *and* $P \xrightarrow{\alpha} P'$*, for* $\alpha = \tau$ *or* $\alpha = \tilde{x}!(\boldsymbol{\nu}\tilde{z} : \widetilde{B})\tilde{v}$*, then* $\Gamma \vdash \alpha$.

Proof sketch. If the label is τ, then the result is immediate, since $\Gamma \vdash \tau$ for any Γ by [T-TAU]. Otherwise, we have a number of cases to examine, depending on which rule was used to conclude the transition. However, the two most interesting cases are for the output and open rules:

- If [E-OUT] was used, then the conclusion is of the form $\tilde{x}\langle\tilde{e}\rangle.P \xrightarrow{\tilde{x}!\tilde{v}} P$. The label is a free output. From the premise we have that $\tilde{e} \rightarrow_e \tilde{v}$. Now $\Gamma \vdash \tilde{x}\langle\tilde{e}\rangle.P$ must have been concluded by [T-OUT], and from its premises we know that $\Gamma; \Gamma \vdash \tilde{x} : \mathrm{ch}(\widetilde{B})$ and $\Gamma \vdash \tilde{e} : \widetilde{B}$. By Lemma 30, we then have that $\Gamma \vdash \tilde{v} : \widetilde{B}$. Hence, by [T-SND], we have that $\Gamma \vdash \tilde{x}!\tilde{v}$.
- If [E-OPEN] was used, then the conclusion is of the form

$$(\boldsymbol{\nu}\tilde{z} : \widetilde{B}_z)P \xrightarrow{\tilde{x}!(\boldsymbol{\nu}\tilde{y},\tilde{z}:\widetilde{B}_y,\widetilde{B}_z)\tilde{v}} P'$$

From the premise and side condition, we have that $P \xrightarrow{\tilde{x}!(\boldsymbol{\nu}\tilde{y}:\widetilde{B}_y)\tilde{v}} P'$, $\tilde{z} \subseteq \tilde{v}$ and $\tilde{z} \# \tilde{x}, \tilde{y}$. Now $\Gamma \vdash (\boldsymbol{\nu}\tilde{z} : \widetilde{B}_z)P$ must have been concluded by [T-RES], and from its premise we know that $\Gamma, \tilde{z} : \widetilde{B}_z \vdash P$. By the induction hypothesis, we then have that $\Gamma, \tilde{z} : \widetilde{B}_z \vdash \tilde{x}!(\boldsymbol{\nu}\tilde{y} : \widetilde{B}_y)\tilde{v}$. We can then conclude $\Gamma \vdash \tilde{x}!(\boldsymbol{\nu}\tilde{y}, \tilde{z} : \widetilde{B}_y, \widetilde{B}_z)\tilde{v}$ by [T-SND].

For the remaining α-labelled rules ([E-PAR$_1$], [E-SUM], [E-RES] and [E-REP]), the result follows directly from the induction hypothesis. For [E-IN], the statement holds vacuously, since the label is not an output- nor a τ-label. □

5.8.3 Proof of Theorem 10

Proof. By induction on the inference for $P \xrightarrow{\alpha} P'$. The base case can be inferred in two ways:

- If [E-IN] was used. Then the conclusion is of the form

$$\tilde{x}(\tilde{y}).P \xrightarrow{\tilde{x}?\tilde{v}} P\{\tilde{v}/\tilde{y}\}$$

  Now $\Gamma \vdash \tilde{x}(\tilde{y}).P$ must have been concluded by [T-IN], and from its premise we know that $\Gamma; \Gamma \vdash \tilde{x} : \mathsf{ch}(\widetilde{B})$ and $\Gamma, \tilde{y} : \widetilde{B} \vdash P$. By assumption, $\Gamma \vdash \tilde{x}?\tilde{v}$, which must have been concluded by [T-RCV], and from the premise we know that $\Gamma \vdash \tilde{v} : \widetilde{B}$. By Lemma 29, we can therefore conclude that $\Gamma, \tilde{y} : \widetilde{B} \vdash P\{\tilde{v}/\tilde{y}\}$. As $\tilde{y} \# \mathrm{fn}(P\{\tilde{v}/\tilde{y}\})$, we can then conclude $\Gamma \vdash P\{\tilde{v}/\tilde{y}\}$ by Lemma 28. As we know that the label here is an input label, we therefore have that $\Gamma' = \Gamma$.

- If [E-OUT] was used. Then the conclusion is of the form

$$\tilde{x}\langle\tilde{e}\rangle.P \xrightarrow{\tilde{x}!\tilde{v}} P$$

  and from the premise we have that $\tilde{e} \to_e \tilde{v}$. Now $\Gamma \vdash \tilde{x}\langle\tilde{e}\rangle.P$ must have been concluded by [T-OUT], and from its premises we know that $\Gamma \vdash P$, thus giving us the desired conclusion. As this is a *free* output, we know that $\mathrm{bn}(\alpha) = \emptyset$; so, no new name will be added to the type environment and $\Gamma' = \Gamma$.

For the inductive step, we reason on the last rule of Figure 5.1 used to conclude the inference. Note that well-typedness for τ-labels always holds, and well-typedness for output labels is ensured by Lemma 31; so, it only needs to be assumed for input labels.

- Suppose [E-PAR$_1$] was used. Then the conclusion is of the form

$$P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2$$

and from the premise and side condition we have that $P_1 \xrightarrow{\alpha} P_1'$ and $\mathrm{bn}(\alpha) \# P_2$. Now $\Gamma \vdash P_1 \mid P_2$ must have been concluded by [T-PAR], and from the premise we have that $\Gamma \vdash P_1$ and $\Gamma \vdash P_2$, We then have by induction hypothesis that $\Gamma' \vdash P_1'$. Now, if $\alpha = \tilde{x}!(\boldsymbol{\nu}\tilde{z} : \widetilde{B})\tilde{v}$, then $\Gamma' = \Gamma, \tilde{z} : \widetilde{B}$, and we know from the side condition of [E-PAR$_1$] that $\tilde{z} \# P_2$. Thus, by Lemma 27 (weakening), we also have that $\Gamma' \vdash P_2$. Otherwise, $\Gamma' = \Gamma$. In either case, we can then conclude $\Gamma' \vdash P_1' \mid P_2$ by [T-PAR]. The symmetric case for [PAR$_2$] is similar.

- Suppose [E-SUM] was used. Then the conclusion is of the form

$$\sum [\tilde{e}]\widetilde{P} \xrightarrow{\alpha} P_i'$$

and from the premise and side condition we know that $P_i \to P_i'$ and $e_i \to_e \mathsf{T}$. Then $\Gamma \vdash \sum [\tilde{e}]\widetilde{P}$ must have been concluded by [T-SUM]. From the premise, we know that $\Gamma \vdash e_i : \mathtt{bool}$ and $\Gamma \vdash P_i$ for each $i \in 1, \ldots, |\tilde{e}|$. The former will have been concluded by the type rules for operations and [T-VAL]. We then have by induction hypothesis that $\Gamma' \vdash P_i'$, where either $\Gamma' = \Gamma$ or $\Gamma' = \Gamma, \tilde{z} : \widetilde{B}$ depending on the label.

- Suppose [E-RES] was used. Then the conclusion is of the form

$$(\boldsymbol{\nu}\tilde{x} : \widetilde{B})P \xrightarrow{\alpha} (\boldsymbol{\nu}\tilde{x} : \widetilde{B})P'$$

and from the premise and side condition we know that $P \xrightarrow{\alpha} P'$ and $\tilde{x} \# \alpha$. Now $\Gamma \vdash (\boldsymbol{\nu}\tilde{x} : \widetilde{B})P$ must have been concluded by [T-RES]. From the premise, we know that $\Gamma, \tilde{x} : \widetilde{B} \vdash P$. Therefore, by the induction hypothesis, we have that $\Gamma', \tilde{x} : \widetilde{B} \vdash P'$, where $\Gamma' = \Gamma$ or $\Gamma' = \Gamma, \tilde{z} : \widetilde{B}'$ as determined by the label. By Lemma 28 (strengthening), we can therefore conclude that $\Gamma' \vdash (\boldsymbol{\nu}\tilde{x} : \widetilde{B})P'$.

- Suppose [E-REP] was used. Then the conclusion is of the form $!P \xrightarrow{\alpha} P'$, and from the premise we know that $!P \mid P \xrightarrow{\alpha} P'$. Now $\Gamma \vdash !P$ must have been concluded by [T-REP], and from its premise we know that $\Gamma \vdash P$. Thus, by rule [T-PAR], we have that $\Gamma \vdash !P \mid P$. By induction, we conclude that $\Gamma' \vdash P'$, where either $\Gamma' = \Gamma$ or $\Gamma' = \Gamma, \tilde{z} : \widetilde{B}$ depending on the label.

- Suppose [E-OPEN] was used. Then the conclusion is of the form

$$(\boldsymbol{\nu}\tilde{z} : \widetilde{B}_z)P \xrightarrow{\tilde{x}!(\boldsymbol{\nu}\tilde{y}:\widetilde{B}_y,\tilde{z}:\widetilde{B}_z)\tilde{v}} P'$$

and from the premise we know that $P \xrightarrow{\tilde{x}!(\boldsymbol{\nu}\tilde{y}:\widetilde{B}_y)\tilde{v}} P'$. Now $\Gamma \vdash (\boldsymbol{\nu}\tilde{z} : \widetilde{B}_z)P$ must have been concluded by [T-RES], and from its premise we know that $\Gamma, \tilde{z} : \widetilde{B}_z \vdash P$. Then by induction hypothesis, we have that $\Gamma, \tilde{y} : \widetilde{B}_y, \tilde{z} : \widetilde{B}_z \vdash P'$, where $\Gamma, \tilde{y} : \widetilde{B}_y, \tilde{z} : \widetilde{B}_z = \Gamma'$.

- Suppose [E-Com$_1$] was used. Then the conclusion is of the form

$$P_1 \mid P_2 \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{z} : \widetilde{B})(P_1' \mid P_2')$$

and from the premises and side condition we know that

$$P_1 \xrightarrow{x!(\boldsymbol{\nu}\tilde{z}:\widetilde{B})\tilde{v}} P_1'$$
$$P_2 \xrightarrow{\tilde{x}?\tilde{v}} P_2'$$

and $\tilde{z}\#P_2$. Now $\Gamma \vdash P_1 \mid P_2$ must have been concluded by [T-par], and from its premises we know that $\Gamma \vdash P_1$ and $\Gamma \vdash P_2$. We now have two cases to examine:

1. If the list of bound names $\tilde{z}$ is empty, then P_1 performs a *free* output, and no new name is added to the type environment. Then $\Gamma' = \Gamma$, and thus by the induction hypothesis we have that $\Gamma \vdash P_1'$.
   In the case of P_2, we know it performs an input, so here $\Gamma' = \Gamma$, and thus by the induction hypothesis we have that $\Gamma \vdash P_2'$. Likewise, the restriction in the conclusion is empty, so the conclusion simplifies to $(\boldsymbol{\nu}\epsilon)(P_1' \mid P_2') = P_1' \mid P_2'$. Then $\Gamma \vdash P_1' \mid P_2'$ can be concluded by [T-par].
2. Otherwise, P_1 performs a *bound* output with label $\alpha_1 = x!(\boldsymbol{\nu}\tilde{z} : \widetilde{B})\tilde{v}$ of the names $\tilde{z}$, and thus $\Gamma' = \Gamma, \tilde{z} : \widetilde{B}$. Hence by the induction hypothesis, $\Gamma, \tilde{z} : \widetilde{B} \vdash P_1'$.
   For P_2, we know from the side condition of [E-Com$_1$] that $\tilde{z}\#P_2$. Thus, by Lemma 27 (weakening), we can also conclude $\Gamma, \tilde{z} : \widetilde{B} \vdash P_2$. As we know that P_2 performs an input with label $\alpha_2 = \tilde{x}?\tilde{v}$, we also know that the type environment does not change.
   We must now show that $\Gamma, \tilde{z} : \widetilde{B} \vdash \alpha_2$. As we know that $\Gamma \vdash P_1$ and $P_1 \xrightarrow{x!(\boldsymbol{\nu}\tilde{z}:\widetilde{B})\tilde{v}} P_1'$, we have by Lemma 31 that $\Gamma \vdash x!(\boldsymbol{\nu}\tilde{z} : \widetilde{B})\tilde{v}$, which must have been concluded by [T-snd]. From the premise, we have that $\Gamma \vdash \tilde{x} : \mathsf{ch}(\widetilde{B}')$ and $\Gamma, \tilde{z} : \widetilde{B} \vdash \tilde{v} : \widetilde{B}'$. By [T-rcv] we can then conclude that $\Gamma, \tilde{z} : \widetilde{B} \vdash \tilde{x}?\tilde{v}$.
   Now by the induction hypothesis, we have that $\Gamma, \tilde{z} : \widetilde{B} \vdash P_2'$. By [T-par] we can then conclude $\Gamma, \tilde{z} : \widetilde{B} \vdash P_1' \mid P_2'$, and finally by [T-res] we can conclude $\Gamma \vdash (\boldsymbol{\nu}\tilde{z} : \widetilde{B})(P_1' \mid P_2')$.

The symmetric case for [Com$_2$] is similar. □

5.8.4 Auxiliary lemmas for the proof of Theorem 11

The proof of Theorem 11 relies on some standard lemmas, that can be proved by straightforward inductions. Firstly, weakening and strengthening express that we can add and remove unused type assumptions in Γ when typing env_V:

Lemma 32 (Weakening for env_V). *If* $\Gamma \vdash \mathsf{env}_V$ *and* $x \notin \mathrm{dom}(\mathsf{env}_V)$, *then* $\Gamma, x : B \vdash \mathsf{env}_V$.

Lemma 33 (Strengthening for env_V). *If* $\Gamma, x : B \vdash \mathsf{env}_V$ *and* $x \notin \mathrm{dom}(\mathsf{env}_V)$, *then* $\Gamma \vdash \mathsf{env}_V$.

The next lemma expresses that, if an expression e is judged to have type B and e evaluates to some value v, then v is indeed of type B. This is shown by induction on the rules for semantics of expressions (Figure 5.6), and the rules for evaluating operations op, which we have omitted.

Lemma 34 (Safety for expressions). *Let* $\Gamma \vdash \mathsf{env}_S$ *and* $\Gamma \vdash \mathsf{env}_V$. *If* $\Gamma \vdash e : B$ *and* $\mathsf{env}_{SV} \vdash e \rightarrow_e v$, *then* $\Gamma \vdash v : B$.

The next two lemmas state that we can update an entry (x, v) in env_V with another value v', if v and v' are of the same type, and that we can extend env_V with a new entry (x, v), as long as x and v have the same type:

Lemma 35 (Substitution for env_V). *If* $\Gamma, x : B \vdash \mathsf{env}_V$ *and* $x \in \mathrm{dom}(\mathsf{env}_V)$ *and* $\Gamma, x : B \vdash v : B$, *then* $\Gamma, x : B \vdash \mathsf{env}_V[x \mapsto v]$.

Lemma 36 (Extension of env_V). *If* $\Gamma, x : B \vdash \mathsf{env}_V$ *and* $x \notin \mathrm{dom}(\mathsf{env}_V)$ *and* $\Gamma, x : B \vdash v : B$, *then* $\Gamma, x : B \vdash (x, v), \mathsf{env}_V$.

We shall need almost the same lemmas for env_S, except extension, since new fields cannot be declared at runtime. As env_S contains nested environments env_F, we shall use the notation $x\#\mathsf{env}_S$ to say that x does not occur in env_S, i.e. neither in its domain, or in the domain of any of its nested env_F environments (nor, in principle, as a value).

Lemma 37 (Weakening for env_S). *If* $\Gamma \vdash \mathsf{env}_S$ *and* $x\#\mathsf{env}_S$, *then* $\Gamma, x : B \vdash \mathsf{env}_S$.

Lemma 38 (Strengthening for env_S). *If* $\Gamma, x : B \vdash \mathsf{env}_S$ *and* $x\#\mathsf{env}_S$, *then* $\Gamma \vdash \mathsf{env}_S$.

Lemma 39 (Substitution for env_S). *Assume* $A \in \mathrm{dom}(\mathsf{env}_S)$ *and* $\mathsf{env}_S(A) = \mathsf{env}_F$. *If* $\Gamma \vdash \mathsf{env}_S$
and $\Gamma \vdash A.p : B$ *and* $\Gamma \vdash v : B$, *then* $\Gamma \vdash \mathsf{env}_S[A \mapsto \mathsf{env}_F[p \mapsto v]]$.

5.8.5 Proof of Theorem 11

Proof. By induction on the inference for $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \rightarrow \mathsf{env}'_{SV}$. We have four possible base cases:

- Suppose the transition was concluded by [WC-SKIP]. Then we know that $S = \texttt{skip}$, and the transition is of the form

$$\mathsf{env}_T \vdash \langle \texttt{skip}, \mathsf{env}_{SV} \rangle \to \mathsf{env}_{SV}$$

  As neither env_S nor env_V are modified by the transition, we have that $\Gamma \vdash \mathsf{env}_{SV}$ by assumption.

- Suppose [WC-WHILE$_\text{F}$] was used. Then we know that $S = \texttt{while}\ e\ \texttt{do}\ S$, and the transition is of the form

$$\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}_{SV} \rangle \to \mathsf{env}_{SV}$$

  As neither env_S nor env_V are modified in the transition, we have that $\Gamma \vdash \mathsf{env}_{SV}$ by assumption.

- Suppose [WC-ASSV] was used. Then we know that $S = x\ \texttt{:=}\ e$, and the transition is of the form

$$\mathsf{env}_T \vdash \langle x\ \texttt{:=}\ e, \mathsf{env}_{SV} \rangle \to \mathsf{env}_S, \mathsf{env}_V[x \mapsto v]$$

  From the premise of that rule we have that $\mathsf{env}_{SV} \vdash e \to_e v$. Now, $\Gamma \vdash x\ \texttt{:=}\ e$ must have been concluded by [T-ASSV], and from the premise of that rule we have that $\Gamma \vdash x : B$ and $\Gamma \vdash e : B$. We can then conclude the following:

$$\begin{aligned} &\Gamma \vdash v : B && \text{by Lemma 34 (expressions safety),} \\ &\Gamma \vdash \mathsf{env}_V[x \mapsto v] && \text{by Lemma 35 (substitution for } \mathsf{env}_V\text{),} \end{aligned}$$

  and $\Gamma \vdash \mathsf{env}_S$ by assumption, since it is not modified by the transition. Thus $\Gamma \vdash \mathsf{env}_S$ and $\Gamma \vdash \mathsf{env}_V[x \mapsto v]$ as desired.

- Suppose [WC-ASSF] was used. Then we know that $S = \texttt{this.}p\ \texttt{:=}\ e$, and the transition is of the form

$$\mathsf{env}_T \vdash \langle \texttt{this.}p\ \texttt{:=}\ e, \mathsf{env}_{SV} \rangle \to \mathsf{env}_S[A \mapsto \mathsf{env}_F[p \mapsto v]], \mathsf{env}_V$$

  From the premise of that rule we have that $\mathsf{env}_{SV} \vdash e \to_e v$, $\mathsf{env}_V(\texttt{this}) = A$, and $\mathsf{env}_S(A) = \mathsf{env}_F$. Now, $\Gamma \vdash \texttt{this.}p\ \texttt{:=}\ e$ must have been concluded by [T-ASSF], and from the premise of that rule we have that $\Gamma \vdash \texttt{this.}p : B$ and $\Gamma \vdash e : B$. We can then conclude that

$$\begin{aligned} &\Gamma \vdash v : B && \text{by Lemma 34 (expressions safety),} \\ &\Gamma \vdash \mathsf{env}_S[A \mapsto \mathsf{env}_F[p \mapsto v]] && \text{by Lemma 39 (substitution for } \mathsf{env}_S\text{),} \end{aligned}$$

  and $\Gamma \vdash \mathsf{env}_V$ by assumption, since it is not modified by the transition. Thus $\Gamma \vdash \mathsf{env}_V$ and $\Gamma \vdash \mathsf{env}_S[A \mapsto \mathsf{env}_F[p \mapsto v]]$ as desired.

For the inductive step, we reason on the rule of Figure 5.7 used for concluding the transition. Note that, when stating the induction hypothesis in the cases below, we shall omit some of the premises for brevity, if they hold by assumption; especially $\Gamma \vdash \mathsf{env}_T$ and $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$.

- Suppose [WC-SEQ] was used. Then we know that $S = S_1; S_2$, and the transition is of the form
$$\mathsf{env}_T \vdash \langle S_1\,;\ \ S_2, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$$
From the premise of that rule we have that $\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \to \mathsf{env}''_{SV}$ and $\mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \to \mathsf{env}'_{SV}$. Now, $\Gamma \vdash S_1; S_2$ must have been concluded by [T-SEQ], and from the premise of that rule we have that $\Gamma \vdash S_1$ and $\Gamma \vdash S_2$. By the induction hypothesis we can then conclude that $\Gamma \vdash \mathsf{env}''_{SV}$ and $\Gamma \vdash \mathsf{env}'_{SV}$, as desired.

- Suppose [WC-IF] was used. Then we know that $S = \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F}$, and the transition is of the form
$$\mathsf{env}_T \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F}\ , \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$$
From the premise of that rule we have that $\mathsf{env}_{SV} \vdash e \to_e b$ and $\mathsf{env}_T \vdash \langle S_b, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$. Now, $\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F}$ must have been concluded by [T-IF], and from the premise of that rule we have that $\Gamma \vdash S_\mathsf{T}$ and $\Gamma \vdash S_\mathsf{F}$. By the induction hypothesis we can then conclude that $\Gamma \vdash \mathsf{env}'_{SV}$, as desired.

- Suppose [WC-WHILE$_\mathsf{T}$] was used. Then we know that $S = \texttt{while}\ e\ \texttt{do}\ S$, and the transition is of the form
$$\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$$
From the premise of that rule we have that $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to \mathsf{env}''_{SV}$ and $\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}''_{SV}\rangle \to \mathsf{env}'_{SV}$. Now, $\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ S$ must have been concluded by [T-WHILE], and from the premise of that rule we have that $\Gamma \vdash S$. As we know the transition was in fact concluded, we also know that the derivation tree for the transition is of finite height, and we can therefore use the induction hypothesis. Thus, by the induction hypothesis we can conclude that $\Gamma \vdash \mathsf{env}''_{SV}$ and $\Gamma \vdash \mathsf{env}'_{SV}$, as desired.

- Suppose [WC-DECV] was used. Then we know that $S = \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S$, and the transition is of the form
$$\mathsf{env}_T \vdash \langle \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$$
From the premise of that rule we have that $\mathsf{env}_{SV} \vdash e \to_e v$ and $\mathsf{env}_T \vdash \langle S, \mathsf{env}_S, ((x, v), \mathsf{env}_V)\rangle \to \mathsf{env}'_S, ((x, v'), \mathsf{env}'_V)$. Now, $\Gamma \vdash \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}$

S must have been concluded by [T-DECV], and from the premise of that rule we have that $\Gamma \vdash e : B$ and $\Gamma, x : B \vdash S$. By Lemma 34 (expressions safety) we conclude that $\Gamma \vdash v : B$ and by Lemma 36 we conclude that $\Gamma, x : B \vdash (x, v), \mathsf{env}_V$. As $x\#\mathsf{env}_S$, we therefore also have that $\Gamma, x : B \vdash \mathsf{env}_S$, by Lemma 37. By the induction hypothesis we can then conclude that $\Gamma, x : B \vdash \mathsf{env}'_S$ and $\Gamma, x : B \vdash (x, v'), \mathsf{env}'_V$. As we know that $x\#\mathsf{env}'_S$ and $x \notin \mathrm{dom}(\mathsf{env}'_V)$, we can apply Lemmas 38 and 33 to conclude that $\Gamma \vdash \mathsf{env}'_S$ and $\Gamma \vdash \mathsf{env}'_V$ as desired.

- Suppose [WC-CALL] was used. Then we know that $S = \texttt{call}\ \ e.f(\tilde{e})$, and the transition is of the form

$$\mathsf{env}_T \vdash \langle \texttt{call}\ \ e.f(\tilde{e}), \mathsf{env}_{SV} \rangle \to \mathsf{env}'_S, \mathsf{env}_V$$

From the premise of that rule we have that

$$\begin{aligned} \mathsf{env}_{SV} &\vdash e \to_e A \\ \mathsf{env}_{SV} &\vdash \tilde{e} \to_e \tilde{v} \\ \mathsf{env}_T &\vdash \langle S, \mathsf{env}_S, \mathsf{env}''_V \rangle \to \mathsf{env}'_{SV} \end{aligned}$$

where $\mathsf{env}_T(A)(f) = (\tilde{x}, S)$ and $\mathsf{env}''_V = (\texttt{this}, A), (x_1, v_1), \ldots, (x_k, v_k)$ for some $k = |\tilde{e}| = |\tilde{x}| = |\tilde{v}|$. Now, $\Gamma \vdash \texttt{call}\ \ e.f(\tilde{e})$ must have been concluded by [T-CALL], and from the premise of that rule we have that $\Gamma \vdash e : I$ and $\Gamma(I)(f) = \mathrm{proc}(\widetilde{B})$ and $\Gamma \vdash \tilde{e} : \widetilde{B}$.

By assumption $\Gamma \vdash \mathsf{env}_T$, which was concluded by rules [T-ENV$_T$] and [T-ENV$_M$]. From the premise of the latter rule, we get that $\Gamma', \tilde{x} : \widetilde{B} \vdash S$ where $\Gamma' = \Gamma, \texttt{this} : I$ and $\Gamma'(I)(f) = \mathrm{proc}(\widetilde{B})$.

Now let $\Gamma' = \Gamma, \texttt{this} : I, \tilde{x} : \widetilde{B}$. We can then conclude the following:

$\Gamma \vdash A : I$	by Lemma 34 (expressions safety),
$\Gamma \vdash \tilde{v} : \widetilde{B}$	by Lemma 34 (expressions safety),
$\Gamma \vdash \mathsf{env}_V^{\varnothing}$	by [T-ENV$^{\varnothing}$],
$\Gamma'' \vdash \mathsf{env}''_V$	by Lemma 36 (extension of env_V) and the preceding,
$\Gamma'' \vdash \mathsf{env}_S$	by Lemma 37 (weakening of env_S),
$\Gamma'' \vdash \mathsf{env}'_S$	by the induction hypothesis.

Note that the env'_V obtained from the execution of S is discarded in the conclusion of [WC-CALL], so we do not need to infer its well-typedness. It is instead replaced with the original env_V, which is unmodified in the transition, and by assumption $\Gamma \vdash \mathsf{env}_V$. Finally, as we know that $\texttt{this} \notin \mathrm{dom}(\mathsf{env}_S)$ and $\tilde{x}\#\mathsf{env}_S$, we can apply Lemma 38 (repeatedly) to conclude that $\Gamma \vdash \mathsf{env}'_S$. Thus we have that $\Gamma \vdash \mathsf{env}_V$ and $\Gamma \vdash \mathsf{env}'_S$ as desired. □

5.8.6 Auxiliary lemmas for the proof of Theorem 12

Lemma 40. $\Gamma \vdash v : B \iff [\![\Gamma]\!] \vdash v : B$.

Proof. By case analysis of the type of v.

For the forward direction:

- If $v \in \mathbb{Z}$ then $\Gamma \vdash v : \mathsf{int}$ by [T-VAL]. We can then conclude $[\![\Gamma]\!] \vdash v : \mathsf{int}$ by [T-VAL], since this does not depend on Γ.
- If $v \in \mathbb{B}$ then $\Gamma \vdash v : \mathsf{bool}$ by [T-VAL]. We can then conclude $[\![\Gamma]\!] \vdash v : \mathsf{bool}$ by [T-VAL], since this does not depend on Γ.
- If $v \in \mathcal{N}$ then $\Gamma \vdash v : \Gamma(v)$ by [T-VAL]. By the syntax of WC (Figure 5.4), the only names that can appear as values are CNames (class names), and not method names, field names or variable names. Thus, v must be a class name A, and $\Gamma(A) = I$, so we can expand the type environment as $\Gamma = A : I, \Gamma'$.

  By the encoding of Γ, we have that $[\![A : I, \Gamma']\!] = A : I, [\![\Gamma']\!]$, so we can conclude that

  $$(A : I, [\![\Gamma']\!])(A) = I$$

For the other direction:

- If $v \in \mathbb{Z}$ then $[\![\Gamma]\!] \vdash v : \mathsf{int}$ by [T-VAL] for any $[\![\Gamma]\!]$. We can then conclude $\Gamma \vdash v : \mathsf{int}$ by [T-VAL].
- If $v \in \mathbb{B}$ then $[\![\Gamma]\!] \vdash v : \mathsf{bool}$ by [T-VAL] for any $[\![\Gamma]\!]$. We can then conclude $\Gamma \vdash v : \mathsf{bool}$ by [T-VAL].
- If $v \in \mathcal{N}$ then $[\![\Gamma]\!] \vdash v : [\![\Gamma]\!](v)$ by [T-VAL]. Hence, $[\![\Gamma]\!]$ can be expanded as $[\![\Gamma]\!] = v : B, [\![\Gamma']\!]$ for some B. Since we know that Γ is used to type WC-programs, we also know by the syntax of WC (Figure 5.4) that the only names that can appear as values are CNames (class names), and not method names, field names or variable names. Thus, v must be a class name A, which must have an interface type I, so $[\![\Gamma]\!] = A : I, [\![\Gamma']\!]$, and therefore $(A : I, [\![\Gamma']\!])(A) = I$.

  By the encoding of Γ, we have that $[\![A : I, \Gamma']\!] = A : I, [\![\Gamma']\!]$, and thus $\Gamma = A : I, \Gamma'$. We can then conclude that $(A : I, \Gamma)(A) = I$, as desired. □

Lemma 41. $\Gamma \vdash e : B \iff [\![\Gamma]\!], z : I^B \vdash [\![e]\!]^\Gamma_z$.

Proof. We have two directions to prove.

For the forward direction, we must show that $\Gamma \vdash e : B \implies [\![\Gamma]\!], z : I^B \vdash [\![e]\!]^\Gamma_z$. We proceed by induction on the structure of e.

- Suppose the expression is v, so $\Gamma \vdash v : B$. By the encoding, $[\![v]\!]^{\Gamma}_{z} = z\langle v\rangle$, and we can then conclude $[\![\Gamma]\!], z : I^B \vdash z\langle v\rangle$ as follows: By Lemma 40, we have that $[\![\Gamma]\!] \vdash v : B$. Then, by [T-VEC$_2$],

$$([\![\Gamma]\!], z : I^B); ([\![\Gamma]\!], z : I^B) \vdash z : \mathsf{ch}(B)$$

since by assumption, type definitions $I^B \mapsto (\mathsf{ch}(B), \varnothing)$ are added to $[\![\Gamma]\!]$ for all the base types used. Finally, we can conclude

$$[\![\Gamma]\!], z : I^B \vdash z\langle v\rangle$$

by [T-OUT], as desired.

- Suppose the expression is x, so $\Gamma \vdash x : B$. This must have been concluded by [T-VAR], and from its premise, we know that $\Gamma(x) = B$. Hence, we can expand the type environment as $\Gamma = x : B, \Gamma'$. Now, by the encoding of expressions:

$$[\![x]\!]^{\Gamma} = x \triangleright (y).z\langle y\rangle = x(y).(x\langle y\rangle \mid z\langle y\rangle)$$

and by the encoding for Γ:

$$[\![x : B, \Gamma']\!] = x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!]$$

so by [T-VEC$_2$] we can then conclude that

$$\begin{aligned}
&(x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!], z : I^B);\\
&(x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!], z : I^B) \vdash x : \mathsf{ch}(B)\\
&(x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!], z : I^B);\\
&(x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!], z : I^B) \vdash z : \mathsf{ch}(B)
\end{aligned}$$

Using this, we can conclude

$$x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!], z : I^B, y : B \vdash x\langle y\rangle$$

by [T-OUT],

$$x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!], z : I^B, y : B \vdash z\langle y\rangle$$

by [T-OUT],

$$x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!], z : I^B, y : B \vdash x\langle y\rangle \mid z\langle y\rangle$$

by [T-PAR], and finally

$$x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!], z : I^B \vdash x(y).(x\langle y\rangle \mid z\langle y\rangle)$$

by [T-IN], as desired.

- Suppose the expression is $e.p$, so $\Gamma \vdash e.p : B$. This must have been concluded by [T-FIELD], and from its premise we know that $\Gamma \vdash e : I_A$ and $\Gamma(I_A)(p) = B$. Thus, we know that Γ can be written as $I_A : (p : B, \Delta), \Gamma'$ for some Δ. By the encoding of Γ, we then have that

  $$[\![I_A : (p : B, \Delta), \Gamma']\!] = I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]$$

  However, we shall leave $[\![\Gamma]\!]$ unexpanded in the following for the sake of readability. Now, by the encoding of expressions:

  $$\begin{aligned}[\![e.p]\!]^\Gamma_z &= (\boldsymbol{\nu} z' : I^{I_A})\big([\![e]\!]^\Gamma_{z'} \mid z'(Y).Y \cdot p \triangleright (y).z\langle y\rangle\big) \\ &= (\boldsymbol{\nu} z' : I^{I_A})\big([\![e]\!]^\Gamma_{z'} \mid z'(Y).Y \cdot p(y).(Y \cdot p\langle y\rangle \mid z\langle y\rangle)\big)\end{aligned}$$

  and by the induction hypothesis $[\![\Gamma]\!], z' : I^{I_A} \vdash [\![e]\!]^\Gamma_{z'}$. Using this, we conclude as follows:

  $$[\![\Gamma]\!], z : I^B, z' : I^{I_A}, y : B \vdash z\langle y\rangle$$

  by [T-OUT],

  $$[\![\Gamma]\!], z : I^B, z' : I^{I_A}, Y : I_A, y : B \vdash Y \cdot p\langle y\rangle$$

  by [T-OUT],

  $$[\![\Gamma]\!], z : I^B, z' : I^{I_A}, Y : I_A, y : B \vdash Y \cdot p\langle y\rangle \mid z\langle y\rangle$$

  by [T-PAR],

  $$[\![\Gamma]\!], z : I^B, z' : I^{I_A}, Y : I_A \vdash Y \cdot p(y).(Y \cdot p\langle y\rangle \mid z\langle y\rangle)$$

  by [T-IN],

  $$[\![\Gamma]\!], z : I^B, z' : I^{I_A} \vdash z'(Y).Y \cdot p(y).(Y \cdot p\langle y\rangle \mid z\langle y\rangle)$$

  by [T-IN],

  $$[\![\Gamma]\!], z : I^B, z' : I^{I_A} \vdash [\![e]\!]^\Gamma_{z'} \mid z'(Y).Y \cdot p(y).(Y \cdot p\langle y\rangle \mid z\langle y\rangle)$$

  by [T-PAR], and finally

  $$[\![\Gamma]\!], z : I^B \vdash (\boldsymbol{\nu} z' : I^{I_A})\big([\![e]\!]^\Gamma_{z'} \mid z'(Y).Y \cdot p(y).(Y \cdot p\langle y\rangle \mid z\langle y\rangle)\big)$$

  by [T-RES], as desired.

- Suppose the expression is $\mathrm{op}(e_1, \ldots, e_n)$, so $\Gamma \vdash \mathrm{op}(e_1, \ldots, e_n) : B$. This must have been concluded by [T-OP], and from its premise we know that

$$\Gamma \vdash e_1 : B_1 \quad \ldots \quad \Gamma \vdash e_n : B_n$$

and $\vdash \mathrm{op} : B_1, \ldots, B_n \to B$. Now, by the encoding of expressions:

$$[\![\mathrm{op}(e_1, \ldots, e_n)]\!]^\Gamma_z = (\boldsymbol{\nu} z_1 : I^{B_1}, \ldots, z_n : I^{B_n})\big([\![e_1]\!]^\Gamma_{z_1} \mid \ldots \mid [\![e_n]\!]^\Gamma_{z_n} \mid z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle\big)$$

and by n applications of the induction hypothesis, we have that $[\![\Gamma]\!], z_1 : I^{B_1} \vdash [\![e_1]\!]^\Gamma_{z_1}$, ..., $[\![\Gamma]\!], z_n : I^{B_n} \vdash [\![e_n]\!]^\Gamma_{z_n}$. By Lemma 27, we can then also conclude that

$$[\![\Gamma]\!], z : I^B, z_1 : I^{B_1}, \ldots, z_n : I^{B_n} \vdash [\![e_i]\!]^\Gamma_{z_i}$$

for each $i \in 1..n$, and thus by $n-1$ applications of rule [T-PAR], we can conclude

$$[\![\Gamma]\!], z : I^B, z_1 : I^{B_1}, \ldots, z_n : I^{B_n} \vdash [\![e_1]\!]^\Gamma_{z_1} \mid \ldots \mid [\![e_n]\!]^\Gamma_{z_n}$$

Furthermore, we can conclude

$$[\![\Gamma]\!], z : I^B, z_1 : I^{B_1}, \ldots, z_n : I^{B_n}, y_1 : B_1, \ldots, y_n : B_n \vdash z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle$$

by [T-OUT], and then, by n applications of [T-IN], we can conclude

$$[\![\Gamma]\!], z : I^B, z_1 : I^{B_1}, \ldots, z_n : I^{B_n} \vdash z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle$$

Thus, we can conclude

$$[\![\Gamma]\!], z : I^B, z_1 : I^{B_1}, \ldots, z_n : I^{B_n} \vdash [\![e_1]\!]^\Gamma_{z_1} \mid \ldots \mid [\![e_n]\!]^\Gamma_{z_n} \mid z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle$$

by another application of [T-PAR], and finally

$$[\![\Gamma]\!], z : I^B \vdash (\boldsymbol{\nu} z_1 : I^{B_1}, \ldots, z_n : I^{B_n})\big([\![e_1]\!]^\Gamma_{z_1} \mid \ldots \mid [\![e_n]\!]^\Gamma_{z_n} \mid z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle\big)$$

by [T-RES], as desired.

This concludes the proof for the forward direction.

For the other direction, we must show that $[\![\Gamma]\!], z : I^B \vdash [\![e]\!]^\Gamma_z \implies \Gamma \vdash e : B$. We proceed again by induction on the structure of e:

- Suppose the expression is v, so $[\![v]\!]^{\Gamma}_{z} = z\langle v\rangle$; by [T-OUT], we have that $[\![\Gamma]\!], z : I^B \vdash z\langle v\rangle$ and, from its premise, we have that $[\![\Gamma]\!], z : I^B \vdash x : \mathsf{ch}(B)$ and $[\![\Gamma]\!], z : I^B \vdash v : B$. By Lemma 28, we have that $[\![\Gamma]\!] \vdash v : B$, since we know that $r\#v$, as the name was introduced by the translation. Then, by Lemma 40, $\Gamma \vdash v : B$, as desired.

- Suppose the expression is x, so

$$[\![x]\!]^{\Gamma} = x(y).(x\langle y\rangle \mid z\langle y\rangle)$$

  and

$$[\![\Gamma]\!], z : I^B \vdash x(y).(x\langle y\rangle \mid z\langle y\rangle)$$

  which must have been concluded by [T-IN], [T-PAR] and [T-OUT]. From the premise of the latter, we get that $[\![\Gamma]\!]; [\![\Gamma]\!] \vdash x : \mathsf{ch}(B)$, so it must be the case that $[\![\Gamma]\!](x) = I^B$, and $[\![\Gamma]\!]$ contains the type assignment $I^B \mapsto (\mathsf{ch}(B), \varnothing)$. By the translation of Γ, we then have that

$$[\![x : B, \Gamma']\!] = x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!]$$

  where $\Gamma = x : B, \Gamma'$, and $x : B, \Gamma' \vdash x : B$ can then be concluded by [T-VAR] as desired.

- Suppose the expression is $e.p$, so

$$[\![e.p]\!]^{\Gamma}_{z} = (\boldsymbol{\nu} z' : I^{I_A})\left([\![e]\!]^{\Gamma}_{z'} \mid z'(Y).Y \cdot p(y).(Y \cdot p\langle y\rangle \mid z\langle y\rangle)\right)$$

  and

$$[\![\Gamma]\!], z : I^B \vdash (\boldsymbol{\nu} z' : I^{I_A})\left([\![e]\!]^{\Gamma}_{z'} \mid z'(Y).Y \cdot p(y).(Y \cdot p\langle y\rangle \mid z\langle y\rangle)\right)$$

  which must have been concluded by [T-RES], [T-PAR], [T-IN] and [T-OUT]. We omit the full derivation here, but see the corresponding case for the forward direction.

  Our goal is to infer $\Gamma \vdash e.p : B$ using [T-FIELD]. The premises of that rule are $\Gamma \vdash e : I_A$ and $\Gamma(I_A)(p) = B$, so we must show that they both are satisfied. Since one of the premises of [T-PAR] is $[\![\Gamma]\!], z' : I^{I_A} \vdash [\![e]\!]^{\Gamma}_{z'}$, by the induction hypothesis we have that $\Gamma \vdash e : I_A$.

  In another branch of the derivation tree, $[\![\Gamma]\!], Y : I_A, y : B \vdash Y \cdot p\langle y\rangle$ is concluded by [T-OUT] (some unused names are omitted for clarity); its premise requires that

$$([\![\Gamma]\!], Y : I_A, y : B); ([\![\Gamma]\!], Y : I_A, y : B) \vdash \mathsf{ch}(B)$$

is concluded by [T-VEC$_1$] and [T-VEC$_2$]. Hence, $[\![\Gamma]\!]$ must contain $I_A \mapsto (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2))$ and $I^B \mapsto (\mathsf{ch}(B), \varnothing)$. Hence, by the encoding of Γ, we have that

$$[\![I_A : (p : B, \Delta), \Gamma']\!] = I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]$$

and clearly $(I_A : (p : B, \Delta), \Gamma')(I_A)(p) = B$. Thus, as all premises of [T-FIELD] are satisfied, we can conclude that $\Gamma \vdash e.p : B$.

- Suppose the expression is $\mathrm{op}(e_1, \ldots, e_n)$, so

$$[\![\mathrm{op}(e_1, \ldots, e_n)]\!]^\Gamma_z = (\boldsymbol{\nu} z_1 : I^{B_1}, \ldots, z_n : I^{B_n})(\\ [\![e_1]\!]^\Gamma_{z_1} \mid \ldots \mid [\![e_n]\!]^\Gamma_{z_n} \mid z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle)$$

and

$$[\![\Gamma]\!], z : I^B \vdash (\boldsymbol{\nu} z_1 : I^{B_1}, \ldots, z_n : I^{B_n})(\\ [\![e_1]\!]^\Gamma_{z_1} \mid \ldots \mid [\![e_n]\!]^\Gamma_{z_n} \mid z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle)$$

which must have been concluded by [T-RES]. From its premise, we then have that

$$[\![\Gamma]\!], z : I^B, z_1 : I^{B_1}, \ldots, z_n : I^{B_n} \\ \vdash [\![e_1]\!]^\Gamma_{z_1} \mid \ldots \mid [\![e_n]\!]^\Gamma_{z_n} \mid z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle$$

and by n applications of the induction hypothesis (and Lemma 28 to remove unused names), we have that $\Gamma \vdash e_i : B_i$, for all i. Furthermore,

$$[\![\Gamma]\!], z : I^B, z_1 : I^{B_1}, \ldots, z_n : I^{B_n} \vdash z_1(y_1) \ldots z_n(y_n).z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle$$

must have been concluded by n applications of [T-IN], and with

$$[\![\Gamma]\!], z : I^B, y_1 : B_1, \ldots, y_n : B_n \vdash z\langle \mathrm{op}(y_1, \ldots, y_n)\rangle$$

as the premise (we omit the full derivation here, but see the corresponding case for the forward direction). Its premise then requires that $[\![\Gamma]\!], z : I^B; [\![\Gamma]\!]; z : I^B \vdash z : \mathsf{ch}(B)$ (which follows from the presence of $z : I^B$ in accordance with our use of interface types for basic types) and of $\vdash \mathrm{op} : B_1, \ldots, B_n \to B$. Thus we can conclude $\Gamma \vdash \mathrm{op}(e_1, \ldots, e_n) : B$ by [T-OP] as desired.

This concludes the proof for the other direction. □

Lemma 42. $\Gamma \vdash S \iff [\![\Gamma]\!], r : I^{ret} \vdash [\![S]\!]^{\Gamma}_{r}$.

Proof. We have two statements to prove. For both directions, we proceed by induction on the structure of S.

For the forward direction, we must show that $\Gamma \vdash S \implies [\![\Gamma]\!], r : I^{\text{ret}} \vdash [\![S]\!]^{\Gamma}_{r}$. We first recall that $I^{\text{ret}} \mapsto (\text{ch}(), \varnothing)$ is included in the definition of $[\![\Gamma]\!]$.

- Suppose the statement is `skip`, so $\Gamma \vdash \texttt{skip}$. This must have been concluded by [T-SKIP], which is an axiom. By the encoding of statements, we have that

$$[\![\texttt{skip}]\!]^{\Gamma}_{r} = r\texttt{<>}$$

  and we can therefore conclude $[\![\Gamma]\!], r : I^{\text{ret}} \vdash r\texttt{<>}$ as desired.

- Suppose the statement is $\texttt{var}\ B\ x\ \texttt{:=}\ e\ \texttt{in}\ S$, so $\Gamma \vdash \texttt{var}\ B\ x\ \texttt{:=}\ e\ \texttt{in}\ S$. This must have been concluded by [T-DECV], and from its premise we have that $\Gamma \vdash e : B$ and $\Gamma, x : B \vdash S$. By the encoding of statements, we have that

$$\begin{aligned}
&[\![\texttt{var}\ B\ x \texttt{:=} e\ \texttt{in}\ S]\!]^{\Gamma}_{r} \\
&= (\boldsymbol{\nu} z : I^{B})\left([\![e]\!]^{\Gamma}_{z} \mid z(y).(\boldsymbol{\nu} x : I^{B} \texttt{:=} y)\left([\![S]\!]^{\Gamma, x:B}_{r}\right)\right) \\
&= (\boldsymbol{\nu} z : I^{B})\left([\![e]\!]^{\Gamma}_{z} \mid z(y).(\boldsymbol{\nu} x : I^{B})\left(x\texttt{<}y\texttt{>} \mid [\![S]\!]^{\Gamma, x:B}_{r}\right)\right)
\end{aligned}$$

  By Lemma 41, we have that $[\![\Gamma]\!], z : I^{B} \vdash [\![e]\!]^{\Gamma}_{z}$ and, by the induction hypothesis, that $[\![\Gamma, x : B]\!], r : I^{\text{ret}} \vdash [\![S]\!]^{\Gamma, x:B}_{r}$. By the encoding of Γ, we have that $[\![\Gamma, x : B]\!] = x : I^{B}, [\![\Gamma]\!]$ where $I^{B} \mapsto I^{B} : (\text{ch}(B), \varnothing)$. We then conclude as follows:

$$[\![\Gamma]\!], x : I^{B}, r : I^{\text{ret}}, y : B \vdash x\texttt{<}y\texttt{>}$$

  by [T-OUT],

$$[\![\Gamma]\!], x : I^{B}, r : I^{\text{ret}}, y : B \vdash [\![S]\!]^{\Gamma, x:B}_{r}$$

  by Lemma 27,

$$[\![\Gamma]\!], x : I^{B}, r : I^{\text{ret}}, y : B \vdash x\texttt{<}y\texttt{>} \mid [\![S]\!]^{\Gamma, x:B}_{r}$$

  by [T-PAR],

$$[\![\Gamma]\!], r : I^{\text{ret}}, y : B \vdash (\boldsymbol{\nu} x : I^{B})\left(x\texttt{<}y\texttt{>} \mid [\![S]\!]^{\Gamma, x:B}_{r}\right)$$

  by [T-RES],

$$[\![\Gamma]\!], r : I^{\text{ret}}, z : I^{B} \vdash z(y).(\boldsymbol{\nu} x : I^{B})\left(x\texttt{<}y\texttt{>} \mid [\![S]\!]^{\Gamma, x:B}_{r}\right)$$

by [T-IN] and Lemma 27,

$$[\![\Gamma]\!], r : I^{\text{ret}}, z : I^B \vdash [\![e]\!]^\Gamma_z$$

by Lemma 27

$$[\![\Gamma]\!], r : I^{\text{ret}}, z : I^B \vdash [\![e]\!]^\Gamma_z \mid z(y).(\boldsymbol{\nu} x : I^B)\left(x\langle y\rangle \mid [\![S]\!]^{\Gamma, x:B}_r\right)$$

by [T-PAR], and finally

$$[\![\Gamma]\!], r : I^{\text{ret}} \vdash (\boldsymbol{\nu} z : I^B)\left([\![e]\!]^\Gamma_z \mid z(y).(\boldsymbol{\nu} x : I^B)\left(x\langle y\rangle \mid [\![S]\!]^{\Gamma, x:B}_r\right)\right)$$

by [T-RES], as desired.

- Suppose the statement is $x := e$, so $\Gamma \vdash x := e$. This must have been concluded by [T-ASSV], and from its premise we have that $\Gamma \vdash e : B$ and $\Gamma \vdash x : B$ By the encoding of statements, we have that

$$\begin{aligned}[\![x := e]\!]^\Gamma_r &= (\boldsymbol{\nu} z : I^B)\left([\![e]\!]^\Gamma_z \mid z(y).x \lhd \langle y\rangle.r\langle\rangle\right) \\ &= (\boldsymbol{\nu} z : I^B)\left([\![e]\!]^\Gamma_z \mid z(y).x(w).(x\langle y\rangle \mid r\langle\rangle)\right)\end{aligned}$$

for some unused name w. By Lemma 41 we have that $[\![\Gamma]\!], z : I^B \vdash [\![e]\!]^\Gamma_z$, and so

$$[\![\Gamma]\!], r : I^{\text{ret}} \vdash (\boldsymbol{\nu} z : I^B)\left([\![e]\!]^\Gamma_z \mid z(y).x(w).(x\langle y\rangle \mid r\langle\rangle)\right)$$

can be easily shown by rules [T-IN], [T-OUT], [T-PAR] and concluded by [T-RES], *if* we can show that $[\![\Gamma]\!]; [\![\Gamma]\!] \vdash x : \mathsf{ch}(B)$, to be concluded by [T-VEC$_2$] for the premise of [T-IN] and [T-OUT].

We show this as follows: Since we know that $\Gamma \vdash x : B$, then we also know that Γ can be expanded as $\Gamma = (x : B), \Gamma'$. By the encoding of Γ, we have that

$$[\![(x : B), \Gamma']\!] = x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!]$$

and hence

$$\begin{aligned}(x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!])(x) &= I^B \\ (x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!])(I^B) &= (\mathsf{ch}(B), \varnothing) \\ \mathsf{fst}(\mathsf{ch}(B), \varnothing) &= \mathsf{ch}(B)\end{aligned}$$

and thus

$$(x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!]); (x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!]) \vdash x : \mathsf{ch}(B)$$

can indeed be concluded by [T-VEC$_2$], as required.

- Suppose the statement is $\texttt{this}.p\ \texttt{=}\ e$, so $\Gamma \vdash \texttt{this}.p\ \texttt{=}\ e$. This must have been concluded by [T-ASSF], and from its premise we have that $\Gamma \vdash \texttt{this}.p : B$ and $\Gamma \vdash e : B$. By the encoding of statements, we have that

$$
\begin{aligned}
&[\![\texttt{this}.p\ \texttt{:=}\ e]\!]^{\Gamma}_{r} \\
&= (\boldsymbol{\nu} z : I^B)\big([\![e]\!]^{\Gamma}_{z} \mid z(y).\texttt{this} \triangleright (Y).Y \cdot p \triangleleft \texttt{<}y\texttt{>}.r\texttt{<>}\big) \\
&= (\boldsymbol{\nu} z : I^B)\big([\![e]\!]^{\Gamma}_{z} \mid z(y).\texttt{this}(Y).\big(\texttt{this<}Y\texttt{>} \mid Y \cdot p \triangleleft \texttt{<}y\texttt{>}.r\texttt{<>}\big)\big) \\
&= (\boldsymbol{\nu} z : I^B)\big([\![e]\!]^{\Gamma}_{z} \mid z(y).\texttt{this}(Y).(\texttt{this<}Y\texttt{>} \mid Y \cdot p(w).(Y \cdot p\texttt{<}y\texttt{>} \mid r\texttt{<>}))\big)
\end{aligned}
$$

and, by Lemma 41, we have that $[\![\Gamma]\!], z : I^B \vdash [\![e]\!]^{\Gamma}_{z}$. As in the case for $x\ \texttt{:=}\ e$ above, it is then straightforward to show that

$$
\begin{aligned}
[\![\Gamma]\!], r : I^{\text{ret}} \vdash (\boldsymbol{\nu} z : I^B)\big([\![e]\!]^{\Gamma}_{z} \mid z(y).\texttt{this}(Y) \\
.(\texttt{this<}Y\texttt{>} \mid Y \cdot p(w).(Y \cdot p\texttt{<}y\texttt{>} \mid r\texttt{<>}))\big)
\end{aligned}
$$

by rules [T-IN], [T-OUT], [T-PAR] and concluded by [T-RES], *if* we can show that

$$
\begin{aligned}
[\![\Gamma]\!]; [\![\Gamma]\!] &\vdash \texttt{this} : \mathsf{ch}(I_A) \\
([\![\Gamma]\!], Y : I_A); ([\![\Gamma]\!], Y : I_A) &\vdash Y \cdot p : \mathsf{ch}(B)
\end{aligned}
$$

We show this as follows: $\Gamma \vdash \texttt{this}.p : B$ must have been concluded by [T-FIELD], and from its premise we know that $\Gamma \vdash \texttt{this} : I_A$ and $\Gamma(I_A)(p) = B$. Thus we know that Γ can be expanded as

$$
\Gamma = \texttt{this} : I_A, I_A : (p : B, \Delta), \Gamma'
$$

and by the encoding for Γ, we have that

$$
\begin{aligned}
&[\![\texttt{this} : I_A, I_A : (p : B, \Delta), \Gamma']\!] \\
&= \texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(I_A), \varnothing), I_A : (\mathsf{nil}, [\![p : B, \Delta]\!]^2), [\![p : B, \Delta]\!]^3, [\![\Gamma']\!] \\
&= \texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(I_A), \varnothing), I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), \\
&\qquad I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]
\end{aligned}
$$

Hence, we can conclude:

$$
\begin{aligned}
&(\texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(I_A), \varnothing), I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), \\
&\qquad I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!])(\texttt{this}) \\
&= I^{I_A}
\end{aligned}
$$

and

$$\begin{aligned}&(\texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(I_A), \varnothing), I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)),\\ &\qquad I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!])(I^{I_A})\\ &= (\mathsf{ch}(I_A), \varnothing)\end{aligned}$$

and $\mathsf{fst}(\mathsf{ch}(I_A), \varnothing) = \mathsf{ch}(I_A)$ by [T-VEC$_2$]. Thus

$$\texttt{this}(Y).(\texttt{this<}Y\texttt{>} \mid Y \cdot p(w).(Y \cdot p\texttt{<}y\texttt{>} \mid r\texttt{<>}))$$

can be typed by [T-IN], and with the continuation typed as

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, Y : I_A \vdash \texttt{this<}Y\texttt{>} \mid Y \cdot p(w).(Y \cdot p\texttt{<}y\texttt{>} \mid r\texttt{<>})$$

Now, what remains to show is that $([\![\Gamma]\!], Y : I_A); [\![\Gamma]\!], Y : I_A \vdash Y \cdot p : \mathsf{ch}(B)$. Like before, we get

$$\begin{aligned}&(\texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(I_A), \varnothing), I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)),\\ &\qquad I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!], Y : I_A)(Y)\\ &= I_A\end{aligned}$$

and

$$\begin{aligned}&(\texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(I_A), \varnothing), I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)),\\ &\qquad I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!], Y : I_A)(I_A)\\ &= (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2))\end{aligned}$$

and $\mathsf{snd}(\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)) = (p : I^B, [\![\Delta]\!]^2)$, which satisfies the first premise of [T-VEC$_1$]. For the second premise, we must conclude $[\![\Gamma]\!]; (p : I^B, [\![\Delta]\!]^2) \vdash p : \mathsf{ch}(B)$. We have that

$$(p : I^B, [\![\Delta]\!]^2)(p) = I^B$$

and

$$\begin{aligned}&(\texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(I_A), \varnothing), I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)),\\ &\qquad I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!], Y : I_A)(I^B)\\ &= (\mathsf{ch}(B), \varnothing)\end{aligned}$$

and $\mathsf{fst}(\mathsf{ch}(B), \varnothing) = \mathsf{ch}(B)$ as required for [T-VEC$_1$]. Thus

$$[\![\Gamma]\!], Y : I_A, y : B \vdash Y \cdot p\texttt{<}y\texttt{>}$$

can be concluded by [T-OUT], as required.

- Suppose the statement is $S_1 ; S_2$, so $\Gamma \vdash S_1 ; S_2$. This must have been concluded by [T-SEQ], and from its premise we have that $\Gamma \vdash S_1$ and $\Gamma \vdash S_2$. By the encoding of statements, we have that

$$[\![S_1 ; S_2]\!]^\Gamma_{r_2} = (\boldsymbol{\nu} r_1 : I^{\mathsf{ret}}) \big([\![S_1]\!]^\Gamma_{r_1} \mid r_1().[\![S_2]\!]^\Gamma_{r_2} \big)$$

  and by the induction hypothesis we have that $[\![\Gamma]\!], r_1 : I^{\mathsf{ret}} \vdash [\![S]\!]^\Gamma_{r_1}$ and $[\![\Gamma]\!], r_2 : I^{\mathsf{ret}} \vdash [\![S]\!]^\Gamma_{r_2}$. Thus we can conclude the following:

$$\begin{aligned}
[\![\Gamma]\!], r_1 : I^{\mathsf{ret}}, r_2 : I^{\mathsf{ret}} &\vdash r_1().[\![S_2]\!]^\Gamma_{r_2} && \text{by [T-IN]} \\
[\![\Gamma]\!], r_1 : I^{\mathsf{ret}}, r_2 : I^{\mathsf{ret}} &\vdash [\![S_1]\!]^\Gamma_{r_1} \mid r_1().[\![S_2]\!]^\Gamma_{r_2} && \text{by [T-PAR]} \\
[\![\Gamma]\!], r_2 : I^{\mathsf{ret}} &\vdash (\boldsymbol{\nu} r_1 : I^{\mathsf{ret}}) \big([\![S_1]\!]^\Gamma_{r_1} \mid r_1().[\![S_2]\!]^\Gamma_{r_2} \big) && \text{by [T-RES]}
\end{aligned}$$

  as desired.

- Suppose the statement is $\texttt{if } e \texttt{ then } S_{\mathsf{T}} \texttt{ else } S_{\mathsf{F}}$, so

$$\Gamma \vdash \texttt{if } e \texttt{ then } S_{\mathsf{T}} \texttt{ else } S_{\mathsf{F}}$$

  This must have been concluded by [T-IF], and from its premise we have that $\Gamma \vdash e : \mathsf{bool}$, $\Gamma \vdash S_{\mathsf{T}}$, and $\Gamma \vdash S_{\mathsf{F}}$. By the encoding of statements, we have that

$$\begin{aligned}
&[\![\texttt{if } e \texttt{ then } S_{\mathsf{T}} \texttt{ else } S_{\mathsf{F}}]\!]^\Gamma_r \\
&= (\boldsymbol{\nu} z : I^{\mathsf{bool}}) \big([\![e]\!]^\Gamma_z \mid z(y).\mathbf{if}\ y\ \mathbf{then}\ [\![S_{\mathsf{T}}]\!]^\Gamma_r\ \mathbf{else}\ [\![S_{\mathsf{F}}]\!]^\Gamma_r \big) \\
&= (\boldsymbol{\nu} z : I^{\mathsf{bool}}) \big([\![e]\!]^\Gamma_z \mid z(y).([y = \mathsf{T}][\![S_{\mathsf{T}}]\!]^\Gamma_r + [y = \mathsf{F}][\![S_{\mathsf{F}}]\!]^\Gamma_r) \big)
\end{aligned}$$

  where we know that $z, y \# [\![S_{\mathsf{T}}]\!]^\Gamma_r, [\![S_{\mathsf{F}}]\!]^\Gamma_r$. Furthermore, by the induction hypothesis, we have that $[\![\Gamma]\!], r : I^{\mathsf{ret}} \vdash [\![S_{\mathsf{T}}]\!]^\Gamma_r$ and $[\![\Gamma]\!], r : I^{\mathsf{ret}} \vdash [\![S_{\mathsf{F}}]\!]^\Gamma_r$; furthermore, by Lemma 41, we have that $[\![\Gamma]\!], z : I^{\mathsf{bool}} \vdash [\![e]\!]^\Gamma_z$. After an application of Lemma 27 to add the type entries $z : I^{\mathsf{bool}}, y : \mathsf{bool}$, we can then conclude the following:

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, z : I^{\mathsf{bool}}, y : \mathsf{bool} \vdash [y = \mathsf{T}][\![S_{\mathsf{T}}]\!]^\Gamma_r + [y = \mathsf{F}][\![S_{\mathsf{F}}]\!]^\Gamma_r$$

  by [T-SUM],

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, z : I^{\mathsf{bool}} \vdash z(y).([y = \mathsf{T}][\![S_{\mathsf{T}}]\!]^\Gamma_r + [y = \mathsf{F}][\![S_{\mathsf{F}}]\!]^\Gamma_r)$$

  by [T-IN],

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, z : I^{\mathsf{bool}} \vdash [\![e]\!]^\Gamma_z \mid z(y).([y = \mathsf{T}][\![S_{\mathsf{T}}]\!]^\Gamma_r + [y = \mathsf{F}][\![S_{\mathsf{F}}]\!]^\Gamma_r)$$

  by [T-PAR], and finally

$$[\![\Gamma]\!], r : I^{\mathsf{ret}} \vdash (\boldsymbol{\nu} z : I^{\mathsf{bool}}) \big([\![e]\!]^\Gamma_z \mid z(y).([y = \mathsf{T}][\![S_{\mathsf{T}}]\!]^\Gamma_r + [y = \mathsf{F}][\![S_{\mathsf{F}}]\!]^\Gamma_r) \big)$$

  by [T-RES], as desired.

- Suppose the statement is `while` e `do` S, so $\Gamma \vdash$ `while` e `do` S. This must have been concluded by [T-WHILE], and from its premise we have that $\Gamma \vdash e : \mathsf{bool}$ and $\Gamma \vdash S$. By the encoding of statements, we have that

$$\begin{aligned}
&[\![\texttt{while } e \texttt{ do } S]\!]^{\Gamma}_{r} \\
&= (\boldsymbol{\nu} r' : I^{\mathsf{ret}})\Big(r'\langle\rangle \mid !r'().(\boldsymbol{\nu} z : I^{\mathsf{bool}}) \\
&\qquad \big([\![e]\!]^{\Gamma}_{z} \mid z(y).\mathbf{if}\ y\ \mathbf{then}\ [\![S]\!]^{\Gamma}_{r'}\ \mathbf{else}\ r\langle\rangle\big)\Big) \\
&= (\boldsymbol{\nu} r' : I^{\mathsf{ret}})\Big(r'\langle\rangle \mid !r'().(\boldsymbol{\nu} z : I^{\mathsf{bool}}) \\
&\qquad \big([\![e]\!]^{\Gamma}_{z} \mid z(y).([y = \mathsf{T}][\![S]\!]^{\Gamma}_{r'} + [y = \mathsf{F}]r\langle\rangle)\big)\Big)
\end{aligned}$$

  where we know that $z, y \# [\![S]\!]^{\Gamma}_{r'}$, and $r' \# [\![e]\!]^{\Gamma}_{z}$. Furthermore, by the induction hypothesis, we have that $[\![\Gamma]\!], r' : I^{\mathsf{ret}} \vdash [\![S]\!]^{\Gamma}_{r'}$ and, by Lemma 41, that $[\![\Gamma]\!], z : I^{\mathsf{bool}} \vdash [\![e]\!]^{\Gamma}_{z}$. After an application of Lemma 27 to add the type entries $r' : I^{\mathsf{ret}}, z : I^{\mathsf{bool}}, y : \mathsf{bool}$, we can then conclude the following:

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, r' : I^{\mathsf{ret}}, z : I^{\mathsf{bool}}, y : \mathsf{bool} \vdash r\langle\rangle$$

  by [T-OUT],

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, r' : I^{\mathsf{ret}}, z : I^{\mathsf{bool}}, y : \mathsf{bool} \vdash [y = \mathsf{T}][\![S]\!]^{\Gamma}_{r'} + [y = \mathsf{F}]r\langle\rangle$$

  by [T-SUM],

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, r' : I^{\mathsf{ret}}, z : I^{\mathsf{bool}} \vdash z(y).([y = \mathsf{T}][\![S]\!]^{\Gamma}_{r'} + [y = \mathsf{F}]r\langle\rangle)$$

  by [T-IN],

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, r' : I^{\mathsf{ret}}, z : I^{\mathsf{bool}} \vdash [\![e]\!]^{\Gamma}_{z} \mid z(y).([y = \mathsf{T}][\![S]\!]^{\Gamma}_{r'} + [y = \mathsf{F}]r\langle\rangle)$$

  by [T-PAR],

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, r' : I^{\mathsf{ret}} \vdash (\boldsymbol{\nu} z : I^{\mathsf{bool}})\big([\![e]\!]^{\Gamma}_{z} \mid z(y).([y = \mathsf{T}][\![S]\!]^{\Gamma}_{r'} + [y = \mathsf{F}]r\langle\rangle)\big)$$

  by [T-RES],

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, r' : I^{\mathsf{ret}} \vdash r'().(\boldsymbol{\nu} z : I^{\mathsf{bool}})\big([\![e]\!]^{\Gamma}_{z} \mid z(y).([y = \mathsf{T}][\![S]\!]^{\Gamma}_{r'} + [y = \mathsf{F}]r\langle\rangle)\big)$$

  by [T-IN],

$$[\![\Gamma]\!], r : I^{\mathsf{ret}}, r' : I^{\mathsf{ret}} \vdash !r'().(\boldsymbol{\nu} z : I^{\mathsf{bool}})$$

$$\big([\![e]\!]^\Gamma_z \mid z(y).([y = \mathsf{T}][\![S]\!]^\Gamma_{r'} + [y = \mathsf{F}]r\!<\!>)\big)$$

by [T-REP],

$$[\![\Gamma]\!], r : I^{\mathrm{ret}}, r' : I^{\mathrm{ret}} \vdash r'\!<\!>$$

by [T-OUT],

$$[\![\Gamma]\!], r : I^{\mathrm{ret}}, r' : I^{\mathrm{ret}} \vdash r'\!<\!> \mid !r'().(\boldsymbol{\nu} z : I^{\mathrm{bool}}) \big([\![e]\!]^\Gamma_z \mid z(y).([y = \mathsf{T}][\![S]\!]^\Gamma_{r'} + [y = \mathsf{F}]r\!<\!>)\big)$$

by [T-PAR], and finally

$$[\![\Gamma]\!], r : I^{\mathrm{ret}} \vdash (\boldsymbol{\nu} r' : I^{\mathrm{ret}})\Big(r'\!<\!> \mid !r'().(\boldsymbol{\nu} z : I^{\mathrm{bool}}) \big([\![e]\!]^\Gamma_z \mid z(y).([y = \mathsf{T}][\![S]\!]^\Gamma_{r'} + [y = \mathsf{F}]r\!<\!>)\big)\Big)$$

by [T-RES], as desired.

- Suppose the statement is `call` $e.f(\tilde{e})$, so $\Gamma \vdash$ `call` $e.f(\tilde{e})$. This must have been concluded by [T-CALL], and from its premise we have that

$$\begin{gathered} \Gamma \vdash e : I_A \\ \Gamma \vdash e_1 : B_1 \\ \vdots \\ \Gamma \vdash e_n : B_n \\ \Gamma(I_A)(f) = \mathrm{proc}(B_1, \dots, B_n) \end{gathered}$$

By the encoding of statements, we have that

$$\begin{aligned} &[\![\texttt{call } e.f(\tilde{e})]\!]^\Gamma_r \\ &= (\boldsymbol{\nu} a : I^{I_A}, \tilde{z} : I^{\widetilde{B}})\Big([\![e]\!]^\Gamma_a \mid [\![\tilde{e}]\!]^\Gamma_{\tilde{z}} \mid a(Y).z_1(y_1) \dots z_n(y_n).Y \cdot f\!<\!r, y_1, \dots, y_n\!>\Big) \end{aligned}$$

where $[\![\tilde{e}]\!]^\Gamma_{\tilde{z}} = [\![e_1]\!]^\Gamma_{z_1} \mid \dots \mid [\![e_n]\!]^\Gamma_{z_n}$ and $\tilde{z} : I^{\widetilde{B}} = z_1 : I^{B_1}, \dots, z_n : I^{B_n}$ for brevity. By Lemma 41, we have that $[\![\Gamma]\!], a : I^{I_A} \vdash [\![e]\!]^\Gamma_a$ and that $[\![\Gamma]\!], z_i : I^{B_i} \vdash [\![e_i]\!]^\Gamma_{z_i}$, for all $i = 1, \dots, n$. Hence, we can immediately conclude

$$[\![\Gamma]\!], r : I^{\mathrm{ret}}, a : I^{I_A}, z_1 : I^{B_1}, \dots, z_n : I^{B_n} \vdash [\![e]\!]^\Gamma_a \mid [\![e_1]\!]^\Gamma_{z_1} \mid \dots \mid [\![e_n]\!]^\Gamma_{z_n}$$

by using rule [T-PAR] and with Lemma 27 used to add the assumption $r : I^{\mathrm{ret}}$ (since we know that $r \# [\![e]\!]^\Gamma_a, [\![\tilde{e}]\!]^\Gamma_{\tilde{z}}$).

What remains to be shown is that also

$$\begin{aligned}[\![\Gamma]\!], r : I^{\mathrm{ret}}, a : I^{I_A}, z_1 : I^{B_1}, \dots, z_n : I^{B_n} \\ \vdash a(Y).z_1(y_1) \dots z_n(y_n).Y \cdot f\!<\!r, y_1, \dots, y_n\!>\end{aligned}$$

which is shown by using rule [T-IN], until we are left with the final premise to show, which is:

$$\begin{aligned}[\![\Gamma]\!], r : I^{\mathrm{ret}}, a : I^{I_A}, z_1 : I^{B_1}, \dots, z_n : I^{B_n}, Y : I_A, y_1 : B_1, \dots, y_n : B_n \\ \vdash Y \cdot f\!<\!r, y_1, \dots, y_n\!>\end{aligned}$$

From $\Gamma(I_A)(f) = \mathrm{proc}(B_1, \dots, B_n)$ we know that Γ can be expanded as

$$\Gamma = I_A : (f : \mathrm{proc}(B_1, \dots, B_n), \Delta), \Gamma'$$

and by the encoding of Γ we have that

$$\begin{aligned}&[\![I_A : (f : \mathrm{proc}(B_1, \dots, B_n), \Delta), \Gamma']\!] \\ &= I_A : (\mathsf{nil}, ([\![f : \mathrm{proc}(B_1, \dots, B_n), \Delta]\!]^2)), [\![f : \mathrm{proc}(B_1, \dots, B_n), \Delta]\!]^3, [\![\Gamma']\!] \\ &= I_A : (\mathsf{nil}, (f : I^{B_1,\dots,B_n}, [\![\Delta]\!]^2)), \\ &\qquad I^{B_1,\dots,B_n} : (\mathsf{ch}(I^{\mathrm{ret}}, B_1, \dots, B_n), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]\end{aligned}$$

Hence, we can conclude

$$[\![\Gamma]\!], \dots, Y : I_A; [\![\Gamma]\!], \dots, Y : I_A \vdash Y \cdot f : \mathsf{ch}(I^{\mathrm{ret}}, B_1, \dots, B_n)$$

by [T-VEC$_1$] (and [T-VEC$_2$] for its premise), and then

$$\begin{aligned}[\![\Gamma]\!], r : I^{\mathrm{ret}}, a : I^{I_A}, z_1 : I^{B_1}, \dots, z_n : I^{B_n}, \\ Y : I_A, y_1 : B_1, \dots, y_n : B_n \vdash Y \cdot f\!<\!r, y_1, \dots, y_n\!>\end{aligned}$$

by [T-OUT]. At last, we can therefore conclude

$$\begin{aligned}[\![\Gamma]\!], r : I^{\mathrm{ret}} \vdash (\boldsymbol{\nu} a : I^{I_A}, \tilde{z} : I^{\tilde{B}})\Big([\![e]\!]^{\Gamma}_{a} \mid [\![\tilde{e}]\!]^{\Gamma}_{\tilde{z}} \\ \mid a(Y).z_1(y_1) \dots z_n(y_n).Y \cdot f\!<\!r, y_1, \dots, y_n\!>\Big)\end{aligned}$$

by [T-PAR] and then [T-RES], as desired.

This concludes the proof for the forward direction.

For the other direction, we must show that $[\![\Gamma]\!], r : I^{\mathrm{ret}} \vdash [\![S]\!]^{\Gamma}_{r} \implies \Gamma \vdash S$ and again the proof is by induction on S.

- Suppose the statement is `skip` and $[\![\Gamma]\!], r : I^{\text{ret}} \vdash [\![\texttt{skip}]\!]^{\Gamma}_{r}$. Then $\Gamma \vdash \texttt{skip}$ holds for any Γ by [T-SKIP], as desired.

- Suppose the statement is $\texttt{var}\ B\ x\ \texttt{:=}\ e\ \texttt{in}\ S$, and

$$[\![\Gamma]\!], r : I^{\text{ret}} \vdash [\![\texttt{var}\ B\ x\ \texttt{:=}\ e\ \texttt{in}\ S]\!]$$

  Then

$$\begin{aligned}&[\![\texttt{var}\ B\ x \texttt{:=} e\ \texttt{in}\ S]\!]^{\Gamma}_{r}\\ &= (\boldsymbol{\nu} z : I^{B})\left([\![e]\!]^{\Gamma}_{z} \mid z(y).(\boldsymbol{\nu} x : I^{B})\left(x\texttt{<}y\texttt{>} \mid [\![S]\!]^{\Gamma, x:B}_{r}\right)\right)\end{aligned}$$

  Our goal is to infer $\Gamma \vdash \texttt{var}\ B\ x\ \texttt{:=}\ e\ \texttt{in}\ S$ using [T-DECV]. Thus, we must show that the premises of that rule, namely $\Gamma \vdash e : B$ and $\Gamma, x : B \vdash S$, can be satisfied.

  We know that

$$[\![\Gamma]\!], r : I^{\text{ret}} \vdash (\boldsymbol{\nu} z : I^{B})\left([\![e]\!]^{\Gamma}_{z} \mid z(y).(\boldsymbol{\nu} x : I^{B})\left(x\texttt{<}y\texttt{>} \mid [\![S]\!]^{\Gamma, x:B}_{r}\right)\right)$$

  which must have been concluded by [T-RES], [T-PAR], [T-IN], [T-RES] again, [T-PAR] again, and finally [T-OUT]. We omit the full derivation here, but see the corresponding case in the proof for the forward direction.

  From their respective premises, we get that $[\![\Gamma]\!], z : I^{B} \vdash [\![e]\!]^{\Gamma}_{z}$ and $[\![\Gamma]\!], x : I^{B}, r : I^{\text{ret}} \vdash [\![S]\!]^{\Gamma}_{r}$. By Lemma 41, the former implies that $\Gamma \vdash e : B$. The latter implies that $[\![\Gamma]\!]$ contains an interface definition for I^{B} of the form $I^{B} \mapsto (\mathsf{ch}(B), \varnothing)$; by the encoding for Γ, we have that

$$[\![\Gamma]\!], x : I^{B}, I^{B} : (\mathsf{ch}(B), \varnothing), r : I^{\text{ret}} = [\![\Gamma, x : B]\!], r : I^{\text{ret}}$$

  and thus $[\![\Gamma, x : B]\!], r : I^{\text{ret}} \vdash [\![S]\!]^{\Gamma}_{r}$. By the induction hypothesis, $\Gamma, x : B \vdash S$;

  We can then conclude $\Gamma \vdash \texttt{var}\ B\ x\ \texttt{:=}\ e\ \texttt{in}\ S$ by [T-DECV], as desired.

- Suppose the statement is $x\ \texttt{:=}\ e$, and $[\![\Gamma]\!], r : I^{\text{ret}} \vdash [\![x\ \texttt{:=}\ e]\!]^{\Gamma}_{r}$. Then

$$[\![x\ \texttt{:=}\ e]\!]^{\Gamma}_{r} = (\boldsymbol{\nu} z : I^{B})\left([\![e]\!]^{\Gamma}_{z} \mid z(y).x(w).(x\texttt{<}y\texttt{>} \mid r\texttt{<>})\right)$$

  Our goal is to conclude by [T-ASSV], thus we must show that its premises can be satisfied, which are $\Gamma \vdash x : B$ and $\Gamma \vdash e : B$.

  We know that

$$[\![\Gamma]\!], r : I^{\text{ret}} \vdash (\boldsymbol{\nu} z : I^{B})\left([\![e]\!]^{\Gamma}_{z} \mid z(y).x(w).(x\texttt{<}y\texttt{>} \mid r\texttt{<>})\right)$$

which must have been concluded by [T-RES], [T-PAR], [T-IN] (twice) and then [T-PAR] and [T-OUT]. We omit the full derivation here, but see the corresponding case in the proof for the forward direction. However, in brief, we infer from the entry $z : I^B$ that x must be typable as $\mathsf{ch}(B)$. In particular, we get that $[\![\Gamma]\!]$ can be expanded as $[\![\Gamma']\!], x : I^B$ (since x is free in the above process). Then $[\![\Gamma']\!], x : I^B, y : B \vdash x\langle y\rangle$, which was concluded by [T-OUT], and for its premise that

$$([\![\Gamma']\!], x : I^B); ([\![\Gamma']\!], x : I^B) \vdash x : \mathsf{ch}(B)$$

which was concluded by [T-VEC$_2$]. This, in turn, requires that $[\![\Gamma']\!]$ must contain $I^B \mapsto (\mathsf{ch}(B), \varnothing)$, and, by the encoding for Γ, we have that

$$[\![\Gamma']\!], x : I^B, I^B : (\mathsf{ch}(B), \varnothing), r : I^{\mathsf{ret}} = [\![\Gamma', x : B]\!], r : I^{\mathsf{ret}}$$

hence, $\Gamma = \Gamma', x : B$, and clearly $\Gamma', x : B \vdash x : B$ by [T-VAR].

Furthermore, from the premise of [T-RES] and [T-PAR], we have $[\![\Gamma]\!], z : I^B \vdash [\![e]\!]^\Gamma_z$ that, by Lemma 41, implies $\Gamma \vdash e : B$. Thus we can conclude $\Gamma \vdash x \ := \ e$ by [T-ASSV], as desired.

- Suppose the statement is $\texttt{this}.p \ := \ e$, and $[\![\Gamma]\!], r : I^{\mathsf{ret}} \vdash [\![\texttt{this}.p \ := \ e]\!]^\Gamma_r$. Then

$$\begin{aligned}&[\![\texttt{this}.p \ := \ e]\!]^\Gamma_r \\ &= (\boldsymbol{\nu} z : I^B)\big([\![e]\!]^\Gamma_z \mid z(y).\texttt{this}(Y).(\texttt{this}\langle Y\rangle \mid Y \cdot p(w).(Y \cdot p\langle y\rangle \mid r\langle\rangle))\big)\end{aligned}$$

Our goal is to conclude by [T-ASSF], thus we must show that its premises can be satisfied, which are $\Gamma \vdash \texttt{this}.p : B$ and $\Gamma \vdash e : B$.

We know that

$$\begin{aligned}[\![\Gamma]\!], r : I^{\mathsf{ret}} \vdash (\boldsymbol{\nu} z : I^B)\big([\![e]\!]^\Gamma_z \mid z(y).\texttt{this}(Y) \\ .(\texttt{this}\langle Y\rangle \mid Y \cdot p(w).(Y \cdot p\langle y\rangle \mid r\langle\rangle))\big)\end{aligned}$$

which must have been concluded by [T-RES], [T-PAR], [T-IN] and [T-OUT]. We omit the full derivation here, but see the corresponding case in the proof for the forward direction. However, in brief, we infer from the entry $z : I^B$ that y must have type B, and therefore $Y \cdot p$ must be typable as $\mathsf{ch}(B)$, which is required to type the output $Y \cdot p\langle y\rangle$. This, in turn, requires that $\texttt{this}$ has a type I^A, which must have an interface definition of the form $I_A \mapsto (\mathsf{nil}, (p : I^B, \Delta))$ for some Δ. Thus we get that $[\![\Gamma]\!] = \texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(A), \varnothing), I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]$ and, by the translation for Γ, we have that

$$\texttt{this} : I^{I_A}, I^{I_A} : (\mathsf{ch}(I_A), \varnothing), I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)),$$

$$\begin{aligned} & I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!] \\ = {} & [\![\texttt{this} : I_A, I_A : (p : B, \Delta), \Gamma']\!] \end{aligned}$$

so $\Gamma = \texttt{this} : I_A, I_A : (p : B, \Delta), \Gamma'$. Then $\texttt{this} : I_A, I_A : (p : B, \Delta), \Gamma' \vdash \texttt{this}.p : B$ can be concluded by [T-FIELD].

Like in the previous case, from the premise of [T-RES] and [T-PAR], we have that $[\![\Gamma]\!], z : I^B \vdash [\![e]\!]^\Gamma_z$ and, by Lemma 41, $\Gamma \vdash e : B$. Thus we can conclude $\Gamma \vdash \texttt{this}.p \texttt{ := } e$ by [T-ASSF], as desired.

- Suppose the statement is $S_1 ; S_2$, and $[\![\Gamma]\!], r_2 : I^{\mathsf{ret}} \vdash [\![S_1 ; S_2]\!]^\Gamma_{r_2}$. Then

$$[\![S_1 ; S_2]\!]^\Gamma_{r_2} = (\boldsymbol{\nu} r_1 : I^{\mathsf{ret}}) \left([\![S_1]\!]^\Gamma_{r_1} \mid r_1().[\![S_2]\!]^\Gamma_{r_2} \right)$$

Our goal is to conclude by [T-SEQ], thus we must show that its premises can be satisfied, which are $\Gamma \vdash S_1$ and $\Gamma \vdash S_S$.

We know that

$$[\![\Gamma]\!], r_2 : I^{\mathsf{ret}} \vdash (\boldsymbol{\nu} r_1 : I^{\mathsf{ret}}) \left([\![S_1]\!]^\Gamma_{r_1} \mid r_1().[\![S_2]\!]^\Gamma_{r_2} \right)$$

which must have been concluded by [T-RES] and [T-PAR], and with the premises:

$$[\![\Gamma]\!], r_1 : I^{\mathsf{ret}}, r_2 : I^{\mathsf{ret}} \vdash [\![S_1]\!]^\Gamma_{r_1} \qquad [\![\Gamma]\!], r_1 : I^{\mathsf{ret}}, r_2 : I^{\mathsf{ret}} \vdash r_1().[\![S_2]\!]^\Gamma_{r_2}$$

By applying [T-IN] to the latter, we also know that $[\![\Gamma]\!], r_1 : I^{\mathsf{ret}}, r_2 : I^{\mathsf{ret}} \vdash [\![S_2]\!]^\Gamma_{r_2}$. Then, after removing the unused type assumptions by Lemma 28 and by two applications of the induction hypothesis, we have the desired $\Gamma \vdash S_1$ and $\Gamma \vdash S_2$.

- Suppose the statement is $\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}$, and

$$[\![\Gamma]\!], r : I^{\mathsf{ret}} \vdash [\![\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}]\!]^\Gamma_r$$

Then

$$\begin{aligned} & [\![\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}]\!]^\Gamma_r \\ = {} & (\boldsymbol{\nu} z : I^{\mathsf{bool}}) \left([\![e]\!]^\Gamma_z \mid z(y).([y = \mathsf{T}][\![S_\mathsf{T}]\!]^\Gamma_r + [y = \mathsf{F}][\![S_\mathsf{F}]\!]^\Gamma_r) \right) \end{aligned}$$

Our goal is to conclude by [T-IF], thus we must show that its premises can be satisfied, which are $\Gamma \vdash e : \mathsf{bool}$ and $\Gamma \vdash S_\mathsf{T}$ and $\Gamma \vdash S_\mathsf{F}$.

We know that

$$[\![\Gamma]\!], r : I^{\mathsf{ret}} \vdash (\boldsymbol{\nu} z : I^{\mathsf{bool}}) \left([\![e]\!]^\Gamma_z \mid z(y).([y = \mathsf{T}][\![S_\mathsf{T}]\!]^\Gamma_r + [y = \mathsf{F}][\![S_\mathsf{F}]\!]^\Gamma_r) \right)$$

must then have been concluded by [T-RES], [T-PAR], [T-IN] and [T-SUM] (see the derivation in the corresponding forward case). From their respective premises, we get that $[\![\Gamma]\!], z : I^{\text{bool}} \vdash [\![e]\!]^{\Gamma}_{z}$ and $[\![\Gamma]\!], r : I^{\text{ret}} \vdash [\![S_{\mathsf{T}}]\!]^{\Gamma}_{r}$ and $[\![\Gamma]\!], r : I^{\text{ret}} \vdash [\![S_{\mathsf{F}}]\!]^{\Gamma}_{r}$. Respectively by Lemma 41 and by two applications of the induction hypothesis, these imply the desired $\Gamma \vdash e : \text{bool}$, $\Gamma \vdash S_{\mathsf{T}}$ and $\Gamma \vdash S_{\mathsf{F}}$.

- Suppose the statement is $\texttt{while}\ e\ \texttt{do}\ S$, and $[\![\Gamma]\!], r : I^{\text{ret}} \vdash [\![\texttt{while}\ e\ \texttt{do}\ S]\!]^{\Gamma}_{r}$. Then

$$
\begin{aligned}
&[\![\texttt{while}\ e\ \texttt{do}\ S]\!]^{\Gamma}_{r} \\
&= (\boldsymbol{\nu} r' : I^{\text{ret}})\Big(r'\texttt{<>} \\
&\qquad \mid\ !r'().(\boldsymbol{\nu} z : I^{\text{bool}})\big([\![e]\!]^{\Gamma}_{z} \mid z(y).([y = \mathsf{T}][\![S]\!]^{\Gamma}_{r'} + [y = \mathsf{F}]r\texttt{<>})\big)\Big)
\end{aligned}
$$

  Our goal is to conclude by [T-WHILE], thus we must show that its premises can be satisfied, which are $\Gamma \vdash e : \text{bool}$ and $\Gamma \vdash S$.

  We know that

$$
\begin{aligned}
[\![\Gamma]\!], r : I^{\text{ret}} \vdash (\boldsymbol{\nu} r' : I^{\text{ret}})\Big(&r'\texttt{<>} \mid\ !r'().(\boldsymbol{\nu} z : I^{\text{bool}}) \\
&\big([\![e]\!]^{\Gamma}_{z} \mid z(y).([y = \mathsf{T}][\![S]\!]^{\Gamma}_{r'} + [y = \mathsf{F}]r\texttt{<>})\big)\Big)
\end{aligned}
$$

  must have been concluded by [T-RES], [T-PAR], [T-OUT], [T-REP], [T-IN] and [T-SUM]. We omit the full derivation here, but see the corresponding case in the proof for the forward direction.

  From the respective premises of these rules, we get that $[\![\Gamma]\!], z : I^{\text{bool}} \vdash [\![e]\!]^{\Gamma}_{z}$ and $[\![\Gamma]\!], r' : I^{\text{ret}} \vdash [\![S]\!]^{\Gamma}_{r'}$, and with unused type assumptions removed by Lemma 28. Again by Lemma 41 and by the induction hypothesis, these allow us to conclude.

- Suppose the statement is $\texttt{call}\ e.f(\tilde{e})$, and $[\![\Gamma]\!], r : I^{\text{ret}} \vdash [\![\texttt{call}\ e.f(\tilde{e})]\!]^{\Gamma}_{r}$. Then

$$
\begin{aligned}
&[\![\texttt{call}\ e.f(\tilde{e})]\!]^{\Gamma}_{r} \\
&= (\boldsymbol{\nu} a : I^{I_A}, \tilde{z} : I^{\widetilde{B}})\Big([\![e]\!]^{\Gamma}_{a} \mid [\![\tilde{e}]\!]^{\Gamma}_{\tilde{z}} \mid a(Y).z_1(y_1) \ldots z_n(y_n).Y \cdot f\texttt{<}r, y_1, \ldots, y_n\texttt{>}\Big)
\end{aligned}
$$

  where $[\![\tilde{e}]\!]^{\Gamma}_{\tilde{z}} = [\![e_1]\!]^{\Gamma}_{z_1} \mid \ldots \mid [\![e_n]\!]^{\Gamma}_{z_n}$ and $\tilde{z} : I^{\widetilde{B}} = z_1 : I^{B_1}, \ldots, z_n : I^{B_n}$ for brevity.

  Our goal is to conclude by [T-CALL], and to do that, we must satisfy its premises, which are

$$
\Gamma \vdash e : B \qquad \Gamma \vdash \tilde{e} : \widetilde{B} \qquad \Gamma(I_A)(f) = \text{proc}(\widetilde{B})
$$

We have that

$$[\![\Gamma]\!], r : I^{\text{ret}} \vdash (\boldsymbol{\nu} a : I^{I_A}, \tilde{z} : I^{\tilde{B}})\Big([\![e]\!]^{\Gamma}_{a} \mid [\![\tilde{e}]\!]^{\Gamma}_{\tilde{z}} \\ \mid a(Y).z_1(y_1) \dots z_n(y_n).Y \cdot f\langle r, y_1, \dots, y_n\rangle\Big)$$

must have been concluded by [T-RES], [T-PAR], [T-IN] and [T-OUT]. We omit the full derivation here, but see the corresponding case in the proof for the forward direction. From their respective premises, we get that

$$\begin{gathered}[\![\Gamma]\!], a : I^{I_A} \vdash [\![e]\!]^{\Gamma}_{a} \\ [\![\Gamma]\!], z_1 : I^{B_1} \vdash [\![e_1]\!]^{\Gamma}_{z_1} \\ \vdots \\ [\![\Gamma]\!], z_n : I^{B_n} \vdash [\![e_n]\!]^{\Gamma}_{z_n}\end{gathered}$$

and that $[\![\Gamma]\!], r : I^{\text{ret}}, Y : I_A, y_1 : B_1, \dots, y_n : B_n \vdash Y \cdot f\langle r, y_1, \dots, y_n\rangle$, with unused type assumptions removed by Lemma 28. This must have been concluded by [T-OUT], and from its premise we then get that

$$([\![\Gamma]\!], r : I^{\text{ret}}, Y : I_A, y_1 : B_1, \dots, y_n : B_n); \\ ([\![\Gamma]\!], r : I^{\text{ret}}, Y : I_A, y_1 : B_1, \dots, y_n : B_n) \vdash Y \cdot f : \mathsf{ch}(I^{\text{ret}}, B_1, \dots, B_n)$$

which was concluded by [T-VEC$_1$] and [T-VEC$_2$]. Again, we omit the full derivation here, since it appears in the forward case. However, it requires that $[\![\Gamma]\!]$ must contain a type definition for I_A, i.e.:

$$[\![\Gamma]\!] = I_A : (\mathsf{nil}, (f : I^{B_1,\dots,B_n}, [\![\Delta]\!]^2)), \\ I^{B_1,\dots,B_n} : (\mathsf{ch}(I^{\text{ret}}, B_1, \dots, B_n), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]$$

and by the encoding of Γ, we have that

$$\begin{aligned}&[\![I_A : (f : \text{proc}(B_1, \dots, B_n), \Delta), \Gamma']\!] \\ &= I_A : (\mathsf{nil}, ([\![f : \text{proc}(B_1, \dots, B_n), \Delta]\!]^2)), \\ &\qquad [\![f : \text{proc}(B_1, \dots, B_n), \Delta]\!]^3, [\![\Gamma']\!] \\ &= I_A : (\mathsf{nil}, (f : I^{B_1,\dots,B_n}, [\![\Delta]\!]^2)), \\ &\qquad I^{B_1,\dots,B_n} : (\mathsf{ch}(I^{\text{ret}}, B_1, \dots, B_n), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]\end{aligned}$$

Thus, Γ can be written as $\Gamma = I_A : (f : \text{proc}(B_1, \dots, B_n), \Delta), \Gamma'$, and clearly

$$(I_A : (f : \text{proc}(B_1, \dots, B_n), \Delta), \Gamma')(I_A)(f) = \text{proc}(B_1, \dots, B_n)$$

Furthermore, by using Lemma 41, we conclude the desired

$$\begin{aligned}&\Gamma \vdash e : I_A\\&\Gamma \vdash e_1 : B_1\\&\quad\vdots\\&\Gamma \vdash e_n : B_n\end{aligned}$$

This concludes the proof for the other direction. □

Lemma 43. *Assume* $\Gamma(A) = I_A$. *Then* $\Gamma \vdash_A \mathsf{env}_M \iff [\![\Gamma]\!] \vdash [\![\mathsf{env}_M]\!]_A^\Gamma$.

Proof. We have two directions to show. For both, we proceed by induction on the structure of env_M. For the forward direction, we must show that $\Gamma \vdash_A \mathsf{env}_M \implies [\![\Gamma]\!] \vdash [\![\mathsf{env}_M]\!]_A^\Gamma$.

- Suppose $\mathsf{env}_M = \mathsf{env}_M^\emptyset$. Then $\Gamma \vdash_A \mathsf{env}_M^\emptyset$ was concluded by [T-ENV$^\emptyset$], which is an axiom. The translation for environments then yields $[\![\mathsf{env}_M^\emptyset]\!]_A^\Gamma = \mathbf{0}$, and $[\![\Gamma]\!] \vdash \mathbf{0}$ can then be concluded by [T-NIL] for any type environment $[\![\Gamma]\!]$.
- Suppose $\mathsf{env}_M = (f, (\tilde{x}, S)), \mathsf{env}'_M$. Then $\Gamma \vdash_A (f, (\tilde{x}, S)), \mathsf{env}'_M$ must have been concluded by [T-ENV$_M$], and from its premise we have that

$$\Gamma(I_A)(f) = \mathrm{proc}(\widetilde{B}) \qquad \Gamma, \texttt{this} : I_A, \tilde{x} : \widetilde{B} \vdash S \qquad \Gamma \vdash_A \mathsf{env}'_M$$

By the translation for environments, we have that

$$\begin{aligned}&[\![(f, (\tilde{x}, S)), \mathsf{env}'_M]\!]_A^\Gamma\\ &= [\![\mathsf{env}'_M]\!]_A^\Gamma \mid\, !A \cdot f(r, \tilde{a}).(\boldsymbol{\nu} \tilde{x} : I^{\widetilde{B}} := \tilde{a})(\boldsymbol{\nu} \texttt{this} : I^{I_A} := A)\left([\![S]\!]_r^\Gamma\right)\\ &= [\![\mathsf{env}'_M]\!]_A^\Gamma \mid\, !A \cdot f(r, \tilde{a}).(\boldsymbol{\nu} x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \texttt{this} : I^{I_A})\\ &\qquad \left(x_1\texttt{<}a_1\texttt{>} \mid \ldots \mid x_n\texttt{<}a_n\texttt{>} \mid \texttt{this<}A\texttt{>} \mid [\![S]\!]_r^\Gamma\right)\end{aligned}$$

where $\Gamma(A) = I_A$ and $\Gamma(I_A)(f) = \mathrm{proc}(\widetilde{B})$. By the induction hypothesis, we have that $[\![\Gamma]\!] \vdash [\![\mathsf{env}_M]\!]_A^\Gamma$. Thus, to allow us to conclude by [T-PAR], what remains to be shown is that

$$\begin{aligned}[\![\Gamma]\!] \vdash\, &!A \cdot f(r, \tilde{a}).(\boldsymbol{\nu} x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \texttt{this} : I^{I_A})\\ &\left(x_1\texttt{<}a_1\texttt{>} \mid \ldots \mid x_n\texttt{<}a_n\texttt{>} \mid \texttt{this<}A\texttt{>} \mid [\![S]\!]_r^\Gamma\right)\end{aligned}$$

We note that

$$\begin{aligned}&[\![\Gamma]\!], r : I^{\mathrm{ret}}, a_1 : B_1, \ldots, a_n : B_n, x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \texttt{this} : I^{I_A}\\ &\quad \vdash x_1\texttt{<}a_1\texttt{>} \mid \ldots \mid x_n\texttt{<}a_n\texttt{>} \mid \texttt{this<}A\texttt{>}\end{aligned}$$

easily holds by using [T-OUT] and [T-PAR]. What remains to be shown is that

$$[\![\Gamma]\!], x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, r : I^{\mathrm{ret}}, \mathtt{this} : I^{I_A} \vdash [\![S]\!]_r^\Gamma$$

where the entries $a_1 : B_1, \ldots, a_n : B_n$ have been removed by Lemma 28, since none of them occur in $[\![S]\!]_r^\Gamma$. As we know that $\Gamma, \mathtt{this} : I_A, \tilde{x} : \widetilde{B} \vdash S$, we have by Lemma 42 that

$$\Gamma, \tilde{x} : \widetilde{B}, \mathtt{this} : I_A \vdash S \implies [\![\Gamma, \tilde{x} : \widetilde{B}, \mathtt{this} : I_A]\!], r : I^{\mathrm{ret}} \vdash [\![S]\!]_r^\Gamma$$

so we must show that

$$[\![\Gamma, \tilde{x} : \widetilde{B}, \mathtt{this} : I_A]\!] = [\![\Gamma]\!], x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, r : I^{\mathrm{ret}}$$

This is seen to be the case from the translation of Γ (local variables), which gives $[\![x : B, \Gamma]\!] = x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma]\!]$. In the case above, the interface definition $I^B : (\mathsf{ch}(B), \varnothing)$ is omitted, since it is already assumed to be in $[\![\Gamma]\!]$, since we assume interface types are defined for each base type B that is used. Thus, by using [T-RES], we can conclude

$$[\![\Gamma]\!], r : I^{\mathrm{ret}}, a_1 : B_1, \ldots, a_n : B_n \vdash (\boldsymbol{\nu} x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \mathtt{this} : I^{I_A})$$
$$\left(x_1\texttt{<}a_1\texttt{>} \mid \ \ldots \ \mid x_n\texttt{<}a_n\texttt{>} \mid \mathtt{this}\texttt{<}A\texttt{>} \mid [\![S]\!]_r^\Gamma\right)$$

Let

$$P \triangleq (\boldsymbol{\nu} x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \mathtt{this} : I^{I_A})$$
$$\left(x_1\texttt{<}a_1\texttt{>} \mid \ \ldots \ \mid x_n\texttt{<}a_n\texttt{>} \mid \mathtt{this}\texttt{<}A\texttt{>} \mid [\![S]\!]_r^\Gamma\right)$$

for readability. What remains to be shown is that

$$[\![\Gamma]\!] \vdash \ !A \cdot f(r, \tilde{a}).P$$

which holds by [T-REP] and [T-IN], *if*, by using [T-VEC$_1$], we can conclude that

$$[\![\Gamma]\!]; [\![\Gamma]\!] \vdash A \cdot p : \mathsf{ch}(I^{\mathrm{ret}}, \widetilde{B})$$

To show this, recall that we know that $\Gamma(A) = I_A$ and $\Gamma(I_A)(f) = \mathrm{proc}(\widetilde{B})$. Thus, we also know that the type environment can be expanded as $\Gamma = A : I_A, I_A : (f : \mathrm{proc}(\widetilde{B}), \Delta), \Gamma'$. and by the encoding of Γ we have that

$$\begin{aligned}
&[\![A : I_A, I_A : (f : \mathrm{proc}(B_1, \ldots, B_n), \Delta), \Gamma']\!] \\
&= A : I_A, I_A : (\mathsf{nil}, ([\![f : \mathrm{proc}(B_1, \ldots, B_n), \Delta]\!]^2)), \\
&\qquad [\![f : \mathrm{proc}(B_1, \ldots, B_n), \Delta]\!]^3, [\![\Gamma']\!] \\
&= A : I_A, I_A : (\mathsf{nil}, (f : I^{B_1,\ldots,B_n}, [\![\Delta]\!]^2)), \\
&\qquad I^{B_1,\ldots,B_n} : (\mathsf{ch}(I^{\mathrm{ret}}, B_1, \ldots, B_n), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]
\end{aligned}$$

which lets us conclude that

$$[\![A : I_A, I_A : (f : \mathrm{proc}(B_1, \ldots, B_n), \Delta), \Gamma']\!](A) = I_A$$

and

$$[\![A : I_A, I_A : (f : \mathrm{proc}(B_1, \ldots, B_n), \Delta), \Gamma']\!](I_A) = (\mathsf{nil}, (f : I^{B_1,\ldots,B_n}, [\![\Delta]\!]^2))$$

and

$$\mathrm{snd}(\mathsf{nil}, (f : I^{B_1,\ldots,B_n}, [\![\Delta]\!]^2)) = (f : I^{B_1,\ldots,B_n}, [\![\Delta]\!]^2)$$

and

$$(f : I^{B_1,\ldots,B_n}, [\![\Delta]\!]^2)(f) = I^{B_1,\ldots,B_n}$$

and

$$\begin{aligned}&[\![A : I_A, I_A : (f : \mathrm{proc}(B_1, \ldots, B_n), \Delta), \Gamma']\!](I^{B_1,\ldots,B_n})\\ &= (\mathsf{ch}(I^{\mathrm{ret}}, B_1, \ldots, B_n), \varnothing)\end{aligned}$$

and finally

$$\mathrm{fst}(\mathsf{ch}(I^{\mathrm{ret}}, B_1, \ldots, B_n), \varnothing) = \mathsf{ch}(I^{\mathrm{ret}}, B_1, \ldots, B_n)$$

as required.

This concludes the proof for the forward direction.
For the other direction, we must show that $[\![\Gamma]\!] \vdash [\![\mathsf{env}_M]\!]^\Gamma_A \implies \Gamma \vdash_A \mathsf{env}_M$.

- Suppose $\mathsf{env}_M = \mathsf{env}^\varnothing_M$. The translation for environments yields $[\![\mathsf{env}^\varnothing_M]\!]^\Gamma_A = \mathbf{0}$, and $[\![\Gamma]\!] \vdash \mathbf{0}$ was then concluded by [T-NIL], which holds for any type environment $[\![\Gamma]\!]$. Then $\Gamma \vdash_A \mathsf{env}^\varnothing_M$ can be concluded by [T-ENV$^\varnothing$].
- Suppose $\mathsf{env}_M = (f, (\tilde{x}, S)), \mathsf{env}'_M$. By the translation for environments, we have that

$$\begin{aligned}&[\![(f, (\tilde{x}, S)), \mathsf{env}'_M]\!]^\Gamma_A\\ &= [\![\mathsf{env}'_M]\!]^\Gamma_A \mid\ !A \cdot f(r, \tilde{a}).(\boldsymbol{\nu} x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \mathtt{this} : I^{I_A})\\ &\qquad (x_1\texttt{<}a_1\texttt{>} \mid \ \ldots\ \mid x_n\texttt{<}a_n\texttt{>} \mid \mathtt{this}\texttt{<}A\texttt{>} \mid [\![S]\!]^\Gamma_r)\end{aligned}$$

  where $\Gamma(A) = I_A$ and $\Gamma(I_A)(f) = \mathrm{proc}(\widetilde{B})$. Our goal is to conclude by [T-ENV$_M$], which requires us to show that the following premises are satisfied:

$$\Gamma(A) = I_A \quad \Gamma(I_A)(f) = \mathrm{proc}(\widetilde{B}) \quad \Gamma, \mathtt{this} : I_A, \tilde{x} : \widetilde{B} \vdash S \quad \Gamma \vdash_A \mathsf{env}'_M$$

We know that

$$\llbracket\Gamma\rrbracket \vdash \llbracket \mathsf{env}'_M \rrbracket^{\Gamma}_A \mid\ !A \cdot f(r, \tilde{a}).(\boldsymbol{\nu} x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \mathtt{this} : I^{I_A}) \\ \left(x_1\texttt{<}a_1\texttt{>} \mid \ \ldots\ \mid x_n\texttt{<}a_n\texttt{>} \mid \mathtt{this}\texttt{<}A\texttt{>} \mid \llbracket S \rrbracket^{\Gamma}_r\right)$$

which must have been concluded by [T-PAR], and from its premise we then get that

$$\llbracket\Gamma\rrbracket \vdash \llbracket \mathsf{env}'_M \rrbracket^{\Gamma}_A \\ \llbracket\Gamma\rrbracket \vdash\ !A \cdot f(r, \tilde{a}).(\boldsymbol{\nu} x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \mathtt{this} : I^{I_A}) \\ \left(x_1\texttt{<}a_1\texttt{>} \mid \ \ldots\ \mid x_n\texttt{<}a_n\texttt{>} \mid \mathtt{this}\texttt{<}A\texttt{>} \mid \llbracket S \rrbracket^{\Gamma}_r\right)$$

By the induction hypothesis applied to the former, we obtain $\Gamma \vdash_A \mathsf{env}'_M$. The other premise must have been concluded by a combination of [T-REP], [T-IN], [T-RES] and [T-PAR]. We omit the lengthy derivation here, but see the corresponding case for the forward direction. However, it yields a premise

$$\llbracket\Gamma\rrbracket, x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, r : I^{\mathrm{ret}}, \mathtt{this} : I^{I_A} \vdash \llbracket S \rrbracket^{\Gamma}_r$$

for one of the applications of [T-PAR], and by Lemma 42 we have that

$$\llbracket\Gamma, \tilde{x} : \widetilde{B}, \mathtt{this} : I_A\rrbracket, r : I^{\mathrm{ret}} \vdash \llbracket S \rrbracket^{\Gamma}_r \implies \Gamma, \tilde{x} : \widetilde{B}, \mathtt{this} : I_A \vdash S$$

Finally, typing the input $A \cdot f(r, \tilde{a}).P$ with the continuation

$$P \triangleq (\boldsymbol{\nu} x_1 : I^{B_1}, \ldots, x_n : I^{B_n}, \mathtt{this} : I^{I_A}) \\ \left(x_1\texttt{<}a_1\texttt{>} \mid \ \ldots\ \mid x_n\texttt{<}a_n\texttt{>} \mid \mathtt{this}\texttt{<}A\texttt{>} \mid \llbracket S \rrbracket^{\Gamma}_r\right)$$

is likewise shown to require the subject $A \cdot p$ to be typable as $\mathsf{ch}(I^{\mathrm{ret}}, B_1, \ldots, B_n)$, which is concluded by [T-VEC$_1$] and [T-VEC$_2$]. It must therefore be the case that

$$\llbracket\Gamma\rrbracket = A : I_A, I_A : (\mathsf{nil}, (f : I^{B_1,\ldots,B_n}, \llbracket\Delta\rrbracket^2)), \\ I^{B_1,\ldots,B_n} : (\mathsf{ch}(I^{\mathrm{ret}}, B_1, \ldots, B_n), \varnothing), \llbracket\Delta\rrbracket^3, \llbracket\Gamma'\rrbracket$$

and, by the encoding of Γ:

$$\llbracket A : I_A, I_A : (f : \mathrm{proc}(B_1, \ldots, B_n), \Delta), \Gamma'\rrbracket \\ = A : I_A, I_A : (\mathsf{nil}, (f : I^{B_1,\ldots,B_n}, \llbracket\Delta\rrbracket^2)), \\ I^{B_1,\ldots,B_n} : (\mathsf{ch}(I^{\mathrm{ret}}, B_1, \ldots, B_n), \varnothing), \llbracket\Delta\rrbracket^3, \llbracket\Gamma'\rrbracket$$

Thus, $\Gamma = A : I_A, I_A : (f : \text{proc}(B_1, \dots, B_n), \Delta), \Gamma'$, and clearly,

$$(A : I_A, I_A : (f : \text{proc}(B_1, \dots, B_n), \Delta), \Gamma')(A) = I_A$$
$$(A : I_A, I_A : (f : \text{proc}(B_1, \dots, B_n), \Delta), \Gamma')(I_A)(f) = \text{proc}(B_1, \dots, B_n)$$

Thus, all the premises of [T-ENV$_M$] are satisfied, and we can therefore conclude that $\Gamma \vdash (f, (\tilde{x}, S)), \text{env}'_M$, as desired.

This concludes the proof for the other direction. □

Lemma 44. $\Gamma \vdash \text{env}_T \iff [\![\Gamma]\!] \vdash [\![\text{env}_T]\!]^\Gamma$.

Proof. We have two directions to show. For both, we proceed by induction on the structure of env_T. For the forward direction, we must show that $\Gamma \vdash \text{env}_T \implies [\![\Gamma]\!] \vdash [\![\text{env}_T]\!]^\Gamma$.

- Suppose $\text{env}_T = \text{env}_T^\emptyset$. Then $\Gamma \vdash \text{env}_T^\emptyset$ was concluded by [T-ENV$^\emptyset$], which is an axiom. The translation for environments then yields $[\![\text{env}_T^\emptyset]\!]^\Gamma = \mathbf{0}$, and $[\![\Gamma]\!] \vdash \mathbf{0}$ can then be concluded by [T-NIL] for any type environment $[\![\Gamma]\!]$.
- Suppose $\text{env}_T = (A, \text{env}_M), \text{env}'_T$. Then $\Gamma \vdash (A, \text{env}_M), \text{env}'_T$ must have been concluded by [T-ENV$_T$], and from its premise we have that $\Gamma \vdash_A \text{env}_M$ and $\Gamma \vdash \text{env}'_T$, where $\Gamma(A) = I_A$ for some interface type I_A. By the encoding for environments, we have that

  $$[\![(A, \text{env}_M), \text{env}'_T]\!]^\Gamma = [\![\text{env}'_T]\!]^\Gamma \mid [\![\text{env}_M]\!]^\Gamma_A$$

  By the induction hypothesis, we have that $[\![\Gamma]\!] \vdash [\![\text{env}'_T]\!]$ and, by Lemma 43, that $[\![\Gamma]\!] \vdash [\![\text{env}_M]\!]^\Gamma_A$. Hence

  $$[\![\Gamma]\!] \vdash [\![\text{env}'_T]\!]^\Gamma \mid [\![\text{env}_M]\!]^\Gamma_A$$

  can be concluded by [T-PAR] as desired.

This concludes the proof for the forward direction.
For the other direction, we must show that $[\![\Gamma]\!] \vdash [\![\text{env}_T]\!]^\Gamma \implies \Gamma \vdash \text{env}_T$.

- Suppose $\text{env}_T = \text{env}_T^\emptyset$. The translation for environments yields $[\![\text{env}_T^\emptyset]\!]^\Gamma = \mathbf{0}$, and $[\![\Gamma]\!] \vdash \mathbf{0}$ must therefore have been concluded by [T-NIL], which is an axiom and holds for any type environment $[\![\Gamma]\!]$. Then $\Gamma \vdash \text{env}_T^\emptyset$ can be concluded by [T-ENV$^\emptyset$].

- Suppose $\mathsf{env}_T = (A, \mathsf{env}_M), \mathsf{env}'_T$. By the encoding for environments, we have that
$$[\![(A, \mathsf{env}_M), \mathsf{env}'_T]\!]^\Gamma = [\![\mathsf{env}'_T]\!]^\Gamma \mid [\![\mathsf{env}_M]\!]^\Gamma_A$$
and $[\![\Gamma]\!] \vdash [\![\mathsf{env}'_T]\!]^\Gamma \mid [\![\mathsf{env}_M]\!]^\Gamma_A$ must therefore have been concluded by [T-PAR]. From its premise, we then know that $[\![\Gamma]\!] \vdash [\![\mathsf{env}'_T]\!]^\Gamma$ and $[\![\Gamma]\!] \vdash [\![\mathsf{env}_M]\!]^\Gamma_A$. By the induction hypothesis on the former and Lemma 43 to the latter, we obtain $\Gamma \vdash \mathsf{env}'_T$ and $\Gamma \vdash_A \mathsf{env}_M$. Then we can conclude $\Gamma \vdash (A, \mathsf{env}_M), \mathsf{env}'_T$ by [T-ENV$_T$], as desired.

This concludes the proof for the other direction. □

Lemma 45. *Assume* $\Gamma(A) = I_A$. *Then* $\Gamma \vdash_A \mathsf{env}_F \iff [\![\Gamma]\!] \vdash [\![\mathsf{env}_F]\!]_A$.

Proof. We have two directions to show. For both, we proceed by induction on the structure of env_F. For the forward direction, we must show that $\Gamma \vdash_A \mathsf{env}_F \implies [\![\Gamma]\!] \vdash [\![\mathsf{env}_F]\!]_A$.

- Suppose $\mathsf{env}_F = \mathsf{env}^\varnothing_F$. Then $\Gamma \vdash \mathsf{env}^\varnothing_F$ was concluded by [T-ENV$^\varnothing$], which is an axiom. The translation for environments then yields $[\![\mathsf{env}^\varnothing_F]\!] = \mathbf{0}$, and $[\![\Gamma]\!] \vdash \mathbf{0}$ can then be concluded by [T-NIL] for any type environment $[\![\Gamma]\!]$.

- Suppose $\mathsf{env}_F = (p, v), \mathsf{env}'_F$. Then $\Gamma \vdash_A (p, v), \mathsf{env}'_F$ must have been concluded by [T-ENV$_F$], and from its premise we have that
$$\Gamma(A) = I_A \qquad \Gamma(I_A)(p) = B \qquad \Gamma \vdash v : B \qquad \Gamma \vdash_A \mathsf{env}'_F$$
By the encoding of environments, we have that
$$[\![(p, v), \mathsf{env}'_F]\!]_A = [\![\mathsf{env}'_F]\!]_A \mid A \cdot p\langle v\rangle$$
By Lemma 40, we have that $[\![\Gamma]\!] \vdash v : B$ and, by the induction hypothesis, that $[\![\Gamma]\!] \vdash [\![\mathsf{env}'_F]\!]$. What remains to be shown is therefore that $[\![\Gamma]\!]; [\![\Gamma]\!] \vdash A \cdot p : \mathsf{ch}(B)$. As we know that $\Gamma(A) = I_A$ and $\Gamma(I_A)(p) = B$, we can write Γ as $A : I_A, I_A : (p : B, \Delta), \Gamma'$ for some Δ. By the encoding of Γ, we then have that
$$\begin{aligned} &[\![A : I_A, I_A : (p : B, \Delta), \Gamma']\!] \\ &= A : I_A, I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!] \end{aligned}$$
and we can therefore conclude
$$(A : I_A, I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!])(A) = I_A$$

and

$$(A : I_A, I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!])(I_A) = (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2))$$

and

$$\mathsf{snd}(\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)) = (p : I^B, [\![\Delta]\!]^2)$$
$$(p : I^B, [\![\Delta]\!]^2)(p) = I^B$$

and

$$(A : I_A, I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!])(I^B) = (\mathsf{ch}(B), \varnothing)$$

and finally $\mathsf{fst}(\mathsf{ch}(B), \varnothing) = \mathsf{ch}(B)$, by [T-VEC$_1$] and [T-VEC$_2$]. Then, by [T-OUT] and [T-PAR], we can conclude

$$[\![\Gamma]\!] \vdash [\![\mathsf{env}'_F]\!]_A \mid A \cdot p\langle v \rangle$$

as desired.

This concludes the proof for the forward direction.
For the other direction, we must show that $[\![\Gamma]\!] \vdash [\![\mathsf{env}_F]\!]_A \implies \Gamma \vdash_A \mathsf{env}_F$.

- Suppose $\mathsf{env}_F = \mathsf{env}_F^{\varnothing}$. The translation for environments yields $[\![\mathsf{env}_F^{\varnothing}]\!] = \mathbf{0}$, and $[\![\Gamma]\!] \vdash \mathbf{0}$ was concluded by [T-NIL] for any type environment $[\![\Gamma]\!]$. Then $\Gamma \vdash \mathsf{env}_F^{\varnothing}$ can be concluded by [T-ENV$^{\varnothing}$].
- Suppose $\mathsf{env}_F = (p, v), \mathsf{env}'_F$. By the encoding of environments, we have that

  $$[\![(p, v), \mathsf{env}'_F]\!]_A = [\![\mathsf{env}'_F]\!]_A \mid A \cdot p\langle v \rangle$$

  so $[\![\Gamma]\!] \vdash [\![\mathsf{env}'_F]\!]_A \mid A \cdot p\langle v \rangle$ must have been concluded by [T-PAR], and from its premise we have that $[\![\Gamma]\!] \vdash [\![\mathsf{env}'_F]\!]_A$ and $[\![\Gamma]\!] \vdash A \cdot p\langle v \rangle$. by the induction hypothesis on the former, we have that $\Gamma \vdash_A \mathsf{env}'_F$. The latter must have been concluded by [E-OUT], and from its premise we then get that $[\![\Gamma]\!]; [\![\Gamma]\!] \vdash A \cdot p : \mathsf{ch}(B)$ and $[\![\Gamma]\!] \vdash v : B$. by Lemma 40 to the latter, we conclude $\Gamma \vdash v : B$.

  What remains to be shown is that $\Gamma(A) = I_A$ and $\Gamma(I_A)(p) = B$. As we know that $[\![\Gamma]\!]; [\![\Gamma]\!] \vdash A \cdot p : \mathsf{ch}(B)$ was concluded by [T-VEC$_1$] and [T-VEC$_2$], we also know that $[\![\Gamma]\!]$ must contain the entries

  $$[\![\Gamma]\!] = A : I_A, I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]$$

(see the derivation in the corresponding case in the forward direction). Hence, by the translation of Γ we then have that

$$\begin{aligned}&[\![A : I_A, I_A : (p : B, \Delta), \Gamma']\!]\\ =\ &A : I_A, I_A : (\mathsf{nil}, (p : I^B, [\![\Delta]\!]^2)), I^B : (\mathsf{ch}(B), \varnothing), [\![\Delta]\!]^3, [\![\Gamma']\!]\end{aligned}$$

and clearly

$$\begin{aligned}(A : I_A, I_A : (p : B, \Delta), \Gamma')(A) &= I_A\\ (A : I_A, I_A : (p : B, \Delta), \Gamma')(I_A)(p) &= B\end{aligned}$$

This allows us to conclude $\Gamma \vdash_A (p, v), \mathsf{env}'_F$ by [T-ENV$_F$] as desired.

This concludes the proof for the other direction. □

Lemma 46. $\Gamma \vdash \mathsf{env}_S \iff [\![\Gamma]\!] \vdash [\![\mathsf{env}_S]\!]$.

Proof. We have two directions to show. For both, we proceed by induction on the structure of env_S. For the forward direction, we must show that $\Gamma \vdash \mathsf{env}_S \implies [\![\Gamma]\!] \vdash [\![\mathsf{env}_S]\!]$.

- Suppose $\mathsf{env}_S = \mathsf{env}_S^{\varnothing}$. Then $\Gamma \vdash \mathsf{env}_S^{\varnothing}$ was concluded by [T-ENV$^{\varnothing}$], which is an axiom. The translation for environments then yields $[\![\mathsf{env}_S^{\varnothing}]\!] = \mathbf{0}$, and $[\![\Gamma]\!] \vdash \mathbf{0}$ can then be concluded by [T-NIL] for any type environment $[\![\Gamma]\!]$.
- Suppose $\mathsf{env}_S = (A, \mathsf{env}_F, \mathsf{env}'_S)$. Then $\Gamma \vdash (A, \mathsf{env}_F), \mathsf{env}'_S$ must have been concluded by [T-ENV$_S$], and from its premise we have that $\Gamma \vdash_A \mathsf{env}_F$ and $\Gamma \vdash \mathsf{env}'_S$, where $\Gamma(A) = I_A$ for some interface type I_A. By the encoding of environments, we have that

  $$[\![(A, \mathsf{env}_F), \mathsf{env}'_S]\!] = [\![\mathsf{env}'_S]\!] \mid [\![\mathsf{env}_F]\!]_A$$

  By the induction hypothesis, we have that $[\![\Gamma]\!] \vdash [\![\mathsf{env}'_S]\!]$ and, by Lemma 45, that $[\![\Gamma]\!] \vdash [\![\mathsf{env}_F]\!]_A$. Hence

  $$[\![\Gamma]\!] \vdash [\![\mathsf{env}'_S]\!]^{\Gamma} \mid [\![\mathsf{env}_F]\!]_A$$

  can be concluded by [T-PAR] as desired.

This concludes the proof for the forward direction.
For the other direction, we must show that $[\![\Gamma]\!] \vdash [\![\mathsf{env}_S]\!] \implies \Gamma \vdash \mathsf{env}_S$.

- Suppose $\mathsf{env}_S = \mathsf{env}_S^{\varnothing}$. Then $[\![\mathsf{env}_S^{\varnothing}]\!] = \mathbf{0}$, and $[\![\Gamma]\!] \vdash \mathbf{0}$ was concluded by [T-NIL], which is an axiom and holds for any $[\![\Gamma]\!]$. Then $\Gamma \vdash \mathsf{env}_S^{\varnothing}$ can be concluded by [T-ENV$^{\varnothing}$].

- Suppose $\mathsf{env}_S = (A, \mathsf{env}_F, \mathsf{env}'_S)$. Then $[\![(A, \mathsf{env}_F), \mathsf{env}'_S]\!] = [\![\mathsf{env}'_S]\!] \mid [\![\mathsf{env}_F]\!]_A$, and
$$[\![\Gamma]\!] \vdash [\![(A, \mathsf{env}_F), \mathsf{env}'_S]\!] = [\![\mathsf{env}'_S]\!] \mid [\![\mathsf{env}_F]\!]_A$$
was concluded by [T-PAR], and from its premise we know that $[\![\Gamma]\!] \vdash [\![\mathsf{env}'_S]\!]$ and $[\![\Gamma]\!] \vdash [\![\mathsf{env}_F]\!]_A$. By the induction hypothesis on the former and Lemma 45 on the latter, we have that $\Gamma \vdash \mathsf{env}'_S$ and $\Gamma \vdash_A \mathsf{env}_F$. Then we can conclude $\Gamma \vdash (A, \mathsf{env}_F), \mathsf{env}'_S$ by [T-ENV$_S$], as desired.

This concludes the proof for the other direction. □

Lemma 47. *Assume* $[\![\Gamma]\!] \vdash P$ *for some process* P. *Then* $\Gamma \vdash \mathsf{env}_V \iff [\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^\Gamma\,[P]$.

Proof. We have two directions to show. For both, we proceed by induction on the structure of env_V. For the forward direction, we must show that $\Gamma \vdash \mathsf{env}_V \implies [\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^\Gamma\,[P]$.

- Suppose $\mathsf{env}_V = \mathsf{env}_V^\varnothing$. Then $\Gamma \vdash \mathsf{env}_V^\varnothing$ was concluded by [T-ENV$^\varnothing$], which is an axiom. The translation for environments then yields $[\![\mathsf{env}_V^\varnothing]\!]^\Gamma = [\ ]$, hence $[\![\mathsf{env}_V^\varnothing]\!]^\Gamma\,[P] = P$, and $[\![\Gamma]\!] \vdash P$ holds by assumption.
- Suppose $\mathsf{env}_V = (x, v), \mathsf{env}'_V$. Then $\Gamma \vdash (x, v), \mathsf{env}'_V$ must have been concluded by [T-ENV$_V$], and from its premise we have that $\Gamma(x) = B$ and $\Gamma \vdash v : B$ and $\Gamma \vdash \mathsf{env}'_V$.

  By the encoding of environments, we have that
$$\begin{aligned} &[\![(x, v), \mathsf{env}'_V]\!]^\Gamma \\ &= (\boldsymbol{\nu} x : I^{\Gamma(x)} := v)\left([\![\mathsf{env}'_V]\!]^\Gamma\right) \\ &= (\boldsymbol{\nu} x : I^B)\left(x\langle v\rangle \mid [\![\mathsf{env}'_V]\!]^\Gamma\right) \end{aligned}$$
  and since $\Gamma(x) = B$, we can expand Γ as $\Gamma = x : B, \Gamma'$. The translation for Γ then yields
$$[\![x : B, \Gamma']\!] = x : I^B, I^B : (\mathsf{ch}(B), \varnothing), [\![\Gamma']\!]$$
  so we know that the interface definition $I^B : (\mathsf{ch}(B), \varnothing)$ is present in $[\![\Gamma]\!]$. Thus we can conclude
$$\begin{aligned} [\![\Gamma]\!] &\vdash [\![\mathsf{env}'_V]\!]^\Gamma\,[P] && \text{by the induction hypothesis} \\ [\![\Gamma]\!], x : I^B &\vdash x\langle v\rangle \mid [\![\mathsf{env}'_V]\!]^\Gamma\,[P] && \text{by [T-OUT] and [T-PAR]} \\ [\![\Gamma]\!] &(\boldsymbol{\nu} x : I^B)\left(x\langle v\rangle \mid [\![\mathsf{env}'_V]\!]^\Gamma\,[P]\right) && \text{by [T-RES]} \end{aligned}$$
  as desired.

This concludes the proof for the forward direction.
For the other direction, we must show that $[\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^\Gamma\,[P] \implies \Gamma \vdash \mathsf{env}_V$.

- Suppose $\mathsf{env}_V = \mathsf{env}_V^\varnothing$, so $[\![\mathsf{env}_V^\varnothing]\!]^\Gamma = [\,]$, and $[\![\Gamma]\!] \vdash [\![\mathsf{env}_V^\varnothing]\!]^\Gamma\,[P]$ therefore holds by assumption. Then $\Gamma \vdash \mathsf{env}_V^\varnothing$ can be concluded by [T-ENV$^\varnothing$] for any Γ.
- Suppose $\mathsf{env}_V = (x_1, v_1), \ldots, (x_n, v_n), \mathsf{env}_V^\varnothing$, so

  $$[\![\Gamma]\!] \vdash [\![(x_1, v_1), \ldots, (x_n, v_n), \mathsf{env}_V^\varnothing]\!]^\Gamma\,[P]$$

  By the encoding of env_V, we have that it is translated as a process context consisting of a list of scopes and bound outputs

  $$(\boldsymbol{\nu} x_1 : I^{\Gamma(x_1)}, \ldots, x_n : I^{\Gamma(x_n)})\,(x_1\langle v_1\rangle \mid \ldots \mid x_n\langle v_n\rangle \mid [\,])$$

  So, $[\![\Gamma]\!] \vdash [\![\mathsf{env}_V]\!]^\Gamma\,[P]$ is concluded by [T-RES] and [T-PAR], and from the premise of the latter, we get that

  $$[\![\Gamma]\!], x_1 : I^{\Gamma(x_1)}, \ldots, x_n : I^{\Gamma(x_n)} \vdash x_1\langle v_1\rangle \mid \ldots \mid x_n\langle v_n\rangle \mid P$$

  However, by Lemma 41 applied to $[\![\Gamma]\!], x : I^{\Gamma(x)} \vdash x$, we also know that $\Gamma, x : \Gamma(x) \vdash x : \Gamma(x)$, hence Γ must already contain a type entry B for x, which is used in the encoding of env_V given above. Then by the encoding of Γ

  $$\begin{aligned}&[\![x_1 : B_1, \ldots, x_n : B_n, \Gamma']\!]\\ &= x_1 : I^{B_1}, I^{B_1} : (\mathsf{ch}(B_1), \varnothing), \ldots, x_n : I^{B_n}, I^{B_n} : (\mathsf{ch}(B_n), \varnothing), [\![\Gamma']\!]\end{aligned}$$

  so the type entries $x_1 : I^{B_1}, \ldots, x_n : I^{B_n}$ are already in $[\![\Gamma]\!]$.

  Also by the premise of [T-OUT], we have that $[\![\Gamma]\!] \vdash v_i : B_i$ for each of the object values, where the corresponding subject has type I^{B_i}, since the definition of each such interface is $I^{B_i} \mapsto (\mathsf{ch}(B_i), \varnothing)$.

  Then, by the translation of Γ, we have that entries $x_i : B_i$ are translated as $x_i : I^{B_i}$, so we know that Γ can be written as $\Gamma = x_1 : B_1, \ldots, x_n : B_n, \Gamma'$. Then by repeated applications of [T-ENV$_V$], we can therefore conclude that

  $$x_1 : B_1, \ldots, x_n : B_n, \Gamma' \vdash (x_1, v_1), \ldots, (x_n, v_n), \mathsf{env}_V^\varnothing$$

  as desired.

This concludes the proof for the other direction. □

A technical detail: The entry $x : I^B$ still appears in $[\![\Gamma]\!]$, even though the name is bound in $[\![\mathsf{env}_V]\!]^\Gamma$, and can thus be removed afterwards. This happens, because the actual type assignment is transferred to the encoding of env_V via the parameter Γ. Hence, even though there might exist a different Γ', such that $[\![\Gamma']\!]$ does not contain the entry $x : I^V$, and $[\![\Gamma']\!] \vdash [\![\mathsf{env}_V]\!]^\Gamma\,[P]$ still holds, then $[\![\Gamma']\!] \vdash [\![\mathsf{env}_V]\!]^{\Gamma'}\,[P]$ would *not* hold.

6 A Sound Type System for Secure Currency Flow[1]

6.1 Introduction

The classic notion of noninterference [47] is a well-known concept that has been applied in a variety of settings to characterise both integrity and secrecy in programming. In particular, this property has been defined by Volpano et al. [130] in terms of a lattice model of security levels (e.g. 'High' and 'Low', or 'Trusted' and 'Untrusted'); the key point being that information must not flow from a higher to a lower level. Thus, the lower levels are unaffected by the higher ones, and, conversely, the higher levels are 'noninterfering' with the lower ones.

Ensuring noninterference seems particularly relevant in a setting where not only information, but also *currency*, flows between programs. This is a core feature of *smart contracts*, which are programs that run atop a blockchain and are used to manage financial assets of users, codify transactions, and implement custom tokens; see e.g. [116] for an overview of the architecture. The code of a smart contract resides on the blockchain itself, and is therefore both immutable and publicly visible. This is one of the important ways in which the 'smart-contract programming paradigm' differs from conventional programming languages.

Public visibility means that *vulnerabilities* in the code can be found and exploited by a malicious user. Moreover, if a vulnerability is discovered, immutability prevents the contract creator from correcting the error. Thus, it is obviously desirable to ensure that a smart contract is safe and correct *before* it is deployed onto the blockchain.

The combination of immutability and visibility has led to huge financial losses in the past (see, e.g., [10; 35; 93; 94; 124]). A particularly spectacular example was the infamous DAO-attack on the Ethereum platform in 2016, which led to a loss of 60 million dollars [35]. This was made possible because a certain contract (the DAO contract, storing assets of users) was *reentrant*, that is, it allowed itself to be

[1]The present chapter is joint work with Luca Aceto and Daniele Gorla. It was presented at ECOOP 2024 and published in the conference proceedings [4]. A version with an appendix containing the proofs is available online in [2]. The text of the present chapter is equivalent to the online version, but several proofs have been updated with corrections compared to previous versions. A longer journal version, containing also some material from Chapter 8 of this thesis, is currently in preparation.

```
1  contract X {
2    ...
3    field called := F;
4    func transfer(z) {
5      if ¬called ∧ this.balance ≥ 1 then
6         call z.deposit(this)$1;
7         this.called := T;
8      else skip
9    }
10 }
11
12 contract Y {
13   ...
14   func deposit(x) {
15     call x.transfer(this)$0
16   }
17 }
```

Figure 6.1: Illustration of reentrancy written in the language TinySol.

called back by the recipient of a transfer *before* recording that the transfer had been completed.

Reentrancy is a pattern based on mutual recursion, where one method f calls another method g whilst also transferring an amount of currency along with the call. If g then immediately calls f back, it may yield a recursion where f will keep transferring funds to g. We can illustrate the problem as in Figure 6.1, using a simple, imperative and class-based model language called TinySol [12]. This model language, which we shall formally describe in Section 6.2, captures some of the core features of the smart-contract language Solidity [41], which is the standard high-level language used to write smart contracts for the Ethereum platform. A key feature of this language is that contracts have an associated `balance`, representing the amount of currency stored in each contract, which cannot be modified *except* through method calls to other contracts. Each method call has an extra parameter, representing the amount of currency to be transferred along with the call, and a method call thus represents a (potential) outgoing currency flow.

In Figure 6.1, `X.transfer(z)` first does a sanity check to ensure that it has not already been called and that the contract contains sufficient funds, which are stored in the `balance` field. Then it calls `z.deposit(this)` and transfers `1` unit of currency along with the call, where `z` is the address received as parameter. However, suppose the address received is `Y`. Then `Y.deposit(x)` immediately calls `X.transfer(z)` back, with `this` as actual parameter; this yields a mutual recursion, because the field

`called` will never be set to `T`. A transaction that invokes `X.transfer(Y)` with any number of currency units will trigger the recursion.

The problem is that currency cannot be transferred without also transferring control to the recipient, and the execution of `X.transfer(z)` comes to depend on unknown and untrusted code in the contract residing at the address received as the actual parameter. Simply switching the order of lines 6 and 7 in `X` solves the problem in this particular case, but it might not always be possible to move external calls to the last position in a sequence of statements. Furthermore, the execution of a function f can also depend on external fields, and not only on external calls. Thus, reentrancy is not just a purely syntactic property.

The property of reentrancy in Ethereum smart contracts has been formally characterised by Grishchenko et al. in [51]. Specifically, they define another property, named *call integrity*, which implies the absence of reentrancy (see [51, Theorem 1]) and has been identified in the literature as one of the safety properties that smart contracts should have. Informally, this property requires any call to a method in a 'trusted' contract (say, X) to yield the exact same sequence of currency flows (i.e. method calls) even if some of the other 'untrusted' contracts (or their stored values) are changed. In a sense, the code and values of the other contracts, which could be controlled by an attacker, must not be able to affect the currency flow from X.

A disadvantage of the definition of call integrity given in [51] is that it relies on a universal quantification over all possible execution contexts, which makes it hard to be checked in practice. However, call integrity seems intuitively to be related to noninterference, in the sense that both stipulate that changes in one part of a program should not have an effect upon another part. Even though we discover that the two properties are incomparable, one might hope to be able to apply techniques for ensuring noninterference to also capture call integrity. Specifically, Volpano et al. [130] show that noninterference can be soundly approximated using a type system. In the present chapter, we shall therefore create an adaptation of this type system for secure-flow analysis to the setting of smart contracts and show that the resulting type system *also* captures call integrity.

To recap, our main contributions in this chapter are: (1) a thorough study of the connections between call integrity and noninterference for smart contracts written in the language TinySol, and (2) a sound type system guaranteeing (noninterference and) call integrity for programs written in that language. We choose TinySol because it provides a minimal calculus for Solidity contracts and thus allows us to focus on the gist of our main contributions in a simple setting. In doing this, we also provide a simpler operational semantics for this language; this can be considered a third contribution of our work.

The chapter is organised as follows: In Section 6.2, we describe a revised version of the smart-contract language TinySol [12]. In Section 6.3, we adapt the definition of call integrity from [51] and of noninterference from [118] to this language; we then show that these two desirable properties are actually incomparable. Nevertheless,

there *is* an overlap between them. In Section 6.4, we create a type system for ensuring noninterference in TinySol, along the lines of Volpano et al. [130], and prove a type soundness result (Theorems 13–16). Our main result is Theorem 17, which shows that well-typedness provides a sound approximation to *both* noninterference *and* call integrity. This is used on a few examples in Section 6.5, where we also discuss the limitations of the type system. We survey some related work in Section 6.6 and conclude the chapter with some directions for future research in Section 6.7. Most proofs are deferred to the end of the chapter to avoid distracting from the main narrative of the text; they can be found in Section 6.8.

6.2 The TinySol language

In [12], Bartoletti et al. present the TinySol language, a standard imperative language (similar to Dijkstra's `While` language [92]), extended with classes (contracts) and two constructs: (1) a `throw` command, representing a fatal error, and (2) a procedure call, with an extra parameter n, denoting an amount of some digital asset, which is transferred along with the call from the caller to the callee. TinySol captures (some of) the core features of Solidity, and, in particular, it is sufficient to represent reentrancy phenomena. In this section, we present a version of TinySol which has been adapted to facilitate our later developments of the type system. Compared to the presentation in [12], we have, in particular, added explicit declarations of variables (local to the scope of a method) and fields (corresponding to the *keys* in the original presentation) to have a place for type annotations in the syntax.[2]

6.2.1 Syntax

The syntax of TinySol is given in Figure 6.2, where we use the notation $\tilde{\cdot}$ to denote (possibly empty) sequences of items. The set of *values*, ranged over by v, is formed by the sets of integers $\mathbb{Z}$, ranged over by n, booleans $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$, ranged over by b, and address names ANames, ranged over by X, Y.

We introduce explicit declarations for fields DF, methods DM, and contracts DC. The latter also encompasses declarations of accounts: an *account* is a contract that contains only the declarations of a special field `balance` and of a single special method `send()`, which does nothing and is used only for transferring funds to the account. By contrast, a contract usually contains other declarations of fields and methods. For the sake of simplicity, we make no syntactic distinction between an account and a contract but, for the purpose of distinguishing, we can assume that the set ANames is split into contract addresses and account addresses.

We have four 'magic' keywords in our syntax:

[2] A review of the original presentation of TinySol from [12] can be found in Appendix A.

$$
\begin{array}{rcl}
DF \in \mathsf{Dec}_F & ::= & \epsilon \mid \texttt{field p := } v\texttt{;}DF \\
DM \in \mathsf{Dec}_M & ::= & \epsilon \mid \texttt{func } f(\tilde{x})\ \texttt{\{ } S\ \texttt{\} } DM \\
DC \in \mathsf{Dec}_C & ::= & \epsilon \mid \texttt{contract } X\ \texttt{\{} \\
 & & \quad \texttt{field balance := } n\texttt{; } DF \\
 & & \quad \texttt{func send() \{ skip \} } DM \\
 & & \texttt{\} } DC \\
m \in \mathsf{MVar} & ::= & \texttt{this} \mid \texttt{sender} \mid \texttt{value} \\
L \in \mathsf{LVal} & ::= & x \mid \texttt{this.}p \\
e \in \mathsf{Exp} & ::= & v \mid x \mid m \mid e\texttt{.balance} \mid e\texttt{.}p \mid \mathrm{op}(\tilde{e}) \\
S \in \mathsf{Stm} & ::= & \texttt{skip} \mid \texttt{throw} \mid \texttt{var } x \texttt{ := } e \texttt{ in } S \mid L \texttt{ := } e \\
 & & \mid S_1\texttt{;}S_2 \mid \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} \mid \texttt{while } e \texttt{ do } S \\
 & & \mid \texttt{call } e_1\texttt{.}f(\tilde{e})\texttt{\$}e_2 \\
v \in \mathsf{Val} & ::= & \mathbb{Z} \cup \mathbb{B} \cup \mathsf{ANames}
\end{array}
$$

where $x, y \in \mathsf{VNames}$ (variable names), $p, q \in \mathsf{FNames}$ (field names), $X, Y \in \mathsf{ANames}$ (address names), $f, g \in \mathsf{MNames}$ (method names).

Figure 6.2: The syntax of TinySol.

- `balance` (type `int`), a special field recording the current balance of the contract (or account). It can be read from, but not directly assigned to, except through method calls. This ensures that the total amount of currency 'on-chain' remains constant during execution.
- `value` (type `int`), a special variable that is bound to the currency amount transferred with a method call.
- `sender` (type `address`), a special variable that is always bound to the address of the caller of a method.
- `this` (type `address`), a special variable that is always bound to the address of the contract containing the currently executing method.

The last three of these are local variables, and we collectively refer to them as 'magic variables' $m \in \mathsf{MVar}$. The declaration of variables and fields are very alike: the main difference is that variable bindings will be created at runtime (and with scoped visibility), hence we can let the initial assignment be an *expression* e; whilst the initial assignment to fields must be *values* v.

The core part of the language is the declaration of expressions e and statements S, that are almost the same as in [12]. The main differences are: (1) we introduce fields p in expressions, instead of keys; (2) we explicitly distinguish between (global) fields

and (local) variables, where the latter are declared with a scope limited to a statement S; and (3) we introduce explicit *lvalues* L, to restrict what can appear on the left-hand side of an assignment (in particular, this ensures that the special field `balance` can never be assigned to directly).

As in the original presentation of TinySol, we can also use our new formulation of the language to describe transactions and blockchains. A *transaction* is simply a call, where the caller is an account A, rather than a contract. We denote this by writing $A\texttt{->}X\,.\,f\,(\tilde{v})\$n$, which expresses that the account A calls the method f on the contract (residing at address) X, with actual parameters $\tilde{v}$, and transferring n amount of currency with the call. We can then model blockchains as follows:

Definition 41 (Syntax of blockchains)**.** A *blockchain* $B \in \mathcal{B}$ is a list of initial contract declarations DC, followed by a sequence of transactions $T \in \mathsf{Tr}$:

$$B ::= DC \;\; T \qquad\qquad T ::= \epsilon \mid A\texttt{->}X\,.\,f\,(\tilde{v})\$n, T$$

Notationally, a blockchain with an empty DC will be simply written as the sequence of transactions. ■

6.2.2 Big-step semantics

To define the semantics, we need some environments to record the bindings of variables (including the three magic variable names `this`, `sender` and `value`), fields, methods, and contracts. We define them as sets of partial functions as follows:

Definition 42 (Binding model)**.** We define the following sets of partial functions:

$$\begin{aligned}
\mathsf{env}_V \in \mathsf{Env}_V &: \mathsf{VNames} \cup \mathsf{MVar} \rightharpoonup \mathsf{Val} \\
\mathsf{env}_S \in \mathsf{Env}_S &: \mathsf{ANames} \rightharpoonup \mathsf{Env}_F \\
\mathsf{env}_F \in \mathsf{Env}_F &: \mathsf{FNames} \cup \{\,\texttt{balance}\,\} \rightharpoonup \mathsf{Val} \\
\mathsf{env}_T \in \mathsf{Env}_T &: \mathsf{ANames} \rightharpoonup \mathsf{Env}_M \\
\mathsf{env}_M \in \mathsf{Env}_M &: \mathsf{MNames} \rightharpoonup \mathsf{VNames}^* \times \mathsf{Stm}
\end{aligned}$$

We regard each environment env_X, for any $X \in \{\,V, F, M, S, T\,\}$, as a list of pairs (d, c) where $d \in \mathrm{dom}\,(\mathsf{env}_X)$ and $c \in \mathrm{codom}\,(\mathsf{env}_X)$. The notation $\mathsf{env}_X[d \mapsto c]$ denotes the update of env_X mapping d to c. We write $\mathsf{env}_X^{\emptyset}$ for the empty environment. To simplify the notation, when two or more environments appear together, we shall use the convention of writing the subscripts together (e.g. env_{MF} instead of $\mathsf{env}_M, \mathsf{env}_F$). ■

Our binding model consists of two environments: a *method table* env_T, which maps addresses to method environments, and a *state* env_S, which maps addresses to lists of fields and their values. Thus, for each contract, we have the list of methods

$$[\textsc{dec-f}_2]\frac{\langle DF, \mathsf{env}_F\rangle \to_{DF} \mathsf{env}'_F}{\langle \texttt{field } p \texttt{ := } v\texttt{;}DF, \mathsf{env}_F\rangle \to_{DF} (p, v) : \mathsf{env}'_F}$$

$$[\textsc{dec-m}_2]\frac{\langle DM, \mathsf{env}_M\rangle \to_{DM} \mathsf{env}'_M}{\langle \texttt{func } f(\tilde{x}) \texttt{ \{ } S \texttt{ \} } DM, \mathsf{env}_M\rangle \to_{DM} (f, (\tilde{x}, S)) : \mathsf{env}'_M}$$

$$[\textsc{dec-c}_2]\frac{\begin{array}{c}\left\langle DF, \mathsf{env}_F^{\emptyset}\right\rangle \to_{DF} \mathsf{env}_F \\ \left\langle DM, \mathsf{env}_M^{\emptyset}\right\rangle \to_{DM} \mathsf{env}_M \\ \langle DC, \mathsf{env}_{ST}\rangle \to_{DC} \mathsf{env}'_{ST}\end{array}}{\begin{array}{r}\langle \texttt{contract } X \texttt{ \{ } DF\ DM \texttt{ \} } DC, \mathsf{env}_{ST}\rangle \\ \to_{DC} (X, \mathsf{env}_F) : \mathsf{env}'_S, (X, \mathsf{env}_M) : \mathsf{env}'_T\end{array}}$$

$$[\textsc{dec-f}_1]\frac{}{\langle \epsilon, \mathsf{env}_F\rangle \to_{DF} \mathsf{env}_F} \qquad [\textsc{dec-m}_1]\frac{}{\langle \epsilon, \mathsf{env}_M\rangle \to_{DM} \mathsf{env}_M}$$

$$[\textsc{dec-c}_1]\frac{}{\langle \epsilon, \mathsf{env}_{ST}\rangle \to_{DC} \mathsf{env}_{ST}}$$

Figure 6.3: Semantics of declarations.

it declares and its current state; of course, the method table is constant, once all declarations are performed, whereas the state will change during the evaluation of a program.

6.2.2.1 Declarations

The semantics of declarations builds the field and method environments, env_F and env_M, and the state and method table env_S and env_T. We give the semantics in a classic big-step style; thus, transitions are of the form $\langle DX, \mathsf{env}_X\rangle \to_{DX} \mathsf{env}'_X$ for $X \in \{F, M, C, S, T\}$, and their defining rules are given in Figure 6.3. Notationally, here and in what follows, we denote with $e : l$ the list that results from prepending an element e to the list l. We assume that field and method names are distinct within each contract; therefore, the rules in Figure 6.3 define partial, finite functions.

6.2.2.2 Expressions

Figure 6.4 gives the semantics of expressions e. Expressions have no side effects, so they cannot contain method calls, but they can access both local variables and fields of any contract. Thus expression evaluations are of the form $\mathsf{env}_{SV} \vdash e \to_e v$, i.e. they are relative to the state and variable environments. We use k to range over `this`,

$$[\textsc{exp-var}]\frac{k \in \mathrm{dom}\,(\mathrm{env}_V) \quad \mathrm{env}_V(k) = v}{\mathrm{env}_{SV} \vdash k \to_e v}$$

$$[\textsc{exp-val}]\frac{}{\mathrm{env}_{SV} \vdash v \to_e v}$$

$$[\textsc{exp-op}]\frac{\mathrm{env}_{SV} \vdash \tilde{e} \to_e \tilde{v} \quad \mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v}{\mathrm{env}_{SV} \vdash \mathrm{op}(\tilde{e}) \to_e v}$$

$$[\textsc{exp-field}]\frac{\mathrm{env}_{SV} \vdash e \to_e X \quad q \in \mathrm{dom}\,(\mathrm{env}_S(X)) \quad \mathrm{env}_S(X)(q) = v}{\mathrm{env}_{SV} \vdash e.q \to_e v}$$

Figure 6.4: Semantics of expressions.

`sender`, `value` and variables x (i.e. $k \in \mathrm{dom}\,(\mathrm{env}_V)$), and q to range over `balance` and fields p (i.e. $q \in \mathrm{dom}\,(\mathrm{env}_F)$).

We do not give explicit rules for the boolean and integer operators subsumed under op, but simply assume that they can be evaluated to a unique value by some semantics $\mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v$.[3] It follows that each expression evaluates to a unique value relative to some given state and variable environments. Note that we assume that no operation is defined for addresses X, so we disallow any form of pointer arithmetic.

6.2.2.3 Statements

The semantics of statements describes the actual execution steps of a program. In Figures 6.5–6.6 we give the semantics in big-step style, where a step describes the execution of a statement in its entirety. Statements can read from the method table and they can modify the state (i.e., the variable and field bindings). The result of executing a statement is a new state, so transitions must here be of the form $\mathrm{env}_T \vdash \langle S, \mathrm{env}_{SV}\rangle \to_S \mathrm{env}'_{SV}$ (recall that env'_{SV} stands for $\mathrm{env}'_S, \mathrm{env}'_V$), since both the field values in env_S and the values of the local variables in env_V may have been modified by the execution of S.

Most of the rules are straightforward. The rule [DECV] is used when we declare a new variable x, with scope limited to the statement S; we implicitly assume alpha-conversion to handle shadowing of an existing name. In the premise, we evaluate the expression e to a value v, and then execute the statement S with a variable environment $(x, v) : \mathrm{env}_V$, where we have added the pair (x, v). During the execution of S, this variable environment may of course be updated (by applications of the rule [ASSV]), which may alter any value in the environment, including v. However, outside of the

[3] To simplify the definitions, we assume that all operations are total. If this was not the case, we would have needed some exception handling for partial operations (e.g., division by zero).

$$[\textsc{skip}]\frac{}{\mathsf{env}_T \vdash \langle \texttt{skip}, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}_{SV}}$$

$$[\textsc{seq}]\frac{\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}''_{SV} \quad \mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle S_1\,; S_2, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}$$

$$[\textsc{if}]\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e b \in \mathbb{B} \quad \mathsf{env}_T \vdash \langle S_b, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}\ , \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}$$

$$[\textsc{while}_\mathsf{T}]\frac{\begin{array}{c}\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}''_{SV} \\ \mathsf{env}_{SV} \vdash e \rightarrow_e \mathsf{T} \quad \mathsf{env}_T \vdash \langle \texttt{while } e \texttt{ do } S, \mathsf{env}''_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \texttt{while } e \texttt{ do } S, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}$$

$$[\textsc{while}_\mathsf{F}]\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e \mathsf{F}}{\mathsf{env}_T \vdash \langle \texttt{while } e \texttt{ do } S, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}_{SV}}$$

$$[\textsc{decv}]\frac{\begin{array}{c}\mathsf{env}_{SV} \vdash e \rightarrow_e v \\ \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, (x, v) : \mathsf{env}_V\rangle \rightarrow_S \mathsf{env}'_S, (x, v') : \mathsf{env}'_V\end{array}}{\mathsf{env}_T \vdash \langle \texttt{var } x \texttt{ := } e \texttt{ in } S, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}$$

where:

$$x \notin \mathrm{dom}\,(\mathsf{env}_V)$$

$$[\textsc{assv}]\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e v}{\mathsf{env}_T \vdash \langle x \texttt{ := } e, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}_S, \mathsf{env}_V[x \mapsto v]} \ (x \in \mathrm{dom}\,(\mathsf{env}_V))$$

$$[\textsc{assf}]\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e v}{\mathsf{env}_T \vdash \langle \texttt{this}.p \texttt{ := } e, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]], \mathsf{env}_V}$$

where:

$$X = \mathsf{env}_V(\texttt{this})$$
$$\mathsf{env}_F = \mathsf{env}_S(X)$$
$$p \in \mathrm{dom}\,(\mathsf{env}_F)$$

Figure 6.5: Big-step semantics of statements in TinySol.

$$[\textsc{call}]\ \frac{\begin{array}{ll}\mathsf{env}_{SV} \vdash e_1 \to_e Y & \mathsf{env}_{SV} \vdash \tilde{e} \to_e \tilde{v} \\ \mathsf{env}_{SV} \vdash e_2 \to_e n & \mathsf{env}_T \vdash \langle S, \mathsf{env}''_{SV} \rangle \to_S \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \texttt{call}\ \ e_1.f(\tilde{e})\$e_2, \mathsf{env}_{SV} \rangle \to_S \mathsf{env}'_S, \mathsf{env}_V}$$

where:

$$\begin{aligned} X &= \mathsf{env}_V(\texttt{this}) \\ \mathsf{env}_F^X &= \mathsf{env}_S(X) \\ \mathsf{env}_F^Y &= \mathsf{env}_S(Y) \\ (\tilde{x}, S) &= (\mathsf{env}_T(Y))(f) \\ |\tilde{x}| &= |\tilde{e}| = k \\ n &\leq \mathsf{env}_F^X(\texttt{balance}) \\ \mathsf{env}''_S &= \mathsf{env}_S\big[X \mapsto \mathsf{env}_F^X[\texttt{balance -= } n]\big]\big[Y \mapsto \mathsf{env}_F^Y[\texttt{balance += } n]\big] \\ \mathsf{env}''_V &= (\texttt{this}, Y) : (\texttt{sender}, X) : (\texttt{value}, n) : (x_1, v_1) : \ldots : (x_k, v_k) : \mathsf{env}_V^{\varnothing} \end{aligned}$$

Figure 6.6: Big-step semantics of statements in TinySol (continued).

scope of the declaration, x is not visible and so the pair (x, v') is removed from the environment once S finishes. By contrast, any other change made to env'_V (as well as any change made to the global state env_S) is retained.

The [call] rule is the most complicated, because we need to perform a number of actions. Some of them are obvious (e.g., evaluate the address and the parameters e_1, $\tilde{e}$ and e_2, relatively to the current execution environment env_{SV}; use the obtained address Y of the callee to retrieve the field environment env_F^Y for this contract and, through the method table, to extract the list of formal parameters $\tilde{x}$ and the body of the method S; and check that the number of actual parameters is the same as the number of formal parameters). Then, we also have to check that the `balance` of the caller is at least n, and, in that case, update the state environment by subtracting n from the balance of X and adding n to the balance of Y, in their respective field environments; this yields a new state env''_S. We write

$$\begin{aligned} \mathsf{env}_F[\texttt{balance -= } n] &\triangleq \mathsf{env}_F[\texttt{balance} \mapsto \mathsf{env}_F(\texttt{balance}) - n] \\ \mathsf{env}_F[\texttt{balance += } n] &\triangleq \mathsf{env}_F[\texttt{balance} \mapsto \mathsf{env}_F(\texttt{balance}) + n] \end{aligned}$$

as a short-hand for these two operations.

Finally, we create the new execution environment by creating new bindings for the special variables `this`, `sender` and `value`, and by binding the formal parameters $\tilde{x}$ to the values of the actual parameters $\tilde{v}$ in env''_V. Then we execute the statement S in this new environment. This yields the new state env'_S, and also an updated variable environment env'_V, since S may have modified the bindings in env''_V. However, these bindings are local to the method, and therefore we throw them away once the call finishes. So, the result of this transition is the updated state env'_S and the original variable environment of the caller env_V.

$$[\textsc{Gen}]\ \frac{\left\langle DC, \mathsf{env}_{ST}^{\emptyset} \right\rangle \rightarrow_{DC} \mathsf{env}_{ST}}{\left\langle DC\ \ T, \mathsf{env}_{ST}^{\emptyset} \right\rangle \rightarrow_{B} \langle T, \mathsf{env}_{ST} \rangle}$$

$$[\textsc{Trans}]\ \frac{\mathsf{env}_T \vdash \left\langle X\,.\,f\,(\tilde{v}) : n, \mathsf{env}_S, (\texttt{this}, A)\ :\ \mathsf{env}_V^{\emptyset} \right\rangle \rightarrow_{S} \mathsf{env}'_S, \mathsf{env}_V}{\langle A\texttt{->}X\,.\,f\,(\tilde{v})\,\$n, T, \mathsf{env}_{ST} \rangle \rightarrow_{B} \left\langle T, \mathsf{env}'_S, \mathsf{env}_T \right\rangle}$$

$$[\textsc{Rev}]\ \frac{}{\langle \epsilon, \mathsf{env}_{ST} \rangle \rightarrow_{B} \mathsf{env}_{ST}}$$

Figure 6.7: Semantics of blockchains.

It should be noted that a *local* method call, i.e. a call to a method within the same (calling) contract, is merely a special case of the rule [CALL]. Such a call would have the form `this.`$f(\tilde{e})$`:0`, since transferring any amount of currency will not alter the balance of the contract. Thus, we could introduce some syntactic sugar, omitting both the address and the value, and instead simply write $f(\tilde{e})$.

6.2.2.4 Transactions and blockchains

The semantics for blockchains is given as a transition system defined by the rules given in Figure 6.7. Here, the rule [GEN] describes the 'genesis event' where contracts are declared, whilst [TRANS] describes a single transaction. This is thus a *small-step* semantics, invoking the big-step semantics for declarations and statements for its premises. We remark that the rules of the operational semantics for blockchains (as well as those for statements presented above) define a deterministic transition relation.

Note that, unlike in the original formulation of TINYSOL, we do not include a rule like [Tx2] in [12] for rolling back a transaction in case it is non-terminating or it aborts via a `throw` command. Such a rule would require a premise that cannot be checked effectively for a Turing-complete language like TINYSOL and therefore we omit it, since it is immaterial for the main contributions we give in this chapter.[4] In practice, termination of Ethereum smart contracts is ensured via a 'gas mechanism' and is assumed by techniques for the formal analysis of smart contracts. However, as observed in, for instance, [45], proof of termination for smart contracts is non-trivial even in the presence of a 'gas mechanism.' In the aforementioned paper, the authors present the first mechanised proof of termination of contracts written in EVM bytecode using minimal assumptions on the gas cost of operations (see the

[4]For instance, rule [Tx2] in [12] has an undecidable premise that checks whether the execution of the body of a contract does *not* yield a final state. It is debatable whether such rules should appear in an operational semantics.

study [134] for an empirical analysis of the effectiveness of the 'gas mechanism' in estimating the computational cost of executing real-life transactions). We leave for future work the addition of a 'gas mechanism' to TinySol and the adaption of the results we present in this chapter to that setting.

6.3 Call integrity and noninterference in TinySol

Grishchenko et al. [51] formulate the property of *call integrity* for smart contracts written in the language EVM, which is the 'low-level' bytecode of the Ethereum platform, and the target language to which e.g. Solidity compiles. They then prove [51, Theorem 1] that this property suffices for ruling out reentrancy phenomena, as those described in the example in Figure 6.1. We first formulate a similar property for TinySol; this requires a few preliminary definitions.

Definition 43 (Trace semantics)**.** A *trace* of method invocations is given by

$$\pi ::= \epsilon \mid X\text{->}Y\,.\,f\,(\tilde{v})\$n, \pi$$

where X is the address of the calling contract, Y is the address of the called contract, f is the method name, and $\tilde{v}$ and n are the actual parameters. We annotate the big-step semantics of Figures 6.5–6.6 with a trace containing information on the invoked methods to yield labeled transitions of the form $\xrightarrow{\pi}$. The rules are given in Figures 6.8–6.9. ■

Definition 44 (Projection)**.** The *projection* of a trace to a specific contract X, written $\pi \downarrow_X$, is the trace of calls with X as the calling address. Formally:

$$\epsilon \downarrow_X = \epsilon \qquad (Z\text{->}Y\,.\,f\,(\tilde{v})\$n, \pi) \downarrow_X = \begin{cases} X\text{->}Y\,.\,f\,(\tilde{v})\$n, (\pi \downarrow_X) & \text{if } Z = X \\ \pi \downarrow_X & \text{otherwise} \end{cases}$$ ■

Notationally, given a (partial) function f, we write $f|_X$ for denoting the restriction of f to the subset X of its domain.

Definition 45 (Call integrity)**.** Let $\mathcal{A}$ denote the set of all contracts (addresses), $\mathcal{X} \subseteq \mathcal{A}$ denote a set of trusted contracts, $\mathcal{Y} \triangleq \mathcal{A} \setminus \mathcal{X}$ denote all other contracts, and $\mathsf{env}_{ST}^{\mathcal{X}}$ have domain $\mathcal{X}$. A contract $C \in \mathcal{X}$ has *call integrity* for $\mathcal{Y}$ if, for every transaction T and environments env_{ST}^1 and env_{ST}^2 such that $\mathsf{env}_{ST}^1|_{\mathcal{X}} = \mathsf{env}_{ST}^2|_{\mathcal{X}} = \mathsf{env}_{ST}^{\mathcal{X}}$, it holds that

$$\langle T, \mathsf{env}_{ST}^1 \rangle \xrightarrow{\pi_1}_B \mathsf{env}_{ST}^{1'} \wedge \langle T, \mathsf{env}_{ST}^2 \rangle \xrightarrow{\pi_2}_B \mathsf{env}_{ST}^{2'} \implies \pi_1 \downarrow_C = \pi_2 \downarrow_C$$ ■

The definition is quite complicated and contains a number of elements:

- C is the contract of interest.

$$[\textsc{skip}^{\pi}]\frac{}{\mathsf{env}_T \vdash \langle \texttt{skip}, \mathsf{env}_{SV}\rangle \xrightarrow{\epsilon} \mathsf{env}_{SV}}$$

$$[\textsc{seq}^{\pi}]\frac{\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1} \mathsf{env}''_{SV} \quad \mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \xrightarrow{\pi_2} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle S_1 ; S_2, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1,\pi_2} \mathsf{env}'_{SV}}$$

$$[\textsc{if}^{\pi}]\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e b \in \mathbb{B} \quad \mathsf{env}_T \vdash \langle S_b, \mathsf{env}_{SV}\rangle \xrightarrow{\pi} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \texttt{if } e \texttt{ then } S_{\mathsf{T}} \texttt{ else } S_{\mathsf{F}} \ , \mathsf{env}_{SV}\rangle \xrightarrow{\pi} \mathsf{env}'_{SV}}$$

$$[\textsc{while}^{\pi}_{\mathsf{T}}]\frac{\begin{array}{c}\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1} \mathsf{env}''_{SV} \\ \mathsf{env}_{SV} \vdash e \rightarrow_e \mathsf{T} \quad \mathsf{env}_T \vdash \langle \texttt{while } e \texttt{ do } S, \mathsf{env}''_{SV}\rangle \xrightarrow{\pi_2} \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \texttt{while } e \texttt{ do } S, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1,\pi_2} \mathsf{env}'_{SV}}$$

$$[\textsc{while}^{\pi}_{\mathsf{F}}]\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e \mathsf{F}}{\mathsf{env}_T \vdash \langle \texttt{while } e \texttt{ do } S, \mathsf{env}_{SV}\rangle \xrightarrow{\epsilon} \mathsf{env}_{SV}}$$

$$[\textsc{decv}^{\pi}]\frac{\begin{array}{c}\mathsf{env}_{SV} \vdash e \rightarrow_e v \\ \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, (x, v) : \mathsf{env}_V\rangle \xrightarrow{\pi} \mathsf{env}'_S, (x, v') : \mathsf{env}'_V\end{array}}{\mathsf{env}_T \vdash \langle \texttt{var } x \texttt{ := } e \texttt{ in } S, \ \mathsf{env}_{SV}\rangle \xrightarrow{\pi} \mathsf{env}'_{SV}}$$

where:

$$x \notin \mathrm{dom}\,(\mathsf{env}_V)$$

$$[\textsc{assv}^{\pi}]\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e v}{\mathsf{env}_T \vdash \langle x \texttt{ := } e, \mathsf{env}_{SV}\rangle \xrightarrow{\epsilon} \mathsf{env}_S, \mathsf{env}_V[x \mapsto v]} \ (x \in \mathrm{dom}\,(\mathsf{env}_V))$$

$$[\textsc{assf}^{\pi}]\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e v}{\mathsf{env}_T \vdash \langle \texttt{this}.p \texttt{ := } e, \mathsf{env}_{SV}\rangle \xrightarrow{\epsilon} \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]], \mathsf{env}_V}$$

where:

$$X = \mathsf{env}_V(\texttt{this})$$
$$\mathsf{env}_F = \mathsf{env}_S(X)$$
$$p \in \mathrm{dom}\,(\mathsf{env}_F)$$

Figure 6.8: Trace semantics.

$$[\textsc{call}^{\pi}]\ \frac{\begin{array}{c}\mathsf{env}_{SV} \vdash e_1 \to_e Y \quad \mathsf{env}_{SV} \vdash \tilde{e} \to_e \tilde{v} \\ \mathsf{env}_{SV} \vdash e_2 \to_e n \qquad \mathsf{env}_T \vdash \langle S, \mathsf{env}''_{SV} \rangle \xrightarrow{\pi} \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \texttt{call}\ \ e_1\,.\,f\,(\tilde{e})\,\$e_2, \mathsf{env}_{SV} \rangle \xrightarrow{X\texttt{->}Y\,.\,f\,(\tilde{v})\$n,\pi} \mathsf{env}'_S, \mathsf{env}_V}$$

where:

$$\begin{aligned}
X &= \mathsf{env}_V(\texttt{this}) \\
\mathsf{env}_F^X &= \mathsf{env}_S(X) \\
\mathsf{env}_F^Y &= \mathsf{env}_S(Y) \\
(\tilde{x}, S) &= (\mathsf{env}_T(Y))(f) \\
|\tilde{x}| &= |\tilde{e}| = k \\
n &\leq \mathsf{env}_F^X(\texttt{balance}) \\
\mathsf{env}''_S &= \mathsf{env}_S\big[X \mapsto \mathsf{env}_F^X[\texttt{balance -= } n]\big]\big[Y \mapsto \mathsf{env}_F^Y[\texttt{balance += } n]\big] \\
\mathsf{env}''_V &= (\texttt{this}, Y) : (\texttt{sender}, X) : (\texttt{value}, n) : (x_1, v_1) : \ldots : (x_k, v_k) : \mathsf{env}_V^{\varnothing}
\end{aligned}$$

Figure 6.9: Trace semantics (continued).

- $\mathcal{X}$ is a set of *trusted* contracts, which we assume are allowed to influence the behaviour of C. This set must obviously contain C, since C at least must be assumed to be trusted. Thus, a contract C can have call integrity for all contracts, if $\mathcal{X} = \{ C \}$.
- Conversely, the set $\mathcal{Y} = \mathcal{A} \setminus \mathcal{X}$ is the set of addresses of all contracts that are *untrusted*.[5]
- env^1_{ST} and env^2_{ST} are any two pairs of method/field environments that coincide (both in the code and in the values) for all the trusted contracts.[6] The point is that the contracts in $\mathcal{X}$ are assumed to be known, and hence invariant, whereas any contract in $\mathcal{Y}$ is assumed to be unknown and may be controlled by an attacker. Thus, we are actually quantifying over all possible contexts where the contracts in $\mathcal{X}$ can be run.
- T is any transaction; it may be issued from any account and to any contract. Thus we also quantify over all possible transactions, since an attacker may request an arbitrary transaction, that is thus part of the execution context as well.

Then, the call integrity property intuitively requires that, if we run the trusted part of the code in any execution context, the behavior of C remains the same, i.e. C

[5] Note that this is formulated inversely by Grishchenko et al., who instead formulate the property for a set of *untrusted* contracts $\mathcal{A}_{\mathcal{C}}$, corresponding to $\mathcal{Y}$ in the present formulation. However, using the set of *trusted* addresses $\mathcal{X}$ seems more straightforward.

[6] This too is inversely formulated by Grishchenko et al.

must make exactly the same method calls (and in exactly the same order). Thus, to *disprove* that C has call integrity, it suffices to find two environments and a transaction that will induce a difference in the call trace of C.

The idea in the property of call integrity is that the behaviour of C should not depend on any untrusted code (i.e. contracts in $\mathcal{Y}$), even if control is transferred to a contract in $\mathcal{Y}$. The latter could for example happen if C calls a method on $B \in \mathcal{X}$, and B then calls a method on a contract in $\mathcal{Y}$. This also means that C cannot directly call any contract in $\mathcal{Y}$, since that can only happen if C calls a method on a contract, where the address is received as a parameter, or if it calls a method on a 'hard-coded' contract address. In both cases, we can easily pick up two environments able to induce different behaviors, for example by choosing a non-existing address for one context (in the first case), or by ensuring that no contract exists on the hard-coded address in one context (in the second case). The latter possibility can seem somewhat contrived, especially if we assume that all contracts are created at the genesis event, and it might therefore be reasonable to require also that $\mathrm{dom}\left(\mathsf{env}^1_{ST}\right) = \mathrm{dom}\left(\mathsf{env}^2_{ST}\right)$, such that we at least assume that contracts exist on the same addresses. However, on an actual blockchain, new contracts can be deployed (and in some cases also deleted) at any time, and if such a degree of realism is desired, this extra constraint should not be imposed.

The main problem with the definition of call integrity is that it relies on a universal quantification over all possible executions contexts. This makes it hard to be checked in practice. However, our previous discussion indicates that call integrity may intuitively be viewed as a form of *noninterference* between the trusted and the untrusted contracts. We now see to what extent this intuition is true and formally compare the two notions.

First of all, we consider a basic lattice of security levels, made up by just two levels, namely H (for *high*) and L (for *low*), with $L < H$. We tag every contract to be high or low through a contracts-to-levels mapping $\lambda : \mathcal{A} \to \{L, H\}$; this induces a bipartition of the contract names $\mathcal{A}$ into the following sets:

$$\mathcal{L} = \{\, X \in \mathcal{A} | \lambda(X) = L \,\}$$
$$\mathcal{H} = \{\, X \in \mathcal{A} | \lambda(X) = H \,\}$$

In this way, we create a bipartition of the state into low and high, corresponding to the fields of the low and of the high contracts, respectively. Then, we define *low-equivalence* $=_L$ to be the equivalence on states such that $\mathsf{env}^1_S =_L \mathsf{env}^2_S$ if and only if $\mathsf{env}^1_S(X) = \mathsf{env}^2_S(X)$, for every $X \in \mathcal{L}$.

We can now adapt the notion of noninterference for multi-threaded programs by Smith and Volpano [118] to the setting of TinySol.

Definition 46 (Noninterference)**.** Given a contracts-to-levels mapping $\lambda : \mathcal{A} \to \{L, H\}$ and a contract environment env_T, the contracts satisfy *noninterference* if, for every env^1_S and env^2_S and for every transaction T such that

- $\mathsf{env}_S^1 =_L \mathsf{env}_S^2$, and
- $\langle T, \mathsf{env}_S^1, \mathsf{env}_T \rangle \rightarrow_B \mathsf{env}_S^{1'}, \mathsf{env}_T$, and
- $\langle T, \mathsf{env}_S^2, \mathsf{env}_T \rangle \rightarrow_B \mathsf{env}_S^{2'}, \mathsf{env}_T$

it holds that $\mathsf{env}_S^{1'} =_L \mathsf{env}_S^{2'}$. ■

Remark 1 (Incomparability)**.** Call integrity and noninterference seem strongly related, in the sense that the first requires that the behaviour of a contract is not influenced by the (bad) execution context, whereas the second one requires that a part of the computation (the 'low' one) is not influenced by the remainder context (the 'high' one). So, one may try to prove a statement like: "$C \in \mathcal{X}$ has call integrity for $\mathcal{Y} \triangleq \mathcal{A} \setminus \mathcal{X}$ if and only if it satisfies noninterference w.r.t. λ such that $\mathcal{L} = \mathcal{X}$ and $\mathcal{H} = \mathcal{Y}$." However, both directions are false.

For the direction from right to left, consider:

```
contract X {
  field balance = 0
  func go() { }
}

contract Y {
  field balance = v
  func go() {
    call X.go()$this.balance
  }
}
```

where `X` is trusted and `Y` untrusted. Since `X` cannot invoke any method, this example satisfies call integrity. However, it does not satisfy noninterference. To see this, consider two environments, one assigning 1 to `Y`'s balance and the other one assigning 0, and the transaction `Y->Y.go():0`.

For the direction from left to right, consider the following:

```
contract X {
  func go() {
      if Y.balance = 0 then call Z.a()$0 else call Z.b()$0
  }
}

contract Y {
  field balance = v;
}


```

```
12 contract Z {
13   func a() { }
14   func b() { }
15 }
```

Assuming that both `X` and `Z` are low, the example satisfies noninterference: there is no way for `Y` to influence the low memory. By contrast, the code does not satisfy call integrity. Indeed, let `v` be `0` in one environment and `1` in the other, and consider T to be `X->X.go():0`: in the first environment, it generates `X->Z.a():0`, whereas in the second one it generates `X->Z.b():0`. ■

6.4 A type system for noninterference and call integrity

As demonstrated in Remark 1, call integrity and noninterference are incomparable properties. This is so because noninterference is a 2-property on the pair of *stores* $(\mathsf{env}_S^{1'}, \mathsf{env}_S^{2'})$ resulting from two different executions, whereas call integrity is a 2-property on the pair of *call traces* (π_1, π_2) generated during two executions. However, the two properties have an interesting overlap, because an outgoing currency flow (i.e. a method call) may also result, at least potentially, in a change of the stored values of the `balance` fields of the sender and recipient. Every method call is therefore *also* an information flow between the two, even when no amount of currency is transferred. In [130], Volpano et al. devise a type system for checking information flows, which, as they show, yields a sound approximation to noninterference. In the following, we create an adaptation of this type system to TinySol and show that it may *also* be used to soundly approximate call integrity.

6.4.1 Type syntax

We begin by assuming a finite lattice $(\mathcal{S}, \sqsubseteq)$ consisting of a set of *security levels* $\mathcal{S}$, ranged over by s, and equipped with a partial order $\sqsubseteq$. We write $s_\bot$, and $s_\top$ for the least and largest elements in $\mathcal{S}$.

In the simplest setting, we can let $\mathcal{S} \triangleq \{L, H\}$ (for 'low' and 'high') and define $L \sqsubseteq L$, $L \sqsubseteq H$, and $H \sqsubseteq H$. This is sufficient for ensuring bi-partite noninterference, but the type system can also handle more fine-grained security control. With this, we can define the types:

Definition 47. We use the following language of types, where $I \in \mathsf{TNames}$ is a *type name* (or 'interface name'):

$$
\begin{aligned}
B \in \mathcal{B} &::= s \mid I_s \\
T \in \mathcal{T} &::= B \mid \mathsf{var}(B) \mid \mathsf{cmd}(s) \mid \mathsf{proc}(\widetilde{B}) : s \\
\Gamma \in \mathcal{E} &::= \mathcal{N} \rightharpoonup \mathcal{T} \cup \mathcal{E} \\
\mathcal{N} &::= \mathsf{ANames} \cup \mathsf{FNames} \cup \mathsf{VNames} \cup \mathsf{MNames} \cup \mathsf{TNames}
\end{aligned}
$$

We write $\widetilde{T}$ for a tuple of types $(T_1, \ldots, T_n)$. ■

Note that for the purpose of the type system, unless otherwise noted, we shall assume that the four 'magic names' MVar are contained in the respective sets of field and variable names; i.e. `balance` ∈ FNames and `this`, `sender`, `value` ∈ VNames.

The meaning of the types is as follows:

- $\mathcal{B}$ is a set of *base types*, which can either be a security level s, or an interface name I, annotated with a security level, I_s. Security levels are assigned to plain data, i.e. *values* of type `int` or `bool`, as well as *expressions* yielding values of these types. The annotated interface type is assigned to *addresses*, as well as expressions yielding addresses. In either case, the meaning of the type s (resp. I_s), when given to an expression e, is that all variables *read from* within e, are of level s *or lower*.

  Note that for the purpose of the present type system, we do not distinguish between values of type `int` and `bool`, in the sense that we do not check whether these type constraints are preserved. Instead, we shall just assume that all programs are well-typed w.r.t. these simple type constraints, such that e.g. expressions in the guards of `if` and `while` constructs indeed yield boolean values. The present type system can easily be extended to incorporate such a simple type check by extending the set of base types with annotated value types $\texttt{int}_s$ and $\texttt{bool}_s$, similar to the annotated interface types.

- var(B) is a box type given to value *containers*, i.e. variables and fields. It denotes that the container can store data of type B. In the case of var(s), it denotes that the box can store data of level s *or lower*, whereas in the case of var(I_s) it additionally denotes that the address stored in the variable must be of type I.

- cmd(s) is a *phrase type* given to *code*, i.e. commands S. It denotes that all *assignments* in the code are made to variables whose security level is s *or higher*.

- proc($\widetilde{B}$):s is a procedure type given to methods $f(\tilde{x})$ `{` S `}`. It denotes that the body S can be typed as cmd(s), under the assumption that the formal parameters $\tilde{x}$ have types var($\widetilde{B}$). We shall discuss the types assigned to the 'magic variables' `this`, `sender` and `value` below.

  Note that every method declaration contains an implicit write to the `balance` field of the containing contract: hence, given the meaning of cmd(s), this also means that the security level of `balance` must always be s *or higher* than the level of any method declared in an interface.

Finally, Γ is a type environment, which is a partial function from names to types *or type environments*. The latter possibility is included because we shall represent each contract declaration as its own type environment, containing box types and procedure

types for the fields and methods of the contract, and pointed to by the corresponding interface name. Thus, if a contract has address X, then $\Gamma(X) = I_s$ for some interface name I and security level s, and $\Gamma(I) = \Gamma_I$, where Γ_I is a type environment containing the signatures of the methods and fields of the contract. We shall use the following simple interface declaration language for the interfaces of contracts:

$$\begin{array}{rcl} IC & ::= & \epsilon \mid \texttt{interface}\ I\ \{\ IF\ \ IM\ \}\ IC \\ IF & ::= & \epsilon \mid p\ :\ \mathsf{var}(B);\ \ IF \\ IM & ::= & \epsilon \mid f\ :\ \mathsf{proc}(B){:}s;\ \ IM \end{array}$$

mirroring the syntax of contract declarations.

We require that all interface declarations be *well-formed* in the sense that they must at least contain a declaration for the mandatory members, i.e. the `balance` field and the `send()` method. This ensures that we can define a minimal interface declaration called $I^\top$, such that every well-formed interface declaration is a specialisation of $I^\top$. This minimal interface contains just the signatures of the mandatory `balance` field and of the `send()` method; i.e.

```
interface I^⊤ {
  balance : var(s_⊤);
  send    : proc():s_⊥;
}
```

in the aforementioned interface declaration syntax.

Intuitively, this definition ensures that, for any valid interface definition I (containing at least `balance` and `send`) and any security level annotation s, it must hold that I_s is a subtype of $I^\top_{s_\top}$, thus always allowing us to type I_s up to $I^\top_{s_\top}$. In the following section, we shall give a definition of a subtyping relation that will ensure that this indeed is the case.

The inclusion of a contract 'supertype' $I^\top_{s_\top}$ is similar to what is done in the type system developed for Featherweight Solidity by Crafa et al. in [33]. This is necessary to enable us to give a type to the 'magic variable' `sender`, which is available within the body of every method, since this variable can be bound to the address of any contract or account. We shall assume that $I^\top \in \mathrm{dom}(\Gamma)$ for any Γ we shall consider.

We shall also use a *typed* syntax of TinySol, where local variables are now declared as

$$\mathsf{var}(B)\ \ x\ \ \texttt{:=}\ \ e$$

where B is the type of the value of the expression e. Likewise, we add annotated type names I_s to contract declarations thus:

$$\texttt{contract}\ \ X\ \ :\ \ I_s\ \ \{\ \ DF;\ \ DM\ \ \}$$

$$[\text{T-SUBS-NAME}]\frac{\Gamma \vdash \Gamma(I^1) <: \Gamma(I^2)}{\Gamma \vdash I^1_{s^1} <: I^2_{s^2}} \ (s_1 \sqsubseteq s_2)$$

$$[\text{T-SUBS-SEC}]\frac{}{\Gamma \vdash s_1 <: s_2} \ (s_1 \sqsubseteq s_2)$$

$$[\text{T-SUBS-VAR}]\frac{\Gamma \vdash B_1 <: B_2}{\Gamma \vdash \mathsf{var}(B_1) <: \mathsf{var}(B_2)}$$

$$[\text{T-SUBS-ENV}]\frac{\forall n \in \mathrm{dom}\,(\Gamma_2)\ .\ \Gamma_1(n) <: \Gamma_2(n)}{\Gamma \vdash \Gamma_1 <: \Gamma_2} \ (\mathrm{dom}\,(\Gamma_2) \subseteq \mathrm{dom}\,(\Gamma_1))$$

$$[\text{T-SUBS-CMD}]\frac{}{\Gamma \vdash \mathsf{cmd}(s_1) <: \mathsf{cmd}(s_2)} \ (s_2 \sqsubseteq s_1)$$

$$[\text{T-SUBS-PROC}]\frac{\Gamma \vdash \widetilde{B}_1 <: \widetilde{B}_2}{\Gamma \vdash \mathsf{proc}(\widetilde{B}_1) : s_1 <: \mathsf{proc}(\widetilde{B}_2) : s_2} \ (s_2 \sqsubseteq s_1)$$

Figure 6.10: Subtyping rules.

where I is a declared type name. Note that the security level is given on the *contract*, rather than on the interface. This is intentional, since multiple contracts may implement the same interface but nevertheless be categorised into different security levels. For the sake of simplicity, we shall omit the explicit definition of interfaces in the code and merely assume that an interface declaration Γ_I with an associated name I is provided for each contract.

6.4.2 Subtyping

We shall introduce a parametrised subtyping relation $\Gamma \vdash \cdot <: \cdot$ on types. For each choice of Γ, we define it as the least preorder satisfying the rules given in Figure 6.10. The parameter Γ is needed to handle subtyping for interface names I in rule [T-SUBS-NAME]. Note that by this rule we have, for each well-formed interface I_s (with security level s and interface name I) declared in Γ, that $\Gamma \vdash I_s <: I^{\top}_{s_\top}$ as expected. Also note that we write $\Gamma \vdash \widetilde{B}_1 <: \widetilde{B}_2$ to mean $\Gamma \vdash B^i_1 <: B^i_2$ for each i ($1 \le i \le n$, where $|\widetilde{B}_1| = n = |\widetilde{B}_2|$).

By rule [T-SUBS-SEC], subtyping is covariant in the types of *data*, i.e. the security level s, and likewise, the box type constructor $\mathsf{var}(B)$ is covariant by rule [T-SUBS-VAR]. On the other hand, the type constructor for commands, $\mathsf{cmd}(s)$, is *contravariant* by rule [T-SUBS-CMD]. Lastly, the type constructor for methods, $\mathsf{proc}(\widetilde{B}) : s$, is covariant in the input parameters $\widetilde{B}$ by rule [T-SUBS-PROC], but contravariant in the 'return'

type s, which indicates the level of the underlying command type. These variances are consistent with the intended meaning of the types:

- A box of type $\mathsf{var}(B)$ can store something of B or lower (where B is either s or I_s). Hence, if $\Gamma \vdash B_1 <: B_2$, then a box type $\mathsf{var}(B_2)$ can safely be used wherever a box type $\mathsf{var}(B_1)$ is needed.
- A command of type $\mathsf{cmd}(s)$ will assign to variables whose level is s or higher. Hence, if $s_1 \sqsubseteq s_2$, then a command type $\mathsf{cmd}(s_1)$ can safely be used wherever a command type $\mathsf{cmd}(s_2)$ is needed.
- A method of type $\mathsf{proc}(\widetilde{B}) : s$ expects parameters of types $\widetilde{B}$ and promises that the method body will only assign to variables that are level s or higher. Hence, if $\Gamma \vdash \widetilde{B}_1 <: \widetilde{B}_2$ and $s_2 \sqsubseteq s_1$, a command type $\mathsf{proc}(\widetilde{B}_2) : s_2$ can safely be used wherever a command type $\mathsf{proc}(\widetilde{B}_1) : s_1$ is needed. This is consistent with the type for the body S since, if S can be typed to level $\mathsf{cmd}(s_1)$, then it can also safely be typed to level $\mathsf{cmd}(s_2)$.

6.4.3 Type judgments

We can now give the rules for concluding type judgments, starting with the type rules for declarations given in Figure 6.11.

Type judgments for contract declarations are of the form $\Gamma \vdash DC$, stating that the declarations DC are *well-typed* w.r.t. the environment Γ. This holds if the declarations are consistent with the type information recorded in Γ, i.e. every field and method must have a type, and the body of each method must be typable according to the assumptions of the type. Note that the check here only ensures that every declared contract member has a type; the converse check (i.e. that every declared type in an interface also has an implementation) should also be performed. However, we shall omit this in the present treatment.

After the initial reduction step, all declarations are stored in the two environments env_{ST}, and further reductions also use the variable environment env_V for local variable declarations. Hence, we also need to be able to conclude *agreement* between these environments and Γ. These rules are given in Figure 6.12, closely mirroring those of Figure 6.11. We omit the type rules for empty environments (since an empty environment is always well-typed). As with declarations above, we also omit the rules for ensuring that all declared types in an interface also have an implementation in any contract claiming to implement that interface.

Next, we consider the type rules for statements appearing in the body of method declarations; they are given in Figure 6.13. Here, judgments are of the form $\Gamma \vdash S : \mathsf{cmd}(s)$, indicating that s is the *lowest* level of any variable written to within S. This is derived from the types of the variables occurring in S, i.e. the types $\mathsf{var}(B)$. However,

$$[\text{T-DEC-C}]\frac{\Gamma \vdash DC \quad \Gamma_1 \vdash DF \quad \Gamma_1 \vdash DM}{\Gamma \vdash \texttt{contract}\ X\ :\ I_s\ \{\ DF\ DM\ \}\ DC}$$
where:
$$\Gamma(X) = I_s$$
$$\Gamma_1 = \Gamma, \texttt{this} : \text{var}(I_s)$$

$$[\text{T-DEC-F}]\frac{\Gamma \vdash DF}{\Gamma \vdash \texttt{field}\ p\ :=\ v;\ DF}$$
where:
$$\Gamma(\texttt{this}) = \text{var}(I_s)$$
$$p \in \text{dom}\,(\Gamma(I))$$

$$[\text{T-DEC-M}]\frac{\Gamma \vdash \texttt{this.balance} : \text{var}(s) \quad \Gamma_1 \vdash S : \text{cmd}(s) \quad \Gamma \vdash DM}{\Gamma \vdash \texttt{func}\ f(\tilde{x})\ \{\ S\ \}\ DM}$$
where:
$$\Gamma(\texttt{this}) = \text{var}(I_{s_1})$$
$$\Gamma(I)(f) = \text{proc}(\widetilde{B}) : s$$
$$\Gamma_1 = \Gamma, \tilde{x} : \text{var}(\widetilde{B}), \texttt{value} : \text{var}(s), \texttt{sender} : \text{var}(I_{s_\top}^{\top})$$

Figure 6.11: Type rules for declarations.

as B can be either s or I_s, we need a way to extract just the security level and drop the interface name. For this, we write $B \rightsquigarrow s$, defined in the obvious way:

$$s \rightsquigarrow s \qquad\qquad I_s \rightsquigarrow s$$

This is used in the rules for assignments (rules [T-ASSV] and [T-ASSF]). Note that in the rules [T-IF] and [T-WHILE], we know (by our assumption that all contracts are well-typed w.r.t. simple type preservation) that e will evaluate to a boolean value, which therefore necessarily will have a type s. Thus, we do not need the extra step of $B \rightsquigarrow s$ here.

All rules are straightforward, except for [T-CALL]. According to the semantics for call (cf. rule [CALL]), every call includes an implicit read and write of the `balance` field of the calling contract, since the call will only be performed if the value of e_2 is less than, or equal to, the value of `balance` (to ensure that the subtraction will not yield a negative number). There is thus an implicit flow from `this.balance` to the body S of the method call, similar to the case for the guard expression e in an if-statement. Furthermore, there is an implicit write to the `balance` field of the callee, and thus a flow of information from one field to the other. This might initially seem like it would require both caller and callee to have the same security level for their `balance` field. However, the levels *can* differ, since by subtyping we can coerce one up to match the level of the other. For this reason, we have $\Gamma \vdash$ `this.balance` in

$$[\textsc{t-env-t}]\frac{\Gamma, \texttt{this} : \Gamma(X) \vdash \mathsf{env}_M \quad \Gamma \vdash \mathsf{env}_T}{\Gamma \vdash \mathsf{env}_T, (X, \mathsf{env}_M)}$$

$$[\textsc{t-env-m}]\frac{\Gamma \vdash \mathsf{env}_M \quad \Gamma \vdash \texttt{this.balance} : \mathsf{var}(s) \quad \Gamma_1 \vdash S : \mathsf{cmd}(s)}{\Gamma \vdash \mathsf{env}_M, (f, (\tilde{x}, S))}$$

where:

$$\Gamma(\texttt{this}) = \mathsf{var}(I_{s_1})$$
$$\Gamma(I)(f) = \mathsf{proc}(\widetilde{B}) : s$$
$$\Gamma_1 = \Gamma, \tilde{x} : \mathsf{var}(\widetilde{B}), \texttt{value} : \mathsf{var}(s), \texttt{sender} : \mathsf{var}(I_{s_\top}^\top)$$

$$[\textsc{t-env-s}]\frac{\Gamma, \texttt{this} : \Gamma(X) \vdash \mathsf{env}_F \quad \Gamma \vdash \mathsf{env}_S}{\Gamma \vdash \mathsf{env}_S, (X, \mathsf{env}_F)}$$

$$[\textsc{t-env-f}]\frac{\Gamma \vdash \mathsf{env}_F}{\Gamma \vdash \mathsf{env}_F, (p, v)}$$

where:

$$\Gamma(\texttt{this}) = \mathsf{var}(I_s)$$
$$p \in \mathrm{dom}\,(\Gamma(I))$$

$$[\textsc{t-env-v}]\frac{\Gamma \vdash \mathsf{env}_V}{\Gamma \vdash \mathsf{env}_V, (x, v)} \ (x \in \mathrm{dom}\,(\Gamma))$$

Figure 6.12: Type rules for environment agreement.

$$[\textsc{t-skip}]\frac{}{\Gamma \vdash \texttt{skip} : \mathsf{cmd}(s_{\mathsf{T}})}$$

$$[\textsc{t-throw}]\frac{}{\Gamma \vdash \texttt{throw} : \mathsf{cmd}(s_{\mathsf{T}})}$$

$$[\textsc{t-assv}]\frac{\Gamma \vdash x : \mathsf{var}(B) \quad \Gamma \vdash e : B}{\Gamma \vdash x \ \texttt{:=}\ e : \mathsf{cmd}(s)}\ (B \rightsquigarrow s)$$

$$[\textsc{t-seq}]\frac{\Gamma \vdash S_1 : \mathsf{cmd}(s) \quad \Gamma \vdash S_2 : \mathsf{cmd}(s)}{\Gamma \vdash S_1 ; \ S_2 : \mathsf{cmd}(s)}$$

$$[\textsc{t-while}]\frac{\Gamma \vdash e : s \quad \Gamma \vdash S : \mathsf{cmd}(s)}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ S : \mathsf{cmd}(s)}$$

$$[\textsc{t-subs-stm}]\frac{\Gamma \vdash S : \mathsf{cmd}(s_1) \quad \Gamma \vdash \mathsf{cmd}(s_1) <: \mathsf{cmd}(s_2)}{\Gamma \vdash S : \mathsf{cmd}(s_2)}$$

$$[\textsc{t-decv}]\frac{\Gamma \vdash e : B \quad \Gamma, x : \mathsf{var}(B) \vdash S : \mathsf{cmd}(s)}{\Gamma \vdash \mathsf{var}(B)\ \ x\ \ \texttt{:=}\ e\ \texttt{in}\ S : \mathsf{cmd}(s)}$$

$$[\textsc{t-assf}]\frac{\Gamma \vdash e_1.p : \mathsf{var}(B) \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1 . p \ \texttt{:=}\ e_2 : \mathsf{cmd}(s)}\ (B \rightsquigarrow s)$$

$$[\textsc{t-call}]\frac{\begin{array}{ll}\Gamma \vdash e_1.f : \mathsf{proc}(\widetilde{B}) : s & \Gamma \vdash \tilde{e} : \widetilde{B} \\ \Gamma \vdash \texttt{this.balance} : \mathsf{var}(s) & \Gamma \vdash e_2 : s\end{array}}{\Gamma \vdash \texttt{call}\ e_1 . f(\tilde{e})\$e_2 : \mathsf{cmd}(s)}$$

$$[\textsc{t-if}]\frac{\Gamma \vdash e : s \quad \Gamma \vdash S_{\mathsf{T}} : \mathsf{cmd}(s) \quad \Gamma \vdash S_{\mathsf{F}} : \mathsf{cmd}(s)}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ S_{\mathsf{T}}\ \texttt{else}\ S_{\mathsf{F}} : \mathsf{cmd}(s)}$$

Figure 6.13: Type rules for statements.

$$[\text{T-VAR}]\frac{\Gamma \vdash x : \mathsf{var}(B)}{\Gamma \vdash x : B}$$

$$[\text{T-FIELD}]\frac{\Gamma \vdash e.p : \mathsf{var}(B)}{\Gamma \vdash e.p : B}$$

$$[\text{T-SUBS-EXP}]\frac{\Gamma \vdash e : B_1 \quad \Gamma \vdash B_1 <: B_2}{\Gamma \vdash e : B_2}$$

$$[\text{T-VAL}]\frac{}{\Gamma \vdash v : B}\left(B = \begin{cases}\Gamma(v) & \text{if } v \in \mathsf{ANames} \\ s & \text{otherwise}\end{cases}\right)$$

$$[\text{T-OP}]\frac{\Gamma \vdash e_1 : B_1 \quad \ldots \quad \Gamma \vdash e_n : B_n}{\Gamma \vdash \mathsf{op}(e_1, \ldots, e_n) : s}\begin{pmatrix} B_1 \rightsquigarrow s \\ \vdots \\ B_n \rightsquigarrow s \end{pmatrix}$$

Figure 6.14: Type rules for expressions

the premise, to be explicitly *concluded*, rather than as a simple lookup. This enables calls from a lower security level into a higher security level, but not the other way around.

Next, we consider the type rules for expressions e, given in Figure 6.14. Here, judgments are of the form $\Gamma \vdash e : B$. There are a few things to note:

- In rule [T-VAL], the type of a value v can be chosen freely, if v is a value type, i.e. of type `int` or `bool`. This rule is a consequence of the fact that there is no simple relationship between the datatype of a value and its security level. The actual security level will then be determined by the type of the variable (resp. field) to which it is assigned.

- The rules [T-VAR] and [T-FIELD] simply unwrap the type of the contained value from the box type of the container. Note that here we assume that x also covers the 'magic variable' names `this`, `sender` and `value`, and that p also covers the field name `balance`.

- Finally, in rule [T-OP], we require that all arguments and the return value must be typable to the same security level s. Note in particular that we assume that *no* operation is defined with an address *return* type; i.e. we do not allow any form of pointer arithmetic. Operations may be defined on addresses for their *arguments*, e.g. equality testing, but the return type must be one of the other value types, which can be given a security level. Thus, in the rule [T-OP], we also need to extract the security level s from the types of the argument expressions.

$$[\textsc{t-box-x}]\frac{}{\Gamma \vdash x : \mathsf{var}(B)}\ (\Gamma(x) = \mathsf{var}(B))$$

$$[\textsc{t-m-sub}]\frac{\begin{array}{c}\Gamma \vdash e.f : \mathsf{proc}(\widetilde{B}_1) : s_1 \\ \Gamma \vdash \mathsf{proc}(\widetilde{B}_1) : s_1 <: \mathsf{proc}(\widetilde{B}_2) : s_2\end{array}}{\Gamma \vdash e.f : \mathsf{proc}(\widetilde{B}_2) : s_2}$$

$$[\textsc{t-box-f}]\frac{\Gamma \vdash e : I_s}{\Gamma \vdash e.p : \mathsf{var}(B)}\ \left(\begin{array}{l}\Gamma(I)(p) = \mathsf{var}(B) \\ B \rightsquigarrow s\end{array}\right)$$

$$[\textsc{t-meth}]\frac{\Gamma \vdash e : I_s}{\Gamma \vdash e.f : \mathsf{proc}(\widetilde{B}) : s}\ \left(\Gamma(I)(f) = \mathsf{proc}(\widetilde{B}) : s\right)$$

Figure 6.15: Type rules for method, variable and field lookup.

Finally, we have the look-up rules for methods, variables and fields, given in Figure 6.15.

- In rule [T-BOX-X] we assume that x also covers the magic variable names `this`, `sender` and `value`.
- In rule [T-BOX-F] we assume that p also covers the special field name `balance`. Furthermore, we require e in $e.p$ to resolve to an *interface name* rather than variable; i.e. the expression must yield an address. This is again warranted by our assumption that expressions are well-typed w.r.t. simple type preservation.
- The same is the case in rule [T-METH] for method lookup $e.f$, which is used in the premise of the rule [T-CALL].

In the lookup rules, the expression e is an *object path*, which must resolve to an address. As we disallow operations op to return addresses, the object paths form a proper subset of the set of expressions, since they can only consist of variable lookups, field reads or addresses given as pure values. Note that, in the rules [T-BOX-F] and [T-METH], we require that the object path e must be typable as an interface with the *same* security level s as the value (resp. method) that is being looked up. This is necessary to ensure that values residing in a higher-level part of the memory cannot affect values at lower levels, in this case by altering the *path* to the object being resolved.

6.4.4 Safety and soundness

As is the case for the type system proposed in [130], our type system does not have a now-safety predicate in the usual sense, since (invariant) safety in simple type

$$[\textsc{eq-env-}\emptyset]\frac{}{\Gamma \vdash \mathsf{env}_X^{\emptyset} =_s \mathsf{env}_X^{\emptyset}} \ (X \in \{V, S, F, T, M\})$$

$$[\textsc{eq-env}_V]\frac{\Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2}{\Gamma \vdash \mathsf{env}_V^1, (x, v_1) =_s \mathsf{env}_V^2, (x, v_2)} \left(\begin{array}{l} \Gamma(x) = \mathsf{var}(s') \\ s' \sqsubseteq s \implies v_1 = v_2 \end{array} \right)$$

$$[\textsc{eq-env}_S]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma(\Gamma(X)) \vdash \mathsf{env}_F^1 =_s \mathsf{env}_F^2}{\Gamma \vdash \mathsf{env}_S^1, (X, \mathsf{env}_F^1) =_s \mathsf{env}_S^2, (X, \mathsf{env}_F^2)}$$

$$[\textsc{eq-env}_F]\frac{\Gamma \vdash \mathsf{env}_F^1 =_s \mathsf{env}_F^2}{\Gamma \vdash \mathsf{env}_F^1, (p, v_1) =_s \mathsf{env}_F^2, (p, v_2)} \left(\begin{array}{l} \Gamma(p) = \mathsf{var}(s') \\ s' \sqsubseteq s \implies v_1 = v_2 \end{array} \right)$$

$$[\textsc{eq-env}_T]\frac{\Gamma \vdash \mathsf{env}_T^1 =_s \mathsf{env}_T^2}{\Gamma \vdash \mathsf{env}_T^1, (X, \mathsf{env}_M^1) =_s \mathsf{env}_T^2, (X, \mathsf{env}_M^2)} \left(\begin{array}{l} \Gamma(X) = I_{s'} \\ s' \sqsubseteq s \implies \mathsf{env}_M^1 = \mathsf{env}_M^2 \end{array} \right)$$

$$[\textsc{eq-env}_{SV}]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2}{\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2}$$

$$[\textsc{eq-env}_{ST}]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma \vdash \mathsf{env}_T^1 =_s \mathsf{env}_T^2}{\Gamma \vdash \mathsf{env}_{ST}^1 =_s \mathsf{env}_{ST}^2}$$

Figure 6.16: Rules for the s-parameterised equivalence relation.

systems is a 1-property, whereas noninterference is a hyper-property (specifically, a 2-property). Instead, the meaning of 'safety' is expressed directly in the meaning of the types. Specifically:

- If an expression e has type B such that $B \rightsquigarrow s$, then it denotes that all variables *read from* in the evaluation of e are of level s or *lower*, i.e. no read-up.
- If a statement S has type $\mathsf{cmd}(s)$, then it denotes that all variables *written to* in the execution of S are of level s or *higher*, i.e. no write-down.

Intuitively, the meaning of these two types together imply that information from higher-level variables cannot flow into lower-level variables. For a statement such as $x \ := \ e$ to be well-typed, it must therefore be the case that, if $\Gamma \vdash x : \mathsf{var}(s_1)$ and $\Gamma \vdash e : s_2$, then $s_2 \sqsubseteq s_1$. Since s_2 can be coerced up to s_1 through subtyping to match the level of the variable, the statement itself can then be typed as $\mathsf{cmd}(s_1)$. We shall prove that our type system indeed ensures these properties in Theorems 13-15 below.

Before proceeding, we need to define a way to express that two *states*, i.e. two collections of variable and field environments env_{SV}, are *equal* up to a certain security

level s. This relation, written $\Gamma \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$, is given by the rules in Figure 6.16. Note in particular that the definition implies that env^1_{SV} and env^2_{SV} must have the same domain, and this carries over to the inner environments env_F inside env_S. The above definition gives us the following obvious result, which can be shown by induction on the rules of $=_s$:

Lemma 48 (Restriction)**.** *If*

- $\Gamma \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$*, and*
- $s' \sqsubseteq s$*,*

then $\Gamma \vdash \mathsf{env}^1_{SV} =_{s'} \mathsf{env}^2_{SV}$.

Given our annotation of security levels on interfaces as well, we also extend the $=_s$ relation to method tables env_T, and finally to the combined representation of state and code, i.e. env_{ST}. Next, we need the standard lemmas for strengthening and weakening of the variable environment:

Lemma 49 (Strengthening)**.** *If*

$$\Gamma, x : \mathsf{var}(B) \vdash (x, v_1) : \mathsf{env}^1_V =_s (x, v_2) : \mathsf{env}^2_V$$

then also $\Gamma \vdash \mathsf{env}^1_V =_s \mathsf{env}^2_V$.

Lemma 50 (Weakening)**.** *If*

- $\Gamma \vdash \mathsf{env}^1_V =_s \mathsf{env}^2_V$*, and*
- $x \notin \mathrm{dom}\left(\mathsf{env}^1_V\right)$*, and*
- $x \notin \mathrm{dom}\left(\mathsf{env}^2_V\right)$

then also

$$\Gamma, x : \mathsf{var}(B) \vdash (x, v_1) : \mathsf{env}^1_V =_s (x, v_2) : \mathsf{env}^2_V$$

for any B, v, x.

Both results can be shown by induction on the rules of $=_s$. Furthermore, both of the lemmas can then be directly extended to $\Gamma \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$. With this, we can now state the first of our main theorems:

Theorem 13 (Preservation)**.** *Assume that*

- $\Gamma \vdash S : \mathsf{cmd}(s)$*, and*
- $\Gamma \vdash \mathsf{env}_T$*, and*

- $\Gamma \vdash \mathsf{env}_{SV}$, *and*
- $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$.

Then, $\Gamma \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}'_{SV}$ *for any* s' *such that* $s \not\sqsubseteq s'$.

The proof proceeds by induction on the derivation of $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$. It can be found in Section 6.8.1.

The Preservation theorem assures us that the promise made by the type $\mathsf{cmd}(s)$ is actually fulfilled. If $\Gamma \vdash S : \mathsf{cmd}(s)$, then every variable or field written to in S will be of level s *or higher*; hence every variable or field of a level that is *strictly lower* than, or incomparable to, s will be unaffected. Thus, the pre- and post-transition states will be equal on all values stored in variables or fields of level s' or lower, since they cannot have been changed during the execution of S. In other words, what is shown to be 'preserved' in this theorem is the *values* at levels lower than, or incomparable to, s.

Note that the theorem does not show preservation of *well-typedness* for the environments (as is otherwise usually required in preservation proofs for type systems). Indeed, a result saying that also $\Gamma \vdash \mathsf{env}'_{SV}$ would be pointless. As can be seen in Figure 6.12, the type judgment $\Gamma \vdash \mathsf{env}_{SV}$ only ensures that every field and variable in env_{SV} has *any* type in Γ. The *number* of declared fields and variables cannot change between the pre- and post-states of a transition (this is ensured by the rule [DECV]); only the stored values can change, but there is no inherent relationship between a value and its assigned security level.

Our next theorem assures us that the type of an expression is also in accordance with the intended meaning, namely: if $\Gamma \vdash e : s$, then every variable (or field) read from in e will be of level s or lower (i.e. no read-down of values from a higher level). We express this by considering two different states, env^1_{SV} and env^2_{SV}, which must agree on all values of level s and lower. Evaluating e w.r.t. either of these states should then yield the same result.

Theorem 14 (Safety for expressions)**.** *Assume that*

- $\Gamma \vdash e : B$ *where* $B \rightsquigarrow s$, *and*
- $\Gamma \vdash \mathsf{env}^1_{SV}$, *and*
- $\Gamma \vdash \mathsf{env}^2_{SV}$, *and*
- $\Gamma \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$.

Then, $\mathsf{env}^1_{SV} \vdash e \to v$ *and* $\mathsf{env}^2_{SV} \vdash e \to v$.

The proof is by induction on the derivation of $\Gamma \vdash e : B$. It can be found in Section 6.8.2.

Finally, we can use the preceding two theorems to show soundness for the type system. The soundness theorem expresses that, if a statement S is well-typed *to any level* s_1 and we execute S with any two states env^1_{SV} and env^2_{SV} that agree *up to any level* s_2, then the resulting states $\mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^{2'}_{SV}$ will still agree on all values up to level s_2. This ensures noninterference, since any difference in values of a *higher* level than s_2 cannot induce a difference in the computation of values at any lower levels.

Theorem 15 (Soundness)**.** *Assume that*

- $\Gamma \vdash S : \mathsf{cmd}(s_1)$*, and*
- $\Gamma \vdash \mathsf{env}_T$*, and*
- $\Gamma \vdash \mathsf{env}^1_{SV}$*, and*
- $\Gamma \vdash \mathsf{env}^2_{SV}$*, and*
- $\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$*, and*
- $\mathsf{env}_T \vdash \langle S, \mathsf{env}^1_{SV} \rangle \to \mathsf{env}^{1'}_{SV}$*, and*
- $\mathsf{env}_T \vdash \langle S, \mathsf{env}^2_{SV} \rangle \to \mathsf{env}^{2'}_{SV}$.

Then, $\Gamma \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$.

There are two cases to consider in the proof:

1. When $s_1 \not\sqsubseteq s_2$, i.e. s_2 is either strictly below s_1, or they are incomparable, we can conclude the following by Theorem 13:

$$\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^{1'}_{SV}$$
$$\Gamma \vdash \mathsf{env}^2_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}.$$

   From $\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$ we can then conclude $\Gamma \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$.

2. When $s_1 \sqsubseteq s_2$, we proceed by induction on the sum of the lengths of the derivations of the transitions. The proof can be found in Section 6.8.3.

Theorem 15 corresponds to the soundness theorem proved by Volpano, Smith and Irvine [130] for their While-like language. However, given the object-oriented nature of TinySol, we can actually take this one step further and allow even parts of the code to vary. Specifically, given two 'method table' environments, env^1_T and env^2_T, we just require that these two environments agree up to the same level s_2 to ensure agreement of the resulting two states $\mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^{2'}_{SV}$. We state this in the following theorem:

Theorem 16 (Extended soundness)**.** *Assume that*

- $\Gamma \vdash S : \mathsf{cmd}(s_1)$*, and*
- $\Gamma \vdash \mathsf{env}_T^1$*, and*
- $\Gamma \vdash \mathsf{env}_T^2$*, and*
- $\Gamma \vdash \mathsf{env}_T^1 =_{s_2} \mathsf{env}_T^2$*, and*
- $\Gamma \vdash \mathsf{env}_{SV}^1$*, and*
- $\Gamma \vdash \mathsf{env}_{SV}^2$*, and*
- $\Gamma \vdash \mathsf{env}_{SV}^1 =_{s_2} \mathsf{env}_{SV}^2$*, and*
- $\mathsf{env}_T^1 \vdash \langle S, \mathsf{env}_{SV}^1 \rangle \to \mathsf{env}_{SV}^{1'}$*, and*
- $\mathsf{env}_T^2 \vdash \langle S, \mathsf{env}_{SV}^2 \rangle \to \mathsf{env}_{SV}^{2'}$.

Then, $\Gamma \vdash \mathsf{env}_{SV}^{1'} =_{s_2} \mathsf{env}_{SV}^{2'}$.

Proof. By induction on the sum of the lengths of the derivations

$$\mathsf{env}_T^1 \vdash \langle S, \mathsf{env}_{SV}^1 \rangle \to_S \mathsf{env}_{SV}^{1'}$$
$$\mathsf{env}_T^2 \vdash \langle S, \mathsf{env}_{SV}^2 \rangle \to_S \mathsf{env}_{SV}^{2'}.$$

All the cases are like the corresponding ones in the proof of Theorem 15, except for the case of [CALL], which is the only case where env_T plays a role. There are then two cases to consider:

- If $s_1 \not\sqsubseteq s_2$, then env_T^1 and env_T^2 only agree up to level s_2, but may differ on other levels, including s_1 (which is either strictly above s_2, or they are incomparable). Thus the call may be to two different methods (albeit with the same signature), namely $\mathsf{env}_T^1(Y)(f) = (\tilde{x}, S_1)$ and $\mathsf{env}_T^2(Y)(f) = (\tilde{x}, S_2)$. However, as S is typed to level s_1, this cannot induce a difference at level s_1 or lower. In particular, by the premises of rule [CALL] we have that $\mathsf{env}_T^1 \vdash \langle S_1, \mathsf{env}_{SV}^1 \rangle \to_S \mathsf{env}_{SV}^{1'}$ and $\mathsf{env}_T^2 \vdash \langle S_2, \mathsf{env}_{SV}^2 \rangle \to_S \mathsf{env}_{SV}^{2'}$. By twice application of Theorem 13, we can then conclude that

  $$\Gamma \vdash \mathsf{env}_{SV}^1 =_{s_2} \mathsf{env}_{SV}^{1'}$$
  $$\Gamma \vdash \mathsf{env}_{SV}^2 =_{s_2} \mathsf{env}_{SV}^{2'}.$$

  As we know by assumption that $\Gamma \vdash \mathsf{env}_{SV}^1 =_{s_2} \mathsf{env}_{SV}^2$, we can therefore also conclude that $\Gamma \vdash \mathsf{env}_{SV}^{1'} =_{s_2} \mathsf{env}_{SV}^{2'}$.

- If $s_1 \sqsubseteq s_2$, then env_T^1 and env_T^2 agree on all levels up to level s_2 *including* level s_1. By assumption, $\Gamma \vdash \mathsf{env}_T^1 =_{s_2} \mathsf{env}_T^2$ and so, by [EQ-ENV$_T$], $\mathsf{env}_T^1(Y) = \mathsf{env}_T^2(Y)$, since Y resides at level s_1. In particular, $\mathsf{env}_T^1(Y)(f) = \mathsf{env}_T^2(Y)(f) = (\tilde{x}, S')$. The rest of the proof then proceeds like in the case for [CALL] in the proof of Theorem 15 (Section 6.8.3). □

6.4.5 Extending the type system to transactions

A transaction is nothing but a method call with real-valued parameters and `sender` set to an *account* address, which corresponds to a minimal implementation of I^{T}. Thus, the theorems from the preceding section can easily be extended to transactions and blockchains.

A blockchain consists of a set of contract declarations DC, followed by a list of transactions $\widetilde{T}$. Hence, we can conclude $\Gamma \vdash DC\ \widetilde{T} : \mathsf{cmd}(s)$, if it holds that $\Gamma \vdash DC$ and $\Gamma \vdash \widetilde{T} : \mathsf{cmd}(s)$. The latter can be simply concluded by the following rules:

$$[\text{T-EMPTY}]\frac{}{\Gamma \vdash \epsilon : \mathsf{cmd}(s)}$$

$$[\text{T-TRANS}]\frac{\Gamma \vdash X\,.\,f\,(\tilde{v}) : n : \mathsf{cmd}(s) \quad \Gamma \vdash \widetilde{T} : \mathsf{cmd}(s)}{\Gamma \vdash A\text{->}X\,.\,f\,(\tilde{v})\,\$n, \widetilde{T} : \mathsf{cmd}(s)}$$

This gives us the following two results:

Lemma 51. *If* $\Gamma \vdash DC$ *and* $\left\langle DC, \mathsf{env}_{ST}^{\emptyset} \right\rangle \to \mathsf{env}_{ST}$, *then* $\Gamma \vdash \mathsf{env}_{ST}$.

Lemma 52. *If*

- $\Gamma \vdash A\text{->}X\,.\,f\,(\tilde{v})\,\$n, \widetilde{T} : \mathsf{cmd}(s)$, *and*
- $\Gamma \vdash \mathsf{env}_{ST}$, *and*
- $\left\langle A\text{->}X\,.\,f\,(\tilde{v})\,\$n, \widetilde{T}, \mathsf{env}_{ST} \right\rangle \to \left\langle \widetilde{T}, \mathsf{env}'_S, \mathsf{env}_T \right\rangle$

then also $\Gamma \vdash \mathsf{env}'_S$.

As the initial step (the 'genesis event') does nothing except transforming the declaration DC into the environment representation env_{ST}, the first result is obvious, and as the rule [TRANS] just unwraps a transaction step into a call to the corresponding method, the second result follows directly from the Preservation theorem. This can then be generalised in an obvious way to the whole transaction list. Likewise, the Safety and Soundness theorems can be extended to transactions in the same manner.

6.4.6 Noninterference and call integrity

As immediately evident from Definition 46 and Theorem 15, well-typedness ensures noninterference:

Corollary 11 (Noninterference). *Assume a set of security levels* $\mathcal{S} \triangleq \{L, H\}$*, with* $L \sqsubseteq L$*,* $L \sqsubseteq H$ *and* $H \sqsubseteq H$*, and furthermore that*

- $\Gamma \vdash \widetilde{T} : \mathsf{cmd}(s)$*, and*
- $\Gamma \vdash \mathsf{env}^1_{ST}$*, and*
- $\Gamma \vdash \mathsf{env}^2_{ST}$*, and*
- $\Gamma \vdash \mathsf{env}^1_{ST} =_L \mathsf{env}^2_{ST}$*, and*
- $\langle \widetilde{T}, \mathsf{env}^1_{ST} \rangle \rightarrow^* \mathsf{env}^{1'}_{ST}$*, and*
- $\langle \widetilde{T}, \mathsf{env}^2_{ST} \rangle \rightarrow^* \mathsf{env}^{2'}_{ST}$.

Then, $\Gamma \vdash \mathsf{env}^{1'}_{ST} =_L \mathsf{env}^{2'}_{ST}$.

Proof. By Theorem 15 and Lemma 52. □

From Corollary 11, we then obviously also have that $\Gamma \vdash \mathsf{env}^{1'}_{S} =_L \mathsf{env}^{2'}_{S}$, regardless of whether s is L or H. In particular, we can assign security levels to entire contracts, as well as all their members. Thus, our type system can be used to ensure noninterference according to Definition 46.

As we previously argued in Remark 1, noninterference and call integrity are incomparable properties. However, as our next theorem shows, *well-typedness* actually *also* ensures call integrity. This is surprising, so before stating the theorem, we should give some hints as to why this is the case.

The definition of call integrity (Definition 45) requires the execution of any code in a contract C to be unaffected by all contracts in an 'untrusted set' $\mathcal{Y}$, regardless of whether parts of the code in $\mathcal{Y}$ execute before, meanwhile or after the code in C. This is expressed by a quantification over all possible traces resulting from a change in $\mathcal{Y}$, i.e. either in the code or in the values of the fields. Regardless of any such change, it must hold that the sequence of method calls originating from C be the same.

Noninterference, on the other hand, says nothing about execution traces, but only speaks of the correspondence between values residing in the memory before and after the execution step. The two counter-examples used in Remark 1 made use of this fact:

- The first counter-example had C be unable to perform any method calls at all, thus obviously satisfying call integrity, but allowed different `balance` values to be transferred into it from a 'high' context by means of a method

call, thereby violating noninterference. However, this situation is ruled out by well-typedness, because well-typedness disallows *any* method calls from a 'high' to a 'low' context, precisely because every method call may transfer the `value` parameter along with each call.

- The second counter-example had an `if` statement in C (the 'low' context) depend on a field value in a 'high' context. The two branches then perform two different method calls, thus enabling a change of the 'high' context to induce two different execution traces for C. Thus, the example satisfies noninterference, because no value stored in memory is changed, but it obviously does not satisfy call integrity. However, this situation is also ruled out by well-typedness, because the rule [T-IF] does not allow the boolean guard expression e in a 'low' context to depend on a value from a 'high' context.

Thus, both of the two counter-examples would be rejected by the type system. With a setting of L for the 'trusted' segment and H for the 'untrusted',[7] no values or computations performed in the untrusted segment can affect the values in the trusted segment, *nor* the value of any expression in this segment, nor can it even perform a call into the trusted segment. On the other hand, the trusted segment *can* call out into the untrusted part, but such a call cannot then reenter the trusted segment: it must return before any further calls from the trusted segment can happen.

Theorem 17 (Well-typedness implies call integrity). *Let* $\mathcal{S} \triangleq \{L, H\}$ *with* $L \sqsubseteq L$, $L \sqsubseteq H$ *and* $H \sqsubseteq H$. *Fix the two sets of addresses* $\mathcal{X}$ *and* $\mathcal{Y}$ *as in Definition 45, such that* $\mathcal{A} = \mathcal{X} \cup \mathcal{Y}$ *and* $\mathcal{A} = \mathrm{dom}(\mathsf{env}_T)$. *Fix a type assignment* Γ *such that*

- $\forall X \in \mathcal{X} \,.\, \Gamma(X) = I_L$ *for some* I *where*
  - $\forall p \in \Gamma(I) \,.\, \Gamma(I)(p) = \mathsf{var}(B)$ *where* $B \rightsquigarrow L$, *and*
  - $\forall f \in \Gamma(I) \,.\, \Gamma(I)(f) = \mathsf{proc}(\widetilde{B}) : L$ *for any* $\widetilde{B}$
- *and with the level* H *given to all other interfaces, fields and methods.*

Also assume that

- $\Gamma \vdash T : \mathsf{cmd}(s)$, *and*
- $\Gamma \vdash \mathsf{env}^1_{ST}$, *and*
- $\Gamma \vdash \mathsf{env}^2_{ST}$, *and*
- $\Gamma \vdash \mathsf{env}^1_{ST} =_L \mathsf{env}^2_{ST}$, *and*

[7] This counter-intuitive naming can perhaps best be thought of as indicating our level of *distrust* in a contract.

- $\langle T, \mathsf{env}_{ST}^1 \rangle \xrightarrow{\pi_1} \mathsf{env}_{ST}^{1'}$, *and*
- $\langle T, \mathsf{env}_{ST}^2 \rangle \xrightarrow{\pi_2} \mathsf{env}_{ST}^{2'}$.

Then, $\pi_1 \downarrow_X = \pi_2 \downarrow_X$, *for any* $X \in \mathcal{X}$.

Proof. By induction on the length of the transaction T. The base case (i.e., $T = \epsilon$) is trivial, since $\pi_1 = \pi_2 = \epsilon$.

For the inductive case, let $T = A\texttt{->}Y\,.\,f\,(\tilde{v})\,\$n, T'$. We consider two possible cases.

Case 1: If $Y \in \mathcal{Y}$, then the result is immediate by [T-CALL], because no call into $\mathcal{X}$ is allowed due to the implicit write to `balance` in every method call. Thus, neither of the traces π_1 or π_2 would contain any method call from any X in $\mathcal{X}$, and therefore $\pi_1 \downarrow_X \; = \pi_2 \downarrow_X$ obviously holds since $\pi_1 \downarrow_X \; = \epsilon = \pi_2 \downarrow_X$.

Case 2: If $Y \in \mathcal{X}$, we know by assumption that $\Gamma(Y) = I_L$ for some I. Hence $\mathsf{env}_T^1(Y) = \mathsf{env}_T^2(Y) = \mathsf{env}_M^Y$ and the method called is the same in both runs, say $\mathsf{env}_M^Y(f) = (\tilde{x}, S)$. Let

$$\mathsf{env}_V = (\tilde{x}, \tilde{v}), (\texttt{this}, Y), (\texttt{sender}, A), (\texttt{value}, n)$$

Then

$$\pi_1 = A\texttt{->}Y\,.\,f\,(\tilde{v})\,\$n, \pi_1', \pi_1''$$
$$\pi_2 = A\texttt{->}Y\,.\,f\,(\tilde{v})\,\$n, \pi_2', \pi_2''$$

where

$$\begin{aligned} \mathsf{env}_T^1 \vdash \langle S, \mathsf{env}_S^1, \mathsf{env}_V \rangle &\xrightarrow{\pi_1'}_S \mathsf{env}_S^{1''}, \mathsf{env}_V^{1'} \\ \mathsf{env}_T^2 \vdash \langle S, \mathsf{env}_S^1, \mathsf{env}_V \rangle &\xrightarrow{\pi_2'}_S \mathsf{env}_S^{2''}, \mathsf{env}_V^{2'} \end{aligned} \tag{6.1}$$

and

$$\langle T', \mathsf{env}_S^{1''}, \mathsf{env}_T^1 \rangle \xrightarrow{\pi_1''}_B \mathsf{env}_{ST}^{1'}$$
$$\langle T', \mathsf{env}_S^{2''}, \mathsf{env}_T^2 \rangle \xrightarrow{\pi_2''}_B \mathsf{env}_{ST}^{2'}$$

By the induction hypothesis, we have that $\pi_1'' \downarrow_X \; = \pi_2'' \downarrow_X$, for every $X \in \mathcal{X}$. To prove that $\pi_1' \downarrow_X \; = \pi_2' \downarrow_X$, we then proceed by a second induction, on the inference of the transitions in (6.1):

Firstly, by Theorem 16 we know that $\Gamma \vdash \mathsf{env}_S^{1'}, \mathsf{env}_V^{1'} =_L \mathsf{env}_S^{2'}, \mathsf{env}_V^{2'}$. Hence, the last rule used in both inferences is the same: in all cases, this is syntax-driven; and moreover, whenever there is a guard that may change the evolution, this guard is evaluated at the same value, because of Theorem 14.

There are four base cases: [SKIP], [ASSV], [ASSF] and [WHILE$_F$]. All cases are trivial, since no call is performed and so $\pi_1' = \epsilon = \pi_2'$.

There are five inductive cases, according to the last rule used in the inferences:

- [DECV]: Then $S = \mathsf{var}(B)\ \ x\ \texttt{:=}\ e\ \texttt{in}\ S'$. By Theorem 14, $\mathsf{env}_S^1, \mathsf{env}_V \vdash e \to_e v$ and $\mathsf{env}_S^2, \mathsf{env}_V \vdash e \to_e v$. Then,

$$\mathsf{env}_T^1 \vdash \langle S', \mathsf{env}_S^1, \mathsf{env}_V, (x, v)\rangle \xrightarrow{\pi_1'}_S \mathsf{env}_S^{1'}, \mathsf{env}_V^{1'}, (x, v_1)$$
$$\mathsf{env}_T^2 \vdash \langle S', \mathsf{env}_S^2, \mathsf{env}_V, (x, v)\rangle \xrightarrow{\pi_2'}_S \mathsf{env}_S^{1'}, \mathsf{env}_V^{2'}, (x, v_2)$$

  and, by the second induction hypothesis, we conclude that $\pi_1' \downarrow_X\ = \pi_2' \downarrow_X$, for all $X \in \mathcal{X}$.

- [SEQ]: Then $S = S_1; S_2$ and so $\pi_1' = \pi_1^1, \pi_1^2$ and $\pi_2' = \pi_2^1, \pi_2^2$. By two applications of the second induction hypothesis, we obtain that $\pi_1^1 \downarrow_X\ = \pi_2^1 \downarrow_X$ and that $\pi_1^2 \downarrow_X\ = \pi_2^2 \downarrow_X$, which allow us to easily conclude.

- [IF]: Then, $S = \texttt{if}\ \ e\ \ \texttt{then}\ \ S_\mathsf{T}\ \ \texttt{else}\ \ S_\mathsf{F}$. By Theorem 14, $\mathsf{env}_S^1, \mathsf{env}_V \vdash e \to_e b$ and $\mathsf{env}_S^2, \mathsf{env}_V \vdash e \to_e b$; so, the same branch is chosen in both contexts and the same command S_b is executed. The case then holds for S_b by the second induction hypothesis.

- [WHILE$_T$]: The argument is the same as for [IF] above.

- [CALL]: Then $S = \texttt{call}\ \ e_1\texttt{.}g(\tilde{e})\texttt{\$}e_2$. By Theorem 14, $\mathsf{env}_S^1, \mathsf{env}_V \vdash e_1 \to_e Z$ and $\mathsf{env}_S^2, \mathsf{env}_V \vdash e_1 \to_e Z$, for some address Z such that $\Gamma(Z) = I_s$ for some interface I and security level s. Moreover, $\mathsf{env}_T^1(Z)(\mathrm{g}) = (\widetilde{x_1}, S_1)$ and $\mathsf{env}_T^2(Z)(\mathrm{g}) = (\widetilde{x_2}, S_2)$, for $|\widetilde{x_1}| = |\widetilde{x_2}| = |\tilde{e}|$. Moreover, $\tilde{e}$ and e_2 are both typed as L and, by Theorem 14, are evaluated at the same values $\widetilde{v'}$ and n'. There are then two cases, depending on the level s:

  - If $s = L$, then $S_1 = S_2 = S'$ and $\widetilde{x_1} = \widetilde{x_2} = \widetilde{x'}$. Now, let

$$\mathsf{env}_V' = (\widetilde{x'}, \widetilde{v'}), (\texttt{this}, Z), (\texttt{sender}, Y), (\texttt{value}, n')$$

    Then,

$$\mathsf{env}_T^1 \vdash \langle S', \mathsf{env}_S^1, \mathsf{env}_V'\rangle \xrightarrow{\hat{\pi}_1}_S \mathsf{env}_S^{1'}, \mathsf{env}_V^{1'}$$
$$\mathsf{env}_T^2 \vdash \langle S', \mathsf{env}_S^2, \mathsf{env}_V'\rangle \xrightarrow{\hat{\pi}_2}_S \mathsf{env}_S^{1'}, \mathsf{env}_V^{2'}$$

    By the second induction hypothesis, $\hat{\pi}_1 \downarrow_X\ = \ \hat{\pi}_2 \downarrow_X$, for any $X \in \mathcal{X}$. Thus

$$\pi_1' \downarrow_X\ = \big(Y\texttt{->}Z\texttt{.}g(\widetilde{v'})\texttt{\$}n', \hat{\pi}_1\big) \downarrow_X\ = \big(Y\texttt{->}Z\texttt{.}g(\widetilde{v'})\texttt{\$}n', \hat{\pi}_2\big) \downarrow_X\ = \pi_2' \downarrow_X .$$

– If $s = H$, then S_1 and S_2 may differ. However, the first scenario depicted in the proof of this theorem then applies, since $Z \in \mathcal{Y}$. Thus, by letting $\hat{\pi}_1$ and $\hat{\pi}_2$ be the traces generated by S_1 and S_2 respectively, we know that $\hat{\pi}_1 \downarrow_X = \epsilon = \hat{\pi}_2 \downarrow_X$, for all $X \in \mathcal{X}$, because the call can never *reenter* any contract in the Low segment $\mathcal{X}$; in particular, no further call from X can appear in the traces. Hence,

$$\pi_1' \downarrow_X = \left(Y\texttt{->}Z.g(\widetilde{v'})\$n'\right) \downarrow_X = \pi_2' \downarrow_X . \qquad \square$$

Theorem 17 tells us that *every* contract X in the trusted segment $\mathcal{X}$ has call integrity w.r.t. the untrusted segment $\mathcal{Y}$. This is thus a stronger condition than that of Definition 45, which only defines call integrity for a *single* contract $C \in \mathcal{X}$, rather than for the whole set. This means that our type system will reject cases where e.g. C calls another contract $Z \in \mathcal{X}$ and Z calls `send()` methods of different contracts, depending on a 'high' value. As `send()` is always ensured to do nothing, such calls could never lead to C being reentered, so this would actually still be safe, even though Z itself would not satisfy call integrity. Thus, this is an example of what resides in the 'slack' of our type system.

However, this situation seems rather contrived, since it depends specifically on the `send()` method, which is always ensured to do nothing except returning. For practical purposes, it would be strange to imagine a contract $C \in \mathcal{X}$ having call integrity w.r.t. $\mathcal{Y}$, but *without* the other contracts in $\mathcal{X}$ also satisfying call integrity w.r.t. $\mathcal{Y}$. Thus, our type system seems to yield a reasonable approximation to the property of call integrity.

6.5 Examples and limitations

Let us see a few examples of the application of the type system. To begin with, consider the first counter-example in Remark 1, which should be ill-typed by the type system. In the counter-example we say that `X` is Low and `Y` is High, so we let them both implement the interface *I*`<s>` defined as follows:

```
1 interface I<s> {
2   balance : var(s)
3   go      : proc():s
4 }
5
6 contract X : I<L> { ... }
7 contract Y : I<H> { ... }
```

where *I*`<L>` (resp. *I*`<H>`) is a shorthand for I_L (resp. I_H) with all occurrences of s within the interface definition replaced by L (resp. H). A part of the failing typing

derivation for the body of the method `Y.go()` in the declaration of contract `Y` is:

$$\dfrac{\dfrac{\Gamma(\texttt{this}) = \texttt{I<H>}}{\Gamma \vdash \texttt{this} : \texttt{I<H>}} \quad \Gamma(\texttt{I<H>})(\texttt{balance}) = \mathsf{var}(H)}{\Gamma \vdash \texttt{this.balance} : \mathsf{var}(H)} \qquad \dfrac{\dfrac{\Gamma(\texttt{X}) = \texttt{I<L>}}{\Gamma \nvdash \texttt{X} : \texttt{I<H>}} \quad \Gamma(\texttt{I<L>})(\texttt{go}) \neq \mathsf{proc}():H}{\Gamma \nvdash \texttt{X.go} : \mathsf{proc}():H}$$
$$\Gamma \nvdash \texttt{call } X\texttt{.go()\$this.balance} : \mathsf{cmd}(H) \tag{6.2}$$

We have that $\Gamma \vdash \texttt{this.balance} : \mathsf{var}(H)$ in contract `Y`, so in order for the method declaration `go() { X.go():this.balance }` in `Y` to be well-typed, the body of the method must be typable as $\mathsf{proc}():H$ by rule [T-DEC-M]. However, as the derivation in (6.2) illustrates, this constraint cannot be satisfied, because the lookup $\Gamma(\texttt{I<L>})(\texttt{go})$ yields $\mathsf{proc}():L$, but $\mathsf{proc}():H$ is needed, and this cannot be obtained through subtyping, because the $\mathsf{proc}(\widetilde{B}):s$ type constructor is contravariant in s.

The above example is simple, since the name `X` is 'hard-coded' directly in the body of `Y.go()`, and therefore the type check fails already while checking the contract definition. However, suppose `X` were instead received as a parameter. Then the signature of the method `Y.go` would have to be `method go :` $\mathsf{proc}(\texttt{I<H>}):H$ instead, and the type check would then fail at the call-site, if a Low address were passed. The following shows a part of the failing typing derivation for the call `Y.go(X):this.balance`, where the parameter `X` is assumed to implement the interface `I<L>` as before:

$$\dfrac{\dfrac{\Gamma(\texttt{X}) = \texttt{I<L>}}{\Gamma \vdash \texttt{X} : \texttt{I<L>}} \quad L \sqsubseteq H \quad \dfrac{\dfrac{L \sqsubseteq H}{\Gamma \vdash L <: H}}{\Gamma \vdash \mathsf{var}(L) <: \mathsf{var}(H)} \quad \dfrac{H \not\sqsubseteq L}{\Gamma \nvdash \mathsf{proc}():L <: \mathsf{proc}():H}}{\dfrac{\Gamma \nvdash \texttt{I<L>} <: \texttt{I<H>}}{\Gamma \nvdash \texttt{X} : \texttt{I<H>}}}$$
$$\Gamma \nvdash \texttt{call } Y\texttt{.go(}X\texttt{)\$this.balance} : \mathsf{cmd}(H) \tag{6.3}$$

Here $\Gamma \vdash \texttt{Y.go} : \mathsf{proc}(\texttt{I<H>}):H$ (not shown). The method call expects a parameter of type `I<H>`, but `I<L>` cannot be coerced up to `I<H>` through subtyping, because its definition of the method `go()` has type $\mathsf{proc}():L$, as given in the code listing above, and $\Gamma \nvdash \mathsf{proc}():L <: \mathsf{proc}():H$ due again to contravariance of the type constructor. Thus we see that the type system indeed prevents calls from High to Low, regardless of whether the Low address is 'hard-coded' or passed as a parameter to a High method. However, the aforementioned examples also illustrate a limitation of our type system approach to ensuring call integrity: the entire blockchain must be checked, i.e. both the contracts *and* the transactions. This is necessary since the type check can fail at the call-site of a method, as in the example shown in (6.3), and the call-site of any method can be a transaction.

Next, we shall briefly consider two examples, reported by Grishchenko et al. in [51], of Solidity contracts that are misclassified w.r.t. reentrancy by the static analyser Oyente [72]; a false positive and a false negative example.

The false negative example relies on a misplaced update of a field value, just as in the example in Figure 6.1 (page 198).[8] In this example, suppose `X` were assigned the level L and `Y` the level H. With a transaction A`->`X`.transfer(`Y`)$`n (for any address A and any amount of currency n), the type system would then correctly reject this blockchain because of the inherent flow from High to Low that is implicit in the call `X.transfer(this)` issued by `Y`. The typing derivation would fail in a similar manner as the situation depicted in (6.3).

The false positive example of Grishchenko et al. from [51] is also similar to the example in Figure 6.1, but this time just with the assignment to the guard variable correctly placed *before* the method call (i.e. with lines 6 and 7 switched in Figure 6.1). This too would be rejected by our type system, since it does not take the ordering of statements in sequential composition into account (i.e. rule [T-SEQ]). Thus, this example constitutes a false positive for our type system as well, which is hardly surprising.

Finally, let us consider a true positive example. Figure 6.17 illustrates a part of the code for two banks, which would allow users to store some of their assets and also to transfer assets between them.[9] We assume both banks implement the same interface `IBank`, but with different security settings: `X` is L and `Y` is H, meaning the latter is untrusted. There is no callback from `Y`, so in this setup a blockchain with a transaction A`->`X`.transfer(`Y`,1)$0 would actually be accepted by the type system, because the Low values from `X` can safely be coerced up (via subtyping) to match the setting of High on `Y`.

6.6 Related work

In light of the visibility and immutability of smart contracts, which makes it hard to correct errors once they are deployed in the wild, it is not surprising that there has been a substantial research effort within the formal methods community on developing formal techniques to prove safety properties of those programs—see, for instance, [124] for a survey. The literature on this topic is already huge and the whole gamut of techniques from the field of verification and validation has been adapted to the smart-contract setting. For example, this includes contributions employing frameworks based on finite-state machines to design and synthesise Ethereum smart

[8] It also involves the presence of a 'default function', which is a special feature of Solidity. It is a parameterless function that is implicitly invoked by `send()`, thus allowing the recipient to execute code upon reception of a currency transfer. This feature is not present in TinySol, yet we can achieve a similar effect by simply allowing the mandatory `send()` method to have an arbitrary method body, rather than just `skip`. This has no effect on the type system and associated proofs, since the `send()` method is treated as any other method therein. Hence, this situation is in principle the same as if the sender had invoked some other method than `send()`, similarly to the example in Figure 6.1.

[9] TinySol does not have a 'mapping' type such as in Solidity, so the setup here is limited to a single user.

```
contract X : IBank_L {
  field owner = A;

  func transfer(recipient, amount) {
    if this.sender = this.owner then
      call recipient.deposit(this.sender)$amount
    else skip
  }
  ...
}

contract Y : IBank_H {
  field credit = 0;

  func deposit(owner) {
    this.credit := this.value;
    this.owner := owner
  }
  ...
}
```

Figure 6.17: A two-bank setup.

contracts [79], a variety of static analysis techniques and accompanying tools, such as those presented in [40; 68; 114; 125], and deductive verification [28; 29; 95], amongst others. The Dafny-based approach reported in [29] is able to model arbitrary reentrancy in a setting with the 'gas mechanism', whereas [21] presents a way to analyse safety properties of smart contracts exhibiting reentrancy in a gas-free setting.

The study in [58] is close in spirit to ours in that it uses a sound type system to guarantee the absence of information flows that violate integrity policies in Solidity smart contracts. That work also presents a type verifier and its prototype implementation within the K-framework [108], which is then applied to analyse more than one hundred smart contracts. However, their technique has not been related to call integrity, which, by contrast, is the focus of our work. Thus, our contribution in the present chapter complements this work and serves to further highlight the utility and applicability of secure-flow types in the smart-contract setting. However, there are also clear differences between this aforementioned work and the present one. Most notably, our type system uses a more refined subtyping relation, which also handles subtyping of method and address types, whereas subtyping is not defined for the former in [59], and the latter is not given a type altogether. This gives us a more fine-grained control over the information flow, since it allows us to assign different security levels to a contract and its members. For example, a High contract might

have certain Low methods, which hence would not be callable from another High contract, whereas High methods would. This is in line with standard object-oriented principles, e.g. Java-style visibility modifiers.

Another approach to using a type system to ensure smart-contract safety in a Solidity-like language is presented by Crafa et al. in [33]. This work is indeed related to ours in that both are based on well-known typing principles from object-oriented languages, especially subtyping for contract/address types and the inclusion of a 'default' supertype for all contracts, similar to our I^{T}. However, the aim of [33] is rather different from ours, in that the type system offered in that paper seeks to prevent *runtime errors* that do not stem from a negative account balance, e.g. those resulting from attempts to access nonexistent members of a contract. Incidentally, such runtime errors would *also* be prevented by our type system (rules [T-CALL] and [T-FIELD] in particular), due to our use of 'interfaces' as address types, if the converse check (ensuring every declared type in an interface has an implementation) were also performed. However, our focus has been on checking the currency flow, rather than preventing runtime errors of this kind.

The aforementioned paper [33] introduced Featherweight Solidity (FS). Just like TinySol, FS is a calculus that formalises the core features of Solidity and, as mentioned above, it supports the static analysis of safety properties of smart contracts via type systems. Therefore, the developments in the present chapter might conceivably have been carried out in FS instead of TinySol. Our rationale for using TinySol is that it provided a very simple language that was sufficient to express the property of call integrity, thus allowing us to focus on the core of this property. Of course, 'simplicity' is a subjective criterion and the choice of one language instead of another is often a matter of preference and convenience. To our mind, TinySol is slightly simpler than FS, which includes functionalities such as callback functions and revert labels. Moreover, the big-step semantics of TinySol provided was more convenient for the development of our type system than the small-step semantics given for FS. Furthermore, unlike FS, TinySol also formalises the semantics of blockchains. Having said so, TinySol and FS are quite similar and it would be interesting to study their similarities in more detail. To this end, in future work, we intend to carry out a formal comparison of these two core languages and to see which adaptations to our type system are needed when formulated for FS. In particular, we note that FS handles the possibility of an explicit type conversion (type cast) of `address` to `address payable` by augmenting the `address` type with type information about the contract to which it refers. This distinction is not present in our version of TinySol, as we require all contracts and accounts to have a default `send()` function, so all addresses are in this sense 'payable'. However, our type system does not depend on the presence of a `send()` function, so this difference is not important here.

6.7 Conclusion and future work

In this chapter we studied two security properties, namely call integrity and noninterference, in the setting of TinySol, a minimal calculus for Solidity smart contracts. To this end, we rephrased the syntax of TinySol to emphasise its object-oriented flavour, gave a new big-step operational semantics for that language and used it to define call integrity and noninterference. Those two properties have some similarities in their definition, in that they both require that some part of a program is not influenced by the other part. However, we showed that the two properties are actually incomparable. Nevertheless, we provided a type system for noninterference and showed that well-typed programs *also* satisfy call integrity. Hence, programs that are accepted by our type systems lie at the intersection between call integrity and noninterference.

A challenging development of our work would be to prove whether the type system exactly characterises the intersection of these two properties, or to find another characterisation of this set of programs. Orthogonally, it would be important to devise type inference algorithms for the present type system, to be used in practical situations where the typing environment is hard to guess. It would also be interesting to compare our typing-based proof method with those proposed, e.g., in [51; 72; 114]. Finally, we also aim at applying our static analysis methodology to many concrete case studies, to better understand the benefits of using a completely static proof technique for call integrity. To do so, it would be useful to extend TinySol with a 'gas mechanism' allowing one to prove the termination of transactions and to compute their computational cost.

A potential limitation of the approach presented in this chapter is that the entire blockchain must be checked to show call integrity of a contract. Indeed, since a typing derivation can fail at the call-site and the call-site of a method can be a transaction, transactions must be well-typed too. In passing, we note that this kind of problem is also present in [118; 130] (and, in general, in many works on type systems for security), where the whole code needs to be typed in order to obtain the desired guarantees. We think that an important avenue for future work, and one we intend to pursue, is to explore whether, and to what extent, other typing disciplines can be employed to mitigate this problem. As mentioned earlier, we also plan to extend the language (and the type system) to enable checking of real-life Solidity contracts; this will also allow us to better assess how (un)feasible it would be to check the whole blockchain.

6.8 Proofs

6.8.1 Proof of Theorem 13

Proof. By induction on the derivation of $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to_S \mathsf{env}'_{SV}$.

There are four base cases (corresponding to the rules of Figure 6.5 that have no occurrence of $\to_S$ in their premise):

- If [SKIP] was used, then $S = \texttt{skip}$ and the result is immediate, since `skip` does not affect env_{SV}.
- If [WHILE$_\mathsf{F}$] was used, then $S = \texttt{while}\ e\ \texttt{do}\ S'$, with e that evaluates to F. The result is immediate, since env_{SV} is not modified in the transition.
- If [ASSV] was used, then $S = x\ \texttt{:=}\ e$ and from its premise we know that

$$\begin{aligned}\mathsf{env}_{SV} &\vdash e \to_e v\\ \mathsf{env}'_V &= \mathsf{env}_V[x \mapsto v]\,.\end{aligned}$$

  By the premise and side condition of [T-ASSV],

$$\begin{aligned}\Gamma(x) &= \mathsf{var}(B)\\ \Gamma &\vdash e : B\\ B &\rightsquigarrow s\end{aligned}$$

  As s' is strictly lower than s, we therefore conclude

$$\Gamma \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}_S, \mathsf{env}_V[x \mapsto v]\,.$$

- If [SKIP] was used, then $S = \texttt{this.}p\ \texttt{:=}\ e$. The argument is the same as above, except that the update affects env_S, rather than env_V.

For the inductive step, we distinguish the last rule used in the derivation. We have the following five cases to consider:

- If [SEQ] was used, then $S = S_1\,;S_2$ and from its premise we know that

$$\begin{aligned}\mathsf{env}_T &\vdash \langle S_1, \mathsf{env}_{SV}\rangle \to_S \mathsf{env}''_{SV}\\ \mathsf{env}_T &\vdash \langle S_2, \mathsf{env}''_{SV}\rangle \to_S \mathsf{env}'_{SV}\end{aligned}$$

  The last rule used to type S is therefore [T-SEQ], and from its premise we have that

$$\begin{aligned}\Gamma &\vdash S_1 : \mathsf{cmd}(s)\\ \Gamma &\vdash S_2 : \mathsf{cmd}(s).\end{aligned}$$

By two applications of the induction hypothesis, the statement holds for both premises. Thus, we obtain that

$$\Gamma \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}''_{SV}$$
$$\Gamma \vdash \mathsf{env}''_{SV} =_{s'} \mathsf{env}'_{SV}$$

and we conclude by transitivity.

- If [IF] was used, then $S = \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F}$ and from its premise we know that

$$\mathsf{env}_T \vdash \langle S_b, \mathsf{env}_{SV} \rangle \to_S \mathsf{env}'_{SV}$$

where $b \in \{\mathsf{T}, \mathsf{F}\}$ depending on the evaluation value of e. Then S was typed by [T-IF], and from its premise we know that

$$\Gamma \vdash S_\mathsf{T} : \mathsf{cmd}(s)$$
$$\Gamma \vdash S_\mathsf{F} : \mathsf{cmd}(s).$$

Regardless of which branch was chosen, the statement then holds by the induction hypothesis.

- If [WHILE$_\mathsf{T}$] was used, then $S = \texttt{while}\ e\ \texttt{do}\ S'$, with e that evaluates to T. From its premise, we know that

$$\mathsf{env}_T \vdash \langle S', \mathsf{env}_{SV} \rangle \to_S \mathsf{env}''_{SV}$$
$$\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S', \mathsf{env}''_{SV} \rangle \to_S \mathsf{env}'_{SV}.$$

Then S was typed by [T-WHILE], and from its premise we know that $\Gamma \vdash S' : \mathsf{cmd}(s)$. The statement then holds by two applications of the induction hypothesis.

- If [DECV] was used, then $S = \mathsf{var}(B_{s''})\ \ x\ \ \texttt{:=}\ \ e\ \ \texttt{in}\ \ S'$, where $x \notin \mathsf{dom}\left(\mathsf{env}'_V\right)$. From its premise we know that

$$\mathsf{env}_{SV} \vdash e \to_e v$$
$$\mathsf{env}_T \vdash \langle S', \mathsf{env}_S, \mathsf{env}_V, (x, v) \rangle \to_S \mathsf{env}'_S, \mathsf{env}'_V, (x, v').$$

Then S was typed by [T-DECV], and from its premise we have that

$$\Gamma \vdash e : B_{s''}$$
$$\Gamma, x : \mathsf{var}(B_{s''}) \vdash S' : \mathsf{cmd}(s).$$

We can then apply the induction hypothesis to conclude that

$$\Gamma, x : \mathsf{var}(B_{s''}) \vdash \mathsf{env}_S, \mathsf{env}_V, (x, v) =_{s'} \mathsf{env}'_S, \mathsf{env}'_V, (x, v').$$

By Lemma 49, we can then conclude $\Gamma \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}'_{SV}$, as required.

- If [CALL] was used, then $S = \texttt{call}\ e_1.f(\tilde{e})\$e_2$ and it was typed by using [T-CALL]. From the premise of [CALL], we know that

$$\begin{aligned} \mathsf{env}_V(\texttt{this}) &= X && \text{(the caller)}, \\ \mathsf{env}_{SV} \vdash e_1 &\to_e Y && \text{(the callee)}, \end{aligned}$$

and there are two writes to the `balance` fields of caller and callee (which are typed as $\mathsf{var}(\mathsf{int}, s_2)$ and $\mathsf{var}(\mathsf{int}, s_3)$, for $s \sqsubseteq s_2, s_3$), that generate the new environment env''_S (that only differs from env_S for the values assigned to the balances). Thus, $\Gamma \vdash \mathsf{env}_S =_{s'} \mathsf{env}''_S$. Moreover,

$$\begin{aligned} \mathsf{env}_{SV} &\vdash e_2 \to_e n \\ \mathsf{env}_{SV} &\vdash \tilde{e} \to_e \tilde{v} \\ \mathsf{env}_T &\vdash \langle S', \mathsf{env}''_{SV} \rangle \to_S \mathsf{env}'_{SV} \end{aligned}$$

where S' is the body of the method and

$$\mathsf{env}''_V = (\texttt{this}, Y), (\texttt{sender}, X), (\texttt{value}, n), (\tilde{x}, \tilde{v}).$$

By the premises of [T-CALL], we know that

$$\begin{aligned} \Gamma &\vdash e_1 : I_s^Y \\ \Gamma &\vdash \tilde{e} : \widetilde{B} \text{ where } B \rightsquigarrow s \\ \Gamma &\vdash e_2 : s \\ \Gamma(\texttt{this}) &= \mathsf{var}(I_{s_1}^X) \\ \Gamma(I^Y)(f) &= \mathsf{proc}(\widetilde{B}) : s \end{aligned}$$

Moreover, from [T-DEC-M] we also know that $\Gamma_1 \vdash S' : \mathsf{cmd}(s)$, where

$$\Gamma_1 = \Gamma, \texttt{this} : \mathsf{var}(I_s^Y), \texttt{sender} : \mathsf{var}(I_{s_\top}^\top), \texttt{value} : \mathsf{var}(s), \tilde{x} : \widetilde{\mathsf{var}(B)}$$

and where $\Gamma \vdash I_{s_1}^X <: I_{s_\top}^\top$. Hence, $\Gamma_1 \vdash \mathsf{env}''_V$ holds by [T-ENV-V]. By the induction hypothesis, $\Gamma_1 \vdash \mathsf{env}''_{SV} =_{s'} \mathsf{env}'_{SV}$, so $\Gamma \vdash \mathsf{env}_S =_{s'} \mathsf{env}'_S$. That trivially implies $\Gamma \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}'_S, \mathsf{env}_V$, as desired. □

6.8.2 Proof of Theorem 14

Proof. By induction on the derivation of $\Gamma \vdash e : B$. There are two base cases:

- If [T-VAL] was used to conclude $\Gamma \vdash e : B$, then $e = v$. The result is immediate, since the value does not depend on env_{SV}.
- If [T-VAR] was used, then $e = x$, and from the premise we have that $\Gamma \vdash x : \mathsf{var}(B)$, which was concluded by [T-BOX-X]. By assumption, $B \rightsquigarrow s$ and $\Gamma \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$ so $\mathsf{env}_V(x) = \mathsf{env}_V(x)$. Thus we can conclude by [EXP-VAR].

For the inductive case, we distinguish the last rule used in the derivation. We have the following three cases:

- If [T-FIELD] was used, then $e = e'.p$, and from the premise we have that $\Gamma \vdash e'.p : \mathsf{var}(B)$, which was concluded by [T-BOX-F]. From the premise and side condition of that rule, we then know that $\Gamma \vdash e' : I_s$, $\Gamma(I)(p) = \mathsf{var}(B)$ and $B \rightsquigarrow s$. By assumption, $\Gamma \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$. Therefore by the induction hypothesis

$$\mathsf{env}^1_{SV} \vdash e' \to v'$$
$$\mathsf{env}^2_{SV} \vdash e' \to v'$$

  where v' is an address X. Then $\mathsf{env}^1_S(X)(p) = \mathsf{env}^2_S(X)(p)$ Thus we can conclude by [EXP-FIELD].

- If [T-OP] was used, then $e = \mathsf{op}(e_1, \ldots, e_n)$. From the premise, we know that each of the arguments e_i are typable as $\Gamma \vdash e_i : B$ where $B \rightsquigarrow s$ so by n applications of the induction hypothesis, we get that $\mathsf{env}^1_{SV} \vdash e_i \to v_i$ and $\mathsf{env}^2_{SV} \vdash e_i \to v_i$, so we can conclude by rule [EXP-OP].
- If [T-SUBS-EXP] was used, then we know from the premise that $\Gamma \vdash e : B'$ and $\Gamma \vdash B' <: B$. By the subtyping rules [T-SUBS-SEC] resp. [T-SUBS-NAME], we know that $s' \sqsubseteq s$. Hence, by Lemma 48, $\Gamma \vdash \mathsf{env}^1_{SV} =_{s'} \mathsf{env}^2_{SV}$ also holds. Then by the induction hypothesis, the statement holds for $\Gamma \vdash e : B'$ where $B' \rightsquigarrow s'$. □

6.8.3 Proof of Theorem 15

Proof. There are two cases to consider: In the first case, assume $s_1 \not\sqsubseteq s_2$ (i.e. s_2 is either strictly below s_1, or they are incomparable). By Theorem 13 we can then conclude

the following:

$$\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^{1'}_{SV}$$
$$\Gamma \vdash \mathsf{env}^2_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$$

and from $\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$ we can then conclude $\Gamma \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$.

For the other case, where $s_1 \sqsubseteq s_2$, we proceed by induction on the sum of the lengths of the derivations of the transitions

$$\mathsf{env}_T \vdash \langle S, \mathsf{env}^1_{SV} \rangle \to_S \mathsf{env}^{1'}_{SV}$$
$$\mathsf{env}_T \vdash \langle S, \mathsf{env}^2_{SV} \rangle \to_S \mathsf{env}^{2'}_{SV}.$$

Initially, we make the following observation: The choice of the last rule in both inferences is syntax-driven (according to the outermost operator in S), and therefore both inferences terminate with the same rule from Figures 6.5–6.6. The only case when the syntax does not identify the rule to use is when S is a `while`. Thus, we shall preliminarily prove that also in this case no uncertainty is possible:

> Let
>
> $$\mathsf{env}^1_{SV} \vdash e \to_e b_1$$
> $$\mathsf{env}^2_{SV} \vdash e \to_e b_2$$
>
> where $b_1, b_2 \in \{\mathsf{T}, \mathsf{F}\}$. Since we are assuming that $s_1 \sqsubseteq s_2$ and by hypothesis $\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, we can conclude that $b_1 = b_2$ by Theorem 14. Hence, both inferences terminate either with [WHILE$_\mathsf{F}$] or [WHILE$_\mathsf{T}$], according to the evaluation of the guard.

We now continue with the induction. There are four base cases (corresponding to the rules of Figure 6.5 that have no occurrence of $\to_S$ in their premise):

- If [SKIP] was used to infer both transitions, then S = `skip` and the result is immediate, since `skip` does not affect env^1_{SV} and env^2_{SV}.
- If [WHILE$_\mathsf{F}$] is used to conclude both transitions, then S = `while` e `do` S' and the result is immediate, since

$$\mathsf{env}^{1'}_{SV} = \mathsf{env}^1_{SV}$$
$$\mathsf{env}^{2'}_{SV} = \mathsf{env}^2_{SV}.$$

- If [ASSV] was used to infer both transitions, then $S = x := e$. From the premises of that rule, we know that

$$\begin{aligned}\mathsf{env}^1_{SV} &\vdash e \to_e v_1\\ \mathsf{env}^2_{SV} &\vdash e \to_e v_2.\end{aligned}$$

  Thus

$$\begin{aligned}\mathsf{env}^{1'}_{SV} &= \mathsf{env}^1_S, \mathsf{env}^1_V[x \mapsto v_1]\\ \mathsf{env}^{2'}_{SV} &= \mathsf{env}^2_S, \mathsf{env}^2_V[x \mapsto v_2]\,.\end{aligned}$$

  By the premise of [T-ASSV], $\Gamma(x) = \mathsf{var}(B)$ and $\Gamma \vdash e : B$ where $B \rightsquigarrow s_1$. Since by assumption $s_1 \sqsubseteq s_2$ and $\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, we can conclude that $v_1 = v_2 = v$ by Theorem 14. Thus,

$$\Gamma \vdash \mathsf{env}^1_S, \mathsf{env}^1_V[x \mapsto v] =_{s_2} \mathsf{env}^2_S, \mathsf{env}^2_V[x \mapsto v]$$

  also holds.

- If [ASSF] was used, then $S = \texttt{this}.p := e$. The argument is then the same as above, except that it is env_S, rather than env_V, that is updated.

For the inductive step, we distinguish the last rule used in both derivations (as we argued above, the two transitions have been inferred by using the same last rule). We have the following five cases to consider:

- If [SEQ] was used to conclude both transitions, then $S = S_1 ; S_2$. From the premises of that rule, we know that

$$\begin{aligned}\mathsf{env}_T &\vdash \langle S_1, \mathsf{env}^1_{SV}\rangle \to_S \mathsf{env}^{1''}_{SV}\\ \mathsf{env}_T &\vdash \langle S_2, \mathsf{env}^{1''}_{SV}\rangle \to_S \mathsf{env}^{1'}_{SV}\\ \mathsf{env}_T &\vdash \langle S_1, \mathsf{env}^2_{SV}\rangle \to_S \mathsf{env}^{2''}_{SV}\\ \mathsf{env}_T &\vdash \langle S_2, \mathsf{env}^{2''}_{SV}\rangle \to_S \mathsf{env}^{2'}_{SV}\end{aligned}$$

  Moreover, from the premises of [T-SEQ], we know that

$$\begin{aligned}\Gamma &\vdash S_1 : \mathsf{cmd}(s_1)\\ \Gamma &\vdash S_2 : \mathsf{cmd}(s_1).\end{aligned}$$

  By two applications of the induction hypothesis, we conclude that

$$\begin{aligned}\Gamma &\vdash \mathsf{env}^{1''}_{SV} =_{s_2} \mathsf{env}^{2''}_{SV}\\ \Gamma &\vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}\end{aligned}$$

  as required.

- If [IF] was the last rule used in both inferences, then $S = \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F}$. Like in the case of the `while` discussed at the beginning of this proof, we have that $b_1 = b_2 = b \in \{\mathsf{T}, \mathsf{F}\}$, where

$$\begin{aligned}\mathsf{env}^1_{SV} &\vdash e \to_e b_1\\ \mathsf{env}^2_{SV} &\vdash e \to_e b_2.\end{aligned}$$

  Thus, the same branch S_b is chosen in both inferences, that is

$$\begin{aligned}\mathsf{env}_T &\vdash \langle S_b, \mathsf{env}^1_{SV}\rangle \to_S \mathsf{env}^{1'}_{SV}\\ \mathsf{env}_T &\vdash \langle S_b, \mathsf{env}^2_{SV}\rangle \to_S \mathsf{env}^{2'}_{SV}.\end{aligned}$$

  Regardless of the value of b, from the premise of [T-IF] we know that $\Gamma \vdash e : s_1$ and $\Gamma \vdash S_b : \mathsf{cmd}(s_1)$. The statement then holds by the induction hypothesis.

- If [WHILE_T] was the last rule used, then $S = \texttt{while}\ e\ \texttt{do}\ S'$ and from the premises of that rule we know that

$$\begin{aligned}\mathsf{env}_T &\vdash \langle S', \mathsf{env}^1_{SV}\rangle \to_S \mathsf{env}^{1''}_{SV}\\ \mathsf{env}_T &\vdash \left\langle \texttt{while}\ e\ \texttt{do}\ S', \mathsf{env}^{1''}_{SV}\right\rangle \to_S \mathsf{env}^{1'}_{SV}\\ \mathsf{env}_T &\vdash \langle S', \mathsf{env}^2_{SV}\rangle \to_S \mathsf{env}^{2''}_{SV}\\ \mathsf{env}_T &\vdash \left\langle \texttt{while}\ e\ \texttt{do}\ S', \mathsf{env}^{2''}_{SV}\right\rangle \to_S \mathsf{env}^{2'}_{SV}\end{aligned}$$

  Moreover, by the premise of [T-WHILE], we know that

$$\begin{aligned}\Gamma &\vdash e : s_1\\ \Gamma &\vdash S' : \mathsf{cmd}(s_1).\end{aligned}$$

  The statement then holds by two applications of the induction hypothesis, like in the case for [SEQ] above.

- If [DECV] was the last rule used, then $S = \mathsf{var}(B)\ x\ \texttt{:=}\ e\ \texttt{in}\ S'$. From the premises of that rule, we have that

$$\begin{aligned}\mathsf{env}^1_{SV} &\vdash e \to_e v_1\\ \mathsf{env}_T &\vdash \langle S', \mathsf{env}^1_S, \mathsf{env}^1_V, (x, v_1)\rangle \to_S \mathsf{env}^{1'}_S, \mathsf{env}^{1'}_V, (x, v'_1)\\ \mathsf{env}^2_{SV} &\vdash e \to_e v_2\\ \mathsf{env}_T &\vdash \langle S', \mathsf{env}^2_S, \mathsf{env}^2_V, (x, v_2)\rangle \to_S \mathsf{env}^{2'}_S, \mathsf{env}^{2'}_V, (x, v'_2)\end{aligned}$$

where $x \notin \mathrm{dom}\left(\mathsf{env}_V^1\right) \cup \mathrm{dom}\left(\mathsf{env}_V^2\right)$. From the premise of [T-DECV], we know that

$$\Gamma \vdash e : B \text{ where } B \rightsquigarrow s$$
$$\Gamma, x : \mathsf{var}(B) \vdash S' : \mathsf{cmd}(s_1).$$

In order to apply the induction hypothesis, we must show that

$$\Gamma, x : \mathsf{var}(B) \vdash \mathsf{env}_S^1, \mathsf{env}_V^1, (x, v_1) =_{s_2} \mathsf{env}_S^2, \mathsf{env}_V^2, (x, v_2) \tag{6.4}$$

There are now two cases:

- If $s \sqsubseteq s_2$, then (6.4) holds by Theorem 14: indeed, $v_1 = v_2 = v$, since all variables read within e are of a lower level than s_2, and by assumption, env_{SV}^1 and env_{SV}^2 agree on all values up to, and including, s_2.
- If $s \not\sqsubseteq s_2$, then s is either strictly higher than s_2, or they are incomparable. Hence, (6.4) holds by definition of the $\Gamma \vdash \cdot =_s \cdot$ relation.

Thus, by the induction hypothesis we have that

$$\Gamma, x : \mathsf{var}(B) \vdash \mathsf{env}_S^{1'}, \mathsf{env}_V^{1'}, (x, v_1') =_{s_2} \mathsf{env}_S^{2'}, \mathsf{env}_V^{2'}, (x, v_2')$$

and we can therefore conclude $\Gamma \vdash \mathsf{env}_{SV}^{1'} =_{s_2} \mathsf{env}_{SV}^{2'}$ by Lemma 49, as required.

- If [CALL] was the last rule, then $S = \texttt{call}\ e_1.f(\tilde{e})\$e_2$, and S was typed by using [T-CALL].

  Firstly, from the premises of [CALL] we know that

$$\mathsf{env}_V^1(\texttt{this}) = X_1$$
$$\mathsf{env}_V^2(\texttt{this}) = X_2$$

  and from the premise of [T-CALL], we have that $\Gamma(\texttt{this}) = \mathsf{var}(I_{s_1'}^X)$ with $s_1' \sqsubseteq s_1$. Since $\Gamma \vdash \mathsf{env}_V^1 =_{s_2} \mathsf{env}_V^2$ and $s_1 \sqsubseteq s_2$ by assumption, we can conclude that $X_1 = X_2 = X$, by [EQ-ENV$_V$].

  Secondly, from the premises of [CALL] and [T-CALL], we have that

$$\mathsf{env}_{SV}^1 \vdash e_1 \to_e Y_1$$
$$\mathsf{env}_{SV}^2 \vdash e_1 \to_e Y_2$$
$$\Gamma \vdash e_1 : I_{s_1}^Y$$

$$\mathsf{env}_{SV}^1 \vdash e_2 \to_e n_1$$
$$\mathsf{env}_{SV}^2 \vdash e_2 \to_e n_2$$
$$\Gamma \vdash e_2 : s_1$$

By Theorem 14, we conclude that $Y_1 = Y_2 = Y$ and $n_1 = n_2 = n$, since we are assuming that $s_1 \sqsubseteq s_2$.

Thirdly, for the list of actual parameters, fix an arbitrary expression $e_i \in \tilde{e}$. By the premises of [CALL] and [T-CALL], we have that

$$\begin{aligned} \mathsf{env}^1_{SV} &\vdash e_i \to_e v_i^1 \\ \mathsf{env}^2_{SV} &\vdash e_i \to_e v_i^2 \\ \Gamma &\vdash e_i : B_i \text{ where } B_i \rightsquigarrow s_i. \end{aligned}$$

The values v_i^1, resp. v_i^2, will be bound to the variable x_i within the newly created variable environments $\mathsf{env}^{1\prime\prime}_V$, resp. $\mathsf{env}^{2\prime\prime}_V$, which are used in the two executions of the method body. For each value, there are two cases:

- If $s_i \sqsubseteq s_2$, then by Theorem 14 we have that $v_i^1 = v_i^2$.
- Otherwise, it may be the case that $v_i^1 \neq v_i^2$, but since $s_i \not\sqsubseteq s_2$, then the variable bindings (x_i, v_i^1), resp. (x_i, v_i^2), will still satisfy the condition that the newly created variable environments $\mathsf{env}^{1\prime\prime}_V$, resp. $\mathsf{env}^{2\prime\prime}_V$, will agree up to level s_2.

As we concluded above, the caller X is the same in both transitions, and likewise, the callee Y is the same, as is the value n. Thus by [EQ-ENV$_V$] it easily follows that

$$\Gamma' \vdash \mathsf{env}^{1\prime\prime}_V =_{s_2} \mathsf{env}^{2\prime\prime}_V \tag{6.5}$$

where

$$\begin{aligned} \mathsf{env}^{1\prime\prime}_V &= (\mathtt{this}, Y), (\mathtt{sender}, X), (\mathtt{value}, n), (x_1, v_1^1), \ldots, (x_k, v_k^1) \\ \mathsf{env}^{2\prime\prime}_V &= (\mathtt{this}, Y), (\mathtt{sender}, X), (\mathtt{value}, n), (x_1, v_1^2), \ldots, (x_k, v_k^2) \end{aligned}$$

and

$$\Gamma' = \Gamma, \mathtt{this} : \mathsf{var}(I^Y_{s_1}), \mathtt{sender} : \mathsf{var}(I^{\mathsf{T}}_{s_\mathsf{T}}), \mathtt{value} : \mathsf{var}(s_1), \tilde{x} : \widetilde{\mathsf{var}(B_s)}$$

and $\Gamma \vdash I^X_{s_1'} <: I^{\mathsf{T}}_{s_\mathsf{T}}$.

Regarding the two state environments $\mathsf{env}^{1\prime\prime}_S$ and $\mathsf{env}^{2\prime\prime}_S$, recall again that the caller X, resp. the callee Y, are the same in both transitions. By [EQ-ENV$_F$] we then have that

$$\begin{aligned} \Gamma &\vdash_{I^X} \mathsf{env}^{1X}_F =_{s_2} \mathsf{env}^{2X}_F \\ \Gamma &\vdash_{I^Y} \mathsf{env}^{1Y}_F =_{s_2} \mathsf{env}^{2Y}_F \end{aligned}$$

where $\mathsf{env}_F^{iZ} = \mathsf{env}_S^i(Z)$ for $i \in \{1,2\}$ and $Z \in \{X, Y\}$. Since [T-CALL] entails that

$$\Gamma(I^X)(\texttt{balance}) = \mathsf{var}(s_2')$$
$$\Gamma(I^Y)(\texttt{balance}) = \mathsf{var}(s_3')$$

with $s_1 \sqsubseteq s_2', s_3'$, we have that

$$\Gamma \vdash \mathsf{env}_S^{1''} =_{s_2} \mathsf{env}_S^{2''} \tag{6.6}$$

where

$$\mathsf{env}_S^{1''} = \mathsf{env}_S^1[X \mapsto \mathsf{env}_F^{1X}[\texttt{balance -= } n]][Y \mapsto \mathsf{env}_F^{1Y}[\texttt{balance += } n]]$$
$$\mathsf{env}_S^{2''} = \mathsf{env}_S^2[X \mapsto \mathsf{env}_F^{2X}[\texttt{balance -= } n]][Y \mapsto \mathsf{env}_F^{2Y}[\texttt{balance += } n]]$$

since both `balance` fields are modified by the same value n in both executions. Finally, since the callee Y is the same in both executions, we have that

$$\mathsf{env}_T(Y)(f) = (\tilde{x}, S')$$

in both executions; i.e. the method body S' is the same, and from the premise of [T-DEC-M], we have that $\Gamma' \vdash S' : \mathsf{cmd}(s_1)$. Moreover, from the premises of [CALL] we then have that

$$\begin{aligned} \mathsf{env}_T &\vdash \langle S', \mathsf{env}_{SV}^{1''} \rangle \to_S \mathsf{env}_{SV}^{1'} \\ \mathsf{env}_T &\vdash \langle S', \mathsf{env}_{SV}^{2''} \rangle \to_S \mathsf{env}_{SV}^{2'} \end{aligned} \tag{6.7}$$

We can then conclude the following:

$\Gamma' \vdash \mathsf{env}_{SV}^{1''} =_{s_2} \mathsf{env}_{SV}^{2''}$ by (6.6) and (6.5)

$\Gamma' \vdash \mathsf{env}_{SV}^{1'} =_{s_2} \mathsf{env}_{SV}^{2'}$ by and (6.7) and the induction hypothesis

By assumption $\Gamma \vdash \mathsf{env}_V^1 =_{s_2} \mathsf{env}_V^2$, so this finally allows us to conclude that

$$\Gamma \vdash \mathsf{env}_S^{1'}, \mathsf{env}_V^1 =_{s_2} \mathsf{env}_S^{2'}, \mathsf{env}_V^2$$

as required. □

7 Preventing Out-of-Gas Exceptions by Typing[1]

7.1 Introduction

Smart contracts running on a blockchain can be viewed as a collective system: they are typically deployed in large numbers and interact in an adversarial environment, each pursuing its own individual goal, without specific centralised control. Many languages have been deployed to program them, with the Ethereum Virtual Machine language [132] (EVM) as one of the most successful proposals. It is a Turing-complete language and, thus, it is in principle undecidable whether a given EVM smart contract will halt or not. This is problematic in a blockchain setting, since smart contracts may be called in transactions, which are executed by miners as part of their attempts to find the next block, as well as when validating existing blocks. If a transaction were to run forever, it would therefore in effect block the miner from making progress, which for instance could be used to cause a denial of service for the entire network.

To limit such undesirable scenarios, EVM introduced the concept of *gas*, which effectively ensures termination of all transactions. This mechanism works as follows (see [51] for details): The gas represents a unit of computation, and thus also a unit of work for the miner. Each EVM instruction has an associated *gas cost*,[2] and the total gas cost of a transaction is thus proportional to the amount of computational work required by the miner to execute the transaction. Whenever a user schedules a transaction, two additional parameters have to be specified: the *gas limit* and the *gas price*. The gas limit is an upper bound on the gas cost of the transaction. If this limit is exceeded, an out-of-gas exception is thrown, the transaction aborts and its effects are rolled back. The gas price is the amount of currency that the user is willing to pay to the miner for each unit of gas consumed in the transaction. This ensures that users will not schedule long-running transactions and just pick an arbitrarily high gas limit,

[1] The present chapter is joint work with Luca Aceto, Daniele Gorla and Mohammad Hamdaqa. It was presented at the REoCAS colloquium in 2024 and published in the proceedings, [6]. A version with an appendix containing the proofs is available online in [5]. The present chapter is equivalent to the online version with all proofs included.

[2] Usually 3–5 units, although some complex operations such as `CREATE` have a much higher cost. An overview of the EVM instructions and their gas costs can be found in [119].

since the gas price will still be paid for each unit of gas consumed in a transaction, even if the transaction aborts with an out-of-gas exception.

The concept of gas is, of course, also inherent in Solidity, since this language compiles to EVM. However, the original presentation of TinySol [12], a minimal object-oriented language based on Solidity, does not include such a mechanism. This limits its ability to realistically model, e.g., aborting transactions and rollback. In particular, the latter phenomenon is modelled in the operational semantics of that language through a rule with an undecidable premise. In our previous work on TinySol [4] (see Chapter 6), we simply decided to omit this rule, thus in effect allowing non-terminating transactions and also leaving transactions stuck, if an exception-raising command were ever to be reached.

In the present chapter, we therefore add a gas mechanism to our version of TinySol and present a first attempt at a type system ensuring that transactions never run of out gas at runtime. The syntax of the language is provided in Section 7.2 and is essentially the one we used in [4], where we gave a big-step operational semantics for TinySol. However, handling exceptions in a big-step semantics is rather unnatural, as exceptions can be raised at any intermediate point of a computation. For this reason, a first contribution of this work is to provide a small-step operational semantics for TinySol; this is done in Section 7.3.

We then move to the core contribution of this chapter: a type system for guaranteeing that a transaction never runs out-of-gas, that is, an out-of-gas exception is never raised at runtime. The type system is defined in Section 7.4.1 and its properties are stated in Section 7.4.2. Never running out-of-gas is essentially a problem of checking termination of a program, a well-known undecidable problem in its general formulation. To make it decidable, and efficiently checkable, we have to reduce the set of programs that the type system accepts; this is discussed thoroughly in Section 7.4.3. We conclude the chapter in Section 7.5, where we also hint at possible directions for future enhancements of this work. Longer proofs are deferred to the end of the chapter; they can be found in Section 7.6.

7.1.1 Related work

The literature on methods for proving that programs terminate is vast and we cannot do justice to related work fully in this chapter. We therefore limit ourselves to mentioning the survey of program termination proof techniques for sequential programs given in [32] and the textbook [91], which offers an introduction to Hoare-style methods to prove total correctness and to give bounds on the execution time of imperative programs [90].

In the setting of smart-contract languages, as observed in [45], proofs of termination are non-trivial even in the presence of a gas mechanism. That article presents the first mechanised proof of termination of contracts written in EVM bytecode using minimal assumptions on the gas cost of operations. Efficient static-analysis techniques

$$
\begin{array}{rcl}
DF \in \mathsf{Dec}_F & ::= & \epsilon \mid \texttt{field p := } v\texttt{;}DF \\
DM \in \mathsf{Dec}_M & ::= & \epsilon \mid \texttt{func } f(\tilde{x})\ \texttt{\{ } S\ \texttt{\}}\ DM \\
DC \in \mathsf{Dec}_C & ::= & \epsilon \mid \texttt{contract } X\ \texttt{\{} \\
 & & \quad \texttt{field balance := } n\texttt{; } DF \\
 & & \quad \texttt{func send() \{ skip \}}\ DM \\
 & & \texttt{\}}\ DC \\
m \in \mathsf{MVar} & ::= & \texttt{this} \mid \texttt{sender} \mid \texttt{value} \\
L \in \mathsf{LVal} & ::= & x \mid \texttt{this.}p \\
e \in \mathsf{Exp} & ::= & v \mid x \mid m \mid e\texttt{.balance} \mid e\texttt{.}p \mid \mathrm{op}(\tilde{e}) \\
S \in \mathsf{Stm} & ::= & \texttt{skip} \mid \texttt{throw} \mid \texttt{var } x \texttt{ := } e \texttt{ in } S \mid L \texttt{ := } e \\
 & & \mid S_1\texttt{;}S_2 \mid \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} \mid \texttt{while } e \texttt{ do } S \\
 & & \mid \texttt{call } e_1\texttt{.}f(\tilde{e})\texttt{\$}e_2 \\
v \in \mathsf{Val} & ::= & \mathbb{Z} \cup \mathbb{B} \cup \mathsf{ANames}
\end{array}
$$

where $x, y \in \mathsf{VNames}$ (variable names), $p, q \in \mathsf{FNames}$ (field names), $X, Y \in \mathsf{ANames}$ (address names), $f, g \in \mathsf{MNames}$ (method names).

Figure 7.1: The syntax of TinySol.

for detecting gas-focused vulnerabilities in smart contracts and their implementation in the tool MadMax have been presented in [50].

Among the plethora of studies on type systems for ensuring program termination in several settings, we mention the work by Boudol on typing termination in a higher-order concurrent imperative language; see [20] and the references therein. Our contribution is inspired by the work by Deng and Sangiorgi in [37] for the π-calculus. In particular, we build on their type system 1, which imposes limitations similar to those for our type system, e.g. it prohibits (the π-calculus equivalent of) recursive function calls.

7.2 TinySol with gas and exceptions

The syntax of TinySol is given in Figure 7.1, where we use the notation $\tilde{\cdot}$ to denote (possibly empty) sequences of items. The set of *values*, ranged over by v, is formed by the sets of integers $\mathbb{Z}$, ranged over by n, booleans $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$, ranged over by b, and address names ANames, ranged over by X, Y.

We introduce explicit declarations for fields DF, methods DM, and contracts DC. The latter also encompasses declarations of accounts: an *account* is a contract that contains only the declarations of a special field `balance` and of a single special method `send()`, which does nothing and is used only for transferring funds to the account. By contrast, a contract usually contains other declarations of fields and

methods. For the sake of simplicity, we make no syntactic distinction between an account and a contract but, to distinguish them, we can assume that the set ANames is split into contract addresses and account addresses.

We have four 'magic' keywords in our syntax. The keyword `balance` (type `int`) stands for a special field recording the current balance of the contract (or account). It can be read from, but not directly assigned to, except through method calls. This ensures that the total amount of currency 'on-chain' remains constant during execution. The keyword `value` (type `int`) denotes a special variable that is bound to the currency amount transferred with a method call. The special variable `sender` (type `address`) is always bound to the address of the caller of a method. Finally, `this` (type `address`) is a special variable that is always bound to the address of the contract containing the currently executing method. The last three of these are local variables, and we collectively refer to them as 'magic variables' $m \in \mathsf{MVar}$.

The declaration of variables and fields (DV and DF) are very alike: the main difference is that variable bindings will be created at runtime (and with scoped visibility), hence we can let the initial assignment be an expression e; by contrast, the initial assignment to fields must be a value v.

The core part of the language is the declaration of expressions e and statements S, which are almost the same as in [12]. The main differences are:

1. We introduce fields p in expressions, instead of keys.
2. We explicitly distinguish between (global) fields and (local) variables, where the latter are declared with a scope limited to a statement S.
3. We introduce explicit *lvalues* L, to restrict what can appear on the left-hand side of an assignment (which, in particular, ensures that the special field `balance` can never be assigned to directly).
4. We have a `for` loop, instead of the more general (and less controllable) `while` loop. This last difference is only introduced for the sake of the type system we are going to present and, indeed, in [4] the `while` loop is present.

We now use TinySol to describe transactions and blockchains. A *transaction* is simply a call, where the caller is an account A, rather than a contract, together with some gas quantity, used to specify how much computational cost the transaction can consume. We denote this by writing $A\texttt{->}X\,.\,f\,(\tilde{v})\,\$(n, g)$, which expresses that the account A calls the method f on the contract (residing at address) X, with actual parameters $\tilde{v}$, transferring the amount of currency n with the call, and with an upper bound on the computational cost of the call given by the gas amount g (a positive integer). Hence, blockchains can be defined as follows:

Definition 48 (Syntax of blockchains)**.** A *blockchain* $B \in \mathcal{B}$ is a list of initial contract declarations DC, followed by a sequence of transactions $T \in \mathsf{Tr}$:

$$B \in \mathcal{B} ::= DC \;\; T$$
$$T \in \mathsf{Tr} ::= \epsilon \mid A\texttt{->}X\,.\,f\,(\tilde{v})\$(n, g),\; T$$

where $g > 0$ is the *gas limit* of the transaction. Notationally, a blockchain with an empty DC will be simply written as the sequence of transactions. ■

7.3 A small-step semantics with exceptions and gas

The semantics given in [4] describes the execution of a complex statement as one single computational step. This is sensible from the point of view of a user, since a transaction is a method call and transactions are atomic. However, we can also create a small-step semantics for statements in TinySol to describe the individual computational steps involved; this form of semantics is more suitable than a big-step one when exception handling is introduced.

To define the semantics, we need some environments to record the bindings of variables (including the three magic variable names `this`, `sender` and `value`), fields, methods, and contracts. We define them as sets of partial functions:

Definition 49 (Binding model)**.** The binding model consists of the following:

$$\begin{aligned}
\mathsf{env}_V \in \mathsf{Env}_V &: \mathsf{VNames} \cup \mathsf{MVar} \rightharpoonup \mathsf{Val} \\
\mathsf{env}_S \in \mathsf{Env}_S &: \mathsf{ANames} \rightharpoonup \mathsf{Env}_F \\
\mathsf{env}_F \in \mathsf{Env}_F &: \mathsf{FNames} \cup \{\texttt{balance}\} \rightharpoonup \mathsf{Val} \\
\mathsf{env}_T \in \mathsf{Env}_T &: \mathsf{ANames} \rightharpoonup \mathsf{Env}_M \\
\mathsf{env}_M \in \mathsf{Env}_M &: \mathsf{MNames} \rightharpoonup \mathsf{VNames}^* \times \mathsf{Stm}
\end{aligned}$$

We write each environment env_X, for $X \in \{V, F, M, S, T\}$, as an unordered sequence of pairs (d, c) where $d \in \mathrm{dom}(\mathsf{env}_X)$ and $c \in \mathrm{codom}(\mathsf{env}_X)$. The notation $\mathsf{env}_X[d \mapsto c]$ denotes the update of env_X mapping d to c. We write $\mathsf{env}_X^{\emptyset}$ for the empty X-environment. ■

To simplify the notation, when two or more environments appear together, we shall use the convention of writing the subscripts together; e.g., we write env_{MF} instead of $\mathsf{env}_M, \mathsf{env}_F$, and env'_{SV} instead of $\mathsf{env}'_S, \mathsf{env}'_V$.

The semantics of expressions is given in the following Figure 7.2, where we assume an evaluation function $\mathrm{op}(\tilde{v}) \rightarrow_{\mathrm{op}} v$ for every basic operation op of the language.

The semantics of declarations turns a field/method/contract declaration into an environment representation, to be used in the semantics of transactions. The rules are given in the following Figure 7.3. Statements are executed in the context of the following environments: a *method table* env_T, which maps addresses to method environments, and a *state* env_{SV}, which maps addresses to lists of fields and their values, and local variables to their values. Thus, for each contract, we have the list of

$$[\textsc{exp-val}]\frac{}{\mathsf{env}_{SV} \vdash v \to_e v}$$

$$[\textsc{exp-var}]\frac{x \in \mathrm{dom}\,(\mathsf{env}_V)}{\mathsf{env}_{SV} \vdash x \to_e v}\ (\mathsf{env}_V(x) = v)$$

$$[\textsc{exp-field}]\frac{\mathsf{env}_{SV} \vdash e \to_e X}{\mathsf{env}_{SV} \vdash e.p \to_e v}\ \begin{pmatrix} p \in \mathrm{dom}\,(\mathsf{env}_S(X)) \\ \mathsf{env}_S(X)(p) = v \end{pmatrix}$$

$$[\textsc{exp-op}]\frac{\mathsf{env}_{SV} \vdash \tilde{e} \to_e \tilde{v} \quad \mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v}{\mathsf{env}_{SV} \vdash \mathrm{op}(\tilde{e}) \to_e v}$$

Figure 7.2: Semantics of expressions.

$$[\textsc{dec-f}_1]\frac{}{\langle \epsilon, \mathsf{env}_F \rangle \to_{DF} \mathsf{env}_F}$$

$$[\textsc{dec-f}_2]\frac{\langle DF, \mathsf{env}_F \rangle \to_{DF} \mathsf{env}'_F}{\langle \texttt{field } p \texttt{ := } v\texttt{;}DF, \mathsf{env}_F \rangle \to_{DF} (p, v), \mathsf{env}'_F}$$

$$[\textsc{dec-m}_1]\frac{}{\langle \epsilon, \mathsf{env}_M \rangle \to_{DM} \mathsf{env}_M}$$

$$[\textsc{dec-m}_2]\frac{\langle DM, \mathsf{env}_M \rangle \to_{DM} \mathsf{env}'_M}{\langle \texttt{func } f\texttt{(}\tilde{x}\texttt{) \{ } S \texttt{ \} } DM, \mathsf{env}_M \rangle \to_{DM} (f, (\tilde{x}, S)), \mathsf{env}'_M}$$

$$[\textsc{dec-c}_1]\frac{}{\langle \epsilon, \mathsf{env}_{ST} \rangle \to_{DC} \mathsf{env}_{ST}}$$

$$[\textsc{dec-c}_2]\frac{\begin{array}{c}\left\langle DF, \mathsf{env}_F^{\emptyset} \right\rangle \to_{DF} \mathsf{env}_F \\ \left\langle DM, \mathsf{env}_M^{\emptyset} \right\rangle \to_{DM} \mathsf{env}_M \\ \langle DC, \mathsf{env}_{ST} \rangle \to_{DC} \mathsf{env}'_{ST}\end{array}}{\begin{array}{r}\langle \texttt{contract } X \texttt{ \{ } DF\ DM \texttt{ \} } DC, \mathsf{env}_{ST} \rangle \\ \to_{DC} ((X, \mathsf{env}_F), \mathsf{env}'_S), ((X, \mathsf{env}_M), \mathsf{env}'_T)\end{array}}$$

Figure 7.3: Semantics of declarations.

methods it declares and its current state. Of course, the method table is constant, once all declarations are performed, whereas the state will change during the evaluation of a program.

The semantics relies on the concept of a *stack* Q that holds the code that has not yet been executed and the variable environments needed to switch execution frames during a call:

Definition 50 (Syntax of stacks)**.** The syntax of *stacks* Q is given as follows:

$$Q ::= \bot \mid S :: Q \mid \mathsf{env}_V :: Q \mid \mathsf{del}(x) :: Q \mid \mathsf{exc}(l) :: Q$$
$$l ::= \texttt{rte} \mid \texttt{neg} \mid \texttt{oog} \mid \texttt{pge}$$ ■

In the above definition, $\bot$ denotes the bottom of the stack, S is a statement, env_V is a variable environment, $\mathsf{del}(x)$ marks the end of the scope of a locally declared variable x, and $\mathsf{exc}(l)$ denotes raising of an exception of kind l, whose possibilities are as follows:

- `rte` denotes an exception raised at runtime because of ill-formed code (e.g., when attempting to access a field not present in a contract or when invoking a method not provided by a contract);
- `neg` denotes an exception raised when a contract's balance becomes negative;
- `oog` denotes an exception raised when a transaction consumes all its gas; and
- `pge` denotes an exception raised by a `throw` command.

In the small-step semantics, we always execute the element at the top of the stack. Transitions have the form $\mathsf{env}_T \vdash \langle Q, \mathsf{env}_{SV}, g\rangle \to \langle Q', \mathsf{env}'_{SV}, g'\rangle$, where $g \geq g' \geq 0$, and are defined by the rules in Figure 7.4 and Figure 7.5. To keep the language and the semantics easy, we make a few simplifying assumptions:

1. We assume a unitary gas cost for all TinySol commands, rather than having different prices for different instructions.
2. We only let commands consume gas; this is in contrast to EVM, which also has gas costs for expression operations such as `ADD`, `MUL` etc.
3. We omit the *gas price*, since the TinySol model of blockchains does not include the concept of miners and blocks. Instead, we shall simply say that the actual amount of gas consumed in a transaction is subtracted from the user's account `balance`.

$$[\textsc{skip}]\frac{}{\mathsf{env}_T \vdash \langle \texttt{skip} :: Q, \mathsf{env}_{SV}, g\rangle \to \langle Q, \mathsf{env}_{SV}, g-1\rangle}\ (g \geq 1)$$

$$[\textsc{seq}]\frac{}{\mathsf{env}_T \vdash \langle S_1\,; S_2 :: Q, \mathsf{env}_{SV}, g\rangle \to \langle S_1 :: S_2 :: Q, \mathsf{env}_{SV}, g\rangle}\ (g \geq 1)$$

$$[\textsc{if}]\frac{\mathsf{env}_{SV} \vdash e \to_e b \in \mathbb{B}}{\begin{array}{l}\mathsf{env}_T \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F} :: Q, \mathsf{env}_{SV}, g\rangle \\ \qquad \to \langle S_b :: Q, \mathsf{env}_{SV}, g-1\rangle\end{array}}\ (g \geq 1)$$

$$[\textsc{for}_\mathsf{T}]\frac{\mathsf{env}_{SV} \vdash e \to v}{\begin{array}{l}\mathsf{env}_T \vdash \langle \texttt{for}\ e\ \texttt{do}\ S :: Q, \mathsf{env}_{SV}, g\rangle \\ \qquad \to \langle S :: \texttt{for}\ v'\ \texttt{do}\ S :: Q, \mathsf{env}_{SV}, g-1\rangle\end{array}}\ \left(\begin{array}{c} g \geq 1 \\ v \geq 1 \\ v' = v-1\end{array}\right)$$

$$[\textsc{for}_\mathsf{F}]\frac{\mathsf{env}_{SV} \vdash e \to v}{\mathsf{env}_T \vdash \langle \texttt{for}\ e\ \texttt{do}\ S :: Q, \mathsf{env}_{SV}, g\rangle \to \langle Q, \mathsf{env}_{SV}, g-1\rangle}\ \left(\begin{array}{c} g \geq 1 \\ v < 1\end{array}\right)$$

$$[\textsc{decv}]\frac{\mathsf{env}_{SV} \vdash e \to_e v}{\begin{array}{l}\mathsf{env}_T \vdash \langle \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S :: Q, \mathsf{env}_{SV}, g\rangle \\ \qquad \to \langle S :: \mathsf{del}(x) :: Q, \mathsf{env}_S, ((x, v), \mathsf{env}_V), g-1\rangle\end{array}}\ (g \geq 1)$$

where:

$$x \notin \mathrm{dom}\,(\mathsf{env}_V)$$

$$[\textsc{assv}]\frac{\mathsf{env}_{SV} \vdash e \to_e v}{\begin{array}{l}\mathsf{env}_T \vdash \langle x\ \texttt{:=}\ e :: Q, \mathsf{env}_{SV}, g\rangle \\ \qquad \to \langle Q, \mathsf{env}_S, \mathsf{env}_V[x \mapsto v], g-1\rangle\end{array}}\ \left(\begin{array}{c} g \geq 1 \\ x \in \mathrm{dom}\,(\mathsf{env}_V)\end{array}\right)$$

$$[\textsc{assf}]\frac{\mathsf{env}_{SV} \vdash e \to_e v}{\begin{array}{l}\mathsf{env}_T \vdash \langle \texttt{this}.p\ \texttt{:=}\ e :: Q, \mathsf{env}_{SV}, g\rangle \\ \qquad \to \langle Q, \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]], \mathsf{env}_V, g-1\rangle\end{array}}\ (g \geq 1)$$

where:

$$X = \mathsf{env}_V(\texttt{this})$$
$$\mathsf{env}_F = \mathsf{env}_S(X)$$
$$p \in \mathrm{dom}\,(\mathsf{env}_F)$$

Figure 7.4: Small-step semantics transition rules for statements with gas.

$$[\textsc{call}]\frac{\mathsf{env}_{SV} \vdash e_1 \to_e Y \quad \mathsf{env}_{SV} \vdash e_2 \to_e n \quad \mathsf{env}_{SV} \vdash \tilde{e} \to_e \tilde{v}}{\mathsf{env}_T \vdash \langle \texttt{call}\ \ e_1.f(\tilde{e})\$e_2 :: Q, \mathsf{env}_{SV}, g\rangle \to \langle S :: \mathsf{env}_V :: Q, \mathsf{env}'_{SV}, g-1\rangle} \ (g \geq 1)$$

where:

$$\begin{aligned} X &= \mathsf{env}_V(\texttt{this}) \\ \mathsf{env}_F^X &= \mathsf{env}_S(X) \\ \mathsf{env}_F^Y &= \mathsf{env}_S(Y) \\ (\tilde{x}, S) &= (\mathsf{env}_T(Y))(f) \\ |\tilde{x}| &= |\tilde{v}| = k \\ n &\leq \mathsf{env}_F^X(\texttt{balance}) \\ \mathsf{env}'_S &= \mathsf{env}_S\big[X \mapsto \mathsf{env}_F^X[\texttt{balance -= } n]\big]\big[Y \mapsto \mathsf{env}_F^Y[\texttt{balance += } n]\big] \\ \mathsf{env}'_V &= (\texttt{this}, Y) : (\texttt{sender}, X) : (\texttt{value}, n) : (x_1, v_1) : \ldots : (x_k, v_k) : \mathsf{env}_V^{\varnothing} \end{aligned}$$

$$[\textsc{throw}]\frac{}{\mathsf{env}_T \vdash \langle \texttt{throw} :: Q, \mathsf{env}_{SV}, g\rangle \to \langle \mathsf{exc}(\texttt{pge}) :: Q, \mathsf{env}_{SV}, g\rangle} \ (g \geq 1)$$

$$[\textsc{oog}]\frac{}{\mathsf{env}_T \vdash \langle S :: Q, \mathsf{env}_{SV}, 0\rangle \to \langle \mathsf{exc}(\texttt{oog}) :: S :: Q, \mathsf{env}_{SV}, 0\rangle}$$

$$[\textsc{delv}]\frac{}{\mathsf{env}_T \vdash \langle \mathsf{del}(x) :: Q, \mathsf{env}_S, ((x, v), \mathsf{env}_V), g\rangle \to \langle Q, \mathsf{env}_{SV}, g\rangle}$$

$$[\textsc{return}]\frac{}{\mathsf{env}_T \vdash \langle \mathsf{env}'_V :: Q, \mathsf{env}_{SV}, g\rangle \to \langle Q, \mathsf{env}_S, \mathsf{env}'_V, g\rangle}$$

Figure 7.5: Small-step semantics transition rules for statements with gas (cont.).

With these assumptions, gas consumption is modelled by introducing the side condition $g \geq 1$ in most of the rules, since g is decremented by 1 in the reduct of these rules.[3] The only rules that do not consume gas are [SEQ], since this rule does not describe the execution of an operation but merely transforms the top element of the stack, and [THROW]. Notice, however, that those two rules have the side condition $g \geq 1$ to ensure determinism of the operational semantics: in this way, when the gas becomes 0, only rule [OOG] can be used. Finally, the only rules that do not check the remaining amount of gas are [DELV] and [RETURN], since these rules do not correspond to commands, but are only used to restore a previous state of the variable environment.

Rule [THROW] says that a `throw` command is simply converted into an exception exc(`pge`), denoting that the exception was thrown from within the program. The remaining exception labels are used if one of the side conditions fails; so, the execution of most of the statements can lead to an exception in several different ways:

- An **o**ut-**o**f-**g**as exception `oog` is thrown if we have a command on top of the stack but no gas is left, see rule [OOG].
- A **neg**ative balance exception `neg` is thrown if executing the statement would lead to a negative `balance`. Hence, this exception can only be thrown from the rule [CALL], if the side condition $n \leq \mathsf{env}_F^X(\texttt{balance})$ does not hold.
- Finally, a **r**un**t**ime **e**xception `rte` is thrown if any of the other side conditions does not hold. We omit the rules for generating these exceptions, since they are heavy and do not add anything to our presentation. Furthermore, by using a very standard type system, such exceptions can be easily prevented at compile time.

Notice that, when an exception appears on top of the stack, the execution halts as there are no rules to conclude a transition in such a case. Furthermore, the exception label l has no meaning in the semantics, and the different labels are used only to distinguish between different situations that raise an exception. We could have added a command `catch` to handle exceptions according to their label (like in many programming languages). However, since the purpose of this chapter is to develop techniques that prevent out-of-gas exceptions through static checks, for the sake of simplicity we preferred to omit such a command.

The semantics for blockchains is given in Figure 7.6 and contains rules [GEN] and [REV], to activate and reveal the outcome of a sequence of transactions. Notice that the first rule invokes the semantics for declarations from Figure 7.3.

[3] It is conceptually easy, albeit notationally cumbersome, to provide each operation and command with its own cost $c(\cdot)$ and modify the semantics to take such costs into account in the side conditions and conclusions of each rule. This would relax the simplifying assumptions 1 and 2 given above.

$$[\textsc{Gen}]\frac{\left\langle DC, \mathsf{env}_{ST}^{\emptyset}\right\rangle \to_{DC} \mathsf{env}_{ST}}{\left\langle DC\ \ T, \mathsf{env}_{ST}^{\emptyset}\right\rangle \to_{B} \langle T, \mathsf{env}_{ST}\rangle} \qquad [\textsc{Rev}]\frac{}{\langle \epsilon, \mathsf{env}_{ST}\rangle \to_{B} \mathsf{env}_{ST}}$$

$$[\mathrm{Tx}_1]\frac{\begin{array}{c}\mathsf{env}_T \vdash \langle \texttt{call}\ \ X.f(\tilde{v})\$n :: \bot, \mathsf{env}_S, (\texttt{this}, A), g\rangle \\ \quad \to^* \left\langle \bot, \mathsf{env}'_S, \mathsf{env}_V, g'\right\rangle\end{array}}{\langle A\texttt{->}X.f(\tilde{v})\$(n,g), T, \mathsf{env}_{ST}\rangle \to_B \left\langle T, \mathsf{env}''_S, \mathsf{env}_T\right\rangle}$$

where:

$$\begin{aligned}\mathsf{env}_F^A &= \mathsf{env}_S(A)\\ g &\leq \mathsf{env}_F^A(\texttt{balance}) - n\\ \mathsf{env}''_S &= \mathsf{env}'_S[A \mapsto \mathsf{env}_F^A[\texttt{balance -= } (g - g')]]\end{aligned}$$

$$[\mathrm{Tx}_2]\frac{\begin{array}{c}\mathsf{env}_T \vdash \langle \texttt{call}\ \ X.f(\tilde{v})\$n :: \bot, \mathsf{env}_S, (\texttt{this}, A), g\rangle \\ \quad \to^* \left\langle \mathsf{exc}(l) :: Q, \mathsf{env}'_S, \mathsf{env}_V, g'\right\rangle\end{array}}{\langle A\texttt{->}X.f(\tilde{v})\$(n,g), T, \mathsf{env}_{ST}\rangle \to_B \left\langle T, \mathsf{env}''_S, \mathsf{env}_T\right\rangle}$$

where:

$$\begin{aligned}\mathsf{env}_F^A &= \mathsf{env}_S(A)\\ g &\leq \mathsf{env}_F^A(\texttt{balance}) - n\\ \mathsf{env}''_S &= \mathsf{env}'_S[A \mapsto \mathsf{env}_F^A[\texttt{balance -= } (g - g')]]\end{aligned}$$

Figure 7.6: Transition rules for blockchains.

The key rules for transactions are $[\mathrm{Tx}_1]$ and $[\mathrm{Tx}_2]$ for describing the execution of a transaction that either succeeds or fails. Rule $[\mathrm{Tx}_1]$ describes the situation where the transaction succeeds. Having a gas limit g, the side condition requires that g must not be greater than the available `balance` of the account A initiating the transaction, *after* any funds n transferred along with the call have been deducted from the `balance` (since that could otherwise allow the `balance` of A to become negative). After the transaction succeeds, some amount of gas will have been consumed. The remaining gas g' is then used to calculate how much gas was used, and then this too is subtracted from the `balance` of A. Rule $[\mathrm{Tx}_2]$ describes the situation where the transaction fails with an exception $\mathsf{exc}(l)$. Here, the gas difference $g - g'$ is *still* subtracted from the `balance` of A, but otherwise the state env_S is unaltered. Thus, all other effects of the transaction that raised the exception are rolled back, and the execution can continue with the next transaction.

7.4 A type system for termination

The gas mechanism ensures that all transactions will eventually terminate, thereby preventing denial-of-service attacks resulting from diverging method calls. However, it does not prevent potentially diverging contracts from being added to the blockchain.

This presents a problem to the users of the smart contracts: how to ensure that they do not inadvertently create a transaction invoking a diverging contract, since that would consume all available gas supplied to the transaction. Furthermore, even if a method call does not actually diverge, it may still require many execution steps, and the user must therefore be sure to supply sufficient gas for the transaction to allow the execution to finish. In the present section, we shall describe a type system for statically checking termination of transactions, whilst also providing upper (and lower) bounds on the number of execution steps.

7.4.1 The type system

We wish to create a type system that will allow us to derive an upper bound on the number of execution steps required to run a statement S, since that is proportional to the required amount of gas. Hence, type judgments for statements will be of the form $\Gamma \vdash S : n$ where n is an integer denoting the maximum number of steps. This number may be affected by the (integer) values of the loop condition expressions in S; thus we need the integer type (for both variables and expressions) to provide explicit upper and lower bounds on those values. We write this as int_ℓ^u, where u (resp. ℓ) is the upper (resp. lower) bound (both included).

The int_ℓ^u type is quite limiting; for instance, it will disallow some common operations such as increments/decrements of a variable (e.g., x `:=` x`+1`). Therefore, we shall also include an ordinary (unbounded) integer type int for integer-valued variables and fields that do not in any way contribute to the expressions that guard `for` loops. To allow variables of both types to appear in operations, we shall also introduce a subtyping relation $<:$ to allow bounded integer types to be coerced up to the unbounded type.

We also need the type of methods to have an upper bound on the number of steps required to execute the method body. This type is written $\mathsf{proc}(\widetilde{B})_\ell^u : n$, to be reads as: assuming that the formal parameters are of (base) types $\widetilde{B}$ and that the transferred amount of currency is in the interval $[\ell..u]$, then the method body requires at most n steps to execute. The bounds on the `value` variable are necessary, since the variable might be used in a loop expression[4]. Thus, we also handle divergence resulting from recursive method calls, since any calls within the method body would need to have a value strictly lower than n (as calling the method itself also requires one step).

Finally, we need to give types to contract names, which must provide information on the signatures of fields and methods implemented in the contract. Here, we shall employ a method similar to the one used in [4] and assume a set of *type names* (or 'interface names') TNames, ranged over by I. We then let type environments Γ be

[4]We could also introduce an 'unbounded' method type (similarly to what we did for int and int_ℓ^u), but we shall omit this to avoid complicating the matter further. The 'bounded' method type also has the added benefit of allowing a programmer to specify a minimum amount of currency that must be provided to call a method.

partial functions from names to types, but also include Γ in the language of types and use these as the types of interface names. The idea is that, if X is a contract name of type I, then $\Gamma(X) = I$ and $\Gamma(I) = \Gamma_I$, where Γ_I then contains the signatures of the fields and methods of contract X. We shall assume that these interfaces are declared besides the contract declarations, e.g. using an interface definition language similar to the one described in [4], and we refer the reader to that work for further details.

Definition 51 (Language of types)**.** We use the following language of types, where $I \in \mathsf{TNames}$ is a *type name* (or 'interface name') and $\mathcal{N} = \mathsf{ANames} \cup \mathsf{FNames} \cup \mathsf{VNames} \cup \mathsf{MNames} \cup \mathsf{TNames}$:

$$\begin{aligned} B &::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{int}^u_\ell \mid I \\ T \in \mathcal{T} &::= B \mid \mathrm{proc}(\widetilde{B})^u_\ell : n \mid \Gamma \\ \Gamma, \Delta &::= \mathcal{N} \rightharpoonup \mathcal{T} \end{aligned}$$

We write $\widetilde{T}$ for a tuple of types $(T_1, \ldots, T_n)$. ■

As in [4], we require that all interface declarations be *well-formed* in the sense that they must at least contain a declaration for the mandatory members, i.e. the `balance` field (type int) and the `send()` method (type $\mathrm{proc}()^{\texttt{INT\_MAX}}_0 : 1$). This ensures that we can define a minimal interface declaration called $I^\top$, containing just the signatures of `balance` and `send()`, such that every well-formed interface declaration is a specialisation of $I^\top$; this is ensured by the subtyping relation and it is necessary to allow us to give a type to the 'magic' variable `sender`, which is available within the body of any method. The subtyping relation is given by the reflexive and transitive closure of the rules in Figure 7.7.

Note that the subtyping relation is parametrised with a type environment Γ: this is necessary for subtyping of interface names I. Also note the two rules for subtyping of the two integer types. The first rule, [SUBS-INT$_1$], allows us to widen the bounds on a bounded integer. This is necessary to make statements such as int^8_2 `x := 5` be well-typed, since the type of `5` is int^5_5 (by rule [T-VAL]). The second rule, [SUBS-INT$_2$], allows any bounded integer type to be coerced up to an unbounded integer type. This is necessary, if e.g. both bounded and unbounded integers appear together in an expression, or if we have an assignment of a bounded integer type to an unbounded integer variable.

We shall also use a *typed* syntax of TinySol, where local variables are now declared as B `x := e` and B is the type of the value of the expression e. Furthermore, we assume that the type information is also stored in env_V, along with the actual value of each variable. We write this as a triple (x, v, B) but otherwise ignore the type information in the semantics. Interface types could also be explicitly given in the contract definitions, i.e. as `contract` X `:` I `{` DF DM `}`, but to simplify the presentation we shall here merely assume that interface definitions for all contracts exist and are added to any type environment Γ that we shall consider in the following.

$$[\textsc{subs-int}_1]\frac{}{\Gamma \vdash \mathsf{int}^{u_1}_{\ell_1} <: \mathsf{int}^{u_2}_{\ell_2}} \begin{pmatrix} u_1 \leq u_2 \\ \ell_1 \geq \ell_2 \end{pmatrix}$$

$$[\textsc{subs-int}_2]\frac{}{\Gamma \vdash \mathsf{int}^{u}_{\ell} <: \mathsf{int}}$$

$$[\textsc{subs-name}]\frac{\Gamma \vdash \Gamma(I^1) <: \Gamma(I^2)}{\Gamma \vdash I^1 <: I^2}$$

$$[\textsc{subs-env}]\frac{\forall n \in \mathrm{dom}\,(\Gamma_2)\ .\ \Gamma_1(n) <: \Gamma_2(n)}{\Gamma \vdash \Gamma_1 <: \Gamma_2}$$

$$[\textsc{subs-proc}]\frac{\Gamma \vdash \widetilde{B}_1 <: \widetilde{B}_2}{\Gamma \vdash \mathrm{proc}(\widetilde{B}_1)^{u_1}_{\ell_1} : n_1 <: \mathrm{proc}(\widetilde{B}_2)^{u_2}_{\ell_2} : n_2} \begin{pmatrix} n_1 \leq n_2 \\ u_1 \leq u_2 \\ \ell_1 \geq \ell_2 \end{pmatrix}$$

Figure 7.7: Subtyping rules.

$$[\textsc{t-dec-c}]\frac{\Gamma, \Delta \vdash DF \quad \Gamma, \Delta \vdash DM \quad \Gamma \vdash DC}{\Gamma \vdash \texttt{contract}\ X\ \texttt{\{}\ DF\ DM\ \texttt{\}}\ DC}$$
where:
$$\Delta = \texttt{this} : \Gamma(X)$$

$$[\textsc{t-dec-f}]\frac{\Gamma, \Delta \vdash v : B \quad \Gamma, \Delta \vdash DF}{\Gamma, \Delta \vdash \texttt{field}\ p\ \texttt{:=}\ v\texttt{;}\ DF}$$
where:
$$I = \Delta(\texttt{this})$$
$$B = \Gamma(I)(p)$$

$$[\textsc{t-dec-m}]\frac{\Gamma, \Delta' \vdash S : n \quad \Gamma, \Delta \vdash DM}{\Gamma, \Delta \vdash \texttt{func}\ f(\tilde{x})\ \texttt{\{}\ S\ \texttt{\}}\ DM}$$
where:
$$I = \Delta(\texttt{this})$$
$$\Gamma(I)(f) = \mathrm{proc}(\widetilde{B})^{u}_{\ell} : n$$
$$\Delta' = \Delta, \tilde{x} : \widetilde{B}, \texttt{value} : \mathsf{int}^{u}_{\ell}, \texttt{sender} : I^{\top}$$

Figure 7.8: Type rules for declarations.

$$[\textsc{t-env-t}]\frac{\Gamma, \Delta \vdash \mathsf{env}_M \quad \Gamma \vdash \mathsf{env}_T}{\Gamma \vdash (X, \mathsf{env}_M), \mathsf{env}_T}$$
where:
$$\Delta = \texttt{this} : \Gamma(X)$$

$$[\textsc{t-env-m}]\frac{\Gamma, \Delta' \vdash S : n \quad \Gamma, \Delta \vdash \mathsf{env}_M}{\Gamma, \Delta \vdash (f, (\tilde{x}, S)), \mathsf{env}_M}$$
where:
$$I = \Delta(\texttt{this})$$
$$\Gamma(I)(f) = \mathrm{proc}(\widetilde{B})^{u}_{\ell} : n$$
$$\Delta' = \Delta, \tilde{x} : \widetilde{B}, \texttt{value} : \mathsf{int}^{u}_{\ell}, \texttt{sender} : I^{\top}$$

$$[\textsc{t-env-s}]\frac{\Gamma, \Delta \vdash \mathsf{env}_F \quad \Gamma \vdash \mathsf{env}_S}{\Gamma \vdash (X, \mathsf{env}_F), \mathsf{env}_S}$$
where:
$$\Delta = \texttt{this} : \Gamma(X)$$

$$[\textsc{t-env-f}]\frac{\Gamma, \Delta \vdash v : B \quad \Gamma, \Delta \vdash \mathsf{env}_F}{\Gamma, \Delta \vdash (p, v), \mathsf{env}_F}$$
where:
$$I = \Delta(\texttt{this})$$
$$B = \Gamma(I)(p)$$

$$[\textsc{t-env-v}]\frac{\Gamma, \Delta \vdash v : B \quad \Gamma, \Delta \vdash \mathsf{env}_V}{\Gamma, \Delta \vdash (x, v, B), \mathsf{env}_V}$$
where:
$$B = \Delta(x)$$

Figure 7.9: Type rules for environment agreement.

$$[\text{T-VAR}]\frac{}{\Gamma, \Delta \vdash x : B}\ (\Delta(x) = B) \qquad [\text{T-FIELD}]\frac{\Gamma, \Delta \vdash e : I}{\Gamma, \Delta \vdash e.p : B}\ (\Gamma(I)(p) = B)$$

$$[\text{T-VAL}]\frac{}{\Gamma, \Delta \vdash v : B}\left(B = \begin{cases} \Gamma(v) & \text{if } v \in \mathsf{ANames} \\ \mathsf{bool} & \text{if } v \in \mathbb{B} \\ \mathsf{int}_v^v & \text{if } v \in \mathbb{Z} \end{cases}\right)$$

$$[\text{T-OP}]\frac{\Gamma, \Delta \vdash \tilde{e} : \widetilde{B} \quad \vdash \mathrm{op} : \widetilde{B} \to B}{\Gamma, \Delta \vdash \mathrm{op}(\tilde{e}) : B}$$

$$[\text{T-SUBS}]\frac{\Gamma, \Delta \vdash e : B_1 \quad \Gamma \vdash B_1 <: B_2}{\Gamma, \Delta \vdash e : B_2}$$

Figure 7.10: Type rules for expressions

We can now give the type rules, starting with the rules for declarations, given in Figure 7.8; and for environment agreement, given in Figure 7.9. Note that they are almost identical, since the environments are just an alternative representation of the code. The type judgments here ascertain that each contract, field, variable and method indeed has a type in the type environment, that these types are compatible with the values assigned to them, and, in the case of methods, with the number of steps inferred for the method body S. Note also that we write the type environment as split into two parts: the first part, Γ, contains the types for contract addresses; the second part, Δ, records the types of local variables, including the 'magic' variables `this`, `sender` and `value`. Also, unless otherwise noted, we shall assume in the rules that these reserved names are contained in the respective sets of field and variables names, i.e. `balance` $\in \mathsf{FNames}$ and `this`, `sender`, `value` $\in \mathsf{VNames}$.

Next, we have the rules for typing expressions, which are given in Figure 7.10. Type judgments are here of the form $\Gamma, \Delta \vdash e : B$. Note that in rule [T-VAL], when v is an integer value, we set both the upper and lower bounds in the type to be exactly the value v. To widen the type of the value, e.g., when typing an assignment (such as $\mathsf{int}_2^5\ x$ `:=` 3), we will use the subtyping relation from Figure 7.7, through the subsumption rule [T-SUBS]. The subtyping rules are only used for the right-hand side expression of assignments and for the arguments to method calls, to allow the value of the expression to be coerced up to a less specific type that can match the type of the variable or field.

In the rule [T-OP], we assume the existence of type rules with judgments of the form $\vdash \mathrm{op} : \widetilde{B} \to B$ for each operation op. In particular, we assume that we can derive conclusions about the upper and lower bounds on integer-valued operations from the bounds on the expression operands. This is straightforward for standard

$$[\textsc{t-skip}]\frac{}{\Gamma,\Delta \vdash \texttt{skip} : 1}$$

$$[\textsc{t-throw}]\frac{}{\Gamma,\Delta \vdash \texttt{throw} : 1}$$

$$[\textsc{t-assv}]\frac{\Gamma,\Delta \vdash x : B \quad \Gamma,\Delta \vdash e : B}{\Gamma,\Delta \vdash x \ \texttt{:=}\ e : 1}$$

$$[\textsc{t-assf}]\frac{\Gamma,\Delta \vdash e_1.p : B \quad \Gamma,\Delta \vdash e_2 : B}{\Gamma,\Delta \vdash e_1.p \ \texttt{:=}\ e_2 : 1}$$

$$[\textsc{t-decv}]\frac{\Gamma,\Delta \vdash e : B \quad \Gamma,(\Delta, x : B) \vdash S : n}{\Gamma,\Delta \vdash B\ x \ \texttt{:=}\ e\ \texttt{in}\ S : n+2}$$

$$[\textsc{t-seq}]\frac{\Gamma,\Delta \vdash S_1 : n_1 \quad \Gamma,\Delta \vdash S_2 : n_2}{\Gamma,\Delta \vdash S_1 ; S_2 : n_1 + n_2 + 1}$$

$$[\textsc{t-if}]\frac{\Gamma,\Delta \vdash e : \mathsf{bool} \quad \Gamma,\Delta \vdash S_\mathsf{T} : n_1 \quad \Gamma,\Delta \vdash S_\mathsf{F} : n_2}{\Gamma,\Delta \vdash \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F} : \max(n_1, n_2) + 1}$$

$$[\textsc{t-for}]\frac{\Gamma,\Delta \vdash e : \mathsf{int}^u_\ell \quad \Gamma,\Delta \vdash S : n}{\Gamma,\Delta \vdash \texttt{for}\ e\ \texttt{do}\ S : \max(1, u(n+1)+1)}$$

$$[\textsc{t-call}]\frac{\Gamma,\Delta \vdash e_1 : I \quad \Gamma,\Delta \vdash \tilde{e} : \widetilde{B} \quad \Gamma,\Delta \vdash e_2 : \mathsf{int}^u_\ell}{\Gamma,\Delta \vdash e_1.f(\tilde{e}) : e_2 : n+2}$$

where:
$\Gamma(I)(f) = \mathrm{proc}(\widetilde{B})^u_\ell : n$

Figure 7.11: Type rules for statements.

integer operations. As an example, consider the following rules:

$$\vdash + : (\mathsf{int}^{u_1}_{\ell_1}, \mathsf{int}^{u_2}_{\ell_2}) \to \mathsf{int}^{u_1+u_2}_{\ell_1+\ell_2}$$
$$\vdash - : (\mathsf{int}^{u_1}_{\ell_1}, \mathsf{int}^{u_2}_{\ell_2}) \to \mathsf{int}^{u_1-\ell_2}_{\ell_1-u_2}$$

For example, assuming that $\Gamma,\Delta \vdash x : \mathsf{int}^5_2$, the type of $10 - x$ is int^8_5.

Finally, we have the type rules for statements, given in Figure 7.11. Here, type judgments are of the form $\Gamma,\Delta \vdash S : n$, with n denoting the maximum number of steps required to execute S. This is mostly calculated in an obvious way, by only using the upper bounds u; however, the lower bounds ℓ are implicitly used when calculating the types of expressions (as discussed above). For example, we calculate the upper bound on the number of steps of `for` e `do` S by first calculating the upper

bound u on the value of e and the upper bound n on the steps for S: If $u \geq 1$, we multiply u and $n+1$ (the '+1' is the extra step needed for activating every iteration of the loop, cf. [FOR$_\mathrm{T}$]) and finally we sum 1 (the cost of the final activation of the loop; i.e., when the guard becomes < 1, see [FOR$_\mathrm{F}$]). Otherwise the upper bound is 1 (only the case with a guard < 1). The max is used for covering the two cases with one single rule.

Notice that [T-DECV] and [T-CALL] both have $n+2$ in their conclusion, rather than $n+1$, because both constructs will push an extra symbol onto the stack (viz., $\mathsf{del}(x)$ and env_V, resp.), which will require an extra step in the semantics when their scope ends. A similar reason justifies the '+1' in rules [T-SEQ], [T-IF], and [T-FOR].

We illustrate the use of the type system with a small example. Consider the statement `for` x `do call` $y.f(x)$`:1`, and assume it is to be executed in an environment where the local variables x and y are declared so that $\mathsf{env}_V(x) = 3$ and $\mathsf{env}_V(y) = A$, for some contract A of interface type I containing a method f. Thus, this statement will call the method $A.f$ three times, each time transferring 1 unit of currency along with the call, and with the number 3 as the actual parameter (since the value of x does not change, cf. rule [FOR$_\mathrm{T}$]).

To type the example, assume further that $\Delta(x) = \mathsf{int}_1^5$, $\Delta(y) = I$, and $\Gamma(I)(f) = \mathsf{proc}(\mathsf{int})_1^{10}$: 20. Thus, the `for`-loop can run at most 5 times; moreover, the method call must at least receive 1 unit of currency to execute and will finish in at most 20 steps. The most interesting part of the typing derivation is then as follows:

$$\dfrac{\dfrac{\Delta(x) = \mathsf{int}_1^5}{\Gamma, \Delta \vdash x : \mathsf{int}_1^5} \quad \Gamma \vdash \mathsf{int}_1^5 <: \mathsf{int}}{\Gamma, \Delta \vdash x : \mathsf{int}} \quad \dfrac{\Delta(x) = \mathsf{int}_1^5}{\Gamma, \Delta \vdash x : \mathsf{int}_1^5} \quad \cdots \quad \dfrac{\ldots}{\Gamma, \Delta \vdash 1 : \mathsf{int}_1^{10}}}{\dfrac{\Gamma, \Delta \vdash \texttt{call } y.f(x)\texttt{\$1} : 20 + 2}{\Gamma, \Delta \vdash \texttt{for } x \texttt{ do call } y.f(x)\texttt{\$1} : \max(1, 5((20+2)+1)+1)}}$$

For space reasons, we have omitted the derivation of $\Gamma, \Delta \vdash y : I$, which is concluded by [T-VAR] and of $\Gamma, \Delta \vdash 1 : \mathsf{int}_1^{10}$, which is inferred using the subtyping rules. The latter involves concluding $\Gamma, \Delta \vdash 1 : \mathsf{int}_1^1$ by [T-VAL], and then $\Gamma \vdash \mathsf{int}_1^1 <: \mathsf{int}_1^{10}$ can be proved by [SUBS-INT$_1$], since $1 \leq 10$ for the upper bound, and $1 \leq 1$ for the lower bound. Likewise, the type of x, which is int_1^5, is coerced up to int by the subtyping rule [SUBS-INT$_2$], when x is used as a parameter for the method call, as required by the type of f. Thus we conclude, by [T-CALL], that the method call itself will require at most $20 + 2 = 22$ steps, and, finally, by [T-FOR], that the entire `for`-loop will require at most $\max(1, 5((20+2)+1)+1) = 116$ steps.

7.4.2 Properties of the type system

Our type system ensures that a well-typed statement S, executing in a configuration with well-typed environments env_{TSV}, will terminate after at most n steps. To prove

this property, we wish to show a statement saying that well-typedness is preserved by transitions (subject reduction), and furthermore that the number n decreases after every transition step. However, our semantics does not describe the execution of single statements, but rather of *stacks*, so we shall also need the notion of well-typed stacks and configurations.

The definition of such notions is complicated by the form of our semantic rules (Figure 7.4): Atomic statements are taken from the top of the stack and evaluated directly in the conclusion of a rule, whereas composite statements are rewritten and then pushed back onto the stack. This feature makes exception handling easy, since we do not need extra rules for propagating an exception down through a derivation tree, but it does pose a problem w.r.t. the subject reduction theorem, in particular in dealing with the rules [DECV] and [CALL]. Both of these rules alter the local variable environment env_V, either by adding a new binding to it ([DECV]) or by replacing it with a new environment ([CALL]), but the scope of these changes is not confined to the premise of those rules, unlike in the corresponding type rules ([T-DECV] and [T-CALL]). Instead, the syntax of stacks (Definition 50) introduces some extra symbols ($\mathsf{del}(x)$ and env_V) to mark the end of these scopes, which are then pushed onto the stack *before* pushing the statement S to be executed within the scope of these new bindings. This, in turn, means that new local variables can appear as free in the statement residing on the top of a stack *after* a transition step, if the previously executed statement was a declaration or a call. Thus, to be able to type the stack after a transition step, we need a way to update the Δ part of the type environment with information on these new bindings. This is done as follows:

Definition 52 (Extraction function)**.** The type extraction function from stacks to type environments is given by the following clauses:

$$\mathbf{E}^{\Gamma}_{\Delta}((x, v, B), \mathsf{env}_V :: Q) = x : B,\ \mathbf{E}^{\Gamma}_{\Delta}(\mathsf{env}_V :: Q) \tag{7.1}$$

$$\mathbf{E}^{\Gamma}_{\Delta}(\mathsf{env}^{\varnothing}_V :: Q) = \epsilon \tag{7.2}$$

$$\mathbf{E}^{\Gamma}_{\Delta}(B\ \ x\ \texttt{:=}\ e\ \texttt{in}\ S :: Q) = x : B,\ \Delta \tag{7.3}$$

$$\mathbf{E}^{\Gamma}_{\Delta, x:B}(\mathsf{del}(x) :: Q) = \Delta \tag{7.4}$$

$$\mathbf{E}^{\Gamma}_{\Delta}(e_1.f(\tilde{e}) : e_2 :: Q) = \texttt{this} : I,\ \texttt{sender} : I^{\top},\ \texttt{value} : \mathsf{int}^{u}_{\ell},\ \tilde{x} : \widetilde{B} \tag{7.5}$$
$$\text{where } \Gamma, \Delta \vdash e_1 : I \text{ and } \Gamma(I)(f) = \mathrm{proc}(\widetilde{B})^{u}_{\ell} : n.$$

$$\mathbf{E}^{\Gamma}_{\Delta}(Q) = \Delta \qquad \text{in all other cases} \tag{7.6}$$

■

The function ensures that the type environment for local variables Δ remains in agreement with the currently active variable environment env_V after a transition. This is also the reason why we needed to store the type information of locally declared

$$[\textsc{t-bot}]\frac{}{\Gamma, \Delta \vdash \bot : 0}$$

$$[\textsc{t-exc}]\frac{}{\Gamma, \Delta \vdash \mathsf{exc}(l) :: Q : 0}$$

$$[\textsc{t-del}]\frac{\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\mathsf{del}(x) :: Q) \vdash Q : n}{\Gamma, \Delta \vdash \mathsf{del}(x) :: Q : n}$$

$$[\textsc{t-stm}]\frac{\Gamma, \Delta \vdash S : n_1 \quad \Gamma, \Delta \vdash Q : n_2}{\Gamma, \Delta \vdash S :: Q : n_1 + n_2}$$

$$[\textsc{t-ctx}]\frac{\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\mathsf{env}_V :: Q) \vdash Q : n}{\Gamma, \Delta \vdash \mathsf{env}_V :: Q : n}$$

$$[\textsc{t-cfg}]\frac{\Gamma, \Delta \vdash Q : n \quad \Gamma \vdash \mathsf{env}_S \quad \Gamma, \Delta \vdash \mathsf{env}_V}{\Gamma, \Delta \vdash \langle Q, \mathsf{env}_{SV}, g\rangle} \ (n < g)$$

Figure 7.12: Type rules for stacks and configurations.

variables in env_V, and why we use the convention of writing the type environment as split into two segments (viz., Γ and Δ), since the Γ segment remains unaltered.

Now, we can define the notions of well-typed stacks and configurations, which result from the rules given in Figure 7.12. Note that we use the extraction function in the premises of rules [T-CTX] and [T-DEL] to handle the two special cases when the top of the stack is an end-of-scope symbol. Moreover, rule [T-CFG] states that a configuration $\langle Q, \mathsf{env}_{SV}, g\rangle$ is well-typed (for some type environment Γ, Δ) if the environments agree, and the stack is typable as terminating in n steps, *and* the supplied gas value g is greater than n; this is precisely what we want our type system to ensure.

We can now state our main theorem:

Theorem 18 (Subject reduction)**.** *Assume that*

- $\Gamma \vdash \mathsf{env}_T$, *and*
- $\Gamma, \Delta \vdash \langle Q, \mathsf{env}_{SV}, g\rangle$.

If $\mathsf{env}_T \vdash \langle Q, \mathsf{env}_{SV}, g\rangle \to \langle Q', \mathsf{env}'_{SV}, g'\rangle$ *then* $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(Q) \vdash \langle Q', \mathsf{env}'_{SV}, g'\rangle$.

The proof proceeds as follows: We know $\Gamma, \Delta \vdash \langle Q, \mathsf{env}_{SV}, g\rangle$ was concluded by [T-CFG]. From the premises and side condition of this rule, we know that:

- $\Gamma, \Delta \vdash Q : n$,

- $\Gamma \vdash \mathsf{env}_S$,
- $\Gamma, \Delta \vdash \mathsf{env}_V$, and
- $n < g$.

The transition is of the form

$$\mathsf{env}_T \vdash \langle Q, \mathsf{env}_{SV}, g \rangle \to \langle Q', \mathsf{env}'_{SV}, g' \rangle$$

and our goal is therefore to show that $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(Q) \vdash \langle Q', \mathsf{env}'_{SV}, g' \rangle$ also holds. This must again be concluded by [T-CFG]; hence, we must show that the premises of this rule, i.e.

- $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(Q) \vdash Q' : n'$,
- $\Gamma \vdash \mathsf{env}'_S$,
- $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(Q) \vdash \mathsf{env}'_V$, and
- $n' < g'$

are all satisfied. We therefore proceed by case analysis on the last rule used to infer the aforementioned transition. The proof can be found in Section 7.6.2.

Theorem 18 ensures not only that well-typedness of configurations is preserved, but also that the number of steps always remains lower than the provided gas value, by the side condition in rule [T-CFG]:

Corollary 12. *Assume that*

- $\Gamma \vdash \mathsf{env}_T$, *and*
- $\Gamma, \Delta \vdash \langle Q, \mathsf{env}_{SV}, g \rangle$.

If $\mathsf{env}_T \vdash \langle Q, \mathsf{env}_{SV}, g \rangle \to \langle Q', \mathsf{env}'_{SV}, g' \rangle$ *then* $Q' \neq \mathsf{exc}(\mathsf{oog}) :: Q''$.

It is perhaps also worth emphasising that the type system also ensures termination. This is another simple consequence of Theorem 18, since the gas value decreases at every step, except when the transition is concluded with one of the rules [SEQ], [DELV], [RETURN], [THROW] and [OOG]. Of these, the last two place an exception on the top of the stack, which has no further transitions, so the execution terminates. Since all stacks have a finite depth, the first three rules can only be used finitely many times before another, gas-consuming statement is encountered, or we reach the bottom of the stack. Thus, the number of steps n must necessarily also always eventually decrease.

Lastly, Theorem 18 can be generalised to transactions and blockchains in an obvious way, since a transaction simply corresponds to a method call placed on an

empty stack (see rule [Tx_1]), from which a Δ is derivable using the extraction function from Definition 52. Given well-typed environments env_{TS}, the type system can thus be used to ensure that a transaction is supplied with sufficient gas to allow it to finish without invoking rule [Tx_2].

7.4.3 Limitations of the type system

The type system ensures termination of well-typed configurations (and hence transactions) and absence of out-of-gas exceptions, but at the cost of some limitations. Firstly, we had to abandon `while` loops, in favour of the (more controllable) `for` loops. This does not seem too severe, since smart contracts are not intended to be used to create non-terminating programs.

More limiting, perhaps, is the fact that it also prohibits all forms of recursive calls, including mutual recursions. This is a consequence of rule [T-CALL], which assigns a number of $n + 2$ steps to a call to a method f, whose number itself is n. By rule [T-ENV-M], the declaration of f is well-typed if the statement S in the method body is typeable as terminating in n steps. As n is strictly less than $n + 2$, S therefore cannot contain a recursive call to f, nor a call to any other method that itself calls f. This limitation seems more severe, since it rules out even terminating recursive functions.

Finally, the bounded integer type needed for typing the guards of `for` loops does not permit some common assignments where the assigned variable appears in the expression (e.g., `x := x+1`). This limitation is a consequence of rules [T-ASSV], [T-DECV] and [T-ASSF], which require the assigned variable (resp. field) to have the same base type B as the expression e. Suppose for example that x has type int^{u}_{ℓ}, and we have the assignment `x := x+1`. The computed bounds for the expression (which we assume in the rule [T-OP]) must therefore be $\mathsf{int}^{u+1}_{\ell+1}$, which does not match the type of x, nor can it be coerced down to match it via subtyping. However, we remark that this limitation only affects those variables that appear in the guard of `for` loops, since all other ones can be typed as int.

7.5 Conclusion and future work

In this chapter, we have further extended the TinySol model by equipping it with a small-step semantics, exceptions and a gas mechanism. Using this refined smart-contract model language, we presented a first step towards the development of static analysis techniques for checking the desirable property that a transaction never runs out-of-gas during its execution. This is done by means of a type system whose main aim is to give an upper bound on the number of steps needed by a transaction to complete its task.

The present type system is, in particular, inspired by Type System 1 of [37]. That paper also presents two other type systems, which e.g. allow for some limited

recursion. Thus, one avenue for future work is to extend the present type system in a similar manner to permit some recursive method calls.

Another avenue concerns an altogether different approach to type soundness. The present work uses the usual, inductive (subject reduction) approach of [133]. However, one way to lift some of the restrictions on the loop construct would be to instead use a coinductive approach, sometimes known as *semantic typing* [7; 9; 23]. A key difference lies in the structure of the soundness proof, which in semantic typing allows 'manual proofs' to be used to guarantee well-typedness of some 'unsafe' pieces of code, which cannot otherwise be judged type-safe using the type rules. This approach has in particular been explored by [7; 9] in a setting of proof-carrying code [88], the point being that such proofs can be automatically verified by a proof-checker. A 'manual proof' could for example be used to show that a particular usage of a `while` loop in a contract in fact *will* terminate, and it could then be published on the blockchain along with the contract, thereby taking advantage of the immutable nature of blockchains. This would allow users to still type check transactions, even if they invoke methods containing unsafe code such as `while` loops, because the type checker can take the accompanying 'manual proof' into account. We intend to explore this possibility in future work.

An orthogonal way to evolve our type system is to let it also provide a lower bound on the gas needed to execute a piece of code (not just an upper bound, as it is now). This can be useful for estimating how much currency one needs to have in the account to invoke a certain functionality.

Finally, having introduced the gas mechanism, we plan to implement real-life smart contracts in TinySol and apply our type system to them to check its practical applicability to meaningful case studies.

7.6 Proofs

7.6.1 Auxiliary lemmas for the proof of Theorem 18

The proof of Theorem 18 relies on a number of standard lemmas:

Lemma 53 (Strengthening). *If* $\Gamma, (\Delta, x : B) \vdash \mathsf{env}_V$ *and* $x \notin \mathrm{dom}(\mathsf{env}_V)$, *then* $\Gamma, \Delta \vdash \mathsf{env}_V$.

Proof. By induction on the structure of env_V, by using rule [T-ENV-V]. □

The next two lemmas simply state that well-typedness of the variable environment (resp. the state) is preserved, if we replace a value v_1 with another value v_2 of the same type as v_1. Both are shown by induction on the structure of env_V (resp. env_S and env_F), by using the rules [T-ENV-V] resp. [T-ENV-S] and [T-ENV-F].

Lemma 54 (Update for variables). *If*

- $\Gamma, \Delta \vdash \mathsf{env}_V$, *and*
- $x \in \mathrm{dom}(\mathsf{env}_V)$, *and*
- $\Delta(x) = B$, *and*
- $\Gamma, \Delta \vdash v : B$,

then $\Gamma, \Delta \vdash \mathsf{env}_V[x \mapsto v]$.

Lemma 55 (Update for fields). *If*

- $\Gamma \vdash \mathsf{env}_S, X \in \mathrm{dom}(\mathsf{env}_S)$, *and*
- $\Gamma(X)(p) = B$, *and*
- $\Gamma, \Delta \vdash v : B$,

then $\Gamma \vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]$.

Next, we need a lemma stating that, if e has a type B and e evaluates to a value v relative to well-typed environments env_{SV}, then v will indeed be a value of type B.

Lemma 56 (Safety for expressions). *Assume that*

- $\Gamma \vdash \mathsf{env}_S$, *and*
- $\Gamma, \Delta \vdash \mathsf{env}_V$, *and*
- $\Gamma, \Delta \vdash e : B$.

If $\mathsf{env}_{SV} \vdash e \to v$, *then* $\Gamma, \Delta \vdash v : B$.

Proof. By induction on the structure of e. □

7.6.2 Proof of Theorem 18

Proof. We know $\Gamma, \Delta \vdash \langle Q, \mathsf{env}_{SV}, g\rangle$ was concluded by [T-CFG]. From the premises and side condition of this rule, we know that:

- $\Gamma, \Delta \vdash Q : n$,
- $\Gamma \vdash \mathsf{env}_S$,
- $\Gamma, \Delta \vdash \mathsf{env}_V$, and
- $n < g$.

The transition is of the form

$$\mathsf{env}_T \vdash \langle Q, \mathsf{env}_{SV}, g\rangle \to \langle Q', \mathsf{env}'_{SV}, g'\rangle \tag{7.7}$$

and our goal is therefore to show that $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(Q) \vdash \langle Q', \mathsf{env}'_{SV}, g'\rangle$ also holds. This must again be concluded by [T-CFG]; hence, we must show that the premises of this rule, i.e.

- $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(Q) \vdash Q' : n'$,
- $\Gamma \vdash \mathsf{env}'_S$,
- $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(Q) \vdash \mathsf{env}'_V$, and
- $n' < g'$

are all satisfied. We therefore proceed by case analysis on how the transition (7.7) has been inferred (by considering the rules in Figure 7.4):

- Suppose the transition was concluded by [SKIP]. Then the stack is of the form $\texttt{skip} :: Q$, and (7.7) is of the form

  $$\mathsf{env}_T \vdash \langle \texttt{skip} :: Q, \mathsf{env}_{SV}, g\rangle \to \langle Q, \mathsf{env}_{SV}, g-1\rangle$$

  Now, $\Gamma, \Delta \vdash \texttt{skip} :: Q : n$ must have been concluded by [T-STM], since the top element on the stack is a statement. The conclusion tells us that $n = n_1 + n_2$, and from the premise we have that

  $$\Gamma, \Delta \vdash \texttt{skip} : n_1$$
  $$\Gamma, \Delta \vdash Q : n_2$$

  Here, $\Gamma, \Delta \vdash \texttt{skip} : n_1$ must have been concluded by [T-SKIP], which tells us that $n_1 = 1$. Thus $n = 1 + n_2$, and therefore $n' = n_2 = n - 1$.

  Finally, $\mathbf{E}^{\Gamma}_{\Delta}(\texttt{skip} :: Q) = \Delta$ by case (7.6) of Definition 52. Putting this together, we thus have the following:

- $\Gamma, \Delta \vdash Q : n - 1$ as argued above,
- $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
- $\Gamma, \Delta \vdash \mathsf{env}_V$ by assumption, since the transition does not alter env_V,
- $n' < g'$, since $n' = n - 1$, $g' = g - 1$, and $n < g$.

- Suppose that [SEQ] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle S_1 ; S_2 :: Q, \mathsf{env}_{SV}, g \rangle \to \langle S_1 :: S_2 :: Q, \mathsf{env}_{SV}, g \rangle$$

Furthermore, $\Gamma, \Delta \vdash \langle S_1 ; S_2 :: Q \rangle : n$ was concluded by [T-STM], where $n = n_1 + n_2$, and with premises

$$\Gamma, \Delta \vdash S_1 ; S_2 : n_1$$
$$\Gamma, \Delta \vdash Q : n_2$$

of which the first was concluded by [T-SEQ]. Thus $n_1 = n_1' + n_1'' + 1$ from the conclusion of that rule, and

$$\Gamma, \Delta \vdash S_1 : n_1'$$
$$\Gamma, \Delta \vdash S_2 : n_1''$$

from its premises.

As $\mathbf{E}_\Delta^\Gamma(S_1 ; S_2 :: Q') = \Delta$ by case (7.6) of Definition 52, we can therefore conclude

$$\Gamma, \Delta \vdash S_2 :: Q : n_1'' + n_2$$
$$\Gamma, \Delta \vdash S_1 :: S_2 :: Q : n_1' + n_1'' + n_2$$

by repeated application of rule [T-STM]. Thus we conclude the following:

- $\Gamma, \Delta \vdash S_1 :: S_2 :: Q : n_1' + n_1'' + n_2$ as argued above,
- $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
- $\Gamma, \Delta \vdash \mathsf{env}_V$ by assumption, since the transition does not alter env_V,
- $n_1' + n_1'' + n_2 < g$, since we know that $n_1' + n_1'' + 1 + n_2 < g$, and the transition does not consume any gas.

- Suppose that [IF] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} :: Q, \mathsf{env}_{SV}, g \rangle \to \langle S_b :: Q, \mathsf{env}_{SV}, g - 1 \rangle$$

Furthermore, $\Gamma, \Delta \vdash \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F} :: Q : n$ was concluded by [T-STM], where $n = n_1 + n_2$, and with premises

$$\Gamma, \Delta \vdash \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F} : n_1$$
$$\Gamma, \Delta \vdash Q : n_2$$

of which the first was concluded by [T-IF]. Thus $n_1 = \max(n_\mathsf{T}, n_\mathsf{F}) + 1$ from the conclusion of that rule, and

$$\Gamma, \Delta \vdash S_\mathsf{T} : n_\mathsf{T}$$
$$\Gamma, \Delta \vdash S_\mathsf{F} : n_\mathsf{F}$$

from its premises. Say that $n_b = \max(n_\mathsf{T}, n_\mathsf{F})$, then $n = n_b + 1 + n_2$.
Finally, $\mathbf{E}^\Gamma_\Delta(\texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F} :: Q) = \Delta$ by case (7.6) of Definition 52. Thus we can conclude the following:

- $\Gamma, \Delta \vdash S_b :: Q : n_b + n_2$, which we conclude by [T-STM].
- $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S.
- $\Gamma, \Delta \vdash \mathsf{env}_V$ by assumption, since the transition does not alter env_V.
- $n_b + n_2 < g - 1$, since we know that $n_b + 1 + n_2 < g$.

- Suppose that [FOR$_\mathsf{T}$] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle \texttt{for}\ e\ \texttt{do}\ S :: Q, \mathsf{env}_{SV}, g \rangle$$
$$\rightarrow \langle S :: \texttt{for}\ v'\ \texttt{do}\ S :: Q, \mathsf{env}_{SV}, g - 1 \rangle$$

with $\mathsf{env}_{SV} \vdash e \rightarrow v$, $v \geq 1$, and $v' = v - 1$ as premises. Then $\Gamma, \Delta \vdash \texttt{for}\ e\ \texttt{do}\ S :: Q : n$ was concluded by [T-STM], where

$$n = \max(1, u(n'_1 + 1) + 1) + n_2$$

and with premises

$$\Gamma, \Delta \vdash \texttt{for}\ e\ \texttt{do}\ S : \max(1, u(n'_1 + 1) + 1)$$
$$\Gamma, \Delta \vdash Q : n_2$$

of which the first was concluded by [T-FOR]. From the premises of this rule, we get that

$$\Gamma, \Delta \vdash e : \mathsf{int}^u_\ell$$
$$\Gamma, \Delta \vdash S : n'_1$$

As we know that $v \geq 1$, this implies that $u \geq 1$, since $v \leq u$ by Lemma 56. Thus

$$\max(1, u(n_1' + 1) + 1) = u(n_1' + 1) + 1$$

and therefore

$$n = u(n_1' + 1) + 1 + n_2$$

We can rewrite this as

$$n = (u - 1)(n_1' + 1) + (n_1' + 1) + 1 + n_2$$

Assume w.l.o.g. that $u = v$, since this is the worst case w.r.t. the number of steps required. Then $u - 1 = v - 1 = v'$, and we can therefore rewrite the above as

$$n = v'(n_1' + 1) + (n_1' + 1) + 1 + n_2$$

By case (7.6) of Definition 52, we have that $\mathbf{E}_{\Delta}^{\Gamma}(\texttt{for}\ e\ \texttt{do}\ S\ ::\ Q) = \Delta$. Furthermore, $\Gamma, \Delta \vdash v' : \mathrm{int}_{v'}^{v'}$ by rule [T-VAL]. We can then conclude the following:

$\Gamma, \Delta \vdash \texttt{for}\ v'\ \texttt{do}\ S : \max(1, v'(n_1' + 1) + 1)$ by [T-FOR]
$\Gamma, \Delta \vdash \texttt{for}\ v'\ \texttt{do}\ S :: Q : \max(1, v'(n_1' + 1) + 1) + n_2$ by [T-STM]
$\Gamma, \Delta \vdash S :: \texttt{for}\ v'\ \texttt{do}\ S :: Q : n_1' + \max(1, v'(n_1' + 1) + 1) + n_2$ by [T-STM]

Say that $n' = n_1' + \max(1, v'(n_1' + 1) + 1) + n_2$. As we know that v is positive, we also know that v' cannot be negative. We then distinguish two cases:

1. If $v' = 0$, then $\max(1, v'(n_1' + 1) + 1) = 1$, and so $n' = n_1' + 1 + n_2$. Clearly, $n' < n - 1$; as we know that $n < g$, we therefore have that $n' < g - 1$.
2. If $v' > 0$, then $\max(1, v'(n_1' + 1) + 1) = v'(n_1' + 1) + 1$. Assuming still the worst case $u = v$, we thus have that

$$n = v'(n_1' + 1) + (n_1' + 1) + 1 + n_2$$
$$n' = v'(n_1' + 1) + (n_1' + 1) + n_2$$

   so $n' = n - 1$. Then, as we know that $n < g$, we therefore also have that $n' < g - 1$.

Thus we can conclude the following:

- $\Gamma, \Delta \vdash S :: \texttt{for}\ v'\ \texttt{do}\ S :: Q : n'$ as argued above.
- $\Gamma \vdash \mathrm{env}_S$ by assumption, since the transition does not alter env_S.

  - $\Gamma, \Delta \vdash \mathsf{env}_V$ by assumption, since the transition does not alter env_V.
  - $n' < g - 1$ as argued above.

- Suppose that [FOR$_\text{F}$] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle \texttt{for } e \texttt{ do } S :: Q, \mathsf{env}_{SV}, g \rangle \to \langle Q, \mathsf{env}_{SV}, g - 1 \rangle$$

  with $\mathsf{env}_{SV} \vdash e \to v$ and $v < 1$ as premises. Then $\Gamma, \Delta \vdash \texttt{for } e \texttt{ do } S :: Q : n$ was concluded by [T-STM], where $n = \max(1, u(n_1' + 1) + 1) + n_2$, and with premises

$$\Gamma, \Delta \vdash \texttt{for } e \texttt{ do } S : \max(1, u(n_1' + 1) + 1)$$
$$\Gamma, \Delta \vdash Q : n_2$$

  of which the first was concluded by [T-FOR]. From the premises of this rule, we get that

$$\Gamma, \Delta \vdash e : \mathsf{int}_\ell^u$$
$$\Gamma, \Delta \vdash S : n_1'$$

  We distinguish three cases:

  1. e is just a single number v (this would e.g. be the case if the loop had previously executed at least once). In that case, $\Gamma, \Delta \vdash v : \mathsf{int}_v^v$ by rule [T-VAL] and, as we know $v < 1$, then $\max(1, v(n_1' + 1) + 1) = 1$, hence $n = 1 + n_2$; since $n < g$, we therefore have that $n_2 < g - 1$.
  2. e is a complex expression (containing operations and/or variables) and $u \geq 1$ (this is possible even though $v < 1$). Then $\max(1, u(n_1' + 1) + 1) = u(n_1' + 1) + 1$, hence $n = u(n_1' + 1) + 1 + n_2$. Clearly, $1 < u(n_1' + 1) + 1$. As we know that $n < g$, we therefore have that $n_2 < g - 1$.
  3. e is a complex expression (as in the second case), but $u < 1$. In this case, $\max(1, u(n_1' + 1) + 1) = 1$, hence $n = 1 + n_2$ as in the first case. Thus, as we know that $n < g$, we therefore have that $n_2 < g - 1$.

  By case (7.6) of Definition 52, we have that $\mathbf{E}_\Delta^\Gamma(\texttt{for } e \texttt{ do } S :: Q) = \Delta$. Thus we can conclude the following:

  - $\Gamma, \Delta \vdash Q : n_2$ by assumption.
  - $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S.
  - $\Gamma, \Delta \vdash \mathsf{env}_V$ by assumption, since the transition does not alter env_V.

– $n_2 < g - 1$, as argued above.

- Suppose that [DECV] was used; then (7.7) is

$$\begin{aligned}\mathsf{env}_T \vdash \langle \texttt{B}\ x\ \texttt{:=}\ e\ \texttt{in}\ S :: Q, \mathsf{env}_{SV}, g\rangle \\ \to \langle S :: \mathsf{del}(x) :: Q, \mathsf{env}_S, ((x, v, B), \mathsf{env}_V), g - 1\rangle\end{aligned}$$

Furthermore, $\Gamma, \Delta \vdash \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S :: Q : n$ was concluded by [T-STM], where $n = n_1 + n_2$, and with premises

$$\begin{aligned}\Gamma, \Delta \vdash \texttt{B}\ x\ \texttt{:=}\ e\ \texttt{in}\ S : n_1 \\ \Gamma, \Delta \vdash Q : n_2\end{aligned}$$

of which the first was concluded by [T-DECV]. From the premises of this rule, we get that

$$\begin{aligned}\Gamma, \Delta \vdash e : B \\ \Gamma, (\Delta, x : B) \vdash S : n_1'\end{aligned}$$

and from the conclusion that $n_1 = n_1' + 2$. Thus $n = n_1' + 2 + n_2$.

From Definition 52, we get that

$$\begin{aligned}\mathbf{E}^{\Gamma}_{\Delta}(\texttt{B}\ x\ \texttt{:=}\ e\ \texttt{in}\ S :: Q) = \Delta, x : B && \text{by case (7.3)} \\ \mathbf{E}^{\Gamma}_{\Delta, x:B}(\mathsf{del}(x) :: Q) = \Delta && \text{by case (7.4)}\end{aligned}$$

We can then conclude the following:

$$\begin{aligned}&\Gamma, (\Delta, x : B) \vdash \mathsf{del}(x) :: Q : n_2 && \text{by [T-DEL]} \\ &\Gamma, (\Delta, x : B) \vdash S :: \mathsf{del}(x) :: Q : n_1' + n_2 && \text{by [T-STM]} \\ &\Gamma, (\Delta, x : B) \vdash (x, v, B), \mathsf{env}_V && \text{by [T-ENV-V]}\end{aligned}$$

where, in the last case, the premise $\Gamma, (\Delta, x : B) \vdash v : B$ holds by Lemma 56, and is concluded by [T-VAL] (or with a subtyping rule). In sum, we thus have the following:

– $\Gamma, (\Delta, x : B) \vdash S :: \mathsf{del}(x) :: Q : n_1 + n_2$ as argued above,
– $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
– $\Gamma, (\Delta, x : B) \vdash (x, v, B), \mathsf{env}_V$ as argued above,
– $n_1' + n_2 < g - 1$, since we know that $n_1' + 2 + n_2 < g$.

- Suppose that [ASSV] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle x \texttt{ := } e :: Q, \mathsf{env}_{SV}, g\rangle \to \langle Q, \mathsf{env}_S, \mathsf{env}_V[x \mapsto v], g-1\rangle$$

  Furthermore, $\Gamma, \Delta \vdash x \texttt{ := } e :: Q : n$ was concluded by [T-STM], where $n = 1 + n_2$, and with premises

$$\Gamma, \Delta \vdash x \texttt{ := } e : 1$$
$$\Gamma, \Delta \vdash Q : n_2$$

  of which the first was concluded by [T-ASSV]. From the premises of this rule, we get that $\Gamma, \Delta \vdash x : B$ and $\Gamma, \Delta \vdash e : B$. We know that $\mathsf{env}_{SV} \vdash e \to v$. We can then conclude:

$$\begin{aligned} \mathbf{E}^{\Gamma}_{\Delta}(x \texttt{ := } e :: Q) &= \Delta && \text{by case (7.6) of Def. 52}\\ \Gamma, \Delta &\vdash v : B && \text{by Lemma 56}\\ \Gamma, \Delta &\vdash \mathsf{env}_V[x \mapsto v] && \text{by Lemma 54}\end{aligned}$$

  In sum, we thus have the following:

  - $\Gamma, \Delta \vdash Q : n_2$ as argued above,
  - $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
  - $\Gamma, \Delta \vdash \mathsf{env}_V[x \mapsto v]$ as argued above,
  - $n_2 < g - 1$, since we know that $1 + n_2 < g$.

- Suppose that [ASSF] was used; then (7.7) is

$$\begin{aligned}\mathsf{env}_T \vdash &\langle \texttt{this.}p \texttt{ := e} :: Q, \mathsf{env}_{SV}, g\rangle\\ &\to \langle Q, \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]], \mathsf{env}_V, g-1\rangle\end{aligned}$$

  Furthermore, $\Gamma, \Delta \vdash \texttt{this.}p \texttt{ := e} :: Q : n$ was concluded by [T-STM], where $n = 1 + n_2$, and with premises

$$\Gamma, \Delta \vdash \texttt{this.}p \texttt{ := e} : 1$$
$$\Gamma, \Delta \vdash Q : n_2$$

  of which the first was concluded by [T-ASSF]. From the premises of this rule, we get that $\Gamma, \Delta \vdash \texttt{this.}p : B$ and $\Gamma, \Delta \vdash e : B$.

We know from the premises of [ASSF] that $\mathsf{env}_{SV} \vdash e \to v$ and $\mathsf{env}_V(\texttt{this}) = X$ and $\mathsf{env}_S(X) = \mathsf{env}_F$. We can then conclude:

$$\begin{aligned}
\mathbf{E}^{\Gamma}_{\Delta}(\texttt{this}.p \ := \ e :: Q) &= \Delta && \text{by case (7.6) of Def. 52}\\
\Gamma, \Delta &\vdash v : B && \text{by Lemma 56}\\
\Gamma, \Delta &\vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]] && \text{by Lemma 55}
\end{aligned}$$

In sum, we thus have the following:

- $\Gamma, \Delta \vdash Q : n_2$ as argued above,
- $\Gamma \vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]$ as argued above,
- $\Gamma, \Delta \vdash \mathsf{env}_V$ by assumption, since the transition does not alter env_V,
- $n_2 < g - 1$, since we know that $1 + n_2 < g$.

- Suppose that [CALL] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle \texttt{call} \ \ e_1 . f(\tilde{e}) \$ e_2 :: Q, \mathsf{env}_{SV}, g\rangle \to \langle S :: \mathsf{env}_V :: Q, \mathsf{env}'_{SV}, g - 1\rangle$$

Furthermore, $\Gamma, \Delta \vdash \texttt{call} \ \ e_1 . f(\tilde{e}) \$ e_2 :: Q : n$ was concluded by [T-STM], where $n = n_1 + 2 + n_2$, and with premises

$$\begin{aligned}
&\Gamma, \Delta \vdash \texttt{call} \ \ e_1 . f(\tilde{e}) \$ e_2 : n_1 + 2\\
&\Gamma, \Delta \vdash Q : n_2
\end{aligned}$$

of which the first was concluded by [T-CALL]. From the premises of this rule, we get that

$$\begin{aligned}
&\Gamma, \Delta \vdash e_1 : I\\
&\Gamma, \Delta \vdash \tilde{e} : \widetilde{B}\\
&\Gamma, \Delta \vdash e_2 : \mathsf{int}^u_\ell
\end{aligned}$$

and $\Gamma(I)(f) = \mathrm{proc}(\widetilde{B})^u_\ell : n_1$.

We know from the premise of [CALL] that $\mathsf{env}_{SV} \vdash e_1 \to X$, and by Lemma 56 we get that $\Gamma, \Delta \vdash X : I$.

We also know by assumption that $\Gamma \vdash \mathsf{env}_T$, which was concluded by [T-ENV-T]. From its premises and side condition, we get that $\Gamma, \texttt{this} : I \vdash \mathsf{env}_M$, where $\mathsf{env}_T(X) = \mathsf{env}_M$ by the premise of [CALL]. This, in turn, was concluded by [T-ENV-M]; from the premise of this rule, we get that

$$\Gamma, (\texttt{this} : I, \tilde{x} : \widetilde{B}, \texttt{value} : \mathsf{int}^u_\ell, \texttt{sender} : I^\top) \vdash S : n_1$$

and, from the premise of [CALL], that $\mathsf{env}_M(f) = (f, (\tilde{x}, S))$.

Finally, by case (7.5) of Definition 52, we have that

$$\mathbf{E}^{\Gamma}_{\Delta}(e_1 \,.\, f\,(\tilde{e}) : e_2 :: Q) = \texttt{this} : I, \texttt{sender} : I^{\top}, \texttt{value} : \mathsf{int}^{u}_{\ell}, \tilde{x} : \widetilde{B}$$

which gives us the contents of the new type environment after the transition. We can now conclude

$$\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\texttt{call}\ \ e_1 \,.\, f\,(\tilde{e})\,\$e_2 :: Q) \vdash \mathsf{env}_V :: Q : n_2 \qquad \text{by [T-CTX]}$$
$$\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\texttt{call}\ \ e_1 \,.\, f\,(\tilde{e})\,\$e_2 :: Q) \vdash S :: \mathsf{env}_V :: Q : n_1 + n_2 \qquad \text{by [T-STM]}$$

In sum, we thus have the following:

- $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\texttt{call}\ \ e_1 \,.\, f\,(\tilde{e})\,\$e_2 :: Q) \vdash S :: \mathsf{env}_V :: Q : n_1 + n_2$ as argued above,
- $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
- $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\texttt{call}\ \ e_1 \,.\, f\,(\tilde{e})\,\$e_2 :: Q) \vdash \mathsf{env}'_V$ by simple inspection of the new env'_V, which contains only the bindings for the formal parameters of f,
- $n_1 + n_2 < g - 1$, since we know that $n_1 + 2 + n_2 < g$.

- Suppose that [THROW] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle \texttt{throw} :: Q, \mathsf{env}_{SV}, g\rangle \to \langle \mathsf{exc}(\texttt{pge}) :: Q, \mathsf{env}_{SV}, g\rangle$$

and no further transition from the resulting configuration is then possible. We know that $\Gamma, \Delta \vdash \texttt{throw} :: Q : n$ was concluded by [T-STM], where $n = 1 + n_2$, and with premises

$$\Gamma, \Delta \vdash \texttt{throw} : 1$$
$$\Gamma, \Delta \vdash Q : n_2$$

of which the first was concluded by [T-THROW]. By case (7.6) of Definition 52, we have that $\mathbf{E}^{\Gamma}_{\Delta}(\texttt{throw} :: Q) = \Delta$. Thus we can conclude the following:

- $\Gamma, \Delta \vdash \mathsf{exc}(\texttt{pge}) :: Q : 0$ by rule [T-EXC],
- $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
- $\Gamma, \Delta \vdash \mathsf{env}_V$ by assumption, since the transition does not alter env_V,
- $0 < g$, by the side condition of [THROW].

- Suppose that [DELV] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle \mathsf{del}(x) :: Q, \mathsf{env}_S, (x, v, B) : \mathsf{env}_V, g\rangle \to \langle Q, \mathsf{env}_{SV}, g\rangle$$

  Furthermore, $\Gamma, \Delta, x : B \vdash \mathsf{del}(x) :: Q : n$ was concluded by [T-DEL]. From its premise, we have that

$$\Gamma, \mathbf{E}^{\Gamma}_{\Delta, x:B}(\mathsf{del}(x) :: Q) \vdash Q : n$$

  and by case (7.4) of Definition 52, we have that

$$\mathbf{E}^{\Gamma}_{\Delta, x:B}(\mathsf{del}(x) :: Q) = \Delta$$

  so we know that $\Gamma, \Delta \vdash Q : n$. Finally, by Lemma 53 and $\Gamma, (\Delta, x : B) \vdash (x, v, B) : \mathsf{env}_V$, we obtain that $\Gamma, \Delta \vdash \mathsf{env}_V$. Thus we can conclude the following:

  - $\Gamma, \Delta \vdash Q : n$ as argued above,
  - $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
  - $\Gamma, \Delta \vdash \mathsf{env}_V$ as argued above,
  - $n < g$ by assumption, from the side condition of [T-CFG], which was used to type the initial configuration.

- Suppose that [RETURN] was used; then (7.7) is

$$\mathsf{env}_T \vdash \langle \mathsf{env}'_V :: Q, \mathsf{env}_{SV}, g\rangle \to \langle Q, \mathsf{env}_S, \mathsf{env}'_V, g\rangle$$

  and $\Gamma, \Delta \vdash \mathsf{env}'_V :: Q : n$ was concluded by [T-CTX]. From its premise, we have that

$$\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\mathsf{env}'_V :: Q) \vdash Q : n$$

  and, by cases (7.1) and (7.2) of Definition 52, we have that $\mathbf{E}^{\Gamma}_{\Delta}(\mathsf{env}'_V :: Q)$ yields a type environment for local variables which is formed by the variable names and the associated type information stored in env'_V. Clearly, $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\mathsf{env}'_V :: Q) \vdash \mathsf{env}'_V$ for any Q. Thus we can conclude the following:

  - $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\mathsf{env}'_V :: Q) \vdash Q : n$ as argued above,
  - $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
  - $\Gamma, \mathbf{E}^{\Gamma}_{\Delta}(\mathsf{env}'_V :: Q) \vdash \mathsf{env}'_V$ as argued above,
  - $n < g$ by assumption, from the side condition of [T-CFG], which was used to type the initial configuration.

In all of the above cases, we have shown that we can satisfy the premises of [T-CFG] after the transition step, whenever the step is concluded by any of the ordinary transition rules for statements. What remains now is to consider the exception rules:

- Suppose the transition was concluded by [OOG]. Then (7.7) is

$$\mathsf{env}_T \vdash \langle S :: Q, \mathsf{env}_{SV}, 0\rangle \to \langle \mathsf{exc}(\mathtt{oog}) :: S :: Q, \mathsf{env}_{SV}, 0\rangle$$

  This case holds vacuously, since $g = 0$, but from the side condition of [T-CFG] we know that $n < g$, which would imply that n is negative, which is impossible. Thus $\langle S :: Q, \mathsf{env}_{SV}, 0\rangle$ cannot be well-typed for any Γ, Δ.

- Suppose any other exception rule was used. Then (7.7) is

$$\mathsf{env}_T \vdash \langle S :: Q, \mathsf{env}_{SV}, g\rangle \to \langle \mathsf{exc}(l) :: S :: Q, \mathsf{env}_{SV}, g\rangle$$

  where l is one of the two labels `rte` or `neg`. In either case, we have that

$$\Gamma, \Delta \vdash \mathsf{exc}(l) :: S :: Q : 0$$

  for any Γ, Δ by rule [T-EXC]. Thus we can conclude the following:

  - $\Gamma, \Delta \vdash \mathsf{exc}(l) :: S :: Q : 0$ as argued above,
  - $\Gamma \vdash \mathsf{env}_S$ by assumption, since the transition does not alter env_S,
  - $\Gamma, \Delta \vdash \mathsf{env}_V$ by assumption, since the transition does not alter env_V,
  - $0 < g$, since we know that g is positive.

This concludes the proof. □

8 A Semantic Approach to Security-Type Safety[1]

In 2017, a vulnerability in a smart contract on the Ethereum platform led to the theft of approximately 31 million dollars' worth of Ether.[2] This vulnerability, known as the *Parity Multisig Wallet attack*, involved an intricate interplay between two peculiar, low-level features of the Solidity smart-contract language: *delegate calls* and *fallback functions*, which can be used for library development to facilitate code reuse. The attacker took advantage of a missing visibility modifier on an initialisation method in a library contract, which was used to set the owner of the contract. The method should have been marked as `internal`, but the missing modifier meant that it defaulted to `public` visibility. This allowed the attacker to invoke this method on contracts *using* the library, thereby enabling him to acquire ownership of these contracts and drain them of all funds. Notably, this omission did not cause the library contract *itself* to become 'unsafe,' if by 'safety' we mean preventing unauthorised access to the funds stored in a contract. Rather, it induced a vulnerability in every contract that *depended* on this library. This is important, because it indicates that this notion of safety is neither purely syntactical, nor always compositional. Later in this chapter we shall see how to formally express under which circumstances safety *is* preserved by composition.

The root cause of the aforementioned vulnerability was an unintended and unwanted flow of data from an untrusted source to a trusted variable, which was used to authorise access to contracts using the library. In terms of security, we would say that the *integrity* of that variable had been compromised. Preserving integrity is integral in ensuring non-interference [47], and in [130] Volpano, Smith and Irvine created a type system for ensuring non-interference, using the concept of *security types*; i.e. types for disciplining the data flow between memory segments of different security levels. This line of work was taken up by the authors of [58], who created a type system with security types for a Solidity-like language in an attempt to prevent such unwanted data flows.

Unfortunately, despite using the Parity Multisig Wallet attack as their motivating example, the authors of the type system in [58] do not actually provide a type rule for

[1] This chapter is unpublished work.

[2] See [77] for a code example with comments explaining the details of the attack.

typing the fallback function, which was a key factor in enabling the aforementioned vulnerability.[3] This is hardly surprising, because the fallback function in effect enables a limited form of *reflection*, or at least introspection, which is notoriously difficult to handle using a syntactic type system, as we also illustrated in Chapter 3. The source of this problem is that the fallback function is invoked when a call is issued to a non-existing method, and the body of the fallback function can then use information *about* the failed call to determine how the computation shall proceed. This information is only available at runtime, hence it cannot be assigned a static type.

In Chapter 6, we also created a type system for ensuring non-interference in TinySol, based on the work of Volpano et al. [130]. Unfortunately, as we shall argue in Section 8.1, we cannot simply extend the type system of Chapter 6 with extra type rules to make the fallback function typable, due to the reflective features of this construct. This is a limitation of the so-called *syntactic* or *subject-reduction* approach to type soundness, that we employed in Chapter 6. In the present chapter, we shall instead use a different approach, known as the *semantic approach to type soundness*, or just *semantic typing* for short, which is capable of reasoning about even seemingly 'unsafe' programming constructs such as reflection. We describe the differences between syntactic and semantic typing in some detail in Section 8.2; however, briefly, the idea is as follows:

A syntactic type system consists of a collection of syntax-directed type rules, which constitute a (usually incomplete) proof system for showing well-typedness of a given term (i.e. a value, an expression, a statement, a method definition, or a contract), based on its syntactic structure. Well-typedness *implies* type safety, but it is seldom *equal* to type safety, due to incompleteness of the proof system. However, suppose that a term can be shown to preserve the safety properties *implied* by well-typedness, even though it may not be possible to *prove* well-typedness itself by means of the ordinary syntactic type rules. The term would reside in the *slack* of the type system, because the fact of its safety is *true* but not *provable*. Such slack is an unavoidable consequence of an incomplete proof system.

Suppose now that we modified our type rules, so that their premises only require the constituents of a term to *be* type safe, rather than requiring that type safety must be *provable* by the proof system. If we can show that type safety itself (and not just well-typedness) is preserved by the composing constructs of the language, this will then allow us to supply alternative proofs of safety for the premises, in the case of unsafe code such as the fallback function. In effect, each type rule now becomes a compatibility lemma to be proved, leading to a different and more modular proof structure than the monolithic induction proofs of the subject-reduction approach.

[3] On a separate note, the type system in [58] also does not assign types to contracts. It fails to take into account the fact that contract addresses can be passed as values, and method calls can be issued *to* such addresses, which in effect means that the *path* to the called method can be changed. This can be used to induce a difference in the 'low' segment of the memory, depending on changes in the 'high' segment, which thus breaks the non-interference property.

The semantic approach to type soundness originated in the works of Milner [81], and Milner and Tofte [84], but was superseded by the subject-reduction approach introduced by Wright and Felleisen [133]. However, semantic typing has recently seen some renewed interest [8; 9; 7; 23; 67; 123], especially in the area of proof-carrying code [88]. On the other hand, and unrelated to semantic typing, Dickerson et al. [38] proposed that proof-carrying code might be particularly well-suited to use in a blockchain setting, because the immutable nature of the blockchain ensures that neither the code nor the proof can be tampered with. We can merge these two ideas as follows: Suppose a contract makes use of a syntactically untypable construct, such as a fallback function, but does it in such a way that it can be shown to *semantically* preserve type safety. Such a proof can then be supplied along with the code of the contract containing the fallback function, when it is declared on the blockchain, as a form of proof-carrying code. The immutable nature of the blockchain environment ensures that neither the code nor the proof can subsequently be altered, so as long as the proof can be verified by any *user* of that contract (or rather, by the type checker), the untypable fragment of code does not itself need to be type checked using the syntactic rules. Of course, *finding* such a proof may then be challenging, but this obligation now falls on the contract creator, rather than on the user of the contract, who only needs to check the validity of the provided 'proof certificate' of type safety. In this way, we may not be able to make the fallback function itself typable by syntactic means, but we *can* provide a method for reasoning about, and ensuring, type safety of contracts *using* this construct.

The rest of this chapter is structured as follows: Section 8.1 gives an overview of the Parity Multisig Wallet attack and describes the necessary changes to the TinySol model, that will allow us to represent this vulnerability. We give some examples that also illustrate why the syntactic approach to type soundness from Chapter 6 is insufficient to handle the problem. Then, in Section 8.2, we give a more detailed introduction to the semantic approach to type soundness and also discuss some related works. This sets the stage for the developments later in this chapter.

In Section 8.3, we describe the syntax and small-step semantics of an extended version of TinySol, which includes the two new constructs; delegate calls and fallback functions. Section 8.4 then presents an updated version of the syntactic type system from Chapter 6, which has been extended with a type rule for delegate calls, and adapted to our use of a small-step semantics in the present chapter, in contrast to the big-step semantics of Chapter 6. Additionally, we introduce a new subtyping relation based on static inheritance for interfaces, rather than using structural subtyping, as we did in Chapter 6.

The purpose of the syntactic type system is to act as a reference for our developments a type system based on the *semantic* approach to type soundness. This is carried out in Sections 8.5–8.6, which give the semantics of types for expressions and stacks, including statements, respectively. The semantics of types is given in terms of a typed semantics, and we use it to give a characterisation of type safety for the two

$$[\textsc{dcall}]\frac{\begin{array}{c}\mathsf{env}_{SV} \vdash e \to Y \\ \mathsf{env}_{SV} \vdash \tilde{e} \to \tilde{v} \\ \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, \mathsf{env}''_V \rangle \to \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \texttt{dcall}\ \ e.f(\tilde{e}), \mathsf{env}_{SV} \rangle \to \mathsf{env}'_S, \mathsf{env}_V}$$

where:

$$\begin{aligned} |\tilde{x}| &= |\tilde{v}| = k \\ (\tilde{x}, S) &= \mathsf{env}_T(Y)(f) \\ \mathsf{env}''_V &= \texttt{this} : \mathsf{env}_V(\texttt{this}), \texttt{sender} : \mathsf{env}_V(\texttt{sender}), \\ &\qquad \texttt{value} : \mathsf{env}_V(\texttt{value}), x_1 : v_1, \ldots, x_k : v_k \end{aligned}$$

Figure 8.1: Big-step semantics of delegate calls.

syntactic categories, based on sets of expressions and stacks. Then, in Section 8.7, we show how these developments may be used to ensure safety of some usages of the fallback function construct. Finally, we summarise our findings from this chapter in Section 8.8, and discuss some avenues for future works.

Longer proofs are deferred to the end of the chapter, to avoid breaking up the main narrative of the text. They can be found in Section 8.9.

8.1 Modelling the Parity Multisig Wallet attack

In order to be able to model the Parity Multisig Wallet attack in TinySol, we need to first extend the language with two new constructs; namely, *delegate calls* and *fallback functions*. This requires a number of changes to the TinySol model, in particular its semantics, so here we shall first describe and discuss each feature in relation to the version of TinySol given in Chapter 6 to motivate and illustrate the necessary changes, before providing a full semantics for this extended version of the language.

8.1.1 Delegate calls

Besides the ordinary method call, Solidity also features another kind of method call with a different calling style, known as the *delegate call*. The purpose of this construct is to facilitate code reuse and creation of contract libraries. This is desirable, since the cost of deploying a smart contract on the Ethereum chain depends on the code size,[4] which thus encourages developers to use libraries for common tasks. The special feature of this call style is that the method body is executed in the context of the *caller* contract, rather than in the context of the contract in which it is declared, which allows the calling contract to use the called method *as if* it were part of the calling contract itself.

[4] See the description of the `CREATE` and `CREATE2` opcodes in [119].

$$[\text{T-DCALL}]\frac{\begin{array}{l}\Gamma \vdash (I_1, s_1) <: (I_2, s_2)\\ \Gamma;\Delta \vdash e : (I_2, s_2)\\ \Gamma;\Delta \vdash \tilde{e} : \widetilde{B}\\ \Gamma;\Delta \vdash e.f : \text{proc}(\widetilde{B}) : s\end{array}}{\Gamma;\Delta \vdash \texttt{dcall}\ e.f(\tilde{e}) : \text{cmd}(s)}\ (\Delta(\texttt{this}) = \text{var}(I_1, s_1))$$

Figure 8.2: Type rule for delegate calls.

We can extend the version of TinySol given in Chapter 6 with a delegate call construct as follows: First, we add the construct to the syntax of statements:

$$S ::= \ldots \mid \texttt{dcall}\ e.f(\tilde{e})$$

We can then extend the semantics of TinySol from Chapter 6 with a big-step semantic rule for this construct, which is given in Figure 8.1. There are a few things to note about the semantics of this construct:

- The method body S is executed in the context of the caller; i.e. the magic variables `this`, `sender` and `value` are *not* rebound. In particular, this means that any access to fields (through `this`) are to fields in the *caller* contract, rather than the contract Y in which f is defined.
- There is no `value` parameter, so funds cannot be transferred to the callee with this type of call. This is consistent with the idea that the code is executed in the context of the caller; i.e. *as if* the method had been declared in the calling contract. With the conventional `call` construct, calls to local methods *can* in principle transfer funds, but this has no effect, since the caller and callee are the same contract. As delegate calls are similar to local calls, except for the fact that the actual code resides in another contract, it therefore makes sense that there is no `value` parameter in the call syntax.

Finally, we can extend the type system of Chapter 6 with a type rule for delegate calls, which is given in Figure 8.2. The important point to note here is the subtyping judgment $\Gamma \vdash (I_1, s_1) <: (I_2, s_2)$. We must require that the interface type of the caller contract is a subtype of the contract containing the definition of f. This is necessary, because the body of f might access fields or call methods via `this`, but when the body is executed in the context of the *caller*, this would lead to a runtime error, if the calling contract does not contain fields or methods with the exact same names (and of the same types). This situation is prevented by requiring the caller contract to be a subtype (i.e. a specialisation) of the callee contract, which thus ensures that the caller contract will contain *at least* the same methods and fields as the callee contract.

Note that this implies that the *caller contract* also must contain a method by the same name f, and with the same signature (or a subtype thereof). We could remove

```
contract C {
  field owner := Bob;
  field impl := X;

  func Update(addr) {
    if sender = this.owner then this.impl := addr else skip
  }

  func f_1(x̃) { dcall this.impl.f_1(x̃) }
  ⋮
  func f_n(x̃) { dcall this.impl.f_n(x̃) }
}
```

Figure 8.3: The *pointer to implementation* pattern.

this limitation by introducing a more complicated type rule, but we shall forego that here, since it is consistent with the purpose of `dcall`, i.e. code reuse. A natural way to use `dcall` would be as a body of a stub method declaration, e.g.:

```
func f(x̃) {
  dcall X.f(x̃)
}
```

which simply forwards the call to an implementation residing in X, rather than locally. One could also easily introduce some syntactic sugar for this kind of stub implementations.

A related usage is the so-called *pointer to implementation* pattern described in [38], which is illustrated in Figure 8.3. This pattern can be used to allow a contract to update its implementation, despite the immutable nature of a blockchain. It consists of a contract `C`, which acts as a proxy for the functionality provided by a different contract, which resides on the address `X`. This address is stored in the field `impl`, and `C` simply forwards all calls to methods f_1, ... f_n to their actual implementations in `X`. Finally, `C` provides an `Update` method, which allows the contract owner `Bob` to overwrite the address stored in `impl`. Thus, if the actual implementation needs to be updated, `Bob` can deploy a new version of `X` and then simply update the pointer to the implementation in `C` without affecting any users of the contract.

8.1.2 The fallback function

The *fallback function* is a peculiar feature of Solidity, which allows a contract creator to add a nameless function to a contract, that will be called in case an attempt is made to call a non-existing method on the contract. This ensures that the execution does

$$[\textsc{fcall}]\frac{\begin{array}{c}\mathsf{env}_{SV} \vdash e_1 \to Y \\ \mathsf{env}_{SV} \vdash \tilde{e} \to \tilde{v} \\ \mathsf{env}_{SV} \vdash e_2 \to n \\ \mathsf{env}_T \vdash \langle S, \mathsf{env}''_{SV} \rangle \to \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \texttt{call}\ e_1.m(\tilde{e})\$e_2, \mathsf{env}_{SV} \rangle \to \mathsf{env}'_S, \mathsf{env}_V}$$

where:

$$\begin{aligned} X &= \mathsf{env}_V(\texttt{this}) \\ \mathsf{env}^X_F &= \mathsf{env}_S(X) \\ \mathsf{env}^Y_F &= \mathsf{env}_S(Y) \\ n &\leq \mathsf{env}^X_F(\texttt{balance}) \\ f &= \begin{cases} \mathsf{env}_V(m) & \text{if } m = \texttt{id} \\ m & \text{otherwise} \end{cases} \\ f &\notin \mathrm{dom}\,(\mathsf{env}_T(Y)) \\ (\epsilon, S) &= (\mathsf{env}_T(Y))(\texttt{fallback}) \\ \mathsf{env}''_V &= \texttt{this} : Y, \texttt{sender} : X, \texttt{value} : n, \texttt{id} : f, \texttt{args} : \tilde{v} \\ \mathsf{env}''_S &= \mathsf{env}_S\big[X \mapsto \mathsf{env}^X_F[\texttt{balance -= } n]\big]\big[Y \mapsto \mathsf{env}^Y_F[\texttt{balance += } n]\big] \end{aligned}$$

Figure 8.4: Big-step semantics of the fallback function.

not get stuck, which would otherwise lead to a failed transaction. To add this feature in TinySol, we need to make a few changes:

- We shall assume that all contracts by default contain a parameterless method

  ```
  fallback() { skip }
  ```

  with a body that does nothing, and that this definition can be overridden by the contract creator by providing a different implementation. For the sake of simplicity, we can assume that this method cannot be directly invoked.

- We add the set of method names MNames to the set of values. This is necessary, because the fallback function will contain two extra 'magic variables', called `id` and `args`, which are the name of the (non-existing) called method, and the list of provided arguments.

- We assume that `fallback`, `id` and `args` are reserved keywords in their respective contexts; namely, inside contract definitions for `fallback`, and inside the body of the fallback function for `id` and `args`.

- We use m to range over MNames $\cup \{\,\texttt{id}\,\}$ and replace the syntax for call with `call` $e_1.m(\tilde{e})\$e_2$. We also need to assume that an extra side condition is

```
contract Proxy {
  field owner := Bob;
  field impl := X;

  func Update(addr) {
    if sender = this.owner then this.impl := addr else skip
  }

  func fallback() { dcall this.impl.id(args) }
}
```

Figure 8.5: Forwarding method calls.

added to the ordinary call rule and the rule for delegate calls:

$$f = \begin{cases} \mathrm{env}_V(m) & \text{if } m = \texttt{id} \\ m & \text{otherwise} \end{cases}$$

Thus we can use the ordinary call rule, or the delegate call rule, with both ordinary method names f, and, in the special case of calls inside the body of the fallback function, with the 'magic variable' `id`.

Now, to add the fallback function, we need an extra semantic rule, given in Figure 8.4. This rule is similar to the ordinary call-rule, except that `id` and `args` are now also bound within the newly created env_V. As can be seen, the rule is applicable exactly when $\mathrm{env}_T(Y)$ does *not* contain a definition for the called method f, and instead, the `fallback` function is invoked. Apart from that, the value transfer, and any other bindings, are as in any ordinary method call.

The `id` 'magic variable' in effect admits a limited form of reflection (or more precisely, introspection) into the language, because the body of the fallback function can use information *about* the failed call – namely the name of the non-existing method – to determine the future of the computation. For example, it could be used to forward the call to an arbitrary other contract, which itself may, or may not, contain a method with the same name and with the same signature. Thus, something like the following is possible:

```
  func fallback() {
    call X.id(args);
  }
```

This can be used to implement a form of inheritance, by delegating all calls to non-existing members to a parent contract. Consider again the "pointer to implementation"

```
contract Lib {
  func init(X) { this.owner := X; }
  ⋮
}

contract Wallet {
  field owner := Bob;
  field impl := Lib;

  func fallback() { dcall this.impl.id(args) }
}

contract Attacker {
  func attack() { call Wallet.init(this) }
}
```

Figure 8.6: Simplified representation of the Parity Multisig Wallet attack.

example in Figure 8.3. The same functionality can now be achieved more succinctly by replacing all the stub implementations of f_1, ..., f_n with a single delegate call in a callback function, as in Figure 8.5. Note, however, that this will forward *all* calls to non-existing methods, and not just those to f_1, ..., f_n, which may not always be desirable.

8.1.3 The Parity Wallet Attack

With our new extensions of the language, we are now finally in a position to model the Parity Multisig Wallet attack in TinySol. Consider the code in Figure 8.6, which is similar to the motivating example in [58]: The `Lib` contract provides some common functionality to implement a wallet; for example, it includes an `init` method that sets a field `owner` to the received address. The contract `Wallet`, which is owned by `Bob`, makes use of this functionality, so it forwards all calls to unimplemented methods by using its `fallback` function and delegate calls, which means that the called method in `Lib` will be executed in the context of `Wallet`. Thus far, the example is similar to the one in Figure 8.5. The attack then consists of the following three steps:

1. When `Attacker` calls `Wallet.init(this)`, the call is forwarded to `Lib`, since `Wallet` does not contain a method called `init`.

2. Instead, `Lib.init(Attacker)` is executed, which then executes the assignment `this.owner := Attacker`, but because this is a *delegate call*, the magic

variable `this` is still bound to `Wallet`.

3. Thus, the field `Wallet.owner` is modified to now contain the address of the `Attacker`.

If the field `Wallet.owner` is used for access control to guard other functionalities in `Wallet`, this access control has thus been circumvented.

The key problem in the code in Figure 8.6 is an unwanted data flow to a variable that should not have been modifiable; i.e. the *integrity* of that variable is compromised. In Chapter 6, we developed a type system for TinySol for ensuring integrity (or, conversely, secrecy), and clearly, the aforementioned example falls within the set of programs that should be rejected by such a type system, due to the flow from an untrusted source into a trusted segment of the memory.

However, the example is unfortunately untypable by the type system from Chapter 6, even if it were extended with the type rule for delegate calls given in Figure 8.2. The problem is precisely that the magic variable `id` could be bound to any method name at runtime, and we therefore cannot statically obtain a signature for it from the type environments $\Gamma; \Delta$. However, the type system from Chapter 6 already prevents calls to non-existing contract members by means of its interface types I, so in practice, the fallback function would never be called in a well-typed program. This is of course doubly problematic, because the fallback function already *is* a part of the Solidity language, so the type system from Chapter 6 is too restrictive to be applicable to this language, unless we are willing to reject all contracts that make use of a fallback function, such as the (legitimate) usage in Figure 8.5. Thus, in order to actually make the fallback function usable at all, we would have to allow calls to non-existing methods, rather than rejecting such programs as ill-typed; but as the body of the fallback function is untypable, this would then break type safety, thereby defeating the purpose of the type system. Neither of the two approaches seem like a satisfying solution.

Not all usages of the fallback function cause unsafe data flows, but the type system does not allow us to distinguish between the safe and unsafe usages. Even worse, it does not even provide us with the means to *reason* about whether a given usage would be safe within a particular program, if the body of the fallback function makes any use of the magic variables `id` and `args` at all. Consider for example the following definition:

```
1 func fallback() {
2   if id = f then dcall X.f() else dcall X.g()
3 }
```

Suppose we could conclude that

$$\Gamma;\Delta \vdash \texttt{X.f} : \text{proc}():s$$
$$\Gamma;\Delta \vdash \texttt{X.g} : \text{proc}():s$$
$$\Gamma;\Delta \vdash \texttt{id} : \text{var}(s)$$

Then the body of this fallback function *ought to* be typable as cmd(s) as well. If neither `X.f()` nor `X.g()` would cause any unsafe data flows, if they were called directly, then neither would the above. The `if-else` construct encapsulates the usage of the `id` variable, effectively restricting its possible values to `f` or not-`f`, yet even though we may be able to *reason* that the above usage in fact is safe because of this encapsulation, the type system does not allow us to *prove* this by means of the type rules. This, in turn, means that we would have to judge any program *containing* the above definition as being unsafe, even if we were able to show by other means, that this piece of code in fact does not violate the restrictions denoted by the type judgment cmd(s).

The problem is precisely that the type system does not provide a formal definition of what the type judgment cmd(s) denotes. It does not define the *meaning* of the types and type judgments, except indirectly through their usage in the type rules. In other words, the types have a *syntax*, but not a *semantics*. This is a limitation of the approach to type soundness that we have hitherto been using. It does not require such a semantics to be defined, and it cannot make use of it either. In the following, we shall therefore explore a different approach to type soundness, which *does* allow us to take such semantic reasoning into account.

8.2 Syntactic and semantic typing

The approach to type soundness used in all the previous chapters (Chapters 3–7) is variously known as the 'subject-reduction' approach, or the 'syntactic approach to type soundness.' However, in the following we shall simply refer to it as *syntactic typing* for brevity.[5] It was proposed by Wright and Felleisen [133] in 1994, and has since become the de-facto standard approach to showing type soundness.

In syntactic typing, we begin by defining a language of types $\mathcal{T}$, which, in the case of data types in an imperative setting like TinySol, are attached to atomic identifiers representing *data containers* (fields and variables including the formal parameters of methods), and encapsulating constructs (methods and contracts); see also Chapter 2 for a more detailed account. Then we define a *typing relation* $\vdash$ for each syntactic category in the language, for example:

- $\Gamma \vdash v : T$ for values, denoting that v is a value *of* type T,

[5] The terms 'syntactic typing' and 'semantic typing' are used in [123] as synonymous with 'syntactic approach to type soundness' resp. 'semantic approach to type soundness.' We adopt this usage simply because it is shorter.

- $\Gamma \vdash e : T$ for expressions, denoting that e will evaluate *to* a value of type T,
- $\Gamma \vdash S$ for statements, denoting that the statement is well-typed,
- $\Gamma \vdash \texttt{func}\ f(\tilde{x})\ \{\ S\ \} : \text{proc}(\widetilde{T})$ for methods, denoting that the body S of the method is well-typed, *if* the formal parameters $\tilde{x}$ are instantiated with actual parameters of types $\widetilde{T}$, and
- $\Gamma \vdash C : I$ for contracts, denoting that the contract implements the interface I.

Each of the type judgments above are subject to the proviso that all free identifiers occurring in the term on the right-hand side of $\vdash$ must also have a type assignment in the type environment Γ on the left-hand side. This relation is defined by a collection of syntax-directed *type rules*, i.e. inference rules, which constitute a proof system that lets us conclude that a type judgment, e.g. $\Gamma \vdash S$, in fact holds. If $\Gamma \vdash S$ can be proved by the type rules, then we say that S is *well-typed* w.r.t. Γ; and, conversely, if $\Gamma \nvdash S$ we say that S is *ill-typed* w.r.t. Γ. Finally, we must show a result of *subject reduction*, which says that well-typedness is preserved by the semantics; i.e. a statement of the form

$$\Gamma \vdash P \wedge P \to P' \implies \Gamma \vdash P'$$

for some program P. In other words, a well-typed term cannot reduce to an ill-typed term. This ensures that if well-typedness implies 'program safety,' for whatever notion of safety we might be interested in, then a well-typed program is safe and will *remain* safe for all its execution steps.

The great advantage of syntactic typing is the simplicity and straightforwardness of the approach. Yet, as the examples in the preceding section illustrate, this simplicity comes at the price of a narrowly syntactical view of types, that is ill-suited to handle abstractions and encapsulation. These shortcomings have been noted independently by Caires in [23], and, more recently, by Timany et al. in [123]. As the latter group of authors remark, many modern, statically typed programming languages feature an `unsafe` construct or similar, that allows the programmer to disable type checking for a segment of the code, because it needs to perform operations for which well-typedness cannot automatically be concluded; e.g. all pointer operations in C# or raw pointer dereferencing in Rust. Such unsafe code blocks can then be encapsulated in libraries, but that leads to problems in verifying the actual safety of programs using these libraries, because the syntactic approach does not provide a way to reason about such encapsulations. The programmer may 'promise' (by explicitly writing `unsafe {…}`) that his usage of unsafe operations in fact does not break the invariants implied by well-typedness, that are presumed to hold on the data. However, even if this promise can be verified (i.e. outside of the type system), there is no way to make the type checker take this 'external' fact into account. Conversely, the programmer may declare an abstraction and claim that it enforces some variant on its private data;

e.g. 'this method always returns a new number,' but syntactic type soundness has nothing to say about the truth of such a claim.[6]

As a remedy, both Caires [23] and Timany et al. [123] advocate an alternative approach to type soundness, known as the *semantic* approach, or simply *semantic typing*. This approach is in fact older than the syntactic approach; it originated in the works of Milner [81] in 1978, and Milner and Tofte [84] in 1991. In semantic typing, types and type judgments are first given an independent meaning, e.g. as logical formulae, which directly define the properties (notions of safety) denoted by each type. In the words of Timany et al. [123, p. 3]:

> *Under the semantic soundness approach, one defines a semantic model of types, which offers an extensional view of typing rather than an intensional one. In other words, unlike syntactic typing, which dictates the syntactic structure of well-typed terms, semantic typing merely places restrictions on their observable behavior. Accordingly, it enables us to explain when a term behaves safely at a given type, even if the term employs unsafe or low-level operations internally.*

Instead of the syntactic typing relation, e.g. $\Gamma \vdash P$, we define a *semantic typing relation* for terms in the language, written $\Gamma \vDash P$, which denotes that the program P satisfies the type constraints in Γ. In other words, the *truth* of these predicates is asserted, relative to P.[7] Note the use of the single turnstile symbol $\vdash$ for the syntactic judgments, which denotes *syntactic* implication, i.e. provability (by the type rules), versus the double turnstile symbol $\vDash$, which denotes *semantic* implication, i.e. truth (type safety). Given a syntactic type system, and a semantic model of types, we can then proceed to show that

$$\Gamma \vdash P \implies \Gamma \vDash P$$

i.e. provability implies truth, which is the condition for soundness. Such a statement is known as a *compatibility* theorem, and it expresses that the syntactic type system provides a sound approximation to the set of safe terms.

Having the compatibility theorem alone would afford us nothing more than subject reduction. However, the semantic model of types additionally allows us to formulate and prove a collection of *closure properties* of the semantic typing relation. Specifically, we are interested in showing that semantic type safety is closed under the syntactic constructors of the language. As an example, we might want to show that *if* $\Gamma \vDash S_1$ *and* $\Gamma \vDash S_2$ *then* $\Gamma \vDash S_1; S_2$ for a language with sequential composition. This would

[6] Consider the type system in Chapter 3 and our claim that the 'name generator' always will return a fresh name. Here, we had to impose an extra restriction on the shape of all generated names, but actually *verifying* that all encoded processes obeyed this restriction was simply outside the scope of the syntactic type system itself.

[7] Note the similarity with the satisfaction relation in logics. If the type constraints in Γ were considered as a conjunction of logical formulae, one would write $P \vDash \Gamma$ instead. This is actually how Caires writes his type judgments in [23].

correspond directly to a 'semantic type rule' of the form

$$\frac{\Gamma \vDash S_1 \qquad \Gamma \vDash S_2}{\Gamma \vDash S_1; S_2}$$

and each type rule thus becomes a compatibility lemma to be shown (or proved admissible), rather than a collection of more or less arbitrary definitions given *a priori*. Having such type rules would then finally allow us to incorporate different kinds of proofs of safety, because the premises of e.g. the rule above do not require, that $\Gamma \vDash S_1$ and $\Gamma \vDash S_2$ need to be *provable* using these rules. They only need to be *true*.

There are different approaches to defining a semantic model of the types and type judgments. One is the step-indexed model of Appel and McAllester [9], which was further extended by Ahmed in [7]. This approach has been applied in a variety of settings, for example to reason about the safety of 'unsafe code' in core Rust libraries [67]; see [123] for an overview of work based on this approach.

Another approach is based on *typing interpretations*, which are coinductively defined objects corresponding to fragments of transition systems. It is reminiscent of the logical relations technique of Plotkin [104] and Tait [121], and it is the method used by Caires in [23]. Informally, a typing interpretation for a syntactic category, e.g. programs P, w.r.t. a collection of type predicates Γ, is a Γ-indexed family of sets of programs $\mathcal{R}_\Gamma$, defined such that $P \in \mathcal{R}_\Gamma$ implies

- P satisfies the type predicates in Γ, and
- If $P \to P'$ then $P' \in \mathcal{R}_\Gamma$.

The largest typing interpretation for a given Γ is the union of all typing interpretations for Γ, written $\nu\mathcal{R}_\Gamma$, similar to bisimilarity being the union of all bisimulations (cf. also the discussion in Chapter 2, Section 2.2). $\nu\mathcal{R}_\Gamma$ is the *typing* of Γ, and the semantic typing relation is then defined as

$$\Gamma \vDash P \triangleq P \in \nu\mathcal{R}_\Gamma.$$

Proving semantic type safety of P w.r.t. a given Γ thus becomes a matter of finding a typing interpretation for Γ that contains P.

We shall follow the approach of Caires in the present chapter, because it gives us a straightforward way of defining the proofs of type safety for unsafe code, that are to be stored on the blockchain. For this purpose, we can simply store the typing interpretations themselves. However, that then requires that any client *using* a contract with such an associated proof must be able to automatically verify that it indeed is a typing interpretation. We shall discuss this further in Section 8.7.3.

$$
\begin{array}{rcl}
DF \in \mathsf{Dec}_F & ::= & \epsilon \mid \texttt{field p := } v\texttt{;}DF \\
DM \in \mathsf{Dec}_M & ::= & \epsilon \mid \texttt{func } f(\tilde{x})\ \texttt{\{ } S\ \texttt{\}}\ DM \\
DC \in \mathsf{Dec}_C & ::= & \epsilon \mid \texttt{contract } X\ \texttt{\{} \\
 & & \quad \texttt{field balance := } n\texttt{;}\ DF \\
 & & \quad \texttt{func send() \{ skip \}}\ DM \\
 & & \quad \texttt{func fallback() \{ } S\ \texttt{\}} \\
 & & \texttt{\}}\ DC \\
v \in \mathsf{Val} & ::= & \mathbb{Z} \cup \mathbb{B} \cup \mathsf{ANames} \cup \mathsf{MNames} \\
e \in \mathsf{Exp} & ::= & v \mid x \mid e\texttt{.balance} \mid e\texttt{.}p \mid \mathrm{op}(\tilde{e}) \\
 & \mid & \texttt{this} \mid \texttt{sender} \mid \texttt{value} \mid \texttt{id} \mid \texttt{args} \\
S \in \mathsf{Stm} & ::= & \texttt{skip} \mid \texttt{throw} \mid \texttt{var } x \texttt{ := } e \texttt{ in } S \mid x \texttt{ := } e \\
 & \mid & \texttt{this.}p \texttt{ := } e \mid S_1\texttt{;}S_2 \mid \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} \\
 & \mid & \texttt{while } e \texttt{ do } S \mid \texttt{call } e_1\texttt{.}m(\tilde{e})\texttt{\$}e_2 \mid \texttt{dcall } e\texttt{.}m(\tilde{e}) \\
Q \in \mathcal{Q} & ::= & \bot \mid S\texttt{;}Q \mid \mathsf{del}(x)\texttt{;}Q \mid \mathsf{env}_V\texttt{;}Q
\end{array}
$$

where $x, y \in \mathsf{VNames}$ (variable names), $p, q \in \mathsf{FNames}$ (field names), $X, Y \in \mathsf{ANames}$ (address names), $m \in \{\texttt{id}\} \cup \mathsf{MNames}$ (method names).

Figure 8.7: The extended syntax of TinySol.

8.3 Syntax and semantics of TinySol

For the sake of self-containedness, we shall begin by giving a complete definition of the syntax and semantics of an extended version of TinySol that includes our new syntactic constructs. The syntax is given in Figure 8.7; apart from the extensions, it is similar to the one given in Chapter 6. We shall use a small-step semantics in the present chapter to allow non-terminating behaviour, so we also include the syntax of *stacks* Q, similar to the version in Chapter 7.

We omit the full definition of the binding model and the semantics for declarations, since these are similar to the ones in Chapters 6–7. However, the environments env_T, env_S, env_V, and the 'inner' environments env_F, env_M, are all partial functions, but may alternatively be thought of as sets of pairs, and we use the following notation to extend or modify them:

- We write $\mathsf{env}_V, x : v$ to denote the *extension* of env_V with a *new* entry (x, v).
- We write $\mathsf{env}_V[x \mapsto v]$ to denote an env_V, in which a *pre-existing* binding (x, v') is updated to (x, v), thus replacing the previous value v' with v.

We shall use both notations for convenience, sometimes expanding an environment as a list of tuples to replace a value, and sometimes using the update notation.

$$[\textsc{exp-val}]\frac{}{\langle v, \mathsf{env}_{SV}\rangle \to v}$$

$$[\textsc{exp-var}]\frac{}{\langle x, \mathsf{env}_{SV}\rangle \to v}\ (\mathsf{env}_V(x) = v)$$

$$[\textsc{exp-op}]\frac{\langle \tilde{e}, \mathsf{env}_{SV}\rangle \to \tilde{v} \quad \mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v}{\langle \mathrm{op}(\tilde{e}), \mathsf{env}_{SV}\rangle \to v}$$

$$[\textsc{exp-field}]\frac{\langle e, \mathsf{env}_{SV}\rangle \to X}{\langle e.p, \mathsf{env}_{SV}\rangle \to v}\ (\mathsf{env}_S(X)(p) = v)$$

Figure 8.8: Semantics of expressions.

The semantics of expression configurations is given in Figure 8.8. This is similar to the previous presentations, and for the sake of simplicity we here assume that x also ranges over the 'magic variable' names `this`, `sender`, `value`, `id` and `args`. We write the transitions as configurations, $\langle e, \mathsf{env}_{SV}\rangle \to v$, to avoid the use of the single turnstile symbol, but apart from this stylistic change, the rules are the same as in Chapters 6–7.

Next, we give a small-step semantics of statements. We use a slightly modified version of the one from Chapter 7, given in Figure 8.9 and Figure 8.10. The main differences are:

- we omit the gas parameter;
- we omit the rule for sequential composition and instead simply treat stacks as sequentially composed elements;
- we place env_T within configurations, even though it is not altered during transitions, to again avoid the single turnstile symbol; and
- we add rules for delegate calls and fallback calls.
- We omit the side condition $n \leq \mathsf{env}_F^X(\texttt{balance})$ from the call rules [CALL] and [FCALL]. This side condition ensures that a contract's `balance` can never be negative, and if this were the case it should instead trigger an exception as in Chapter 7. However, as exceptions are not our focus in the present chapter, we prefer to simply omit this, rather than introducing two extra call rules for the exceptional transitions.

The semantics is also subject to the same provisos as in Chapter 7, i.e. we use a typed version of the language, with types appearing in declarations of new variables, and we also record the types in env_V, along with the variable name and its value,

although this is not used in the semantics. Thus, for now, we implicitly discard the type information when performing a lookup $\mathsf{env}_V(x) = (v, B_s)$.

8.4 A syntactic type system for security types

In Chapter 6, we created a type system for ensuring secure data flows in TinySol, based on the work of Volpano, Smith and Irvine [130]. Building on this work, we shall now create a more refined version of this type system for our extended version of the language, also taking into account that we are here using a stack-based version of the language, as in Chapter 7. As we make several changes, especially w.r.t. the subtyping relation, we shall give detailed comments on the differences in the following.

8.4.1 The syntax of types and environments

As in Chapter 6, we begin by assuming a lattice $(\mathcal{S}, \sqsubseteq)$ of security levels s, with $s_\top$ as the highest level, and $s_\bot$ as the lowest level. For ease of notation, we shall use the following convention when comparing multiple levels against each other using the ordering relation $\sqsubseteq$:

$$s_1, \ldots, s_n \sqsubseteq s \triangleq s_1 \sqsubseteq s \wedge \ldots \wedge s_n \sqsubseteq s$$
$$s \sqsubseteq s_1, \ldots, s_n \triangleq s \sqsubseteq s_1 \wedge \ldots \wedge s \sqsubseteq s_n$$
$$s_1, \ldots, s_n \sqsubseteq s'_1, \ldots s'_n \triangleq s_1 \sqsubseteq s'_1 \wedge \ldots \wedge s_n \sqsubseteq s'_n$$

We also assume a set TNames of type names (interface names) ranged over by I. Compared to the presentation in Chapter 6, we shall make some changes to the syntax of types:

- We let each data type be a *pair*, (B, s), consisting of a base type B and a security level s, rather than only having such a pair for interfaces. We shall use the notations (B, s) and B_s interchangeably, preferring the latter except when we need indices on types.
- We add a 'dummy' base type idf for identifiers to the set of base types B; this is merely to be able to assign a base type to the `id` magic variable in the fallback function, which contains the name of the non-existing called method.
- We omit the command type $\mathsf{cmd}(s)$ from the language of types, since this is not a type that is associated with any identifiers and therefore does not need to be recorded in the type environment. Specifically, $\mathsf{cmd}(s)$ is a symbol used in *type judgments* for statements, but it is not itself a type.
- We shall use *explicit inheritance* of interfaces for subtyping, as opposed to the structural subtyping in Chapter 6. This necessitates the introduction of an extra environment, Σ, which records the inheritance hierarchy.

$$[\textsc{skip}]\frac{}{\langle \texttt{skip}; Q, \mathsf{env}_{TSV}\rangle \to \langle Q, \mathsf{env}_{TSV}\rangle}$$

$$[\textsc{if}]\frac{\langle e, \mathsf{env}_{SV}\rangle \to b \in \mathbb{B}}{\langle \texttt{if } e \texttt{ then } S_{\mathsf{T}} \texttt{ else } S_{\mathsf{F}}; Q, \mathsf{env}_{TSV}\rangle \to \langle S_b; Q, \mathsf{env}_{TSV}\rangle}$$

$$[\textsc{while}_{\mathsf{T}}]\frac{\langle e, \mathsf{env}_{SV}\rangle \to \mathsf{T}}{\langle \texttt{while } e \texttt{ do } S; Q, \mathsf{env}_{TSV}\rangle \to \langle S; \texttt{while } e \texttt{ do } S; Q, \mathsf{env}_{TSV}\rangle}$$

$$[\textsc{while}_{\mathsf{F}}]\frac{\langle e, \mathsf{env}_{SV}\rangle \to \mathsf{F}}{\langle \texttt{while } e \texttt{ do } S; Q, \mathsf{env}_{TSV}\rangle \to \langle Q, \mathsf{env}_{TSV}\rangle}$$

$$[\textsc{decv}]\frac{\langle e, \mathsf{env}_{SV}\rangle \to v}{\begin{array}{l}\langle \mathsf{var}(B_s)\ x\ \texttt{:=}\ e\ \texttt{in}\ S; Q, \mathsf{env}_{TSV}\rangle \\ \quad \to \langle S; \mathsf{del}(x); Q, \mathsf{env}_{TS}, ((x, v, B_s), \mathsf{env}_V)\rangle\end{array}}$$
where:
$$x \notin \mathrm{dom}\,(\mathsf{env}_V)$$

$$[\textsc{assv}]\frac{\langle e, \mathsf{env}_{SV}\rangle \to v}{\langle x\ \texttt{:=}\ e; Q, \mathsf{env}_{TSV}\rangle \to \langle Q, \mathsf{env}_{TS}, \mathsf{env}_V[x \mapsto v]\rangle}$$
where:
$$x \in \mathrm{dom}\,(\mathsf{env}_V)$$

$$[\textsc{assf}]\frac{\langle e, \mathsf{env}_{SV}\rangle \to v}{\begin{array}{l}\langle \texttt{this}.p\ \texttt{:=}\ e; Q, \mathsf{env}_{TSV}\rangle \\ \quad \to \langle Q, \mathsf{env}_T, \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]], \mathsf{env}_V\rangle\end{array}}$$
where:
$$\begin{aligned} X &= \mathsf{env}_V(\texttt{this}) \\ \mathsf{env}_F &= \mathsf{env}_S(X) \\ p &\in \mathrm{dom}\,(\mathsf{env}_F)\end{aligned}$$

$$[\textsc{delv}]\frac{}{\langle \mathsf{del}(x); Q, \mathsf{env}_{TS}, ((x, v, B_s), \mathsf{env}_V)\rangle \to \langle Q, \mathsf{env}_{TSV}\rangle}$$

$$[\textsc{return}]\frac{}{\langle \mathsf{env}'_V; Q, \mathsf{env}_{TSV}\rangle \to \langle Q, \mathsf{env}_{TS}, \mathsf{env}'_V\rangle}$$

Figure 8.9: Small-step semantics of stacks.

$$[\textsc{call}]\frac{\langle e_1, \mathsf{env}_{SV}\rangle \to Y \quad \langle e_2, \mathsf{env}_{SV}\rangle \to n \quad \langle \tilde{e}, \mathsf{env}_{SV}\rangle \to \tilde{v}}{\langle \texttt{call}\ \ e_1.m(\tilde{e})\$e_2; Q, \mathsf{env}_{TSV}\rangle \to \langle S; \mathsf{env}_V; Q, \mathsf{env}_T, \mathsf{env}'_{SV}\rangle}$$

where:

$$\begin{aligned}
X &= \mathsf{env}_V(\texttt{this}) \\
\mathsf{env}_F^X &= \mathsf{env}_S(X) \\
\mathsf{env}_F^Y &= \mathsf{env}_S(Y) \\
f &= \begin{cases} \mathsf{env}_V(m) & \text{if } m = \texttt{id} \\ m & \text{otherwise} \end{cases} \\
(\tilde{x}, S) &= \mathsf{env}_T(Y)(f) \\
|\tilde{x}| &= |\tilde{v}| = k \\
\mathsf{env}'_V &= (\texttt{this}, Y), (\texttt{sender}, X), (\texttt{value}, n), (x_1, v_1), \ldots, (x_k, v_k) \\
\mathsf{env}'_S &= \mathsf{env}_S\big[X \mapsto \mathsf{env}_F^X[\texttt{balance -= } n]\big]\big[Y \mapsto \mathsf{env}_F^Y[\texttt{balance += } n]\big]
\end{aligned}$$

$$[\textsc{dcall}]\frac{\langle e, \mathsf{env}_{SV}\rangle \to Y \quad \langle \tilde{e}, \mathsf{env}_{SV}\rangle \to \tilde{v}}{\langle \texttt{dcall}\ \ e.f(\tilde{e}); Q, \mathsf{env}_{TSV}\rangle \to \langle S; \mathsf{env}_V; Q, \mathsf{env}_{TS}, \mathsf{env}'_V\rangle}$$

where:

$$\begin{aligned}
|\tilde{x}| &= |\tilde{v}| = k \\
f &= \begin{cases} \mathsf{env}_V(m) & \text{if } m = \texttt{id} \\ m & \text{otherwise} \end{cases} \\
(\tilde{x}, S) &= \mathsf{env}_T(Y)(f) \\
\mathsf{env}'_V &= (\texttt{this}, \mathsf{env}_V(\texttt{this})), (\texttt{sender}, \mathsf{env}_V(\texttt{sender})), \\
&\quad (\texttt{value}, \mathsf{env}_V(\texttt{value})), (x_1, v_1), \ldots, (x_k, v_k), \mathsf{env}_V^{\varnothing}
\end{aligned}$$

$$[\textsc{fcall}]\frac{\langle e_1, \mathsf{env}_{SV}\rangle \to Y \quad \langle e_2, \mathsf{env}_{SV}\rangle \to n \quad \langle \tilde{e}, \mathsf{env}_{SV}\rangle \to \tilde{v}}{\langle \texttt{call}\ \ e_1.m(\tilde{e})\$e_2; Q, \mathsf{env}_{TSV}\rangle \to \langle S; \mathsf{env}_V; Q, \mathsf{env}_T, \mathsf{env}'_{SV}\rangle}$$

where:

$$\begin{aligned}
X &= \mathsf{env}_V(\texttt{this}) \\
\mathsf{env}_F^X &= \mathsf{env}_S(X) \\
\mathsf{env}_F^Y &= \mathsf{env}_S(Y) \\
f &= \begin{cases} \mathsf{env}_V(m) & \text{if } m = \texttt{id} \\ m & \text{otherwise} \end{cases} \\
f &\notin \mathrm{dom}\,(\mathsf{env}_T(Y)) \\
(\epsilon, S) &= \mathsf{env}_T(Y)(\texttt{fallback}) \\
\mathsf{env}'_V &= (\texttt{this}, Y), (\texttt{sender}, X), (\texttt{value}, n), (\texttt{id}, f), (\texttt{args}, \tilde{v}) \\
\mathsf{env}'_S &= \mathsf{env}_S\big[X \mapsto \mathsf{env}_F^X[\texttt{balance -= } n]\big]\big[Y \mapsto \mathsf{env}_F^Y[\texttt{balance += } n]\big]
\end{aligned}$$

Figure 8.10: Small-step semantics of stacks: method calls

Definition 53 (Syntax of types and environments)**.** We use the following types and environments:

$$\begin{aligned} B \in \mathcal{B} &::= \mathsf{idf} \mid \mathsf{int} \mid \mathsf{bool} \mid I \\ T \in \mathcal{T} &::= B_s \mid \mathsf{var}(B_s) \mid \mathsf{proc}(\widetilde{B_{\tilde{s}}}) : s \\ n \in \mathcal{N} &::= \mathsf{VNames} \cup \mathsf{FNames} \cup \mathsf{MNames} \cup \mathsf{ANames} \cup \mathsf{TNames} \\ \Gamma, \Delta \in \mathcal{E} &::= \mathcal{N} \rightharpoonup \mathcal{T} \cup \mathcal{E} \\ \Sigma &::= \mathsf{TNames} \rightharpoonup \mathsf{TNames} \end{aligned}$$

where VNames are variable names x, FNames are field names p, MNames are method names f, ANames are contract names X, and TNames are interface names I. ■

The intended meaning and usage of the types T is as follows:

- The type B_s is given to expressions e. It denotes that all *reads* within e are from containers of level s *or lower*, and the resulting value is of type B.
- The type $\mathsf{var}(B_s)$ is given to containers; i.e. variables and fields. It denotes that a field p or variable x is capable of storing a value of type B of level s *or lower*.
- The type $\mathsf{proc}(\widetilde{B_{\tilde{s}}}) : s$ is given to methods f. It denotes that the body of f is safe to execute as $\mathsf{cmd}(s)$, given that the formal arguments have types $\mathsf{var}(\widetilde{B_{\tilde{s}}})$, alternatively written $\mathsf{var}(\widetilde{(B, s)})$, which denotes a sequence of types

$$\mathsf{var}(B_1, s_1), \ldots, \mathsf{var}(B_n, s_n).$$

As mentioned above, we do not include a type for commands in the language of types, since such a type would not appear in the type environment. However, if a command S is safe to execute as $\mathsf{cmd}(s)$, it denotes that all (free) *assignments* made during the execution of a statement S are to containers of level s *or higher*.

Furthermore, well-typedness of a statement S (to any level s) also implies that any *guarded* commands within S must be safe to execute at the same level as their guards (or higher), and this level must then be equal to, or higher than, s. This restriction is necessary to ward against indirect information flows, since the guarded statement would execute in a context where information from the expression guard e is indirectly available. In the present setting, we have four such guarded constructs:

- `if` e `then` S_1 `else` S_2, where the choice of the branch depends on the guard e.
- `while` e `do` S, where the number of times S is executed (zero or more) depends on the guard e.
- `call` $e_1 . f(\tilde{e})\$e_2$, where the value of e_1 determines the path to the method to be executed.

- `dcall` $e.f(\tilde{e})$, where e determines the path to the method, similar to the case for `call` above.

The meaning of the base types B is mostly straightforward: For integer and boolean values, we assume that the base type can be determined directly from observing the value itself; i.e. 5 is an integer and F is a boolean. We shall use the following partial function to assign types to values:

Definition 54 (TypeOf)**.** We define the partial function $\textsc{TypeOf}_\Gamma(v)$ from values to types as follows:

$$\textsc{TypeOf}_\Gamma(v) = \begin{cases} (\mathsf{idf}, s_\bot) & \text{if } v \in \mathsf{MNames} \\ (\mathsf{int}, s_\bot) & \text{if } v \in \mathbb{Z} \\ (\mathsf{bool}, s_\bot) & \text{if } v \in \mathbb{B} \\ \Gamma(v) & \text{if } v \in \mathsf{ANames} \text{ and } v \in \mathrm{dom}\,(\Gamma) \end{cases}$$

and otherwise, the result is undefined. ■

$\textsc{TypeOf}_\Gamma(\cdot)$ assigns basic types to values in the obvious way, based on the set they belong to. However, as the type of a value has both a *data* aspect and a *security* aspect, we also need to assign a security level. Thus we choose the lowest possible level, $s_\bot$ for the 'pure' data values, integers and booleans, since they have no inherent security level attached to them, and it will always be safe to treat a piece of data as having a higher security level than it actually does.

However, with addresses $X \in \mathsf{ANames}$, the situation is different, as there is no inherent relationship between the address itself and its type, since addresses are *declared* to have a type (and a security level), according to the contract located at the address. The type of an address X is therefore an *interface* I, stored in Γ and consisting of the signatures of the methods and fields of that contract. The interface could in principle be extracted from the contract definition itself; however, we shall here prefer to let interfaces be defined separately, since it will allow multiple contracts to implement the same interface. For this purpose we use the following interface definition language:

Definition 55 (Interface definition language)**.** Interface declarations ID are given by the grammar:

$$\begin{array}{rcl} ID & ::= & \epsilon \mid \texttt{interface}\ I_1\ \texttt{:}\ I_2\ \texttt{\{}\ \texttt{IF IM}\ \texttt{\}}\ ID \\ IF & ::= & \epsilon \mid p : \mathrm{var}(B_s), IF \\ IM & ::= & \epsilon \mid f : \mathrm{proc}(\widetilde{B}_{\tilde{s}}) : s, IM \end{array}$$

where I_1 is the name of the interface, and I_2 is the name of its supertype. ■

For the sake of simplicity, we shall assume that members common to both interfaces also appear in both the supertype and the subtype; i.e. members are not automatically inherited from the supertype. We also assume that all contracts only implement a single interface.[8]

The interface definitions and other type assignments are recorded in type environments $\Gamma; \Delta$, and the inheritance hierarchy is separately recorded in Σ. We use them as follows:

- Σ records the immediate supertype of an interface. Thus, if we have the declaration `interface` I_1 `:` I_2 `{ IF IM }` then $\Sigma(I_1) = I_2$.
- Γ records the statically declared interface definitions and statically assigned types of contracts. We store interfaces in Γ by letting interface names I return another type environment Γ_I, which records the signatures of fields and methods declared in interface I. Thus, if a contract with address X implements an interface I, we will have that $\Gamma(X) = I_s$ for some security level s, and $\Gamma(I) = \Gamma_I$, where $\Gamma_I = IF, IM$.
- Δ records the types of local variables. We record these in a separate environment, since local variables are declared at runtime. Thus, this environment may change according to the runtime evolution of the program, whereas the other environments are static.

Given the intended usage of Γ, only addresses X and interface names I should appear in its domain. We express this with the following well-formedness criterion:

Definition 56 (Well-formedness of Γ)**.** We say that a type environment Γ is *well-formed*, if the following holds:

- Each contract name X in $\mathrm{dom}(\Gamma)$ has an interface type I; i.e. $\Gamma(X) = I$, and $I \in \mathrm{dom}(\Gamma)$.
- Each interface name I in $\mathrm{dom}(\Gamma)$ has a type environment entry Γ_I; i.e. $\Gamma(I) = \Gamma_I$.
- No other name has an entry in Γ; i.e. none of VNames, FNames, or MNames.

We shall only consider well-formed type environments in this chapter. ■

[8] We could also allow contracts to implement multiple interfaces; this would require the type of an address to be a *set* of interfaces, rather than just a single interface. However, we shall forego that here to avoid complicating the presentation.

$$[\Sigma\text{-TOP}]\frac{}{\Sigma;\Gamma \vdash (I^\top, I^\top)}$$

$$[\Sigma\text{-REC}]\frac{\Sigma;\Gamma \vdash \Sigma' \quad \forall n \in \mathrm{dom}\,(\Gamma_2)\ .\ \Sigma \vdash \Gamma_1(n) <: \Gamma_2(n)}{\Sigma;\Gamma \vdash (I_1, I_2), \Sigma'}$$

where:

$$\begin{aligned} I_1 &\neq I_2 \\ \nexists I'\ .\ \Sigma'(I') &= I_1 \\ \Gamma(I_1) &= \Gamma_1 \\ \Gamma(I_2) &= \Gamma_2 \\ \mathrm{dom}\,(\Gamma_2) &\subseteq \mathrm{dom}\,(\Gamma_1) \end{aligned}$$

Figure 8.11: Consistency rules for Σ.

8.4.2 Inheritance and subtyping

As mentioned above, we now use explicit inheritance, and we therefore introduce an environment Σ, which we use to record the explicit inheritance hierarchy. Furthermore, we only allow single inheritance, and we shall therefore assume the existence of an interface $I^\top$, corresponding to the basic implementation of a contract, which acts as the single root of the inheritance tree. It has the following definition:

```
1 interface I^⊤ : I^⊤ {
2   balance : var(int, s_⊤)
3   send    : proc():s_⊥
4 }
```

Note that this definition specifies itself as its own supertype. Although self-inheritance is permitted by the syntax, we shall introduce rules in the following to prevent self-inheritance in any other cases than $I^\top$. This is natural, since $I^\top$ is the root of the inheritance tree, and the common supertype of all other interfaces; hence it alone can inherit from nothing besides itself.

Σ gives a *static* representation of the subtype-supertype relationship for interfaces. We shall therefore need a way to decide whether a given Σ actually is *consistent* w.r.t. this relationship. Thus, we shall give a set of rules for deciding whether a given interface I_1, declaring that it inherits from another interface I_2, indeed also is a *subtype* of I_2. They are given in Figure 8.11. The judgment is of the form $\Sigma;\Gamma \vdash \Sigma$, which expresses that the inheritance tree recorded in Σ is *consistent* with the actual declarations recorded in Γ. The rules are quite simple, since they just iterate through the inheritance environment, examining each entry in turn. Σ only records the inheritance hierarchy, whereas the actual structures of the interfaces are recorded in Γ. Hence, consistency of any sub-part of Σ must be judged relative to Γ, as well as relative to the full Σ.

$$[\text{SUB-FIELD}]\frac{\Sigma \vdash B_1 <: B_2}{\Sigma \vdash \mathsf{var}(B_1, s_1) <: \mathsf{var}(B_2, s_2)}\ (s_1 \sqsubseteq s_2)$$

$$[\text{SUB-PROC}]\frac{\Sigma \vdash \widetilde{B_2} <: \widetilde{B_1}}{\Sigma \vdash \mathsf{proc}(\widetilde{B_1, s_1}) : s_1 <: \mathsf{proc}(\widetilde{B_2, s_2}) : s_2}\ \begin{pmatrix} s_2 \sqsubseteq s_1 \\ \widetilde{s_2} \sqsubseteq \widetilde{s_1} \end{pmatrix}$$

Figure 8.12: Subtyping rules for interface members.

- The base case is the rule [Σ-TOP], since we assume the interface $I^\top$ always exists and inherits from nothing besides itself. Furthermore, the form of the conclusion ensures that this rule can only be applied to an inheritance tree consisting of the root node itself.
- The recursive case is the rule [Σ-REC], which examines the entry (I_1, I_2). In the side condition, we require that $I_1 \neq I_2$, since no interface except $I^\top$ may inherit from itself.

  We also require that there must not exist another interface I', such that I' inherits from I_1 according to the remainder of the inheritance environment Σ'. This ensures that [Σ-REC] is only applicable to a *leaf node* of the current inheritance tree, which is then trimmed away as we recur in the premise. Hence it ensures that the environment indeed describes a *tree*.

The premise of [Σ-REC] invokes a subtyping judgment

$$\Sigma \vdash \Gamma_1(n) <: \Gamma_2(n)$$

for each member of the interface I_2, i.e. the supertype; and the side condition

$$\mathrm{dom}\,(\Gamma_2) \subseteq \mathrm{dom}\,(\Gamma_1)$$

ensures that all members of the supertype also exist in the subtype. Thus, this rule is very similar to the structural subtyping rule of Chapter 6, except that it is only invoked as part of the consistency check of Σ, rather than during the type checking phase.

The rules defining the subtyping judgments for interface members are given in Figure 8.12. They are quite similar to the subtyping rules given in Chapter 6, except that comparison of the security levels is handled directly in the side conditions by $\sqsubseteq$, and, furthermore, that these rules too are only invoked as part of the consistency check of Σ. Both rules invoke a subtyping judgment for types of the form

$$\Sigma \vdash B_1 <: B_2$$

$$[\text{SUB-REFL}]\frac{}{\Sigma \vdash B <: B}$$

$$[\text{SUB-TRANS}]\frac{\Sigma \vdash B_1 <: B_2 \quad \Sigma \vdash B_2 <: B_3}{\Sigma \vdash B_1 <: B_3}$$

$$[\text{SUB-NAME}]\frac{}{\Sigma \vdash I_1 <: I_2} \ (\Sigma(I_1) = I_2)$$

Figure 8.13: The subtyping relation for expression types B_s

in their premise: this, finally, is the subtyping relation proper, which is defined by the rules in Figure 8.13. We shall also need this relation later in the presentation. Note that we shall sometimes use the abbreviation

$$\Sigma \vdash B_1 <: B_2 <: B_3 \triangleq \Sigma \vdash B_1 <: B_2 \wedge \Sigma \vdash B_2 <: B_3$$

to simplify some premises.

The subtyping rules in Figure 8.13 are now much simpler, compared to the ones given in Chapter 6. They consist only of the reflexive and transitive rules, and the rule [SUB-NAME], which replaces the structural subtyping rule of Chapter 6. In the present setup, the subtyping relation is parametrised with Σ, rather than with Γ, and in [SUB-NAME] we simply look up the supertype in Σ, rather than structurally comparing their respective definitions in Γ. That comparison is no longer necessary, since the relationship will already have been verified, if Σ is consistent with Γ. Having explicit inheritance thus leads to a much simpler subtyping relation.

Remark 2 (On subtyping of methods and fields)**.** Subtyping of interface members (fields and methods) is complicated by their variances. The rule for suptyping variances is:

- Reading is covariant.
- Writing is contravariant.

Methods are comparable to 'write-only' fields (since subtyping should not be invoked when we are typing the *declaration* of a method).[9] Hence, the method type constructor is contravariant in its arguments, as can be seen in the rule [SUB-PROC]. However, due to this contravariance, we get the rather bizarre situation that an interface supertype must be less specific (i.e. have fewer members, as expected), but

[9]This similarity between methods and fields becomes much clearer, when we consider the encoding of an object-oriented language in a process calculus, as in Chapter 7, where both fields and method calls are represented by the same construct; namely channel outputs.

its methods must have *more* specific arguments than in the subtype. This hardly seems useful, but we nevertheless still allow it to give the rules a uniform shape.

With fields, the situation is more complicated. Usually in class-based/object-oriented languages, fields are *both* readable *and* writable. Hence, subtyping for fields should therefore be both covariant *and* contravariant, depending on whether we read from the field or write to it. In other words, it must be *invariant*, since allowing either form of variance would lead to an unsound subtyping relation, which, in the present setup, would mean that we could create an illegal information flow according to the security levels.

However, as can be seen in the rule [SUB-FIELD], we actually do allow subtyping of the field type constructor $\mathrm{var}(B_s)$ to be *covariant* in its argument. This is possible because of a speciality in TinySol, since fields in this language are *not* writable outside of the contract in which they are declared. This is ensured directly in the syntax, since a field is only allowed to appear as the LHS of an assignment of the form `this.`p `:=` e. Thus, fields are actually *read-only* in all those contexts in which subtyping may be invoked, and we *can* therefore allow covariant subtyping for fields. This is important, since we would otherwise be unable to coerce an arbitrary contract up to I^{T}, which is necessary to allow us to give a type to the 'magic variable' `sender`, that is available within every method body.

This is also the reason why we separate the subtyping rules for interface members (Figure 8.12) from the definition of the subtyping relation proper (Figure 8.13), since only the latter may be used in the actual type judgments. In the following presentation, we shall therefore only allow subtyping to be used when judging the type of an expression e, which limits the usage of subtyping to the RHS of assignments and the arguments of a method call. This automatically ensures that the rules in Figure 8.12 cannot be invoked, since an expression cannot return a field name or a method name; i.e. something of type $\mathrm{var}(B_s)$ or $\mathrm{proc}(\widetilde{B}_s) : s$; and even though an expression *can* yield an address X of type I, we no longer need to recur into the interface definition itself to decide its supertype, since we now simply look it up in Σ by rule [SUB-NAME]. Static inheritance thus solves the problem whilst simultaneously leading to a simpler subtyping relation. ■

Remark 3 (Structural subtyping versus static inheritance)**.** Our desire to use static inheritance is partly motivated by an infinite recursion that may occur in structural subtyping, which was used in Chapter 6. Consider the two simple interface definitions in Figure 8.14. To conclude $\Gamma \vdash I_1 <: I_2$ by the structural subtyping rules from Chapter 6, we have to compare the types of the common members, which in this case are the methods f. Thus, we must be able to conclude

$$\Gamma \vdash \mathrm{proc}((I_1, s)) : s <: \mathrm{proc}((I_2, s)) : s$$

which requires a (contravariant) comparison of the arguments; i.e. $\Gamma \vdash I_2 <: I_1$. Now the pattern repeats, but with the arguments switched, such that we now must be able

```
interface I1 {
  f : proc((I2, s)):s
}

interface I2 {
  f : proc((I1, s)):s
}
```

Figure 8.14: Structural subtyping of interface definitions.

to conclude

$$\Gamma \vdash \text{proc}((I_2, s)):s <: \text{proc}((I_1, s)):s$$

which then requires that $\Gamma \vdash I_1 <: I_2$ must hold, and so on. Thus, this interface definition yields an infinite recursion. This is not technically a problem, since it simply means that $\Gamma \vdash I_1 <: I_2$ cannot be concluded, so I_1 cannot be judged a subtype of I_2, but it is at least impractical from an implementation perspective.

Using static inheritance instead, we can assume that I_1 was declared as a subtype of I_2, and I_2 as a subtype of $I^\top$; i.e. we assume a Σ such that $\Sigma(I_1) = I_2$ and $\Sigma(I_2) = I^\top$. Now, to decide the consistency of this Σ, we must compare the matching members by rule [Σ-REC], which means we must be able to conclude

$$\Sigma \vdash \text{proc}((I_1, s)):s <: \text{proc}((I_2, s)):s$$

by rule [SUB-PROC]. This, in turn, requires that $\Sigma \vdash I_2 <: I_1$ holds, in order to satisfy the premise of that rule. However, according to our declarations, $\Sigma(I_2) = I^\top$, so this cannot be concluded. The result is thus the same as with structural subtyping, but now the derivation terminates rather than recurring infinitely. ■

8.4.3 Safety requirement for operations

We have left the syntactic category of operations op undefined, and simply assumed that they can all be evaluated using some semantics $\rightarrow_{\text{op}}$, such that $\text{op}(\tilde{v}) \rightarrow_{\text{op}} v$. Furthermore, we assume that for each operation we can determine its signature, written $\vdash \text{op} : \widetilde{B} \rightarrow B$. To tie these two together, we need to make one further assumption; namely that *if* the signature of an operation is $\widetilde{B} \rightarrow B$, and the actual parameters $\tilde{v}$ are of basic types $\widetilde{B}$, with $|\tilde{v}| = |\widetilde{B}|$, *then* the operation can actually be performed, and the resulting value v is also of type B. As neither the operations, their semantics, or their type rules are specified, all we can do w.r.t. safety is to require that this must hold. Formally, we can specify the requirement as follows:

Definition 57 (Safety requirement for operations)**.** For all operations op, and for any argument list $\tilde{v}$ it must hold that *if*

- $\vdash \mathrm{op} : \widetilde{B} \to B$, and
- $|\widetilde{B}| = |\tilde{v}|$, and
- $\forall v_i \in \tilde{v} \,.\, \textsc{TypeOf}_\Gamma(v_i) = (B_i, s_i)$,

then $\mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v$ and $\textsc{TypeOf}_\Gamma(v) = (B, s_\bot)$, assuming any addresses X appearing in the argument list $\tilde{v}$ has an entry in Γ. ■

Note that by this definition, we allow addresses to appear as arguments to operations. However, as we assume that an operation can never *yield* an address, we also know that the resulting security level will be $s_\bot$, and that the resulting base type B will be one of int or bool.

Remark 4. Operations are only concerned with the actual data, and not with its security level, since all arguments must be actual values; i.e. not variables. Therefore we disregard the security levels here, and instead check the security level in the type rule for operations, given in the following section. This simply ensures that we can give syntax-directed type rules for expressions.

Alternatively, we could have assumed that operations op were evaluated using rules that must be provided directly as part of the semantics for expressions, and that type rules must be provided directly as part of the type rules for expressions. That would instead require us to impose restrictions on the format of such rules; in particular that the security levels of all arguments are ordered below the 'target level' s using $\sqsubseteq$. With the current solution, such checks are only needed in a single type rule, where the expression arguments are judged to have a type. ■

8.4.4 Syntactic type rules

As a precursor to developing our semantics of types, we shall first give a system of type rules in the usual syntactic style, similar to the one in Chapter 6, but appropriately extended to take our changes to the language into account. This will serve as a guiding intuition in our later developments of a semantics for the types themselves. In a sense, this can seem backwards, since the general rule in semantic typing is that the semantics of types should serve as the guiding principle. However, we wish to keep the correspondence with the syntactic type system from Chapter 6 clear, because the meaning of the types should of course match with their usage in this prior work. Hence, we prefer to do this step first.

8.4.4.1 Type rules for expressions

The type rules for expressions e are given in Figure 8.15. They are adapted from the ones given in Chapter 6, with only some minor changes to fit our current setup. The key differences are that we now use two type environments, Γ and Δ, as in

$$[\text{T-VAL}]\frac{\Sigma \vdash B' <: B}{\Sigma;\Gamma;\Delta \vdash v : B_s}\left(\begin{array}{c}\textsc{TypeOf}_\Gamma(v) = (B', s') \\ s' \sqsubseteq s\end{array}\right)$$

$$[\text{T-VAR}]\frac{\Sigma \vdash B' <: B}{\Sigma;\Gamma;\Delta \vdash x : B_s}\left(\begin{array}{c}\Delta(x) = \text{var}(B', s') \\ s' \sqsubseteq s\end{array}\right)$$

$$[\text{T-FIELD}]\frac{\Sigma;\Gamma;\Delta \vdash e : I_s \quad \Sigma \vdash B' <: B}{\Sigma;\Gamma;\Delta \vdash e.p : B_s}\left(\begin{array}{c}\Gamma(I)(p) = \text{var}(B', s') \\ s' \sqsubseteq s\end{array}\right)$$

$$[\text{T-OP}]\frac{\vdash \text{op} : \widetilde{B} \to B \quad \Sigma;\Gamma;\Delta \vdash \tilde{e} : \widetilde{B}_{s,\ldots,s}}{\Sigma;\Gamma;\Delta \vdash \text{op}(\tilde{e}) : B_s}$$

Figure 8.15: Type rules for expressions.

Chapter 7, and we isolate all usage of subtyping to these rules, so we include Σ as well. Expressions can only occur in guards and as the right-hand side (RHS) of assignments, which are exactly those contexts where subtyping may be used. Thus, by limiting the usage of subtyping to expressions we automatically ensure that subtyping can only be used where it is needed. These changes slightly complicate the side condition of some rules, since we now explicitly compare the security levels s by $\sqsubseteq$, but on the other hand they make it clearer exactly *when* subtyping can be invoked, unlike in Chapter 6, which included a general subsumption rule.

A key feature to notice about these rules is that type judgments are of the form

$$\Sigma;\Gamma;\Delta \vdash e : B_s$$

and each of them have a side condition of the form $s' \sqsubseteq s$, where s' is a level derived from the expression itself. The purpose of this side condition is to emulate one of the effects of subtyping in Chapter 6; namely to ensure that it holds that *if* $\Gamma;\Delta \vdash e : (B, s_1)$ for some level s_1, *then* also $\Gamma;\Delta \vdash e : (B, s_2)$ for any level s_2 such that $s_1 \sqsubseteq s_2$. This is in accordance with the intended meaning of the type B_s, which says that all variables read in the evaluation of e are of level s *or lower*, and type-safety should therefore also hold for all levels *higher* than s. We shall refer to this as the *coercion property* for expressions.

Two of the rules deserve a few extra comments:

- In [T-FIELD], the conclusion is of the form $\Sigma;\Gamma;\Delta \vdash e.p : B_s$, where e must resolve to an address X of a contract containing the field p. In the premise of that rule, we must therefore conclude $\Sigma;\Gamma;\Delta \vdash e : I_s$, so, in other words, e can only depend on variables of a level that is *lower* than, or equal to s. This is necessary to ensure that higher-level parts of the program cannot affect the *path* to the field p, and thereby induce a difference in the data being read. Note

that it might be the case that $\Sigma;\Gamma;\Delta \vdash e : (I, s'')$ for some level $s'' \sqsubseteq s$, but by the coercion property we then know that $\Sigma;\Gamma;\Delta \vdash e : I_s$ can be concluded as well.

The field itself must have a type $\mathsf{var}(B', s')$ according to Γ, and the side condition then states that $s' \sqsubseteq s$, similar to the other rules, to ensure that the coercion property holds. However, s'' and s' do not necessarily have to be related; all we need to require is that s is an upper bound on both, which then ensures that the expression will also be well-typed for all levels above s as well.

- In [T-OP], we write the list of types of $\tilde{e}$ as $\widetilde{B}_{s,\ldots,s}$ to indicate that the list of security levels $s, \ldots, s$ must be the *same* as the level that appears in the conclusion of the rule. In effect, we are thus ensuring that none of the argument expressions $\tilde{e}$ can depend on variables of a higher level than s.

  Furthermore, in this rule we do not need the side condition $s' \sqsubseteq s$, because we know that operations op cannot return addresses X. Hence, the returned value must be one of int, bool, and by the definition of $\textsc{TypeOf}_\Gamma(\cdot)$, the level of such a value is always $s_\bot$, which by definition is lower than, or equal to, any other level s.

8.4.4.2 Type rules for statements

The type rules for statements S are given in Figure 8.16. Type judgments are here of the form

$$\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s)$$

and these rules are again derived from the ones given in Chapter 6, and with the addition of the rule [T-DCALL] for delegate calls. Furthermore, we have opted to completely omit subtyping for statements. The presentation in Chapter 6 included a contravariant subtyping rule for $\mathsf{cmd}(s)$, and furthermore a general subsumption rule, which allowed the level of *any* statement to be coerced down to a lower level. As was the case with expressions, we now handle this directly in the side conditions of the rules, by including a condition of the form $s \sqsubseteq s'$ in all rules that place some restriction on s. This ensures that *if* $\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s_1)$, then it also holds that $\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s_2)$ for any level s_2 such that $s_2 \sqsubseteq s_1$. In other words, we can always choose a *lower* security level, which is the same property that was ensured by subtyping in Chapter 6. Similar to the case for expressions, we refer to this as the *coercion property* for statements. This property is now made explicit in the rules, but the downside is that the side conditions of several rules are more complicated, so we shall give some details regarding the underlying intuitions:

- In [T-SKIP] and [T-THROW], s is free in the conclusion, since neither statement has any data flow. Likewise in [T-SEQ], s is determined directly from the levels

$$[\text{T-SKIP}]\frac{}{\Sigma;\Gamma;\Delta \vdash \texttt{skip} : \mathsf{cmd}(s)}$$

$$[\text{T-THROW}]\frac{}{\Sigma;\Gamma;\Delta \vdash \texttt{throw} : \mathsf{cmd}(s)}$$

$$[\text{T-SEQ}]\frac{\Sigma;\Gamma;\Delta \vdash S_1 : \mathsf{cmd}(s) \quad \Sigma;\Gamma;\Delta \vdash S_2 : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash S_1; S_2 : \mathsf{cmd}(s)}$$

$$[\text{T-IF}]\frac{\Sigma;\Gamma;\Delta \vdash e : (\mathsf{bool}, s') \qquad \begin{array}{c}\Sigma;\Gamma;\Delta \vdash S_\mathsf{T} : \mathsf{cmd}(s') \\ \Sigma;\Gamma;\Delta \vdash S_\mathsf{F} : \mathsf{cmd}(s')\end{array}}{\Sigma;\Gamma;\Delta \vdash \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F} : \mathsf{cmd}(s)}\ (s \sqsubseteq s')$$

$$[\text{T-WHILE}]\frac{\Sigma;\Gamma;\Delta \vdash e : (\mathsf{bool}, s') \quad \Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s')}{\Sigma;\Gamma;\Delta \vdash \texttt{while}\ e\ \texttt{do}\ S : \mathsf{cmd}(s)}\ (s \sqsubseteq s')$$

$$[\text{T-DECV}]\frac{\Sigma;\Gamma;\Delta \vdash e : (B, s') \quad \Sigma;\Gamma;\Delta, x : \mathsf{var}(B, s') \vdash S : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash \mathsf{var}(B, s')\ \ x\ \texttt{:=}\ e\ \texttt{in}\ S : \mathsf{cmd}(s)}$$

$$[\text{T-ASSV}]\frac{\Sigma;\Gamma;\Delta \vdash e : (B, s')}{\Sigma;\Gamma;\Delta \vdash x\ \texttt{:=}\ e : \mathsf{cmd}(s)}\ \left(\begin{array}{c}\Delta(x) = \mathsf{var}(B, s') \\ s \sqsubseteq s'\end{array}\right)$$

$$[\text{T-ASSF}]\frac{\Sigma;\Gamma;\Delta \vdash e : (B, s')}{\Sigma;\Gamma;\Delta \vdash \texttt{this}.p\ \texttt{:=}\ e : \mathsf{cmd}(s)}\ \left(\begin{array}{c}\Delta(\texttt{this}) = \mathsf{var}(I, s_1) \\ \Gamma(I)(p) = \mathsf{var}(B, s') \\ s_1 \sqsubseteq s' \\ s \sqsubseteq s'\end{array}\right)$$

$$[\text{T-CALL}]\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vdash e_1 : (I^Y, s') \\ \Sigma;\Gamma;\Delta \vdash e_2 : (\mathsf{int}, s') \quad \Sigma;\Gamma;\Delta \vdash \tilde{e} : \widetilde{B}_{\tilde{s}}\end{array}}{\Sigma;\Gamma;\Delta \vdash \texttt{call}\ e_1.f(\tilde{e})\$e_2 : \mathsf{cmd}(s)}\ \left(\begin{array}{c}s_1 \sqsubseteq s' \\ s' \sqsubseteq s_3, s_4 \\ s \sqsubseteq s'\end{array}\right)$$

where:
$\mathsf{var}(I^X, s_1) = \Delta(\texttt{this})$
$\mathsf{var}(\mathsf{int}, s_3) = \Gamma(I^X)(\texttt{balance})$
$\mathsf{var}(\mathsf{int}, s_4) = \Gamma(I^Y)(\texttt{balance})$
$\mathsf{proc}(\widetilde{B}_{\tilde{s}}) : s' = \Gamma(I^Y)(f)$

$$[\text{T-DCALL}]\frac{\Sigma \vdash I^X <: I^Y \quad \Sigma;\Gamma;\Delta \vdash e : (I^Y, s') \quad \Sigma;\Gamma;\Delta \vdash \tilde{e} : \widetilde{B}_{\tilde{s}}}{\Sigma;\Gamma;\Delta \vdash \texttt{dcall}\ e.f(\tilde{e}) : \mathsf{cmd}(s)}\ \left(\begin{array}{c}s_1 \sqsubseteq s' \\ s \sqsubseteq s'\end{array}\right)$$

where:
$\Delta(\texttt{this}) = \mathsf{var}(I^X, s_1)$
$\mathsf{proc}(\widetilde{B}_{\tilde{s}}) : s' = \Gamma(I)(f)$

Figure 8.16: Type rules for statements

of the separate statements, so as long as we ensure that the coercion property in fact holds, this rule does not need a subtyping-simulating side condition either.

Finally, in [T-DECV], s is also just determined from the level of the statement S. As with [T-SEQ], we do not need to simulate subtyping for the whole assignment statement itself, as long as the coercion property holds for the inner statement S. The level of the whole declaration statement is $\mathsf{cmd}(s)$, *if* it can be concluded that the inner statement S has level $\mathsf{cmd}(s)$, given that x has type $\mathsf{var}(B_2, s')$. Note that we here use the coercion property for expressions to be able to conclude that the RHS e has level s', since if $\Sigma; \Gamma; \Delta \vdash e(B_1, s'')$ can be concluded for some level s'', and $s'' \sqsubseteq s'$, then $\Sigma; \Gamma; \Delta \vdash e : (B_1, s')$ can be concluded as well.

- In [T-IF], we have the side condition $s \sqsubseteq s'$, where s' is the level of the guard and both branches. Here we rely on the coercion property for both expressions and statements, corresponding to the two different kinds of coercions allowed by subtyping in Chapter 6:

  - Suppose $\Sigma; \Gamma; \Delta \vdash e : (\mathsf{bool}, s_1)$ can be concluded for some $s_1 \sqsubseteq s'$. Then s_1 could be coerced up to match s', meaning in our setup that $\Sigma; \Gamma; \Delta \vdash e : (\mathsf{bool}, s')$ can be concluded as well, by the coercion property for expressions.
  - Suppose $\Sigma; \Gamma; \Delta \vdash S_\mathsf{T} : \mathsf{cmd}(s_2)$ and $\Sigma; \Gamma; \Delta \vdash S_\mathsf{F} : \mathsf{cmd}(s_3)$ can concluded for some levels $s' \sqsubseteq s_2, s_3$. Then $\mathsf{cmd}(s_2)$ and $\mathsf{cmd}(s_3)$ could both be coerced *down* to match $\mathsf{cmd}(s')$, which in the present setup means that $\mathsf{cmd}(s')$ can be concluded as well, by the coercion property for statements.

  Either s_1 could be coerced up to match s_2 and s_3, or s_2 and s_3 could be coerced down to match s_1, meaning in effect that we require that there must exist some level s' such that both $s_1 \sqsubseteq s'$ and $s' \sqsubseteq s_2, s_3$. This 'common meeting point' s' is then the level of the whole `if` statement, and $\mathsf{cmd}(s')$ could then be further coerced down to an arbitrary lower level $\mathsf{cmd}(s)$. We achieve the same effect here by including the side condition $s \sqsubseteq s'$.

  The reasoning is the same for the rule [T-WHILE], albeit we here only have a single guarded statement S. Note also that, in both cases, we have that the level of the guard e must be *lower* than, or equal to, the level of the guarded statement(s) (S_1 and S_2, and S, respectively). This wards against a potential indirect downward information flow from the guard to the guarded statement; or, in other words, it ensures that the execution of the guarded statement will not depend on information that is of a *higher* level than itself. This, in yet other words, ensures that a higher-level part of the program cannot induce a difference in a lower level part of the program by altering which statement is

executed (in the case of `if`), or how many times it is executed (in the case of `while`).

- [T-ASSV] and [T-ASSF] are both rules for assignments to *free* containers (variables resp. fields), so in both these cases the level of the container, s', *does* restrict the level of the whole statement, unlike in [T-DECV]. In both cases we therefore rely on the coercion property for expressions to ensure that as long as the RHS e can be typed to some level s_1 such that $s_1 \sqsubseteq s'$, then $\Sigma; \Gamma; \Delta \vdash e : (B_1, s')$ can be concluded as well. In the type system in Chapter 6, the level of the whole command would then be $\mathsf{cmd}(s')$, which subsequently could be coerced down to any lower level by subtyping. To ensure that the coercion property holds, we therefore include the side condition $s \sqsubseteq s'$.

- The rule [T-CALL] is the most complex, since we need to handle no less than five different security levels:

  - The condition $s' \sqsubseteq s_3, s_4$ ensures that the implicit flows from the `balance` field of the caller to the `balance` field of the callee is safe. Here, s' is the level of method body, and we require that the expression e_2, which is the value to be transferred, also can be evaluated to this level. Then, s_3 is the actual level of `balance` of the caller; and s_4 is the actual level of `balance` of the callee.

    The reasoning is as follows: Suppose the *actual* level of e_2 were some level s_2. Per the semantics (Figure 8.10), every method call implicitly performs a write to the `balance` fields of both the caller and the callee, corresponding to the following two lines of code:

    $$\texttt{this.balance := this.balance - } e_2\texttt{;}$$
    $$Y\texttt{.balance := } Y\texttt{.balance + } e_2\texttt{;}$$

    assuming Y is the address of the callee. Typing these statements similar to [T-ASSF] gives us that $s_2 \sqsubseteq s_3$ and $s_2 \sqsubseteq s_4$. In both cases, the level of the left-hand side (LHS) must of course be above the level of the RHS, and since `this.balance` (resp. Y`.balance`) appears on both sides, we have that s_2 must be less than both, since if it were higher, the level of the RHS would be above the level of the LHS. Finally, as both `balance` fields are being modified in the method body, albeit implicitly, it must also be the case that $s' \sqsubseteq s_3, s_4$. We then simplify this by letting $s_2 = s'$ and requiring that e_2 can be evaluated at level s' as well.[10]

[10] We assume $s_2 \sqsubseteq s'$, so e_2 can safely be coerced up to match the level s'. It *could* also be the case that $s' \sqsubseteq s_2$, or that they were unrelated, but we omit this possibility for the sake of simplicity. We indirectly make the same assumption in Chapter 6 through subtyping.

- We also have that $\Sigma; \Gamma; \Delta \vdash e_1 : (I^Y, s')$, where e_1 is an expression that must resolve to an address of a contract implementing the called method f. The level s' is determined by the type of f; i.e. $\text{proc}(\widetilde{B}_{\tilde{s}}) : s'$, so here we again make use of the coercion property for expressions: Assume that the level of e_1 was actually some s_1 such that $s_1 \sqsubseteq s'$; then we know that $\Sigma; \Gamma; \Delta \vdash e_1 : (I^Y, s')$ can be concluded as well. In effect, we thus require that the level of e_1 must be *lower* than, or equal to, s', which is the level of the method body. This condition is thus similar to the one of the other guarded statements (cf. [T-IF] and [T-WHILE]), where the level of the guard (here, s_1) must be *lower* than, or equal to, the level of the statement that it guards. The level of the whole method call is then s', and we therefore have the side condition $s \sqsubseteq s'$ to simulate subtyping and ensure that the coercion property holds.

  Finally, note that in the *list* of types, written $\widetilde{B}_{\tilde{s}}$, the levels $\tilde{s}$ do not have to be related to any of the other levels.

- The rule [T-DCALL] is similar to [T-CALL], but simpler, because we do not have writes to the two `balance` fields; hence, the side condition simplifies to just $s \sqsubseteq s'$.

  On the other hand, we have to use subtyping to ensure that the interface of the caller contract, I^X, is a subtype of the interface of the callee contract, I^Y, following the same argument as in Section 8.1, and also that the security levels of the caller, callee, and the method body are compatible. Here, s_1 is the actual level of the caller, and s' is the actual level of the method, whilst the actual level of e must be some level s_2, such that $s_2 \sqsubseteq s'$. Thus, s_2 can safely be coerced up to s', so we simplify the condition by requiring that e can be evaluated at level s'.

8.4.4.3 Type rules for stacks

The type rules for stacks Q are given in Figure 8.17. These are derived from the ones given in Chapter 7, and are mostly as expected. None of the stack operations place any restrictions on the level s, so we do not need any side conditions to alter the security level. The only slightly unobvious rule is [T-RET], corresponding to the 'return' operation, where a method call ends, and we restore the previous variable environment env_V. As in Chapter 7, we need a way to recover the types of locally declared variables, which, as the reader may recall, we assumed were stored in env_V along with the values. The extraction function $\mathbf{E}(\cdot)$ simply extracts this information from the variable environment found on the stack, to build the type environment Δ' relative to which the remainder of the stack Q is then typed.

$$[\text{T-BOT}]\frac{}{\Sigma;\Gamma;\Delta \vdash \bot : \mathsf{cmd}(s)}$$

$$[\text{T-STM}]\frac{\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s) \quad \Sigma;\Gamma;\Delta \vdash Q : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash S;Q : \mathsf{cmd}(s)}$$

$$[\text{T-DEL}]\frac{\Sigma;\Gamma,\Delta \vdash Q : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta,x : T \vdash \mathsf{del}(x);Q : \mathsf{cmd}(s)}$$

$$[\text{T-RET}]\frac{\Sigma;\Gamma;\mathbf{E}(\mathsf{env}_V) \vdash \mathsf{env}_V \quad \Sigma;\Gamma;\mathbf{E}(\mathsf{env}_V) \vdash Q : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V;Q : \mathsf{cmd}(s)}$$

where:

$$\mathbf{E}(\epsilon) = \epsilon$$
$$\mathbf{E}(x : (v,T), \mathsf{env}'_V) = x : T, \mathbf{E}(\mathsf{env}'_V)$$

Figure 8.17: Type rules for stacks.

8.4.4.4 Type rules for environments

Figure 8.18 gives the type rules for the three environments env_{TSV}. For all environments, the type judgments are relative to a Σ, Γ and Δ, to make them uniform. However, the Δ component is only actually needed for typing env_V, and we therefore write $\emptyset$ for the Δ component in the type rules for the other environments.

- In [T-ENVS] and [T-ENV-M], the Δ component is not needed, because although the type rule [T-ENVF] invokes the type rule [T-VAL] for expressions to conclude a type for the value v stored in a field p, this rule does not rely on Δ. Using [T-VAL] is merely a simplification, but [T-ENVF] and [T-ENVS] could also be rewritten such that they do not depend on Δ. Hence, if $\Sigma;\Gamma;\emptyset \vdash \mathsf{env}_S$ holds, then $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_S$ would also hold for any Δ.
- In [T-ENV-T] and [T-ENV-M], the type judgment only needs Σ and Γ, hence we write $\emptyset$ here for the Δ component. In the side condition of [T-ENV-M], we then build a new Δ from the signature of the method to be able to type the body S in the premise.

Lastly, note the two different versions of the type rule for env_V, [T-ENVV$_t$] and [T-ENVV$_u$]. Hitherto, we have assumed that type information for locally declared variables is stored in the variable environment, and for typing this form of env_V the rule [T-ENVV$_t$] is used. However, later in the presentation we shall keep this information in a separate environment, and in that case we shall instead need the rule [T-ENVV$_u$]. We shall generally assume it is clear from the context which of the two rules is used when writing $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V$, and we only refer explicitly to one of the two rules in proofs.

$$[\text{T-ENV-}\emptyset]\frac{}{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_X^{\emptyset}} \ (X \in \{T, M, S, F, V\})$$

$$[\text{T-ENV-T}]\frac{\Sigma;\Gamma;\emptyset \vdash_X \mathsf{env}_M \quad \Sigma;\Gamma;\emptyset \vdash \mathsf{env}_T}{\Sigma;\Gamma;\emptyset \vdash \mathsf{env}_T, X : \mathsf{env}_M}$$

$$[\text{T-ENV-M}]\frac{\Sigma;\Gamma;\emptyset \vdash_X \mathsf{env}_M \quad \Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s)}{\Sigma;\Gamma;\emptyset \vdash_X \mathsf{env}_M, f : (x_1, \ldots, x_n, S)} \ (s \sqsubseteq s_1)$$

where:

$$\begin{aligned} I_{s_1} &= \Gamma(X) \\ \mathsf{var}(\mathsf{int}_{s_2}) &= \Gamma(I)(\texttt{balance}) \\ \Gamma(I)(f) &= \mathsf{proc}((B_1, s_1'), \ldots, (B_n, s_n')) : s \\ \Delta &= \texttt{this} : \mathsf{var}(I_{s_1}), \texttt{value} : \mathsf{var}(\mathsf{int}_{s_2}), \texttt{sender} : \mathsf{var}(I_{s_\top}^{\top}), \\ &\qquad x_1 : \mathsf{var}(B_1, s_1'), \ldots, x_n : \mathsf{var}(B_n, s_n') \end{aligned}$$

$$[\text{T-ENVS}]\frac{\Sigma;\Gamma;\emptyset \vdash \mathsf{env}_S \quad \Sigma;\Gamma;\emptyset \vdash_X \mathsf{env}_F}{\Sigma;\Gamma;\emptyset \vdash \mathsf{env}_S, X : \mathsf{env}_F}$$

$$[\text{T-ENVF}]\frac{\Sigma;\Gamma;\emptyset \vdash_X \mathsf{env}_F \quad \Sigma;\Gamma;\emptyset \vdash v : B_s}{\Sigma;\Gamma;\emptyset \vdash_X \mathsf{env}_F, p : v} \begin{pmatrix} \Gamma(X) = (I, s') \\ \Gamma(I)(p) = \mathsf{var}(B_s) \end{pmatrix}$$

$$[\text{T-ENVV}_t]\frac{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V \quad \Sigma;\Gamma;\emptyset \vdash v : B_s}{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V, x : (v, B_s)} \ (\Delta(x) = \mathsf{var}(B_s))$$

$$[\text{T-ENVV}_u]\frac{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V \quad \Sigma;\Gamma;\emptyset \vdash v : B_s}{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V, x : v} \ (\Delta(x) = \mathsf{var}(B_s))$$

$$[\text{T-ENVSV}]\frac{\Sigma;\Gamma;\emptyset \vdash \mathsf{env}_S \quad \Sigma;\Gamma;\Delta \vdash \mathsf{env}_V}{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_{SV}}$$

Figure 8.18: Type rules for the method-, state-, and variable environments

8.4.5 Environment consistency

Having a well-typed stack Q, and well-typed environments env_{TSV}, is unfortunately not enough to ensure that the combination $\langle Q, \mathsf{env}_{TSV} \rangle$ also will be safe to execute. The names appearing in a stack Q must obviously also appear within the state env_{SV}, since otherwise the transition might become stuck. However, well-typedness does not ensure the *existence* of a field or variable in env_{SV}, so it is entirely possible that two different states env^1_{SV} and env^2_{SV} can both be well-typed relative to the same $\Sigma; \Gamma; \Delta$, yet nevertheless have no fields or variables in common. Then e.g. $\langle Q, \mathsf{env}^1_{TSV} \rangle$ might be executable, but $\langle Q, \mathsf{env}^2_{TSV} \rangle$ might not be, if the two states do not contain the same names.

Likewise, env_T and env_S represent two different parts of the contract declarations, namely the method declarations and the field declarations, respectively. Yet, well-typedness itself does not ensure that an arbitrary env_T and env_S are actually derived from the *same* set of contract declarations.

Furthermore, addresses can not only appear in the domains of env_S, but also as *values* v in env_{SV}, e.g. as the value of a variable x. Any such address could be 'dereferenced' if x is used in an expression or as the object path in a method call. Well-typedness only ensures that the address has an appropriate interface type, but not that an appropriate *implementation* of the interface also exists on that address.

This is also related to the 'reverse check' mentioned in Chapter 6: Even if a contract C declares that it implements an interface I, well-typedness does not ensure that C implements *every member* of I (no more and no less). It only ensures that the members that C *does* implement are in accordance with the corresponding type definitions in I.

To handle these problems, we shall need to impose some further consistency checks on the environments env_{TSV}, and their inner environments env_M and env_F, containing the methods and fields of each individual contract. Firstly, we shall define two different notions of free names:

Definition 58 (Free names)**.** We use the following notation:

- We write $\mathcal{F}_A(S)$, $\mathcal{F}_A(e)$ for the set of *addresses* X occurring in a statement S and an expression e. We omit a formal definition, since addresses cannot be bound, so they can be read directly from the syntax.
- We write $\mathcal{F}_V(S)$, $\mathcal{F}_V(e)$ for the set of *free variable names* x occurring in a statement S and an expression e. The formal definition is given in Figure 8.19. We assume here that x also ranges over the magic variable names `this`, `sender`, `value`, `id`, and `args`. ■

Now, in Figure 8.20, we give the rules for a consistency check for env_F and env_M. Both are defined in terms of a relation $\rightleftharpoons$ between a code environment (env_F, resp. env_M) and type environment (Γ_F, resp. Γ_M), where the latter represents the interface

$$
\begin{aligned}
\mathcal{F}_V(v) &= \varnothing \\
\mathcal{F}_V(x) &= \{\, x \,\} \\
\mathcal{F}_V(e.p) &= \mathcal{F}_V(e) \\
\mathcal{F}_V(\mathrm{op}(e_1, \ldots, e_n)) &= \mathcal{F}_V(e_1) \cup \ldots \cup \mathcal{F}_V(e_n)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}_V(\texttt{skip}) &= \varnothing \\
\mathcal{F}_V(\texttt{throw}) &= \varnothing \\
\mathcal{F}_V(\texttt{var } x \texttt{ := } e \texttt{ in } S) &= \mathcal{F}_V(e) \cup \mathcal{F}_V(S) \setminus \{\, x \,\} \\
\mathcal{F}_V(x \texttt{ := } e) &= \{\, x \,\} \cup \mathcal{F}_V(e) \\
\mathcal{F}_V(\texttt{this}.p \texttt{ := } e) &= \{\, \texttt{this} \,\} \cup \mathcal{F}_V(e) \\
\mathcal{F}_V(S_1\texttt{; } S_2) &= \mathcal{F}_V(S_1) \cup \mathcal{F}_V(S_2) \\
\mathcal{F}_V(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) &= \mathcal{F}_V(e) \cup \mathcal{F}_V(S_1) \cup \mathcal{F}_V(S_2) \\
\mathcal{F}_V(\texttt{while } e \texttt{ do } S) &= \mathcal{F}_V(e) \cup \mathcal{F}_V(S) \\
\mathcal{F}_V(\texttt{call } e_1\texttt{.}m(e'_1, \ldots, e'_n)\texttt{\$}e_2) &= \mathcal{F}_V(e_1) \cup \mathcal{F}_V(e'_1) \cup \ldots \cup \mathcal{F}_V(e'_n) \\
&\quad \cup \mathcal{F}_V(e_2) \cup \mathcal{F}_V(m) \\
\mathcal{F}_V(\texttt{dcall } e\texttt{.}m(e'_1, \ldots, e'_n)) &= \mathcal{F}_V(e) \cup \mathcal{F}_V(e'_1) \cup \ldots \cup \mathcal{F}_V(e'_n) \cup \{\, \texttt{id} \,\} \\
\mathcal{F}_V(m) &= \begin{cases} \{\, \texttt{id} \,\} & \text{if } m = \texttt{id} \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 8.19: The calculation of free variable names for expressions and statements.

$$[\text{C-ENVF}_1]\frac{}{\mathsf{env}_S \vdash \mathsf{env}_F^{\varnothing} \rightleftharpoons \varnothing}$$

$$[\text{C-ENVF}_2]\frac{\mathsf{env}_S \vdash \mathsf{env}'_F \rightleftharpoons \Gamma'_F}{\mathsf{env}_S \vdash \mathsf{env}'_F, p : v \rightleftharpoons \Gamma'_F, p : \mathsf{var}(B_s)}$$

where:

$$v \in \mathsf{ANames} \implies v \in \mathrm{dom}\,(\mathsf{env}_S)$$

$$[\text{C-ENVM}_1]\frac{}{\mathsf{env}_M^{\varnothing} \rightleftharpoons \varnothing}$$

$$[\text{C-ENVM}_2]\frac{\mathsf{env}_S \vdash \mathsf{env}'_M \rightleftharpoons \Gamma'_F}{\mathsf{env}_S \vdash \mathsf{env}'_M, f : (\tilde{x}, S) \rightleftharpoons \Gamma'_M, f : \mathsf{proc}(\widetilde{B}_{\tilde{s}}) : s}$$

where:

$$\mathcal{F}_A(S) \subseteq \mathrm{dom}\,(\mathsf{env}_S)$$

Figure 8.20: Consistency rules for env_F and env_M.

definitions of fields and methods for a particular contract. The consistency judgments are of the form

$$\mathsf{env}_S \vdash \mathsf{env}_X \rightleftharpoons \Gamma_X$$

for $X \in \{F, M\}$. Note the following:

- In rule [C-ENVF$_2$] we check that each field definition in Γ_F has a corresponding field declaration in env_F; and furthermore that *if* the value v stored in the field p is an address, then this address must also be in the domain of env_S. The first check ensures that all declared fields also have an implementation, whilst the second check ensures that all addresses appearing as values also contain a contract.
- In rule [C-ENVM$_2$] we check that each method signature has a corresponding implementation. Furthermore, in the side condition, we require that any free addresses occurring in the method body S must also exist in env_S. This ensures that even if a 'hard-coded' address appears in S, there must still be a contract on that address. This is necessary, since well-typedness alone only ensures that the address has the correct type for its usage, but not that it contains an implementation.

Next, we can define the consistency rules for env_{TSV}, which are given in Figure 8.21. Note that we use the notation $f|_{\mathcal{X}}$ to denote the *restriction* of the domain of a function f to the set of values in $\mathcal{X}$. We use this to extract the field, resp. method, signatures from an inner type environment Γ_I, corresponding to the declaration of

$$[\text{C-ENVV}_1]\frac{}{\mathsf{env}_S \vdash \mathsf{env}_V^{\emptyset}}$$

$$[\text{C-ENVV}_2]\frac{\mathsf{env}_S \vdash \mathsf{env}_V}{\mathsf{env}_S \vdash \mathsf{env}_V, x : v}$$
where:
$v \in \mathsf{ANames} \implies v \in \mathrm{dom}(\mathsf{env}_S)$

$$[\text{C-ENVS}_1]\frac{}{\Gamma; \mathsf{env}_S \vdash \mathsf{env}_S^{\emptyset}}$$

$$[\text{C-ENVS}_2]\frac{\Gamma; \mathsf{env}_S \vdash \mathsf{env}'_S \quad \mathsf{env}_S \vdash \mathsf{env}_F \rightleftharpoons \Gamma_I|_{\mathsf{FNames}}}{\Gamma; \mathsf{env}_S \vdash \mathsf{env}'_S, X : \mathsf{env}_F} \begin{pmatrix} \Gamma(X) = I_s \\ \Gamma(I) = \Gamma_I \end{pmatrix}$$

$$[\text{C-ENVT}_1]\frac{}{\Gamma; \mathsf{env}_S \vdash \mathsf{env}_T^{\emptyset}}$$

$$[\text{C-ENVT}_2]\frac{\Gamma; \mathsf{env}_S \vdash \mathsf{env}'_T \quad \mathsf{env}_S \vdash \mathsf{env}_M \rightleftharpoons \Gamma_I|_{\mathsf{MNames}}}{\Gamma; \mathsf{env}_S \vdash \mathsf{env}'_T, X : \mathsf{env}_M} \begin{pmatrix} \Gamma(X) = I_s \\ \Gamma(I) = \Gamma_I \end{pmatrix}$$

$$[\text{C-ENVSV}]\frac{\Gamma; \mathsf{env}_S \vdash \mathsf{env}_S \quad \mathsf{env}_S \vdash \mathsf{env}_V}{\Gamma \vdash \mathsf{env}_{SV}}$$

$$[\text{C-ENVTSV}]\frac{\Gamma; \mathsf{env}_S \vdash \mathsf{env}_T \quad \Gamma; \mathsf{env}_S \vdash \mathsf{env}_S \quad \mathsf{env}_S \vdash \mathsf{env}_V}{\Gamma \vdash \mathsf{env}_{TSV}}$$
where:
$\mathrm{dom}(\mathsf{env}_T) = \mathrm{dom}(\mathsf{env}_S)$

Figure 8.21: Consistency rules for the environments env_{TSV}.

an interface I, in rules [C-ENVS$_2$] and [C-ENVT$_2$], which then invoke the consistency rules from Figure 8.20 in their premises. The rules are otherwise straightforward:

- Rules [C-ENVV$_1$] and [C-ENVV$_2$] are used to check the consistency of the variable environment env_V, which simply consists of ensuring that any addresses appearing as values also exist in the domain of env_S, similar to the rule [C-ENVF$_2$] above. Thus, the judgment is relative to env_S.
- Rules [C-ENVS$_1$] and [C-ENVS$_2$] are used to check the consistency of env_S, which contains all field declarations of all contracts. It must be judged consistent relative to *itself*, as well as to Γ, because we need to ensure both that all declared fields have an implementation, *and* that all addresses appearing as values refer to actual contracts.
- Rules [C-ENVT$_1$] and [C-ENVT$_2$] are used to check the consistency of env_T, which contains all method declarations of all contracts. Like [C-ENVS$_2$], we extract the relevant method declarations from Γ for each collection of declarations env_M, which then must correspond.
- Finally, we have the two short-hand rules [C-ENVSV] and [C-ENVTSV], which are used to judge consistency of a collection of environments env_{SV}, resp. env_{TSV}, together. Notably, in both rules, env_S must be judged consistent relative to *itself*; and in the latter rule we also require that the domains of env_T and env_S must coincide. This must be the case, if env_T and env_S are derived from the same collection of contracts.

Well-typedness and consistency together ensure that all addresses actually contain contracts with appropriate interface types, and that all contracts also implement all members of their interfaces. Unless otherwise noted, we shall in the following only consider consistent environments env_{TSV}.

8.4.6 Safety and subject reduction

In an ordinary syntactic type system, we would now proceed to show two results, assuming that $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_{TSV}$ (well-typedness), and $\Gamma \vdash \mathsf{env}_{TSV}$ (consistency), both hold for the environments; namely:

- Safety for expressions: $\Sigma;\Gamma;\Delta \vdash e : B_s$ and $\langle e, \mathsf{env}_{SV}\rangle \to v$ imply that v indeed is of type B (or a subtype thereof), and no variable of a level higher than s was read in the evaluation of e.
- Subject reduction for stacks: $\Sigma;\Gamma;\Delta \vdash Q$, and $\langle Q, \mathsf{env}_{TSV}\rangle \to \langle Q', \mathsf{env}'_{TSV}\rangle$ imply that
  - $\Sigma;\Gamma;\Delta' \vdash Q'$, and

- $\Sigma; \Gamma; \varnothing \vdash \mathsf{env}'_S$, and
- $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_V$

where either $\Delta' = \Delta$, or Δ' is derived from the operation at the top of the stack in the cases of method calls, method returns, variable declarations and end of a variable scope.

However, the syntactic type system is still unable to type the `fallback` function in the cases where the `id` and `args` 'magic variables' appear in its body. Therefore, we shall instead proceed to give a semantics of our types by means of a *typed semantics* for expressions and stacks, by merging the type rules from Section 8.4.4 with the small-step semantics (Figures 8.8-8.10). This will give us *runtime type safety*, which will include all safe usages of the `fallback` function. Then we will proceed to show that static type safety, given by the rules in Figures 8.15-8.17, implies runtime type safety.

8.5 Semantics of types for expressions

To illustrate the semantic typing approach, we begin by giving the runtime safety rules for expressions, which are in Figure 8.22. Transitions are now of the form

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV} \rangle \to v$$

which expresses that e evaluates to a value v *of type* B_s, or a subtype thereof, given env_{SV} and the type assumptions in $\Sigma; \Gamma; \Delta$. We shall refer to the type B_s on the turnstile as the *runtime type* of the expression, which may be different from its *actual type*, due to subtyping. The runtime type is a type which the expression is safely able to *behave as* at runtime. We shall furthermore refer to the B component as the *runtime base type* and the s component as the *runtime security level*.

The rules have some quite complicated side conditions, so we shall again give some intuitions regarding their formulation: As in the type rules for expressions (Figure 8.15), we wish to limit the use of the subtyping relation to expressions, and furthermore we must ensure that the coercion property holds. Thus, all rules have a side condition of the form $s' \sqsubseteq s$, which ensures that the transition can be concluded for any level s or *higher*.

- In [R-VAL], we are evaluating a hard-coded value, so we use $\textsc{TypeOf}_\Gamma(\cdot)$ to obtain its type. However, as this value could be an *address* X, and its type therefore would be an interface I, we need to invoke subtyping on the actual type, in case the runtime base type B in fact were a supertype of I, rather than I itself.

$$[\text{R-VAL}]\frac{\Sigma \vdash B' <: B}{\Sigma;\Gamma;\Delta \vDash_{B_s} \langle v, \mathsf{env}_{SV}\rangle \to v} \left(\begin{array}{c} \textsc{TypeOf}_\Gamma(v) = (B', s') \\ s' \sqsubseteq s \end{array}\right)$$

$$[\text{R-VAR}]\frac{\Sigma \vdash B_1 <: B_2 <: B}{\Sigma;\Gamma;\Delta \vDash_{B_s} \langle x, \mathsf{env}_{SV}\rangle \to v} \left(\begin{array}{c} \mathsf{env}_V(x) = v \\ \textsc{TypeOf}_\Gamma(v) = (B_1, s_1) \\ \Delta(x) = \mathsf{var}(B_2, s_2) \\ s_1 \sqsubseteq s_2 \sqsubseteq s \end{array}\right)$$

$$[\text{R-FIELD}]\frac{\begin{array}{c}\Sigma \vdash B_1 <: B_2 <: B \\ \Sigma;\Gamma;\Delta \vDash_{I_s} \langle e, \mathsf{env}_{SV}\rangle \to X\end{array}}{\Sigma;\Gamma;\Delta \vDash_{B_s} \langle e.p, \mathsf{env}_{SV}\rangle \to v} \left(\begin{array}{c} \mathsf{env}_S(X)(p) = v \\ \textsc{TypeOf}_\Gamma(v) = (B_1, s_1) \\ \Gamma(I)(p) = \mathsf{var}(B_2, s_2) \\ s_1 \sqsubseteq s_2 \sqsubseteq s \end{array}\right)$$

$$[\text{R-OP}]\frac{\Sigma;\Gamma;\Delta \vDash_{\widetilde{B}_{s,\ldots,s}} \langle \tilde{e}, \mathsf{env}_{SV}\rangle \to \tilde{v} \quad \mathsf{op}(\tilde{v}) \to_{\mathrm{op}} v}{\Sigma;\Gamma;\Delta \vDash_{B_s} \langle \mathsf{op}(\tilde{e}), \mathsf{env}_{SV}\rangle \to v} \left(\vdash \mathsf{op} : \widetilde{B} \to B\right)$$

Figure 8.22: Typed semantics of expressions.

- In [R-VAR], we have a similar situation to the one in [R-VAL], but now we need to invoke subtyping twice: First to ensure that the actual value v in fact was able to be stored in the variable x; and, secondly, to obtain the runtime base type B, in case v were an address and the runtime type were a supertype of its interface. The check on the value is necessary, since we cannot know whether env_V in fact is well-typed, so we must check this at runtime as well whenever an entry is read.

  Likewise, we must require that the three security levels form a chain $s_1 \sqsubseteq s_2 \sqsubseteq s$; firstly, since we must have that $s_1 \sqsubseteq s_2$ to ensure that the value in fact was of a lower level than the container x; and secondly, because we must have that $s_2 \sqsubseteq s$ to ensure that the coercion property holds.

- [R-FIELD] is similar to the situation in [R-VAR], except that we must now, in addition, have that the *path* e evaluates to an address X of type I_s. Thus, we require a runtime type I_s in the evaluation. Now, because of subtyping and the coercion property, it might be the case that the *actual* type of X were some interface I' and of level s' such that $\Sigma \vdash I' <: I$ and $s' \sqsubseteq s$, but in that case the transition can be concluded for I_s as well. Note that by requiring the runtime level s in the evaluation of e, we are in effect ensuring that e can be evaluated to a level that is *lower* than, or equal to, s, which ascertains that higher-level parts of the program cannot alter the *path* to the field being read. This, again, is in accordance with the type rule [T-FIELD].

- In [R-OP], we require the runtime types $\widetilde{B}_{s,\ldots s}$ for the operands $\tilde{v}$, and in the side condition we require that $\vdash \text{op} : \widetilde{B} \to B$. Thus, we make use of two assumptions: Firstly, the safety requirement for operations (Definition 57), which assures us that the resulting value v in fact will be of type B; and, secondly, that v *cannot* be an address, since no operation is allowed to *return* an address value. Hence, we can let the type of v, obtained from the signature of op, equal the runtime base type, because B must be one of int or bool, on which subtyping can only be reflexive. Furthermore, by the definition of $\textsc{TypeOf}_\Gamma(\cdot)$ (Definition 54), the level of v can only be $s_\bot$, and therefore it always holds that the level of v is below, or equal to, the runtime level s, since by definition $s_\bot \sqsubseteq s$.

  Lastly, the runtime levels $s, \ldots, s$ required for the operands should be read as the same as the level s in the conclusion. In effect, we thus require that all operands can be evaluated at *some* levels $s_1, \ldots, s_n$, which must all be equal to, or lower than, s.

8.5.1 Properties of typed transitions

The rules of the typed semantics in Figure 8.22 are simply the ordinary semantic rules from Figure 8.8, equipped with the type restrictions imposed by the type rules from Figure 8.15. Hence, it should be immediately clear from a comparison of the respective rules that every transition, that can be concluded by the typed semantic rules, also can be concluded by the ordinary, 'untyped' semantics. We can state and prove this formally:

Theorem 19 (Semantic compatibility for expressions)**.** *For all type environments* Σ, Γ, Δ, *types* B_s, *expressions* e, *and states* env_{SV}*:*

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV} \rangle \to v \implies \langle e, \mathsf{env}_{SV} \rangle \to v.$$

The proof is by induction on the rules of the typed semantics of expressions (Figure 8.22). It can be found in Section 8.9.1.

Although quite obvious from the definition of the typed semantics, we can also formally show that the coercion property holds for both the data type component and the security level. This assures us that subtyping indeed works as expected:

Theorem 20 (Coercion property for expressions)**.** *If* $\Sigma \vdash B_1 <: B_2$ *and* $s_1 \sqsubseteq s_2$*, then*

$$\Sigma; \Gamma; \Delta \vDash_{(B_1, s_1)} \langle e, \mathsf{env}_{SV} \rangle \to v \implies \Sigma; \Gamma; \Delta \vDash_{(B_2, s_2)} \langle e, \mathsf{env}_{SV} \rangle \to v.$$

The proof is by induction on the rules of the typed semantics (Figure 8.22). It can be found in Section 8.9.2.

An expression configuration $\langle e, \mathsf{env}_{SV} \rangle$ is *safe* precisely when it yields a value v by the typed semantics; i.e. the rules ensure both that v indeed is of type B, or a

subtype thereof, and that all containers *read from* in the evaluation of e are of security level s or lower. This precisely captures the intended meaning of safety implied by the expression type B_s. Again, we can quite easily show that this property holds for transitions in the typed semantics:

Theorem 21 (Runtime safety for expressions)**.** *For all type environments* Σ, Γ, Δ, *types* B_s, *expressions* e, *and states* env_{SV}*:*

$$\Sigma;\Gamma;\Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV}\rangle \to v \implies \textsc{TypeOf}_{\Gamma}(v) = (B_1, s_1)$$

for some B_1 *and* s_1*, such that* $\Sigma \vdash B_1 <: B$ *and* $s_1 \sqsubseteq s$*, and every container read from in the evaluation of* e *is of type* $\mathsf{var}(B_2, s_2)$*, for some* B_2 *and* s_2*, such that* $s_2 \sqsubseteq s$*.*

The proof is by induction on the rules of the typed semantics (Figure 8.22). It can be found in Section 8.9.3.

Theorem 21 ensures that if an expression e can be evaluated *at* a given type, then the resulting value v will indeed be of that type. However, suppose v is an address X. It is then technically possible that X could have a type in Γ, but not actually contain an *implementation* in env_S. This trivial error is easily prevented by also requiring well-typedness and consistency of env_{SV}, and that e does not contain any free names.

Corollary 13 (Consistency of expression values)**.** *If*

- $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_{SV}$ *(well-typedness), and*
- $\Gamma \vdash \mathsf{env}_{SV}$ *(consistency), and*
- $\Sigma;\Gamma;\Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV}\rangle \to v$*,*

then $v \in \mathsf{ANames} \implies v \in \mathrm{dom}(\mathsf{env}_S)$.

Proof. By Theorem 21, we know that $\textsc{TypeOf}_{\Gamma}(v) = (B_1, s_1)$ such that $\Sigma \vdash B_1 <: B$ and $s_1 \sqsubseteq s$. Thus v is ensured to have a type. If v is an address X, then by the definition of $\textsc{TypeOf}_{\Gamma}(\cdot)$ (Definition 54), X must be an interface type I from Γ. Well-typedness and consistency of env_{SV} then together ensure that every interface I in Γ also must have an implementation in env_S. Thus we conclude that $v \in \mathrm{dom}(\mathsf{env}_S)$. □

The typed semantics ensures runtime safety for expression *configurations*, which not only concerns a particular expression e, but also a *particular* choice of environments env_{SV} that must be well-typed and consistent. However, given a collection of type environments Σ, Γ and Δ, we can also *construct* environments env_{SV} such that well-typedness and consistency are ensured to hold, and thereby abstract away from a particular choice of these environments. The construction is tedious but straightforward, since it is just a matter of following the rules of well-typedness and consistency 'in reverse' and, for each type $\mathsf{var}(B_s)$, choosing any value inhabiting the

type B_s. The only detail to keep in mind here is that whenever an address name X is chosen as a value, then it obviously must be a *declared* address, i.e. one that also will contain an implementation in the constructed env_S.

Definition 59 (Construction of env_{SV})**.** Given a Σ, Γ and Δ, let $\mathcal{D} \triangleq \mathrm{dom}(\Gamma) \cap \mathsf{ANames}$ denote the set of *declared* address names in Γ.

- Let env_V be a list of pairs $x_1 : v_1, \ldots, x_n : v_n$ such that $\{x_1, \ldots, x_n\} = \mathrm{dom}(\Delta)$, and choose each value v_i such that if $\Delta(x_i) = \mathsf{var}(B_2, s_2)$ then $\textsc{TypeOf}_\Gamma(v_i) = (B_1, s_1)$ and $\Sigma \vdash B_1 <: B_2$ and $s_1 \sqsubseteq s_2$, and $v_i \in \mathsf{ANames} \implies v_i \in \mathcal{D}$.
- Let env_S be a list of pairs $X_1 : \mathsf{env}_F^1, \ldots, X_n : \mathsf{env}_F^n$ such that $\{X_1, \ldots, X_n\} = \mathcal{D}$, and let each environment env_F^i be constructed as follows:
  - Assume $\Gamma(X_i) = I_s$ and $\Gamma(I) = \Gamma_F^I, \Gamma_M^I$, where Γ_F^I is the segment of the type environment describing the field declarations of I.
  - Let env_F^i be a list of pairs $p_1 : v_1, \ldots, p_n : v_n$ such that $\{p_1, \ldots, p_n\} = \mathrm{dom}(\Gamma_F^I)$, and choose each value v_i such that if $\Gamma_F^I(x_i) = \mathsf{var}(B_2, s_2)$ then $\textsc{TypeOf}_\Gamma(v_i) = (B_1, s_1)$ and $\Sigma \vdash B_1 <: B_2$ and $s_1 \sqsubseteq s_2$, and $v_i \in \mathsf{ANames} \implies v_i \in \mathcal{D}$. ■

Lemma 57 (Correctness of construction)**.** *Given a* Σ, Γ *and* Δ, *if* env_{SV} *is built according to Definition 59, then* env_{SV} *satisfies the following:*

- $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$ *(well-typedness), and*
- $\Gamma \vdash \mathsf{env}_{SV}$ *(consistency).*

Using the construction of Definition 59, we can now abstract away from the specific values used in the evaluation of an expression configuration $\langle e, \mathsf{env}_{SV} \rangle$ and thereby obtain a result about the safety of the expression e itself:

Theorem 22 (Safe state abstraction)**.** *For all environments* Σ, Γ *and* Δ, *and* $\mathsf{env}_{SV}^1, \mathsf{env}_{SV}^2$ *built according to Definition 59, it holds that*

$$\begin{aligned} &\forall v_1 \,.\, \Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV}^1 \rangle \to v_1 \\ \implies &\exists v_2 \,.\, \Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV}^2 \rangle \to v_2. \end{aligned}$$

Proof sketch. By induction on the rules of the typed semantics (Figure 8.22). By construction, env_{SV}^1 and env_{SV}^2 contain the same named entries (fields and variables), so since the configuration $\langle e, \mathsf{env}_{SV}^1 \rangle$ was executable, the configuration $\langle e, \mathsf{env}_{SV}^2 \rangle$ cannot be stuck due to a missing variable name. A field or variable in env_{SV}^2 could contain an address X not present in env_S^1, but by Lemma 57 we know that env_{SV}^2 is well-typed and consistent w.r.t. Σ, Γ and Δ, which by Corollary 13 ensures that X must have an implementation in env_S^2, and by Theorem 21 that X has the appropriate type. Thus, this cannot lead to the configuration being stuck either. □

$$[\text{EQ-ENV}_V^{\emptyset}]\frac{}{\Delta \vdash \text{env}_V^{\emptyset} =_s \text{env}_V^{\emptyset}}$$

$$[\text{EQ-ENV}_V]\frac{\Delta \vdash \text{env}_V^1 =_s \text{env}_V^2}{\Delta \vdash \text{env}_V^1, x : v_1 =_s \text{env}_V^2, x : v_2} \left(\begin{array}{l} \Delta(x) = \text{var}(B_{s'}) \\ s' \sqsubseteq s \implies v_1 = v_2 \end{array} \right)$$

$$[\text{EQ-ENV}_F^{\emptyset}]\frac{}{\Gamma \vdash_I \text{env}_F^{\emptyset} =_s \text{env}_F^{\emptyset}}$$

$$[\text{EQ-ENV}_F]\frac{\Gamma \vdash_I \text{env}_F^1 =_s \text{env}_F^2}{\Gamma \vdash_I \text{env}_F^1, p : v_1 =_s \text{env}_F^2, p : v_2} \left(\begin{array}{l} \Gamma(I)(p) = \text{var}(B_{s'}) \\ s' \sqsubseteq s \implies v_1 = v_2 \end{array} \right)$$

$$[\text{EQ-ENV}_S^{\emptyset}]\frac{}{\Gamma \vdash \text{env}_S^{\emptyset} =_s \text{env}_S^{\emptyset}}$$

$$[\text{EQ-ENV}_S]\frac{\Gamma \vdash \text{env}_S^1 =_s \text{env}_S^2 \quad \Gamma \vdash_I \text{env}_F^1 =_s \text{env}_F^2}{\Gamma \vdash \text{env}_S^1, X : \text{env}_F^1 =_s \text{env}_S^2, X : \text{env}_F^2} \ (\Gamma(X) = I_{s'})$$

$$[\text{EQ-ENV}_{SV}]\frac{\Gamma \vdash \text{env}_S^1 =_s \text{env}_S^2 \quad \Delta \vdash \text{env}_V^1 =_S \text{env}_V^2}{\Gamma; \Delta \vdash \text{env}_{SV}^1 =_s \text{env}_{SV}^2}$$

Figure 8.23: Rules for the s-parameterised equivalence relation.

The actual values v_1 and v_2 produced in the two transitions in Theorem 22 may of course differ, but by Theorem 21 above, their *types* will be the same. This intuitively follows because the typed semantics only restricts the *types* of values read from env_{SV}, and not the actual values. The existence of a typed transition with env_{SV}^1 implies that all the values read from this environment in the evaluation of e are in accordance with the type restrictions in Σ, Γ, Δ; and choosing any other state env_{SV}^2, which satisfies the same restrictions, will therefore still yield a transition.

Finally, it is worth recalling that the purpose of the type system from Chapter 6 was to ensure non-interference, which is a 2-property; i.e. on a *pair* of states. For expressions, this property says that if we have two states, env_{SV}^1 and env_{SV}^2, that agree on all values of containers of level s or lower, then any expression that is *safe* to execute at level s, will yield the same value when executed in a configuration with either of the two states. To express this property for our typed semantics, we need the notion of an s-parametrised equivalence relation on states, written $\Gamma; \Delta \vdash \text{env}_{SV}^1 =_s \text{env}_{SV}^2$. It is defined by the rules in Figure 8.23, and is mostly similar to the definition from Chapter 6, but adapted to our usage of two type environments, Γ and Δ. Runtime non-interference for expressions now follows as a simple corollary of Theorem 21:

Corollary 14 (Runtime non-interference for expressions)**.** *If*

- $s_1 \sqsubseteq s_2$, *and*
- $\Gamma; \Delta \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, *and*
- $\Sigma; \Gamma; \Delta \vDash_{B_{s_1}} \langle e, \mathsf{env}^1_{SV} \rangle \to v_1$, *and*
- $\Sigma; \Gamma; \Delta \vDash_{B_{s_1}} \langle e, \mathsf{env}^2_{SV} \rangle \to v_2$,

then $v_1 = v_2$.

Proof. By assumption, all values of entries of level s_2 or lower in env^1_{SV} and env^2_{SV} agree, and by Theorem 21, only entries of level s_1 or lower are read in the transitions, and $s_1 \sqsubseteq s_2$. Thus, the two expressions use the same values and therefore also produces the same result. □

8.5.2 Typing interpretations and compatibility for expressions

In the previous section we have shown a collection of results, which together assure us that the definition of the typed semantics is sensible and indeed expresses our intended meaning of the types:

- Theorem 21 expresses that expression configurations $\langle e, \mathsf{env}_{SV} \rangle$, that have a transition in the typed semantics (w.r.t. a particular Σ, Γ, Δ, and a type B_s), are indeed safe, in the sense that they behave in accordance with the meaning of the type B_s. Furthermore, by Corollary 13, if env_{SV} is well-typed and consistent, then so is the value produced by evaluating the configuration. Thus, these results replace the usual safety theorem (i.e. well-typedness implies now-safety) in the syntactic approach.
- Theorem 20 assures us that the typed transitions respect the subtyping relation and the ordering relation on security levels, in accordance with the intended meaning of the type B_s; i.e. we can always choose a supertype of B and a higher security level than s.
- Theorem 19 states that our typed semantics does not enable any transitions that *cannot* be concluded by the untyped semantics. Hence, the set of *safe* expression configurations form a subset of the set of *all* expression configurations.

Finally, Theorem 22 allows us to abstract away from any *specific* choice of environments env_{SV} and instead focus on expressions that have a transition for *any* sensible choice of these environments. By this theorem, if an expression e has a transition for some env^1_{SV} constructed according to Definition 59, then it will have a transition for *all* similarly constructed environments env^2_{SV}. In a sense, these environments can be

thought of as representing all meaningful 'inputs' to the expression, and for any such appropriately shaped input, a safe expression will yield a value of the correct type. We can now define the set of such safe expressions by using the notion of *typing interpretations* in the style of Caires [23]:

Definition 60 (Typing interpretation for expressions)**.** A *typing interpretation* for expressions e, for some $\Sigma; \Gamma; \Delta$ and a type B_s, written $\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$, is a set of expressions satisfying that $e \in \mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$ implies that there exists an env_{SV} built according to Definition 59, and a value v such that

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV} \rangle \to v.$$

The *typing* of expressions w.r.t. a given Σ, Γ, Δ and B_s, written $\nu\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$, is the union of all typing interpretations $\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$:

$$\nu\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta} \triangleq \bigcup \mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$$

We write $\Sigma; \Gamma; \Delta \vDash e : B_s$, if there exists a typing interpretation $\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$ containing e. Thus:

$$\Sigma; \Gamma; \Delta \vDash e : B_s \triangleq e \in \nu\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$$ ■

A typing interpretation for expressions, w.r.t. a collection of type environments Σ, Γ, Δ and a type B_s, is simply any set of expressions that can be evaluated to a value of a particular type B_s (or subtype thereof), when executed in a configuration with any appropriately shaped state env_{SV}. In other words, any expression from such a set will safely *behave as* a value of the type B_s according to the typed semantics, regardless of the actual values bound to its names, as long as they are of the appropriate types. This is exactly what we would expect it to *mean*, when we say an expression inhabits the type B_s.

However, Definition 60 does not tell us how to actually *decide* whether an expression e inhabits a given type B_s, apart from exhibiting a typing interpretation $\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$ containing e. To remedy this situation, we shall therefore now state and show a theorem, which expresses that our static type rules from Figure 8.15 are *compatible* with Definition 60, meaning they indeed imply safety:

Theorem 23 (Compatibility for expressions)**.**

$$\Sigma; \Gamma; \Delta \vdash e : B_s \implies \Sigma; \Gamma; \Delta \vDash e : B_s.$$

By unfolding the definition of $\Sigma; \Gamma; \Delta \vDash e : B_s$, the statement to be shown becomes

$$\Sigma; \Gamma; \Delta \vdash e : B_s \implies \Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV} \rangle \to v$$

for some v and for any appropriately shaped env_{SV}. This is shown by induction on the type rules (Figure 8.15). The proof can be found in Section 8.9.4.

Combining Theorems 19, 23, and 21 gives us the equivalent of the soundness theorem for expressions, which now follows as a simple corollary:

Corollary 15 (Static soundness for expressions)**.** *If*

- $\Sigma; \Gamma; \Delta \vdash e : B_s$, *and*
- $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$, *and*
- $\Gamma \vdash \mathsf{env}_{SV}$, *and*
- $\mathcal{F}_A(e) \subseteq \mathrm{dom}\,(\mathsf{env}_S)$, *and*
- $\mathcal{F}_V(e) \subseteq \mathrm{dom}\,(\mathsf{env}_V)$,

then there exists some v such that $\langle e, \mathsf{env}_{SV}\rangle \to v$, and $\textsc{TypeOf}_\Gamma(v) = (B_1, s_1)$ for some B_1 and s_1 such that $\Sigma \vdash B_1 <: B$ and $s_1 \sqsubseteq s$, and every container, that will be read from in the evaluation of e, is of type $\mathsf{var}(B_2, s_2)$, for some B_2, such that $s_2 \sqsubseteq s$.

Proof. By Theorem 23, we have that

$$\Sigma; \Gamma; \Delta \vdash e : B_s \implies \Sigma; \Gamma; \Delta \vDash e : B_s$$

Unfolding the definition of $\Sigma; \Gamma; \Delta \vDash e : B_s$ (Definition 60) yields that there must exist a typing interpretation $\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$ containing e, which again means that the transition

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV}\rangle \to v$$

can be concluded by the typed semantics for any consistent and well-typed env_{SV} containing at least all the names used in e. By Theorem 21, this transition implies that $\textsc{TypeOf}_\Gamma(v) = (B_1, s_1)$ and $s_1 \sqsubseteq s$ and every container read from in e is of level s or lower. Finally, by Theorem 19, we have that

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV}\rangle \to v \implies \langle e, \mathsf{env}_{SV}\rangle \to v \qquad \square$$

Finally, we can also obtain the equivalent of the non-interference theorem for expressions from Chapter 6 (Theorem 14) as a simple corollary of Corollaries 14–15:

Corollary 16 (Static non-interference for expressions)**.** *If*

- $\Sigma; \Gamma; \Delta \vdash e : B_s$, *and*
- $\Sigma; \Gamma; \Delta \vdash \mathsf{env}^1_{SV}, \Gamma \vdash \mathsf{env}^1_{SV}, \mathcal{F}_A(e) \subseteq \mathrm{dom}\left(\mathsf{env}^1_S\right), \mathcal{F}_V(e) \subseteq \mathrm{dom}\left(\mathsf{env}^1_V\right)$, *and*
- $\Sigma; \Gamma; \Delta \vdash \mathsf{env}^2_{SV}, \Gamma \vdash \mathsf{env}^2_{SV}, \mathcal{F}_A(e) \subseteq \mathrm{dom}\left(\mathsf{env}^2_S\right), \mathcal{F}_V(e) \subseteq \mathrm{dom}\left(\mathsf{env}^2_V\right)$, *and*

- $\Gamma; \Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$,

then $\langle e, \mathsf{env}^1_{SV} \rangle \to v$ *and* $\langle e, \mathsf{env}^2_{SV} \rangle \to v$.

Proof. From the assumptions, and by two applications of Corollary 15, we get that

$$\langle e, \mathsf{env}^1_{SV} \rangle \to v_1$$
$$\langle e, \mathsf{env}^2_{SV} \rangle \to v_2$$

such that no container of a level higher than s was read from env^1_{SV} resp. env^2_{SV} in the two evaluations of e. By assumption, $\Gamma; \Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$, so the two states agree on all entries of level s or lower. Thus, the two evaluations of e will use the same values, and will therefore yield the same result, so $v_1 = v_2$. □

With this last result, we have recovered the properties ensured by syntactic well-typedness for expressions in Chapter 6, but through a quite different route; namely by starting from a semantics of types for expressions, given in terms of a typed semantics, that is compatible with the untyped semantics and ensures that the restrictions imposed by the type predicates are in fact observed.

8.6 Semantics of types for statements and stacks

The previous section has illustrated the key steps in our approach:

1. define runtime type safety in terms of the semantics;
2. define typing interpretations as sets of configurations that are runtime type safe (for all steps);
3. define typing as the union of all typing interpretations;
4. show that the syntactic type rules are compatible with the semantic interpretation.

We shall now perform the same steps with the syntactic category of *stacks* Q, but rather than showing a single compatibility theorem (as we did in Theorem 23), we shall instead show a series of compatibility *lemmas*, which can be used to conclude safety in a compositional way, similar to how the syntactic type rules are used to judge well-typedness. This will give us a form of 'semantic type rules,' which, unlike the syntactic type rules, will allow us to use typing interpretations to satisfy premises for which safety cannot be concluded by the rules.

However, firstly, we need to make two minor changes to the syntax of stacks: In the small-step semantics of Chapter 7 (recalled and extended in Figures 8.9–8.10), a stack Q is either the empty stack $\bot$, or it is a stack Q' with either a statement S

or one of the two special stack symbols $\mathsf{del}(x)$ and env_V at the top. The latter two denote the end-of-scope of a locally declared variable x, and the return of a method call (which restores the variable environment that existed prior to the method call). We shall now make the following changes:

1. In the untyped small-step semantics, we rather off-handedly mentioned that we assume that type information for locally declared variables is stored in env_V, because this is needed for the syntactic type system (as in Chapter 7). However, in the typed semantics, we have this information directly available in Δ, so we shall simply allow the type environment itself to be placed on the stack beside env_V in method calls. Thus, the return symbol will now be a pair, (env_V, Δ). This is in principle not different from extracting the type information from env_V, so this change makes no difference w.r.t. the semantics, but merely serves to simplify the presentation.

2. Statements can execute at different security levels. Specifically, some statements may need to be able to execute at a *higher* security level s (comparable to executing with heightened privileges), if the statement is placed in a context where higher-level information is available. To be able to capture this in the typed semantics, we shall also allow *security levels* to be stored on the stack, so the lower security level can be restored, once the statement executing in the higher-level security context terminates.

Thus we update the syntax of stacks as follows:

Definition 61 (Syntax of stacks)**.** A stack Q is given by the syntax:

$$
\begin{array}{lll}
Q ::= \bot & & \text{empty stack} \\
\quad \mid S; Q & & \text{statement} \\
\quad \mid \mathsf{del}(x); Q & & \text{end of scope} \\
\quad \mid (\mathsf{env}_V, \Delta); Q & & \text{return} \\
\quad \mid s; Q & & \text{restore}
\end{array}
$$

■

Now let us again remind ourselves of the meaning of 'safety' for statements S w.r.t. a given security level s:

- All containers *written to* within S are of level s or *higher*. Note that this only needs to hold for *free* variables (as well as fields), but not for variables declared at runtime, since their scopes will end before the execution finishes.

- Branching, loops and method calls can all induce a difference in the structure of the stack, and thereby directly or indirectly in the memory. These 'guarded statements' can therefore only depend on information that is of a *lower* level than the level of the variables they modify, which is the level of the statement

under the guard. Information from the expression guard e is indirectly available within the guarded statement S, since the execution of S is contingent upon the value of e; hence, the expression guards in `if` and `while` constructs, and the 'object path' e in method calls `call e.f` must all be runtime type safe up to some level s', which is *lower than*, or equal to, the level of the statement they guard, to avoid a potential downward information flow. This corresponds to requiring, that the statement S under the guard must be executable at level s' or higher.

We now give the typed semantics of stack configurations, which ensures that these two properties hold for any transition. The rules are given in Figure 8.24, Figure 8.25 and Figure 8.26. Note that transitions here are of the form

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'} \langle Q', \mathsf{env}'_{TSV} \rangle$$

because the type environment Δ also may be changed during the execution, if a new variable is declared, or the scope of a local variable ends, or if we enter or return from a method call; and the security level may change, if a statement is executed at a different security level. Thus, we also need to carry the type environment and the security level across the transition. For the sake of consistency, we also include Σ and Γ, as well as env_T, even though neither of these will change during the execution of a program.

Most of the rules are straightforward variants of their untyped semantics counterparts, with security level checks added to ensure adherence to the meaning of the types. The intuitions behind these checks are mostly the same as those given for the type rules for statements (see Section 8.4.4). However, there are a few key points to note:

- In the rules for guarded statements, i.e. [R-IF], [R-WHILE$_\mathrm{T}$], [R-CALL], [R-DCALL], and [R-FCALL], we allow the security level to be raised from s to s', for some level s' such that $s \sqsubseteq s'$. The original level s is then pushed onto the stack *before* the command inside the guarded statement (or the method body, in the cases of method calls). The rule [R-RESTORE] then restores the original level s, and this rule therefore has the side condition $s \sqsupseteq s'$, which is the converse of the condition in the other rules. This side condition is necessary, since it otherwise might be possible to have a stack of the form $S; s'; Q$ and execute it at some level s that is *lower* than s', which would mean that the command S could be executed at a level that is *too low*, because it would lead to the execution level being *raised* when s' is encountered.

- The reversed side condition of [R-RESTORE] means that the coercion property *cannot* hold for arbitrary stacks, since it precisely would disallow the execution of a stack $s'; Q$ at some level s where s is lower than s'; i.e. we cannot choose

$$[\text{R-SKIP}]\frac{}{\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{skip}; Q, \mathsf{env}_{TSV}\rangle \to \Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_{TSV}\rangle}$$

$$[\text{R-IF}]\frac{\Sigma;\Gamma;\Delta \vDash_{(\text{bool},s')} \langle e, \mathsf{env}_{SV}\rangle \to b \in \mathbb{B}}{\begin{array}{l}\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; Q, \mathsf{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta \vDash_{s'} \langle S_b; s; Q, \mathsf{env}_{TSV}\rangle\end{array}} \ (s \sqsubseteq s')$$

$$[\text{R-WHILE}_\mathsf{T}]\frac{\Sigma;\Gamma;\Delta \vDash_{(\text{bool},s')} \langle e, \mathsf{env}_{SV}\rangle \to \mathsf{T}}{\begin{array}{l}\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{while } e \texttt{ do } S; Q, \mathsf{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta \vDash_{s'} \langle S; s; \texttt{while } e \texttt{ do } S; Q, \mathsf{env}_{TSV}\rangle\end{array}} \ (s \sqsubseteq s')$$

$$[\text{R-WHILE}_\mathsf{F}]\frac{\Sigma;\Gamma;\Delta \vDash_{(\text{bool},s')} \langle e, \mathsf{env}_{SV}\rangle \to \mathsf{F}}{\begin{array}{l}\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{while } e \texttt{ do } S; Q, \mathsf{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_{TSV}\rangle\end{array}}$$

$$[\text{R-DECV}]\frac{\Sigma;\Gamma;\Delta \vDash_{(B,s')} \langle e, \mathsf{env}_{SV}\rangle \to v}{\begin{array}{l}\Sigma;\Gamma;\Delta \vDash_s \langle \mathsf{var}(B,s') \ x \ \texttt{:=} \ e \ \texttt{in} \ S; Q, \mathsf{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta, x : \mathsf{var}(B,s') \vDash_s \langle S; \mathsf{del}(x); Q, \mathsf{env}_{TS}; \mathsf{env}_V, x : v\rangle\end{array}}$$

where:

$$x \notin \mathrm{dom}\,(\mathsf{env}_V)\,, \mathrm{dom}\,(\Delta)$$

$$[\text{R-ASSV}]\frac{\Sigma;\Gamma;\Delta \vDash_{(B,s')} \langle e, \mathsf{env}_{SV}\rangle \to v}{\begin{array}{l}\Sigma;\Gamma;\Delta \vDash_s \langle x \ \texttt{:=} \ e; Q, \mathsf{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_{TS}; \mathsf{env}_V[x \mapsto v]\rangle\end{array}} \left(\begin{array}{c} x \in \mathrm{dom}\,(\mathsf{env}_V) \\ \Delta(x) = \mathsf{var}(B,s') \\ s \sqsubseteq s' \end{array}\right)$$

$$[\text{R-ASSF}]\frac{\Sigma;\Gamma;\Delta \vDash_{(B,s')} \langle e, \mathsf{env}_{SV}\rangle \to v}{\begin{array}{l}\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{this}.p \ \texttt{:=} \ e; Q, \mathsf{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]\,; \mathsf{env}_V\rangle\end{array}} \left(\begin{array}{c} s_1 \sqsubseteq s' \\ s \sqsubseteq s' \end{array}\right)$$

where:

$$\begin{aligned} X &= \mathsf{env}_V(\texttt{this}) \\ \mathsf{env}_F &= \mathsf{env}_S(X) \\ p &\in \mathrm{dom}\,(\mathsf{env}_F) \\ \mathsf{var}(I, s_1) &= \Delta(\texttt{this}) \\ \mathsf{var}(B, s') &= \Gamma(I)(p) \end{aligned}$$

Figure 8.24: Typed semantics of statements.

$$
[\textsc{r-call}]\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vDash_{(I^Y,s')} \langle e_1, \mathrm{env}_{SV}\rangle \to Y \\ \Sigma;\Gamma;\Delta \vDash_{(\mathsf{int},s')} \langle e_2, \mathrm{env}_{SV}\rangle \to n \\ \Sigma;\Gamma;\Delta \vDash_{(\widetilde{B},\tilde{s}')} \langle \tilde{e}, \mathrm{env}_{SV}\rangle \to \tilde{v}\end{array}}{\begin{array}{c}\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{call } e_1\,.\,m(\tilde{e})\,\$e_2; Q, \mathrm{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta' \vDash_{s'} \langle S; (\mathrm{env}_V, \Delta); s; Q, \mathrm{env}_T; \mathrm{env}'_{SV}\rangle\end{array}} \left(\begin{array}{c} s_1 \sqsubseteq s' \\ s \sqsubseteq s' \\ n \neq 0 \Rightarrow s' \sqsubseteq s_3, s_4 \end{array}\right)
$$

where:

$$
\begin{aligned}
X &= \mathrm{env}_V(\texttt{this}) \\
\mathrm{env}_F^X &= \mathrm{env}_S(X) \\
\mathrm{env}_F^Y &= \mathrm{env}_S(Y) \\
f &= \begin{cases} \mathrm{env}_V(m) & \text{if } m = \texttt{id} \\ m & \text{otherwise} \end{cases} \\
(\tilde{x}, S) &= \mathrm{env}_T(Y)(f) \\
|\tilde{x}| &= |\tilde{v}| = |\widetilde{B}| = k \\
\mathrm{env}'_V &= \texttt{this} : Y, \texttt{sender} : X, \texttt{value} : n, x_1 : v_1, \ldots, x_k : v_k \\
\mathrm{env}'_S &= \mathrm{env}_S\big[X \mapsto \mathrm{env}_F^X[\texttt{balance -= } n]\big]\big[Y \mapsto \mathrm{env}_F^Y[\texttt{balance += } n]\big] \\
\mathrm{var}(I^X, s_1) &= \Delta(\texttt{this}) \\
(I^Y, s_2) &= \Gamma(Y) \\
\mathrm{var}(\mathsf{int}, s_3) &= \Gamma(I^X)(\texttt{balance}) \\
\mathrm{var}(\mathsf{int}, s_4) &= \Gamma(I^Y)(\texttt{balance}) \\
\mathrm{proc}(\widetilde{B}_{\tilde{s}}) : s' &= \Gamma(I^Y)(f) \\
\Delta' &= \texttt{this} : \mathrm{var}(I^Y, s_2), \texttt{sender} : \mathrm{var}(I^X, s_1), \texttt{value} : \mathrm{var}(\mathsf{int}, s'), \\
&\quad\ x_1 : \mathrm{var}(B_1, s'_1), \ldots, x_k : \mathrm{var}(B_k, s'_k)
\end{aligned}
$$

$$
[\textsc{r-dcall}]\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vDash_{(I^Y,s')} \langle e, \mathrm{env}_{SV}\rangle \to Y \\ \Sigma;\Gamma;\Delta \vDash_{(\widetilde{B},\tilde{s}')} \langle \tilde{e}, \mathrm{env}_{SV}\rangle \to \tilde{v} \qquad \Sigma \vdash I^X <: I^Y\end{array}}{\begin{array}{c}\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{dcall } e\,.\,m(\tilde{e}); Q, \mathrm{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta' \vDash_{s'} \langle S; (\mathrm{env}_V, \Delta); s; Q, \mathrm{env}_{TS}; \mathrm{env}'_V\rangle\end{array}} \left(\begin{array}{c} s_1 \sqsubseteq s' \\ s \sqsubseteq s' \end{array}\right)
$$

where:

$$
\begin{aligned}
|\tilde{x}| &= |\tilde{v}| = |\widetilde{B}| = k \\
f &= \begin{cases} \mathrm{env}_V(m) & \text{if } m = \texttt{id} \\ m & \text{otherwise} \end{cases} \\
(\tilde{x}, S) &= \mathrm{env}_T(Y)(f) \\
\mathrm{env}'_V &= \texttt{this} : \mathrm{env}_V(\texttt{this}), \texttt{sender} : \mathrm{env}_V(\texttt{sender}), \texttt{value} : \mathrm{env}_V(\texttt{value}), \\
&\quad\ x_1 : v_1, \ldots, x_k : v_k \\
\Delta(\texttt{this}) &= \mathrm{var}(I^X, s_1) \\
\mathrm{proc}(\widetilde{B}_{\tilde{s}}) : s' &= \Gamma(I^Y)(f) \\
\Delta' &= \texttt{this} : \Delta(\texttt{this}), \texttt{sender} : \Delta(\texttt{sender}), \texttt{value} : \Delta(\texttt{value}), \\
&\quad\ x_1 : \mathrm{var}(B_1, s'_1), \ldots, x_k : \mathrm{var}(B_k, s'_k)
\end{aligned}
$$

Figure 8.25: Typed semantics of statements: method calls.

$$[\textsc{r-fcall}]\ \frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vDash_{(I^Y,s')} \langle e_1, \mathsf{env}_{SV}\rangle \to Y \\ \Sigma;\Gamma;\Delta \vDash_{(\mathsf{int},s')} \langle e_2, \mathsf{env}_{SV}\rangle \to n \\ \Sigma;\Gamma;\Delta \vDash_{(\widetilde{B},\widetilde{s}')} \langle \tilde{e}, \mathsf{env}_{SV}\rangle \to \tilde{v}\end{array}}{\begin{array}{r}\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{call}\ \ e_1.m(\tilde{e})\,\$e_2; Q, \mathsf{env}_{TSV}\rangle \\ \to \Sigma;\Gamma;\Delta \vDash_{s'} \left\langle S; (\mathsf{env}_V, \Delta); s; Q, \mathsf{env}_T; \mathsf{env}'_{SV}\right\rangle\end{array}} \left(\begin{array}{c} s_1 \sqsubseteq s' \\ s \sqsubseteq s' \\ n \neq 0 \Rightarrow s' \sqsubseteq s_3, s_4 \end{array}\right)$$

where:

$$\begin{aligned}
X &= \mathsf{env}_V(\texttt{this}) \\
\mathsf{env}_F^X &= \mathsf{env}_S(X) \\
\mathsf{env}_F^Y &= \mathsf{env}_S(Y) \\
f &= \begin{cases}\mathsf{env}_V(m) & \text{if } m = \texttt{id} \\ m & \text{otherwise}\end{cases} \\
f &\notin \mathrm{dom}\,(\mathsf{env}_T(Y)) \\
(\epsilon, S) &= \mathsf{env}_T(Y)(\texttt{fallback}) \\
\mathsf{env}'_V &= \texttt{this} : Y, \texttt{sender} : X, \texttt{value} : n, \texttt{id} : f, \texttt{args} : \tilde{v} \\
\mathsf{env}'_S &= \mathsf{env}_S\big[X \mapsto \mathsf{env}_F^X[\texttt{balance -= } n]\big]\big[Y \mapsto \mathsf{env}_F^Y[\texttt{balance += } n]\big] \\
\mathsf{var}(I^X, s_1) &= \Delta(\texttt{this}) \\
(I^Y, s_2) &= \Gamma(Y) \\
\mathsf{var}(\mathsf{int}, s_3) &= \Gamma(I^X)(\texttt{balance}) \\
\mathsf{var}(\mathsf{int}, s_4) &= \Gamma(I^Y)(\texttt{balance}) \\
\mathsf{proc}() : s' &= \Gamma(I^Y)(\texttt{fallback}) \\
\Delta' &= \texttt{this} : \mathsf{var}(I^Y, s_2), \texttt{sender} : \mathsf{var}(I^X, s_1), \texttt{value} : \mathsf{var}(\mathsf{int}, s'), \\
&\quad\ \texttt{id} : \mathsf{var}(\mathsf{idf}, s'), \texttt{args} : \mathsf{var}(\widetilde{B}, \widetilde{s}')
\end{aligned}$$

$$[\textsc{r-delv}]\ \frac{}{\begin{array}{r}\Sigma;\Gamma;\Delta, x : \mathsf{var}(B_{s'}) \vDash_s \langle \mathsf{del}(x); Q, \mathsf{env}_{TS}; \mathsf{env}_V, x : v\rangle \\ \to \Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_{TSV}\rangle\end{array}}$$

$$[\textsc{r-return}]\ \frac{\Sigma;\Gamma;\Delta' \vdash \mathsf{env}'_V}{\begin{array}{r}\Sigma;\Gamma;\Delta \vDash_s \left\langle (\mathsf{env}'_V, \Delta'); Q, \mathsf{env}_{TSV}\right\rangle \\ \to \Sigma;\Gamma;\Delta' \vDash_s \left\langle Q, \mathsf{env}_{TS}; \mathsf{env}'_V\right\rangle\end{array}}$$

$$[\textsc{r-restore}]\ \frac{}{\Sigma;\Gamma;\Delta \vDash_s \langle s'; Q, \mathsf{env}_{TSV}\rangle \to \Sigma;\Gamma;\Delta \vDash_{s'} \langle Q, \mathsf{env}_{TSV}\rangle}\ (s \sqsupseteq s')$$

Figure 8.26: Typed semantics of statements: the fallback function and 'bookkeeping' rules.

s arbitrarily low in this case. However, given the way we have formulated the rules of the typed semantics, a security level can only occur on the stack following a statement that must be executed at a *higher* or equal level than the initial level, so this situation cannot occur, if the stack represents, or derives from the execution of, an actual program. In effect, this means that security-levels on a *well-formed* stack must occur in *descending* order, and the coercion property for stacks Q therefore becomes that we can choose s as low as the *first* (highest) security level occurring in Q. Hence, we can choose s arbitrarily low, if Q does not contain any security levels.

- In [R-IF], we of course only run the selected branch, whilst discarding the other. In terms of type safety, this means that only the selected branch needs to be type safe, since the other branch is not executed. However, in the corresponding type rule, [T-IF], we must type-check both branches, and the type system could therefore potentially reject a program, if one of the branches were ill-typed, even if that branch would not be executed at runtime. The rule [R-WHILE$_\text{F}$] presents a similar case, since the body of the loop is not checked, because it is not run, whereas in the corresponding type rule, [T-WHILE], it is. These are examples of the *slack* of the syntactic type system.

- In the call rules, [R-CALL] and [R-FCALL], we introduce a slight variation of the side condition, compared to the type rule [T-CALL]; namely that $s' \sqsubseteq s_3, s_4$ *only* needs to hold if $n \neq 0$, where n is the value of the expression e_2, i.e. the value to be transferred. This relaxation is possible, because if $n = 0$, then no value is transferred, and the balance fields of the caller and callee are therefore not modified. However, in the syntactic type system, we cannot know what the value of this expression will be at runtime, so we have to always require that $s' \sqsubseteq s_3, s_4$, which again could lead to some programs being rejected, even though they would be safe to run. This is thus another example of the slack of the syntactic type system.

- In [R-CALL] and [R-FCALL], we perform an extra lookup in the environment, namely $\Gamma(Y) = (I^Y, s_2)$. This is necessary to obtain the actual security level s_2 of Y, rather than just the level s', which is required in the evaluation of the path e_1, and which therefore might be higher. We need the actual level, when we create the type binding for `sender` in the new type environment Δ', to prevent the level from increasing beyond necessity.

- The rule [R-FCALL] for fallback function calls is almost similar to [R-CALL], except for some minor differences in the construction of Δ': Firstly, we use the 'dummy type' `idf` for the variable `id`. We also assign the security level s' to this variable, which is the same as the security level obtained from the type of the fallback function. For the sake of simplicity, we assume that this identifier

is not passed as an argument to a method call; i.e. we assume `id` is only ever read from within the body of the fallback function, and that its contents are not passed out of this scope.

Secondly, the `args` magic variable is really an *array* of values, which we assume is expanded at runtime. Therefore, we also assume that its contents are never assigned into another variable, or otherwise modified or read; i.e. `args` is not allowed to appear as the LHS of an assignment, nor in an ordinary expression. It can only be used as the argument to another method call. Therefore, we also do not give it a single security level, or a base type, but just retain the individual types of the elements, written $\mathsf{var}(\widetilde{B}_{\tilde{s}})$.

8.6.1 Properties of typed transitions

As was the case with the typed semantics of expressions, it should again be clear from a comparison of Figures 8.9–8.10, Figures 8.16–8.17, and Figures 8.24–8.26, that the rules of the typed semantics of stacks are the same as the rules for the untyped semantics, but with added restrictions from the type rules for statements and stacks. Every transition that can be concluded by the typed semantics can therefore also be concluded by the untyped semantics, with *one* exception; namely transitions concluded by using [R-RESTORE], which restores a security level s that had previously been placed on the stack. There is nothing similar in the untyped semantics, as we altered the syntax of stacks to allow security levels to be stored there. Hence, to show a statement corresponding to Theorem 19 for stacks, we first need to remove all security levels s, and also merge the type information from Δ into variable environments env_V placed on the stack. We use the following functions, defined on the syntax of return symbols and stacks, respectively:

$$\begin{aligned}
\textsc{mrg}(\mathsf{env}_V^{\varnothing}; \varnothing) &= \mathsf{env}_V^{\varnothing} \\
\textsc{mrg}(x : v, \mathsf{env}_V; x : B_s, \Delta) &= (x, v, B_s), \textsc{mrg}(\mathsf{env}_V, \Delta)
\end{aligned}$$

$$\begin{aligned}
\textsc{rem}(\bot) &= \bot \\
\textsc{rem}(S; Q) &= S; \textsc{rem}(Q) \\
\textsc{rem}(\mathsf{del}(x); Q) &= \mathsf{del}(x); \textsc{rem}(Q) \\
\textsc{rem}((\mathsf{env}_V, \Delta); Q) &= \textsc{mrg}(\mathsf{env}_V, \Delta); \textsc{rem}(Q) \\
\textsc{rem}(s; Q) &= \textsc{rem}(Q)
\end{aligned}$$

Furthermore, we need to impose some restrictions on how security levels can occur in stacks in the typed semantics. As mentioned above, only method calls and guarded statements (`if` and `while`) can place security levels on the stack, when they need to raise the current security levels. Hence, if a stack Q represents a partially evaluated

actual program, then all security levels $s_1, \ldots, s_n$ in Q must occur in *descending* order, i.e.

$$s_1 \sqsupseteq \ldots \sqsupseteq s_n$$

to satisfy the side condition of [R-RESTORE]. We shall later formalise this notion of well-formedness of Q, but for now it will suffice to assume, that Q satisfies this requirement.

Also, the stack must be executed at a level s that is *higher than*, or equal to, the first security level s_1 occurring in Q, i.e. $s \sqsupseteq s_1$. We write $\textsc{fst}(Q)$ for the first security level occurring in Q, which by the aforementioned requirement also will be the highest. The function is straightforwardly defined as follows:

$$\begin{aligned}
\textsc{fst}(\bot) &= s_\bot \\
\textsc{fst}(S; Q) &= \textsc{fst}(Q) \\
\textsc{fst}(\mathsf{del}(x); Q) &= \textsc{fst}(Q) \\
\textsc{fst}((\mathsf{env}_V, \Delta); Q) &= \textsc{fst}(Q) \\
\textsc{fst}(s; Q) &= s
\end{aligned}$$

Notice that the function returns $s_\bot$ for the bottom of the stack. This corresponds to no restriction on the level at which the stack can be executed, so in that case we can choose the execution level arbitrarily low. With this, we can then formally state that the typed semantics is compatible with the untyped semantics:

Theorem 24 (Semantic compatibility for stacks)**.** *Assume Q satisfies that all security levels in Q occur in descending order. For all type environments Σ, Γ, Δ, and security level $s \sqsupseteq \textsc{fst}(Q)$, and environments env_{TSV}, it holds that either*

$$\begin{aligned}
&\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'} \left\langle Q', \mathsf{env}_T; \mathsf{env}'_{SV} \right\rangle \\
\implies &\langle \textsc{rem}(Q), \mathsf{env}_{TS}; \textsc{mrg}(\mathsf{env}_V; \Delta) \rangle \to \left\langle \textsc{rem}(Q), \mathsf{env}_T; \mathsf{env}'_S; \textsc{mrg}(\mathsf{env}'_V; \Delta') \right\rangle
\end{aligned} \tag{8.1}$$

or otherwise there exists a sequence of transitions

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to \ldots \to \Sigma; \Gamma; \Delta \vDash_{s'} \langle Q', \mathsf{env}_{TSV} \rangle$$

of finite length, to a state for which the implication in (8.1) holds.

Proof sketch. Firstly, suppose $Q = s'; Q'$. Then $\textsc{fst}(Q) = s'$ and $s \sqsupseteq s'$ by assumption. Thus

$$\Sigma; \Gamma; \Delta \vDash_s \langle s'; Q', \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta \vDash_{s'} \langle Q', \mathsf{env}_{TSV} \rangle$$

can then be concluded by [R-RESTORE]. It might now again be the case that $Q' = s''; Q''$, but then $s' \sqsupseteq s''$ by the assumption of ordering of the security levels. Hence, the transition can again be concluded by [R-RESTORE], and so on. As we know the

stack is of finite length, it will eventually be the case that the symbol at the top of the stack is *not* a security level, in which case the theorem holds if the first part holds.

When $Q \neq s'; Q'$, we proceed by case analysis of the rules used to conclude the transition

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'} \langle Q', \mathsf{env}_T; \mathsf{env}'_{SV} \rangle$$

In each case, we obtain the required premises and side conditions from the rule of the typed semantics and use it to conclude the same transition in the untyped semantics by the corresponding untyped rule. This is straightforward, since each typed semantic rule (with the exception of [R-RESTORE]) has an untyped variant, whose premises and side conditions are a subset of the premises and side conditions of the typed variant. □

Assuming still that all security levels in Q occur in descending order, we can also show that the coercion property for stacks holds:

Theorem 25 (Coercion property for stacks)**.** *Assume Q satisfies that all security levels in Q occur in descending order. For all type environments Σ, Γ, Δ, and security levels $s_1 \sqsupseteq s_2 \sqsupseteq \textsc{fst}(Q)$, and environments env_{TSV}, it holds that*

$$\begin{aligned} &\forall s'_1 \,.\, \Sigma; \Gamma; \Delta \vDash_{s_1} \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'_1} \langle Q', \mathsf{env}_T; \mathsf{env}'_{SV} \rangle \\ \implies &\exists s'_2 \,.\, \Sigma; \Gamma; \Delta \vDash_{s_2} \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'_2} \langle Q', \mathsf{env}_T; \mathsf{env}'_{SV} \rangle \end{aligned}$$

Proof. By case analysis of the rules used to conclude the transition.

- For rules [R-SKIP], [R-WHILE$_\mathrm{F}$], [R-DECV], [R-DELV], and [R-RETURN], the conclusion follows directly from the fact that these rules do not impose any restrictions on the security level.
- For rules [R-IF], [R-WHILE$_\mathrm{T}$], [R-ASSV], [R-ASSF], [R-CALL], [R-DCALL], and [R-FCALL], we have a side condition of the form $s_1 \sqsubseteq s'$ restricting the security level, where s' is derived from the premise of the rule. As we know that $s_2 \sqsubseteq s_1$, we therefore also have that $s_2 \sqsubseteq s'$ by transitivity of $\sqsubseteq$, which, for each rule, allows us to conclude the transition with s_2 as well.
- Finally, if [R-RESTORE] was used to conclude the transition, we have in the side condition that $s_1 \sqsupseteq s'$, where s' is the top-most element on the stack Q. Hence, $\textsc{fst}(Q) = s'$, and by assumption $s_2 \sqsupseteq \textsc{fst}(Q)$ so $s_2 \sqsupseteq s'$. Thus we can satisfy the side condition with s_2 as well and conclude the transition with [R-RESTORE]. □

Note that Theorem 25 does not directly relate the two resulting security levels s'_1 and s'_2. Indeed, several of the rules allow a higher security level to be chosen (i.e.

those mentioned in the second case of the proof), so, depending on the structure of the security lattice, different choices could be made in the two transitions, which could end up either making $s_1' \sqsupseteq s_2'$ or $s_2' \sqsupseteq s_1'$. However, if we assume that the same choice is made in both executions (e.g. the highest, or lowest, possible security level for which the transition can be concluded), then we would also have that $s_1' \sqsupseteq s_2'$.

We can also show a result similar to Theorem 22 for expressions, which allows us to abstract away from the specific values stored in the state environment env_{SV}. This ensures that *if* a stack Q has a typed transition for some appropriately shaped env_{SV}^1, then any other similarly shaped env_{SV}^2 will also yield a transition:

Theorem 26 (Safe state abstraction)**.** *For all environments* Σ, Γ, Δ, env_T, *and security level* s, *and* $\mathsf{env}_{SV}^1, \mathsf{env}_{SV}^2$ *built according to Definition 59, it holds that*

$$\begin{aligned} &\forall s_1' \,.\, \Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}_{SV}^1\rangle \to \Sigma;\Gamma;\Delta_1' \vDash_{s_1'} \langle Q', \mathsf{env}_T; \mathsf{env}_{SV}^{1'}\rangle \\ \Longrightarrow\ &\exists s_2' \,.\, \Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}_{SV}^2\rangle \to \Sigma;\Gamma;\Delta_2' \vDash_{s_2'} \langle Q', \mathsf{env}_T; \mathsf{env}_{SV}^{2'}\rangle\,. \end{aligned}$$

Proof sketch. By case analysis of the rules of the typed semantics (Figures 8.24–8.26). The environments env_{SV} only affect the execution of a statement through variables and field names appearing in expressions, and for these cases we apply Theorem 22 for the evaluation of all expressions occurring inside statements in Q. □

Finally, we can show a result concerning the type environment Δ and the security level s. Both may have changed after a transition, in case the security level was raised, a local variable was declared, or a method call was issued. However, any such change is temporary in the sense that all of them will have resulted in a special symbol having been pushed onto the stack, which undoes the change if it is reached. Thus, *if* a stack Q terminates normally, by reaching the bottom symbol $\bot$, then all such temporary changes will have been discarded. Assuming Q is to be executed relative to some Δ and s, we can detect the type environment and security level (Δ', s') on which Q will terminate, *if* it terminates, as follows:

Definition 62 (Terminal Δ and s)**.** We define the function $\textsc{fin}(\cdot)$ on triplets (Q, Δ, s) of stacks Q, type environments Δ and security levels s as follows:

$$\begin{aligned} \textsc{fin}(\bot, \Delta, s) &= (\Delta, s) \\ \textsc{fin}(S; Q, \Delta, s) &= \textsc{fin}(Q, \Delta, s) \\ \textsc{fin}(\mathsf{del}(x); Q, (\Delta, x : \mathsf{var}(B_{s'})), s) &= \textsc{fin}(Q, \Delta, s) \\ \textsc{fin}((\mathsf{env}_V, \Delta'); Q, \Delta, s) &= \textsc{fin}(Q, \Delta', s) \\ \textsc{fin}(s'; Q, \Delta, s) &= \textsc{fin}(Q, \Delta, s') \end{aligned}$$ ■

Obviously, we cannot know whether Q *will* terminate, or whether it will terminate abnormally with `throw`, but *if* in fact it does terminate normally, then given any

arbitrary Δ and s with which Q might be executed, we can use $\textsc{fin}(Q, \Delta, s) = (\Delta', s')$ to detect the terminal type environment and security level.

Theorem 27 (Terminal type level)**.** *If*

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to^* \Sigma; \Gamma; \Delta' \vDash_{s'} \langle \bot, \mathsf{env}_T, \mathsf{env}'_{SV} \rangle$$

then $\textsc{fin}(Q, \Delta, s) = (\Delta', s')$.

The proof proceeds by first showing that the pair (Δ', s') obtained from $\textsc{fin}(Q, \Delta, s)$ is unaltered after a single transition step. This is shown by case analysis of the semantic rules used to conclude the transition. This result can then be generalised to the case of $\to^*$ by induction on the length of the transition sequence. The proof is given in Section 8.9.5.

We can now ascertain that the typed semantics indeed ensures the intended notion of type safety. There are several aspects to this: Firstly, we have the standard notion of preservation of the data types B for fields and variables; i.e. that if $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$ and

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'} \langle Q', \mathsf{env}_T, \mathsf{env}'_{SV} \rangle$$

then $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_{SV}$. This relates directly to the data aspect of our types.

Secondly, we have a notion that is particular to the security level aspect of our types; namely that if

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'} \langle Q', \mathsf{env}_T, \mathsf{env}'_{SV} \rangle$$

then only values in containers of level s or *higher* can have been modified in the transition. In other words, env_S and env'_S will agree on all entries that are strictly *lower* than, or unrelated to, s.[11]

Thirdly, we shall need to impose a further well-formedness criterion on stacks Q. We wish to show that well-typedness and consistency of the state env_{SV} are preserved by the typed transitions, but since a transition can restore a variable environment from the stack Q, then we must naturally require that all return symbols (env_V, Δ) occurring in Q also must be well-typed and consistent. We combine this with the requirement on the ordering of security levels in Q by the rules in Figure 8.27, which we use to judge *well-formedness* of stacks w.r.t. type environments $\Sigma; \Gamma$, a state env_{SV}, and a security level s. As usual, some comments on the rules are in order:

- In [WF-STM] we require that any statement S found on the stack must not contain names not found in the domains of env_{SV}. Together with consistency of env_{SV}, this ensures that transitions cannot be stuck due to a free name, or a missing contract implementation.

[11] Note however that we cannot show the same result for env_V, since a transition may create a new variable environment (in the case of method calls) or restore a variable environment from the stack (in the case of method returns).

$$[\textsc{wf-bot}]\frac{}{\Sigma;\Gamma;\mathsf{env}_{SV};s\vdash\bot}$$

$$[\textsc{wf-stm}]\frac{\Sigma;\Gamma;\mathsf{env}_{SV};s\vdash Q}{\Sigma;\Gamma;\mathsf{env}_{SV};s\vdash S;Q}\left(\begin{array}{c}\mathcal{F}_A(S)\subseteq\mathrm{dom}\,(\mathsf{env}_S)\\ \mathcal{F}_V(S)\subseteq\mathrm{dom}\,(\mathsf{env}_V)\end{array}\right)$$

$$[\textsc{wf-del}]\frac{\Sigma;\Gamma;\mathsf{env}_{SV};s\vdash Q}{\Sigma;\Gamma;\mathsf{env}_{SV},x\,:\,v;s\vdash\mathsf{del}(x);Q}$$

$$[\textsc{wf-ret}]\frac{\Sigma;\Gamma;\Delta\vdash\mathsf{env}'_V\quad\mathsf{env}_S\vdash\mathsf{env}'_V\quad\Sigma;\Gamma;\mathsf{env}_S;\mathsf{env}'_V;s\vdash Q}{\Sigma;\Gamma;\mathsf{env}_S;\mathsf{env}_V;s\vdash(\mathsf{env}'_V,\Delta);Q}$$

$$[\textsc{wf-sec}]\frac{\Sigma;\Gamma;\mathsf{env}_{SV};s'\vdash Q}{\Sigma;\Gamma;\mathsf{env}_{SV};s\vdash s';Q}\ (s\sqsupseteq s')$$

Figure 8.27: Well-formedness of stacks Q.

- In [WF-DEL], we remove x from env_V when we encounter an end-of-scope symbol $\mathsf{del}(x)$, since the domain of env_V is used in the rule [WF-STM].
- The rule [WF-RET] handles the of a return symbol (env'_V,Δ) occurring on the stack. As explained above, we must here require that env_V is both well-typed and consistent, and the remainder of the stack must then be well-formed relative to this new env'_V.
- Finally, rule [WF-SEC] ensures that all security levels s' occurring in the stack are in descending order, as in our previous requirement. Thus, this property too is ensured by well-formedness.

Well-formedness ensures that Q and env_{SV} can meaningfully be combined in a configuration $\langle Q,\mathsf{env}_{TSV}\rangle$ and executed by the typed semantics at level s, provided that the initial state env_{SV} is well-typed, and env_{TSV} are consistent, and that s is greater than, or equal to, the first security level occurring in Q. Note that it is necessary to assume consistency of env_T as well to ensure that well-formedness of Q is preserved, since a transition can place code from env_T on the stack.

All of the aforementioned properties are preserved by the typed transitions. We show this in the following theorem:

Theorem 28 (Runtime preservation)**.** *Assume that the following holds:*

- *Well-typedness:* $\Sigma;\Gamma;\Delta\vdash\mathsf{env}_{SV}$*, and*
- *Consistency:* $\Gamma\vdash\mathsf{env}_{TSV}$*, and*

- *Well-formedness:* $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$, *and*

If $\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'} \langle Q', \mathsf{env}_T; \mathsf{env}'_{SV} \rangle$ *then the following holds:*

1. *Well-typedness:* $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_{SV}$,
2. *Consistency:* $\Gamma \vdash \mathsf{env}'_{SV}$,
3. *Well-formedness:* $\Sigma; \Gamma; \mathsf{env}'_{SV}; s' \vdash Q'$, *and*
4. *s-equivalence:* $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}'_S$, *and*

The proof is by case analysis of the rules used to conclude the transition (Figures 8.24–8.26). It can be found in Section 8.9.7.

Theorem 28 assures us that the properties of well-typedness, consistency, and well-formedness are preserved by the typed semantics; and, additionally, that any changes to env_S *must* be to a field that is of level s or higher. The latter is in accordance with the meaning of the static type judgment $\mathsf{cmd}(s)$, corresponding to the parameterisation of the turnstile $\vDash_s$ in the typed semantics. All variables of a level *strictly below*, or incomparable to, s will be unaffected by the transition steps, so the two environments, env_S and env'_S, agree on all such entries. This is the expected safety property induced by the types, since executing the stack at level s means that no variables of a level *lower* than, or incomparable to, s may be modified (cf. rules [R-ASSF], [R-CALL] and [R-FCALL]), and information is not permitted to flow from a higher level to a lower level.

Finally, as we did with expressions (Corollary 14), we can now also use Theorem 28 to show that the typed semantics ensures *non-interference* for stack executions. This result states that if we execute a stack Q at level s with two different, but s-equivalent, memories, env^1_{SV} and env^2_{SV}, then the resulting two memories, $\mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^{2'}_{SV}$, will still be s-equivalent:

Theorem 29 (Runtime non-interference for stacks)**.** *Assume that the following holds:*

- *Well-typedness:* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}^1_{SV}$ *and* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}^2_{SV}$, *and*
- *Consistency:* $\Gamma \vdash \mathsf{env}_T; \mathsf{env}^1_{SV}$ *and* $\Gamma \vdash \mathsf{env}_T; \mathsf{env}^2_{SV}$, *and*
- *Well-formedness:* $\Sigma; \Gamma; \mathsf{env}^1_{SV}, s \vdash Q$ *and* $\Sigma; \Gamma; \mathsf{env}^2_{SV}, s \vdash Q$, *and*
- *s-equivalence:* $\Gamma; \Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$.

If

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}^1_{SV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'} \langle Q_1, \mathsf{env}_T; \mathsf{env}^{1'}_{SV} \rangle$$
$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}^2_{SV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_{s'} \langle Q_2, \mathsf{env}_T; \mathsf{env}^{2'}_{SV} \rangle$$

then $\Gamma; \Delta' \vdash \mathsf{env}^{1'}_{SV} =_s \mathsf{env}^{2'}_{SV}$.

The proof uses Theorem 28 to argue that no entry (field or variable) of a *lower* level than s could have been modified in the two execution steps. For fields of *exactly* level s, the transitions may have modified them in an assignment, but Corollary 14 then ensures that the two evaluations of the expression on the right-hand side of the assignments must yield the *same* value; hence, the fields are updated in the same way. Finally, the proof goes through a case analysis on the rules of the typed semantics to handle updates of variables of level s, again using Corollary 14 to show that any expression evaluated at level s would yield the same value in both executions. The proof can be found in Section 8.9.8.

It is worth noting that the dual executions of the configurations $\langle Q, \mathsf{env}_T; \mathsf{env}^1_{SV} \rangle$ and $\langle Q, \mathsf{env}_T; \mathsf{env}^2_{SV} \rangle$ actually allow for two different methods to be called. Suppose for example that Q is of the form $\texttt{call}\ \ x\,.\,f\,(\vec{e})\,\$e_2$, and x is a variable of type $\mathsf{var}(I, s')$ for some level s' strictly higher than s. This would allow it to contain two different addresses, e.g. X_1 and X_2 in the two executions, which would lead to two different methods being called, albeit with the same signature. Well-typedness and consistency of the two environments ensure that either both methods must exist; or neither of the methods exist, because both X_1 and X_2 must be addresses of contracts that implement all the methods of the interface I. Thus, the two resulting variable environments, $\mathsf{env}^{1'}_V$ and $\mathsf{env}^{2'}_V$, will therefore still contain exactly the same variable entries, and with values agreeing at level s or lower.

On the other hand, calling two different methods would also mean that the two different `balance` fields of the two callees would be different in $\mathsf{env}^{1'}_S$ resp. $\mathsf{env}^{2'}_S$; e.g. in $\mathsf{env}^{1'}_S$ it might be X_1.`balance` that was modified, but X_2.`balance` in $\mathsf{env}^{2'}_S$. This, however, does not lead to a contradiction of Theorem 29 because of the side condition of the rules [R-CALL] and [R-FCALL], which are the only two rules that could have been used to conclude such a transition (rule [R-DCALL] cannot have been used, since it does not modify env_S). This side condition states that $s' \sqsubseteq s_3, s_4$, where s_3 and s_4 are the levels of the `balance` fields of the caller and callee, respectively. As we assumed, for the sake of this argument, that s' is *strictly greater* than s, then so are both s_1 and s_2, which means that this difference will not affect the s-equality of the two environments.

This example also illustrates that the non-interference property cannot be straightforwardly generalised to arbitrarily long transitions sequences, i.e. the reflexive and transitive closure of typed transitions, since the two stacks Q_1 and Q_2 actually may differ after a single transition step. Along with method calls, the two other guarded constructs, $\texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F}$ and $\texttt{while}\ e\ \texttt{do}\ S$, can similarly induce a difference in the stacks, if e.g. the S_T branch of an `if-else` construct is selected in the execution of stack Q_1, whilst the S_F branch is selected in the execution of stack Q_2. However, in all these cases, the execution of the guarded statement (S_b in `if-else`, resp. S in `while`) now happens in an s' context, which by assumption is strictly above s, since the expression guard e otherwise could not have yielded two different values

in the two executions by Corollary 14. The typed semantics will therefore still ensure that no field of a level lower than s is modified, and fields of level s are modified in the same way, but the transition sequences will not necessarily match step by step. This can temporarily induce differences between the two variable environments, because the bodies of the two method calls could declare different local variables. It could even be the case that the body of one method call might contain an infinite loop, whilst the other returns. Thus, generalising Theorem 29 to the case of $\rightarrow^*$ would require some way of expressing that *if* both method calls return, *then* the stacks are again equal from that point on, and s-equality of the two states will then again hold.[12] However, we leave it as future work to state and show such a result.

The example above also illustrates a curious detail of our types: If the methods of a contract are to be executable *at all* by an external caller in the typed semantics, then the level of its `balance` field *must* be higher than, or equal to, the level of the contract address. If it were not, then the fact of the call itself would create a downward information flow (from the address to the `balance`), which is disallowed by the types. However, as mentioned above, this does not apply to *delegate calls*, since env_S is not modified in the [R-DCALL] rule. Thus, one way of ensuring that all methods in a contract can only be invoked as delegate calls would be to let the `balance` field have a strictly lower level than the address of the contract.

8.6.2 Typing interpretations for stacks

In the preceding section, we have shown a number of results regarding typed transitions. Most importantly, we have Theorem 28, which shows that the key property denoted by $\vDash_s$, corresponding to the static type judgment $\mathsf{cmd}(s)$, indeed is ensured by the typed semantics; namely that only containers of level s or higher can be modified by such a transition. However, this result required some assumptions on the type environments $\Sigma; \Gamma; \Delta$, the state and code environments env_{TSV}, and the stack Q; specifically, consistency of env_{TSV}, well-typedness of env_{SV}, and well-formedness of Q. These properties are also preserved by the typed transitions, and, together, they ensure that a configuration $\langle Q, \mathsf{env}_{TSV}\rangle$ does not get stuck due to a malformed stack or an attempt to access a contract member on an address that does not contain an implementation etc. When we rule out such 'trivial' type errors, what remains is that a stuck configuration can only derive from three situations:

- Q is of the form $\bot$, meaning we have reached the bottom of the stack, so the execution terminates normally.
- Q is of the form $\texttt{throw}; Q'$, meaning an exception was thrown, so the execution terminates abnormally.

[12] The issue was not present in Chapter 6, because we used a big-step semantics, which ensures that method calls must return, and loops must terminate, if a transition is to be concluded. It only becomes apparent here, because we are using a small-step semantics.

- The corresponding untyped transition would cause a violation of *type safety* by allowing an insecure data flow.

Of these, the first two forms are of course still safe to execute, since they have no transitions. They can obviously also easily be detected by just examining the structure of Q. Thus, apart from the first two cases, the key point to note here is that *a stuck configuration indicates a runtime type error.* We shall use this to formulate a notion of *typing interpretations* for stacks Q, such that Q either must be of one of the two first forms, or else it *must* have a typed transition (with some appropriately shaped env_{SV}), such that the reduct again is contained in the typing interpretation.

However, unlike in the corresponding formulation for expressions (Definition 60), it is important to note that the type constraints represented by Δ and the current security level s can *change* after a transition step; i.e. typed transitions are of the form

$$\Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_{TSV}\rangle \to \Sigma;\Gamma;\Delta' \vDash_{s'} \left\langle Q', \mathsf{env}_T; \mathsf{env}'_{SV}\right\rangle$$

where Δ' may have been altered, and s' may have been increased or decreased, whereas $\Sigma;\Gamma$ are fixed for each program. Hence, Δ and s must also be included in the typing interpretation, together with Q. Our notion of typing interpretations will therefore not be for stacks Q alone, but for *stack type triplets* (Q, Δ, s), where Δ and s represent the currently active variable type environment and security level under which Q is executing. To make this clearer, we shall first define a new transition system directly on the triplets themselves:

Definition 63 (Type lifted transition)**.** Assume Σ, Γ and env_T are fixed. A *type lifted transition system* is a tuple $(\mathcal{P}, \Mapsto)$, where

$$P \in \mathcal{P} ::= \mathcal{Q} \times \mathcal{E} \times \mathcal{S}$$

is the set of stack type triplets (Q, Δ, s), and the transition relation is defined such that

$$\Sigma;\Gamma;\mathsf{env}_T \vDash (Q, \Delta, s) \Mapsto (Q', \Delta', s')$$

if there exist an env_{SV} built from $\Sigma;\Gamma;\Delta$ according to Definition 59, and an env'_{SV} such that

$$\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash_s \langle Q, \mathsf{env}_{TSV}\rangle \to \Sigma;\Gamma;\Delta';\mathsf{env}_T \vDash_{s'} \left\langle Q', \mathsf{env}_T; \mathsf{env}'_{SV}\right\rangle. \qquad \blacksquare$$

By Theorem 26, we know that if there exists an env^1_{SV} such that the configuration $\Sigma;\Gamma;\Delta \vDash_s \left\langle Q, \mathsf{env}_T; \mathsf{env}^1_{SV}\right\rangle$ has a transition, then a transition will also exist for all similarly shaped env^2_{SV}. All such env_{SV} are ensured to be well-typed, consistent and compatible with (Q, Δ, s) by construction, and one way to view this quantification over all such environments is to think of it as a quantification over all possible, appropriately shaped and typed *inputs* to the program on the stack. This is quite

natural, since we obviously want a notion of safety that does not depend on the *actual* values being passed to the program, but only on the *types* of those values. It is in this sense that the transition system is *type lifted:* the shape of the transition system is not determined by actual values, but by the *types* of those values.

Different inputs may of course yield different reducts, e.g. in case the stack Q had an `if-else` statement at the top, and the environments yield different values for the guard expression e. Hence, there may also be more than one possible reduct for a given triplet P in the type lifted transition system. This is important to keep in mind, because our definition of typing interpretations for stack type triplets P must be such that we ensure both *that* a reduct for P exists, and that *all* reducts of P are again contained in the typing interpretation. We define it as follows:

Definition 64 (Typing interpretation for stacks)**.** A *typing interpretation* for stack type triplets w.r.t. environments Σ, Γ and env_T, written $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, is a set of stack type triplets $P = (Q, \Delta, s)$ satisfying that $(Q, \Delta, s) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ implies

1. $Q = \bot$ or $Q = \texttt{throw}; Q'$, or
2. $\exists P_1'$ such that $\Sigma; \Gamma; \mathsf{env}_T \vDash (Q, \Delta, s) \Mapsto P_1'$, and
   $\forall P_2'$ such that $\Sigma; \Gamma; \mathsf{env}_T \vDash (Q, \Delta, s) \Mapsto P_2'$ it holds that $P_2' \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$.

The *typing* of stack type triplets P w.r.t. a given Σ, Γ, and env_T, written $\nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, is the union of all typing interpretations $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$:

$$\nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T} \triangleq \bigcup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$$

We write $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s)$, if there exists a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing (Q, Δ, s). Thus:

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s) \triangleq (Q, \Delta, s) \in \nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$$

Likewise, we write $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S : \mathsf{cmd}(s)$, if there exists a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing $(S; \bot, \Delta, s)$. Thus:

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S : \mathsf{cmd}(s) \triangleq (S; \bot, \Delta, s) \in \nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T} \qquad \blacksquare$$

The existence of a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing (Q, Δ, s) expresses that Q is safe to execute at level s or lower (down to $\textsc{fst}(Q)$) for *all* steps, according to the type constraints in Σ, Γ and Δ. Note also that typing interpretations for stack type triplets are formulated relative to a *particular* choice of code environment env_T. This is necessary because of the presence of fallback functions, which can place (statically) untypable code on the stack. All we therefore can require is that such code must *be* runtime safe, if it is executed, which is what the existence of a typing interpretation expresses.

Like in the definition of typing interpretations for expressions, we define *typing* of stack type triplets as a greatest fixed point, although in this case, this set is clearly undecidable, since statements can diverge. We therefore use a coinductive definition to allow for the possibility of diverging executions. The definition has two cases:

- In Case 1, we either have an empty stack $\bot$, or a stack with the command `throw` as the top element. Neither of these stacks have any transitions according to the typed semantics, so they are both safe to execute for all steps (since there are none), regardless of the type environment Δ and security level s.
- In Case 2, we handle all other stacks Q. Here, the triplet (Q, Δ, s) *must* have at least one transition, and for all its transitions (including the one to P_1') we require that the reduct P_2' again must be contained in the typing interpretation. This slightly complicated formulation is necessary, because we both need to ensure that the triplet is not stuck, since that would correspond to a type error, and that all of its possible reducts are type-safe as well.

By using the type-lifted transition system in the definition, we abstract away from any particular choice of field- and variable environment env_{SV}. We only require that one can be built from Σ and Γ, and from the Δ in the stack type triplet. In the definition, we discard the environments env'_{SV} from the configuration in the reduct of the underlying typed transition, since we use the type-lifted transition system, but we require that the triplet $P_2' = (Q_2', \Delta_2', s_2')$ again must be in the typing interpretation. This works, because by Theorem 28, we know that the properties of well-typedness, consistency and well-formedness are preserved by the typed semantics, so this env'_{SV} will again be amongst those that can be built from Σ, Γ and Δ_2' in the next step.

Using Definition 64, we can now show that typing is closed under various operations, including, importantly, the syntactic constructors of the language, given certain assumptions. As an example of what is to come, we can now show that typing is closed under s-coercions:

Lemma 58 (Typing coercion for stacks)**.** *If*

- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s_1)$*, and*
- $s_1 \sqsupseteq s_2 \sqsupseteq \textsc{fst}(Q)$*,*

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s_2)$*.*

Proof. By coinduction. From $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s_1)$ we know there exists a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing (Q, Δ, s_1). We create a candidate typing interpretation $\mathcal{R}$ containing (Q, Δ, s_2) thus:

$$\mathcal{R} \triangleq \left\{ (Q', \Delta', s_2') \mid (Q', \Delta', s_1') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T} \wedge s_1' \sqsupseteq s_2' \sqsupseteq \textsc{fst}(Q') \right\}$$

and clearly, since $(Q, \Delta, s_1) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ and $s_1 \sqsupseteq s_2 \sqsupseteq \textsc{fst}(Q)$, we also have that $(Q, \Delta, s_2) \in \mathcal{R}$.

We must now show that $\mathcal{R}$ indeed is a typing interpretation according to Definition 64: Pick any $(Q', \Delta', s_2') \in \mathcal{R}$. There are now two cases:

1. If $Q' = \bot$ or $Q' = \texttt{throw}; Q''$, then it satisfies Case 1 of Definition 64.
2. Otherwise, we know that $(Q', \Delta', s_1') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, and for all appropriately shaped env_{SV} there exists at least one transition

$$\Sigma; \Gamma; \Delta' \vDash_{s_1'} \langle Q', \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta'' \vDash_{s_1''} \left\langle Q'', \mathsf{env}_T; \mathsf{env}'_{SV} \right\rangle$$

such that $(Q'', \Delta'', s_1'') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. By Theorem 25 we then have that the transition

$$\Sigma; \Gamma; \Delta' \vDash_{s_2'} \langle Q', \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta'' \vDash_{s_2''} \left\langle Q'', \mathsf{env}_T; \mathsf{env}'_{SV} \right\rangle$$

also can be concluded for any s_2' such that $s_1' \sqsupseteq s_2' \sqsupseteq \textsc{fst}(Q')$, and by construction of $\mathcal{R}$, we also have that $(Q'', \Delta'', s_2'') \in \mathcal{R}$.

Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. □

Lemma 59 (Typing coercion for statements)**.** *If*

- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S : \mathsf{cmd}(s_1)$, *and*
- $s_1 \sqsupseteq s_2$,

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S : \mathsf{cmd}(s_2)$.

Proof. By Definition 64,

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S : \mathsf{cmd}(s_1) = \Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S; \bot : \mathsf{cmd}(s_1)$$

and $\textsc{fst}(S; \bot) = s_\bot$, so $s_1 \sqsupseteq s_2 \sqsupseteq \textsc{fst}(S; \bot)$ holds for any s_2 such that $s_1 \sqsupseteq s_2$. The conclusion then follows from Lemma 58. □

Lemma 58 and Lemma 59 are instructive for two reasons: Firstly, they illustrate the coinductive proof technique, which we shall also use in the following: We construct a new candidate typing interpretation $\mathcal{R}$ from an existing one, and show that $\mathcal{R}$ is still a typing interpretation. Hence, it too is contained in the *typing* of $\Sigma; \Gamma; \mathsf{env}_T$, since $\nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ is the union of all typing interpretations.

Secondly, both lemmas have a particular *form*, with a number of premises involving typing, implying a single conclusion involving typing. Hence, they can alternatively be written in the form of *inference rules*, given in Figure 8.28. These rules resemble the contravariant coercion rule for statements in Chapter 6, but importantly, unlike the rules from Chapter 6, these rules have been *proved admissible* based on the definition of typing, rather than being defined a priori. This is one of the key differences between the syntactic and the semantic approach.

$$[\textsc{st-stack-sub}]\frac{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash Q : \mathsf{cmd}(s_1)}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash Q : \mathsf{cmd}(s_2)}\ (s_1 \sqsupseteq s_2 \sqsupseteq \textsc{fst}(Q))$$

$$[\textsc{st-stm-sub}]\frac{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s_1)}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s_2)}\ (s_1 \sqsupseteq s_2)$$

Figure 8.28: Semantic coercion rules for stacks and statement.

8.6.3 Semantic type rules for stacks

We shall now state and prove a collection of compatibility lemmas for each of the stack operations. The lemmas are syntax directed, following the syntax of stacks from Definition 61.

In some of the proofs in this, and the following, section, we shall need to concatenate two stacks, that is, take a stack Q_1, remove the bottom symbol $\bot$, and then prefix the remainder onto another stack Q_2. Thus, we write q for a finite sequence of stack symbols *except* $\bot$; i.e.

$$q ::= \epsilon \mid S;q \mid \mathsf{del}(x);q \mid (\mathsf{env}_V,\Delta);q \mid s;q$$

so we can alternatively write a stack Q as $q;\bot$. Hence, q may be regarded as a 'bottomless stack.'

Lemma 60 (Bottom)**.** $\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash \bot : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma;\Gamma;\Delta \vDash \bot : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet $(\bot,\Delta,s)$. For any choice of Δ and s, $\mathcal{R} \triangleq \{(\bot,\Delta,s)\}$ is a typing interpretation by Case 1 of Definition 64. □

Lemma 61 (Statement)**.** *If*

- $\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s)$, *and*
- $\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash Q : \mathsf{cmd}(s)$,

then $\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S;Q : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma;\Gamma;\Delta \vDash S;Q : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet $(S;Q,\Delta,s)$.

From $\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s)$ and $\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash Q : \mathsf{cmd}(s)$ we know there exist typing interpretations $\mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$ and $\mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T}$ such that

$$(S;\bot,\Delta,s) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$$
$$(Q,\Delta,s) \in \mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T}$$

We then construct the following candidate typing interpretation:

$$\mathcal{R} \triangleq \left\{ (q; Q, \Delta', s') \mid (q; \bot, \Delta', s') \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T} \wedge \textsc{fin}(q; \bot, \Delta', s') = (\Delta, s) \right\} \cup \mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T}$$

where q may be empty. As $(S; \bot, \Delta, s) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$ and $\textsc{fin}(S; \bot, \Delta, s) = (\Delta, s)$, we have by construction that $(S; Q, \Delta, s) \in \mathcal{R}$.

Now we must show that $\mathcal{R}$ indeed is a typing interpretation. Consider an arbitrary triplet $(Q_1, \Delta_1, s_1) \in \mathcal{R}$. If Q_1 is of the form $\texttt{throw}; Q_1'$ or $\bot$, then Case 1 of Definition 64 holds. Otherwise, there are three possibilities:

- If Q_1 is of the form $q_1; Q$ for non-empty q_1, then $(q_1; \bot, \Delta_1, s_1) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$ by construction of $\mathcal{R}$. Thus we know the transition

  $$\Sigma; \Gamma; \Delta_1 \vDash_{s_1} \langle q_1; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta_1' \vDash_{s_1'} \langle q_1'; \bot, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$$

  can be concluded by some rule, and $(q_1'; \bot, \Delta_1', s_1') \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$. Therefore, the transition

  $$\Sigma; \Gamma; \Delta_1 \vDash_{s_1} \langle q_1; Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta_1' \vDash_{s_1'} \langle q_1'; Q, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$$

  can be concluded by the same rule, and by construction of $\mathcal{R}$, we have that $(q_1'; Q, \Delta_1', s_1') \in \mathcal{R}$.

- A special case of the above is if q_1' in the reduct is empty. Again we have that $(q_1; \bot, \Delta_1, s_1) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$ by construction of $\mathcal{R}$, and we know the transition

  $$\Sigma; \Gamma; \Delta_1 \vDash_{s_1} \langle q_1; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta_1' \vDash_{s_1'} \langle \bot, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$$

  can be concluded by some rule, and $(\bot, \Delta_1', s_1') \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$, so the execution terminates normally.

  Now we know by construction of $\mathcal{R}$ that $\textsc{fin}(q_1; \bot, \Delta_1, s_1) = (\Delta, s)$, and since the execution terminates, we have by Theorem 27 that $\Delta_1' = \Delta$ and $s_1' = s$. By the same reasoning as above, the transition

  $$\Sigma; \Gamma; \Delta_1 \vDash_{s_1} \langle q_1; Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta \vDash_{s} \langle Q, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$$

  can therefore be concluded, and since $(Q, \Delta, s) \in \mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T}$, we also have that $(Q, \Delta, s) \in \mathcal{R}$ by construction.

- Otherwise, the triplet (Q_1, Δ_1, s_1) came from $\mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T}$. Hence we know that the transition

$$\Sigma; \Gamma; \Delta_1 \vDash_{s_1} \langle Q_1, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta_1' \vDash_{s_1'} \langle Q_1', \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$$

  can be concluded, and $(Q_1', \Delta_1', s_1') \in \mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T}$. Therefore we also have that $(Q_1', \Delta_1', s_1') \in \mathcal{R}$ by construction.

Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ for the triplet $(S; Q, \Delta, s)$, as required. □

Before proceeding it is perhaps worth adding a few further comments on the technique used in the proof of Lemma 61, since this is a crucial point, and we shall use the same technique in several other lemmas: In the construction of the candidate typing interpretation $\mathcal{R}$ we make use of two related facts:

- Transitions only ever modify *the top* of the stack; hence, we can remove the $\bot$ symbol from a stack $q; \bot$ and suffix another stack Q onto it.
- Theorem 27 tells us that *if* $q; \bot$ eventually terminates normally, when executed at level s and with type environment Δ, *then* we can use the function $\textsc{fin}(\cdot)$ to find the security level s' and type environment Δ in the terminal state.

By the definition of $\textsc{fin}(\cdot)$ (Definition 62), we have that $\textsc{fin}(S; \bot, \Delta, s) = (\Delta, s)$, so in the construction of $\mathcal{R}$ we make sure to only pick those triplets $(q; \bot, \Delta', s')$ from $\mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$ for which it holds that $\textsc{fin}(q; \bot, \Delta', s') = (\Delta, s)$. This ensures two things:

- Every stack $q; \bot$ that derives from the execution of S will be suffixed with Q and included in the candidate typing interpretation.
- Only those stacks $q; \bot$ that will terminate on (Δ, s), and hence are *compatible* with Q, will be suffixed with Q and included. This ensures that the newly created stacks $q; Q$ remain well-formed.

It is of course possible that some stacks $q; \bot$ will not terminate, in which case Theorem 27 does not apply. This is, however, not a problem, since the execution will then never reach the suffix Q in the compound stacks $q; Q$. The predicate $\textsc{fin}(q; \bot, \Delta', s') = (\Delta, s)$ still ensures that every compound stack $q; Q$ is well-formed, and by Theorem 28 we know that well-formedness is preserved by the typed semantics, so the construction is still safe, even if the Q suffix is never reached.

Lemma 62 (End of scope). *If* $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s)$, *then* $\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s'}'); \mathsf{env}_T \vDash \mathsf{del}(x); Q : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta, x : \mathsf{var}(B'_{s'}); \mathsf{env}_T \vDash \mathsf{del}(x); Q : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ which is such that

$$(\mathsf{del}(x); Q, (\Delta, x : \mathsf{var}(B'_{s'})), s) \in \mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}.$$

From $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s)$, we know there exists a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet (Q, Δ, s). We then construct a candidate typing interpretation $\mathcal{R}$, such that it contains $(\mathsf{del}(x); Q, (\Delta, x : \mathsf{var}(B'_{s'})), s)$ as follows:

$$\mathcal{R} \triangleq \left\{ (\mathsf{del}(x); Q, (\Delta, x : \mathsf{var}(B'_{s'})), s) \right\} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$$

As we have only added a single triple, all that now is needed is to show that there exists at least one transition

$$\Sigma; \Gamma; \mathsf{env}_T \vDash (\mathsf{del}(x); Q, (\Delta, x : \mathsf{var}(B'_{s'})), s) \Rrightarrow P'_1$$

and that for all such transitions, the reduct P'_2 is again in $\mathcal{R}$.

From $(Q, \Delta, s) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, we know that an appropriately shaped env_{SV} can be built from $\Sigma; \Gamma; \Delta$ for all transitions from Q by the method in Definition 59 (and if Q has no transitions by Case 1 of Definition 64, then the shape of env_{SV} is irrelevant). Pick any such environment, and pick any value v such that $\textsc{TypeOf}_\Gamma(v) = (B_1, s_1)$ and $\Sigma \vdash B_1 <: B'$ and $s_1 \sqsubseteq s'$. Now, the transition

$$\Sigma; \Gamma; \Delta, x : \mathsf{var}(B'_{s'}) \vDash_s \langle \mathsf{del}(x); Q, \mathsf{env}_{TS}; \mathsf{env}_V, x : v \rangle \to \Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle$$

can be concluded by [R-DELV], and by Theorem 26, the transition can also be concluded for any other env'_{SV} having the same structure and satisfying the requirements. No other transition is possible, so we conclude that the transition

$$\Sigma; \Gamma; \mathsf{env}_T \vDash (\mathsf{del}(x); Q, (\Delta, x : \mathsf{var}(B'_{s'})), s) \Rrightarrow (Q, \Delta, s)$$

exists, and as $(Q, \Delta, s) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, we also have that $(Q, \Delta, s) \in \mathcal{R}$ by construction. Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation. □

Lemma 63 (Method return)**.** *If*

- $\Sigma; \Gamma; \Delta'; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s)$, *and*
- $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_V$,

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash (\mathsf{env}'_V, \Delta'); Q : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash (\mathsf{env}'_V, \Delta'); Q : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet $((\mathsf{env}'_V, \Delta'); Q, \Delta, s)$.

From $\Sigma; \Gamma; \Delta'; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s)$, we know there exists a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet (Q, Δ', s). We then construct a candidate typing interpretation $\mathcal{R}'$ containing $((\mathsf{env}'_V, \Delta'); Q, \Delta, s)$ as follows:

$$\mathcal{R} \triangleq \left\{ ((\mathsf{env}'_V, \Delta'); Q, \Delta, s) \right\} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$$

As we have only added a single triple, all that now is needed is to show that there exists at least one transition

$$\Sigma; \Gamma; \mathsf{env}_T \vDash ((\mathsf{env}'_V, \Delta'); Q, \Delta, s) \Rrightarrow P'_1$$

and that for all such transitions, the reduct P'_2 is again in $\mathcal{R}$. Pick any env_{SV} built from $\Sigma; \Gamma; \Delta$ by the method in Definition 59. As we know that $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_V$, the transition

$$\Sigma; \Gamma; \Delta \vDash_s \langle ((\mathsf{env}'_V, \Delta'); Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_s \langle Q, \mathsf{env}_{TS}; \mathsf{env}'_V \rangle$$

can now be concluded by [R-RETURN], and by Theorem 26 the transition can also be concluded for any other env_{SV} built according to Definition 59. No other transition is possible, so we conclude that the transition

$$\Sigma; \Gamma; \mathsf{env}_T \vDash ((\mathsf{env}'_V, \Delta'); Q, \Delta, s) \Rrightarrow (Q, \Delta', s)$$

exists, and as $(Q, \Delta', s) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, we also have that $(Q, \Delta', s) \in \mathcal{R}$ by construction. Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation. □

Lemma 64 (Security level restore)**.** *If*

- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s')$*, and*
- $s \sqsupseteq s'$

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash s'; Q : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash s'; Q : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet $(s'; Q, \Delta, s)$.

From $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash Q : \mathsf{cmd}(s')$, we know there exists a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet (Q, Δ, s'). We then construct a candidate typing interpretation $\mathcal{R}$, such that it contains $(s'; Q, \Delta, s)$ for some $s \sqsupseteq s'$ as follows:

$$\mathcal{R} \triangleq \{ (s'; Q, \Delta, s) \} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$$

As we have only added a single triple, all that now is needed is to show that there exists at least one transition

$$\Sigma; \Gamma; \mathsf{env}_T \vDash (s'; Q, \Delta, s) \Rrightarrow P'_1$$

$$[\textsc{st-bot}]\frac{}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash \bot : \mathsf{cmd}(s)}$$

$$[\textsc{st-stm}]\frac{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s) \quad \Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash Q : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash S;Q : \mathsf{cmd}(s)}$$

$$[\textsc{st-del}]\frac{\Sigma;\Gamma,\Delta;\mathsf{env}_T \vDash Q : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta,x : \mathsf{var}(B_{s'});\mathsf{env}_T \vDash \mathsf{del}(x);Q : \mathsf{cmd}(s)}$$

$$[\textsc{st-ret}]\frac{\Sigma;\Gamma;\Delta' \vdash \mathsf{env}_V \quad \Sigma;\Gamma;\Delta' \vDash Q : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vDash (\mathsf{env}'_V,\Delta');Q : \mathsf{cmd}(s)}$$

$$[\textsc{st-res}]\frac{\Sigma;\Gamma;\Delta \vDash Q : \mathsf{cmd}(s')}{\Sigma;\Gamma;\Delta \vDash s';Q : \mathsf{cmd}(s)}\ (s \sqsupseteq s')$$

Figure 8.29: Semantic type rules for stacks.

and that, for all such transitions, the reduct P_2' is again in $\mathcal{R}$. Pick any env_{SV} built from $\Sigma;\Gamma;\Delta$ by the method in Definition 59. As we know that $s \sqsupseteq s'$, the transition

$$\Sigma;\Gamma;\Delta \vDash_s \langle s';Q,\mathsf{env}_{TSV}\rangle \to \Sigma;\Gamma;\Delta \vDash_{s'} \langle Q,\mathsf{env}_{TSV}\rangle$$

can be concluded by [R-RESTORE], and by Theorem 26 the transition can also be concluded for any other env'_{SV} having the same structure and satisfying the requirements. No other transition is possible, so we conclude that the transition

$$\Sigma;\Gamma;\mathsf{env}_T \vDash (s';Q,\Delta,s) \Rrightarrow (Q,\Delta,s')$$

exists, and as $(Q,\Delta,s') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, we also have that $(Q,\Delta,s') \in \mathcal{R}$ by construction. Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation. □

As was the case with the coercion lemmas, the preceding Lemmas 60–64 can alternatively be written as inference rules. These *semantic type rules* are given in Figure 8.29. As the reader will notice, these rules closely resemble the static type rules from Figure 8.17, except that we now use the double turnstile $\vDash$ in place of the typing relation $\vdash$, and we have env_T on the left-hand side along with the type environment. The rule [ST-RES] is of course new, since it is necessitated by our extended syntax for stacks, and in [ST-RET] the extraction function $\mathbf{E}(\cdot)$ from the corresponding rule [T-RET] is no longer necessary, since we have the type environment Δ' stored directly on the stack. Nevertheless, the correspondence should be clear.

However, the crucial difference between

$$\Sigma;\Gamma;\Delta \vdash Q : \mathsf{cmd}(s) \qquad \text{and} \qquad \Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash Q : \mathsf{cmd}(s)$$

is of course that the former is a *judgment of well-typedness*, concluded inductively by the type rules, whereas the latter is an *assertion of type-safety*, defined independently of the lemmas. This is a subtle, but important difference: The syntactic type rules are first defined, and *then* well-typedness is shown to be preserved by the (untyped) semantics, with well-typedness implying type-safety. The semantic type rules are derived from lemmas, that state conditions under which *type-safety* itself is preserved by the stack constructors. The rules follow from proven statements, rather than the other way around.

8.6.4 Semantic type rules for statements

The preceding section has illustrated the key steps in the semantic approach, by which we prove admissibility of our type rules individually. We shall now continue with type-safety assertions of the form $\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s)$ for the syntactic category of statements S. Most of the proofs are again by coinduction, and as before we shall use q to denote a bottomless stack.

Lemma 65 (Skip)**.** $\Sigma;\Gamma;\Delta \vDash \texttt{skip} : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma;\Gamma;\Delta \vDash \texttt{skip} : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet $(\texttt{skip};\bot,\Delta,s)$. Here is our candidate typing interpretation:

$$\mathcal{R} \triangleq \{(\texttt{skip};\bot,\Delta,s),(\bot,\Delta,s)\}$$

and clearly, $(\texttt{skip};\bot,\Delta,s) \in \mathcal{R}$. We must now show that $\mathcal{R}$ indeed is a typing interpretation by checking the criteria of Definition 64. We examine each of the pairs we have added:

- By rule [R-SKIP] we can conclude the transition

  $$\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{skip};\bot,\mathsf{env}_{TSV}\rangle \to \Sigma;\Gamma;\Delta \vDash_s \langle \bot,\mathsf{env}_{TSV}\rangle$$

  for any Δ, s, and env_{SV}. There are no other possible transitions, so we conclude that the transition

  $$\Sigma;\Gamma;\mathsf{env}_T \vDash (\texttt{skip};\bot,\Delta,s) \Rightarrow (\bot,\Delta,s)$$

  exists, and $(\bot,\Delta,s) \in \mathcal{R}$ by construction.

- Now we must show that the criteria of Definition 64 also hold for the triplet $(\bot,\Delta,s)$. This holds by Case 1.

Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation. □

Lemma 66 (Throw). $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{throw} : \mathsf{cmd}(s)$.

Proof. By Case 1 of Definition 64. □

Lemma 67 (Sequential composition). *If*

- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_1 : \mathsf{cmd}(s)$, *and*
- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_2 : \mathsf{cmd}(s)$,

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_1; S_2 : \mathsf{cmd}(s)$.

Proof. By Definition 64,

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_2 : \mathsf{cmd}(s) = \Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_2; \bot : \mathsf{cmd}(s)$$

and $S_2; \bot$ is a stack Q. The result then follows directly from Lemma 61. □

Lemma 68 (If-then-else). *If*

- $\Sigma; \Gamma; \Delta \vDash e : \mathsf{bool}_s$, *and*
- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_T : \mathsf{cmd}(s)$, *and*
- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_F : \mathsf{cmd}(s)$, *and*

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{if } e \texttt{ then } S_T \texttt{ else } S_F : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ which is such that it contains the triplet $(\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; \bot, \Delta, s)$.

From $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_\mathsf{T} : \mathsf{cmd}(s)$ and $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_\mathsf{F} : \mathsf{cmd}(s)$ we know that there exist typing interpretations $\mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$ and $\mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T}$ such that

$$(S_\mathsf{T}; \bot, \Delta, s) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$$
$$(S_\mathsf{F}; \bot, \Delta, s) \in \mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T}.$$

We then construct the following candidate typing interpretation where q_1 and q_2 may be empty:

$$\begin{aligned}
\mathcal{R} \triangleq\ & \{(\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; \bot, \Delta, s)\} \\
& \cup \left\{ (q_1; s; \bot, \Delta_1, s_1) \mid (q_1; \bot, \Delta_1, s_1) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T} \wedge \textsc{fin}(q_1; \bot, \Delta_1, s_1) = (\Delta, s) \right\} \\
& \cup \left\{ (q_2; s; \bot, \Delta_2, s_2) \mid (q_2; \bot, \Delta_2, s_2) \in \mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T} \wedge \textsc{fin}(q_2; \bot, \Delta_2, s_2) = (\Delta, s) \right\} \\
& \cup \{(\bot, \Delta, s)\}
\end{aligned}$$

and, by construction, we have that $(\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; \bot, \Delta, s) \in \mathcal{R}$.

We must now show that $\mathcal{R}$ in fact is a typing interpretation. First we consider the two single triples that we added:

- As $\bot$ has no transitions, the triplet $(\bot, \Delta, s)$ satisfies Case 1 of Definition 64.
- From $\Sigma; \Gamma; \Delta \vDash e : \mathsf{bool}_s$ and Definition 60, we know that

  $$\Sigma; \Gamma; \Delta \vDash_{\mathsf{bool}_s} \langle e, \mathsf{env}_{SV} \rangle \to b$$

  where b is a boolean of level s for any env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59. Thus, the transition

  $$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle S_b; s; \bot, \mathsf{env}_{TSV} \rangle \end{aligned}$$

  can be concluded by rule [R-IF], where S_b can be either S_T or S_F. Hence, there are two possible transitions:

  $$\begin{aligned} \Sigma; \Gamma; \mathsf{env}_T \vDash (\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; \bot, \Delta, s) &\Rrightarrow (S_\mathsf{T}; s; \bot, \Delta, s) \\ \Sigma; \Gamma; \mathsf{env}_T \vDash (\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; \bot, \Delta, s) &\Rrightarrow (S_\mathsf{F}; s; \bot, \Delta, s) \end{aligned}$$

  which by Theorem 26 can be concluded for any env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59. There are no other possible transitions. Finally, since

  $$\begin{aligned} (S_\mathsf{T}; \bot, \Delta, s) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T} \wedge \textsc{fin}(S_\mathsf{T}; \bot, \Delta, s) = (\Delta, s) \\ (S_\mathsf{F}; \bot, \Delta, s) \in \mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T} \wedge \textsc{fin}(S_\mathsf{F}; \bot, \Delta, s) = (\Delta, s) \end{aligned}$$

  we have that $(S_\mathsf{T}; s; \bot, \Delta, s) \in \mathcal{R}$ and $(S_\mathsf{F}; s; \bot, \Delta, s) \in \mathcal{R}$ by construction.

Next we consider an arbitrary triplet $(q_1; s; \bot, \Delta_1, s_1)$ from the set

$$\left\{ (q_1; s; \bot, \Delta_1, s_1) \mid (q_1; \bot, \Delta_1, s_1) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T} \wedge \textsc{fin}(q_1; \bot, \Delta_1, s_1) = (\Delta, s) \right\}.$$

As q_1 can be empty, we have three cases to consider:

- If q_1 is empty, then it must have come from a triplet $(\bot, \Delta, s) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$, since $\textsc{fin}(\bot, \Delta, s) = (\Delta, s)$. Thus, the triplet in $\mathcal{R}$ is of the form $(s; \bot, \Delta, s)$ by construction of $\mathcal{R}$.

  As $s \sqsupseteq s$, this satisfies the side condition of [R-RESTORE], so the transition

  $$\Sigma; \Gamma; \Delta \vDash_s \langle s; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta \vDash_s \langle \bot, \mathsf{env}_{TSV} \rangle$$

  can be concluded by that rule for any appropriately shaped env_{SV}, and as argued above, $(\bot, \Delta, s) \in \mathcal{R}$ by construction.
- If q_1 is of the form $\texttt{throw}; q_1'$ then this satisfies Case 1 of Definition 64.

- If q_1 is non-empty and not of the form $\texttt{throw}; q_1'$, then all we know is that $(q_1; \bot, \Delta_1, s_1) \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$ and $\textsc{fin}(q_1, \Delta_1, s_1) = (\Delta, s)$, and that there exists at least one transition

$$\Sigma; \Gamma; \Delta_1 \vDash_{s_1} \langle q_1; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta_1' \vDash_{s_1'} \langle q_1'; \bot, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$$

  for all appropriately shaped env_{SV}, and $(q_1'; \bot, \Delta_1', s_1') \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$. There are now two possibilities, depending on the shape of q_1':

  - If q_1' is empty, then the transition above terminates normally. Hence, the reduct is of the form $\Sigma; \Gamma; \Delta \vDash_s \langle \bot, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$. Then, the transition

$$\Sigma; \Gamma; \Delta_1 \vDash_{s_1} \langle q_1; s; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta \vDash_s \langle s; \bot, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$$

    can likewise be concluded, and $(s; \bot, \Delta, s) \in \mathcal{R}$ by construction, as argued above.
  - Otherwise, if q_1' is non-empty, then the reduct is instead of the form $\Sigma; \Gamma; \Delta_1' \vDash_{s_1'} \langle q_1'; \bot, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$ and $(q_1'; \bot, \Delta_1', s_1') \in \mathcal{R}^1_{\Sigma,\Gamma,\mathsf{env}_T}$. Then, the transition

$$\Sigma; \Gamma; \Delta_1 \vDash_{s_1} \langle q_1; s; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta_1' \vDash_{s_1'} \langle q_1'; s; \bot, \mathsf{env}_T; \mathsf{env}_{SV}' \rangle$$

    can likewise be concluded. Finally, by Theorem 27 (Terminal type level) we have that $\textsc{fin}(q_1'; \bot, \Delta_1', s_1') = (\Delta, s)$, so by construction we also have that $(q_1'; s; \bot, \Delta_1', s_1') \in \mathcal{R}$, as required.

The cases for triplets from the other set

$$\left\{ (q_2; s; \bot, \Delta_2, s_2) \mid (q_2; \bot, \Delta_2, s_2) \in \mathcal{R}^2_{\Sigma,\Gamma,\mathsf{env}_T} \wedge \textsc{fin}(q_2; \bot, \Delta_2, s_2) = (\Delta, s) \right\}$$

are similar to the above, so we omit repeating the argument. Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation for the triplet $(\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; \bot, \Delta, s)$. □

Lemma 68 above tells us that we from the premises $\Sigma; \Gamma; \Delta \vDash e : (\mathsf{bool}, s)$, and $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_\mathsf{T} : \mathsf{cmd}(s)$, and $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S_\mathsf{F} : \mathsf{cmd}(s)$, can conclude that

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} : \mathsf{cmd}(s)$$

which, as expected, bears a close resemblance to a syntactic type rule. However, as the reader may have noticed, the list of premises and side conditions does not quite match the corresponding syntactic type rule, [T-IF], since, according to that rule, the conclusion should hold *for each* s' such that $s' \sqsubseteq s$. This is on purpose, since, as the reader may recall from Section 8.4.4, we chose to omit an explicit coercion rule for statements, which instead led to the addition of side conditions of the form $s' \sqsubseteq s$ to

all rules for guarded statements, where the statement under the guard may have to execute at a higher level.

However, by Lemma 59 above, we have derived the admissibility of such a coercion rule, so by combining Lemma 68 and Lemma 59, we can also conclude

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} : \mathsf{cmd}(s')$$

for each $s' \sqsubseteq s$ from these premises. In the following lemmas for guarded statement constructs, we shall therefore likewise omit this extra side condition $s' \sqsubseteq s$.

Lemma 69 (While). *If*

- $\Sigma; \Gamma; \Delta \vDash e : \mathsf{bool}_s$, *and*
- $\Sigma; \Gamma; \Delta \mathsf{env}_T \vDash S : \mathsf{cmd}(s)$,

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{while } e \texttt{ do } S : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{while } e \texttt{ do } S : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet $(\texttt{while } e \texttt{ do } S; \bot, \Delta, s)$.

From $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash S : \mathsf{cmd}(s)$ we know that there exists a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ such that $(S, \Delta, s) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. We then construct the following candidate typing interpretation where q' may be empty:

$$\begin{aligned}\mathcal{R} \triangleq\ & \{(\texttt{while } e \texttt{ do } S; \bot, \Delta, s)\} \\ & \cup \left\{ (q'; s; \texttt{while } e \texttt{ do } S; \bot, \Delta', s') \;\middle|\; \begin{array}{c} (q'; \bot, \Delta', s') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T} \\ \wedge\ \mathrm{FIN}(q'; \bot, \Delta', s') = (\Delta, s) \end{array} \right\} \\ & \cup \{(\bot, \Delta, s)\}\end{aligned}$$

and, by construction, we have that $(\texttt{while } e \texttt{ do } S; \bot, \Delta, s) \in \mathcal{R}$.

We must now show that $\mathcal{R}$ in fact is a typing interpretation. First we consider the two single triplets that we added:

- As $\bot$ has no transitions, the triplet $(\bot, \Delta, s)$ satisfies Case 1 of Definition 64.
- From $\Sigma; \Gamma; \Delta \vDash e : \mathsf{bool}_s$ and Definition 60, we know that

  $$\Sigma; \Gamma; \Delta \vDash_{\mathsf{bool}_s} \langle e, \mathsf{env}_{SV} \rangle \to b$$

  can be concluded for any env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59. Depending on the value of b, we now have a choice between two different rules by which to conclude the transition. By Theorem 26, either of the two transition can then be concluded for any env_{SV} built according to Definition 59. We consider each case in turn:

– If $b = \mathsf{F}$, then we can conclude the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{while}\ e\ \texttt{do}\ S,\ \mathsf{env}_{TSV}; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle \bot, \mathsf{env}_{TSV} \rangle \end{aligned}$$

by [R-WHILE$_\mathsf{F}$], and $(\bot, \Delta, s) \in \mathcal{R}$ by construction, as argued above.

– If $b = \mathsf{T}$, then we can conclude the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{while}\ e\ \texttt{do}\ S; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle S; s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \mathsf{env}_{TSV} \rangle \end{aligned}$$

and since $(S; \bot, \Delta, s) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ and $\textsc{fin}(S; \bot, \Delta, s) = (\Delta, s)$, then we also have that $(S; s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \Delta, s) \in \mathcal{R}$ by construction.

Next we consider an arbitrary triplet $(q'; s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \Delta', s')$ from the set

$$\left\{ (q'; s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \Delta', s') \;\middle|\; \begin{array}{c} (q'; \bot, \Delta', s') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T} \\ \wedge\ \textsc{fin}(q'; \bot, \Delta', s') = (\Delta, s) \end{array} \right\}$$

As q' can be empty, we have three cases to consider:

- If q' is empty, then it must have come from the triplet $(\bot, \Delta, s) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, since $\textsc{fin}(\bot, \Delta, s) = (\Delta, s)$. Thus, the triplet is of the form $(s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \Delta, s)$. As $s \sqsupseteq s$, this satisfies the side condition of [R-RESTORE], so the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{while}\ e\ \texttt{do}\ S; \bot, \mathsf{env}_{TSV} \rangle \end{aligned}$$

can be concluded by that rule for any appropriately shaped env_{SV}, and we have that $(\texttt{while}\ e\ \texttt{do}\ S; \bot, \Delta, s) \in \mathcal{R}$ by construction.

- If q' is of the form $\texttt{throw}; q''$, then this satisfies Case 1 of Definition 64.

- If q' is non-empty and not of the form $\texttt{throw}; q''$, then all we know is that $(q'; \bot, \Delta', s') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ and $\textsc{fin}(q'; \bot, \Delta', s') = (\Delta, s)$, and there exists at least one transition

$$\Sigma; \Gamma; \Delta' \vDash_{s'} \langle q'; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta'' \vDash_{s''} \langle q''; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle$$

for all appropriately shaped env_{SV} built from $\Sigma; \Gamma; \Delta'$ according to Definition 59, and $(q''; \bot, \Delta'', s'') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. There are now two possibilities, depending on the shape of q'':

– If q'' is empty, then the transition above terminates normally. Hence, the reduct is of the form $\Sigma; \Gamma; \Delta \vDash_s \langle \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle$. Then, the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta' \vDash_{s'} \langle q'; s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle \end{aligned}$$

can likewise be concluded, and $(s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \Delta, s) \in \mathcal{R}$ by construction, as argued above.

– Otherwise, if q'' is non-empty, then the reduct is instead of the form $\Sigma; \Gamma; \Delta'' \vDash_{s''} \langle q''; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle$ and $(q''; \bot, \Delta'', s'') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. Then, the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta_1 \vDash_{s'} \langle q'; s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta'' \vDash_{s''} \langle q''; s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle \end{aligned}$$

can likewise be concluded. Finally, by Theorem 27 (Terminal type level) we have that $\textsc{fin}(q''; \bot, \Delta'', s'') = (\Delta, s)$, so by construction we also have that $(q''; s; \texttt{while}\ e\ \texttt{do}\ S; \bot, \Delta'', s'') \in \mathcal{R}$, as required.

Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation containing the triplet $(\texttt{while}\ e\ \texttt{do}\ S; \bot, \Delta, s)$, as required. □

Lemma 70 (Variable declaration)**.** *If*

- $\Sigma; \Gamma; \Delta \vDash e : B_{s_1}$*, and*
- $\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s_1}); \mathsf{env}_T \vDash S : \mathsf{cmd}(s)$*,*

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \mathsf{var}(B_{s_1})\ x\ \texttt{:=}\ e\ \texttt{in}\ S : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \mathsf{var}(B_{s_1})\ x\ \texttt{:=}\ e\ \texttt{in}\ S : \mathsf{cmd}(s)$, we must exhibit a typing interpretation containing $(\mathsf{var}(B_{s_1})\ x\ \texttt{:=}\ e\ \texttt{in}\ S; \bot, \Delta, s)$.

From $\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s_1}); \mathsf{env}_T \vDash S : \mathsf{cmd}(s)$ we know there exists a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ such that

$$(S; \bot, (\Delta, x : \mathsf{var}(B_{s_1})), s) \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$$

We then construct the following candidate typing interpretation:

$$\begin{aligned} \mathcal{R} \triangleq\ & \left\{ (\mathsf{var}(B_{s_1})\ x\ \texttt{:=}\ e\ \texttt{in}\ S; \bot, \Delta, s) \right\} \\ & \cup \left\{ (q'; \mathsf{del}(x); \bot, \Delta', s') \;\middle|\; \begin{array}{c} (q'; \bot, \Delta', s') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T} \\ \wedge\ \textsc{fin}(q'; \bot, \Delta', s') = ((\Delta, x : \mathsf{var}(B_{s_1})), s) \end{array} \right\} \\ & \cup \{ (\bot, \Delta, s) \} \end{aligned}$$

and clearly $(\mathsf{var}(B_{s_1})\ \ x\ \texttt{:=}\ e\ \texttt{in}\ S; \bot, \Delta, s) \in \mathcal{R}$.

We must now show that $\mathcal{R}$ indeed is a typing interpretation. First, we consider the two single triplets that we added:

- As $\bot$ has no transitions, the triplet $(\bot, \Delta, s)$ satisfies Case 1 of Definition 64.
- From $\Sigma; \Gamma; \Delta \vDash e : B_{s_1}$ and Definition 60, we know that

  $$\Sigma; \Gamma; \Delta \vDash_{B_{s_1}} \langle e, \mathsf{env}_{SV} \rangle \to v$$

  where v is a value of type B and level s_1 for any env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59. Thus, the transition

  $$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \mathsf{var}(B_{s_1})\ \ x\ \texttt{:=}\ e\ \texttt{in}\ S; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s_1}) \vDash_s \langle S; \mathsf{del}(x); \bot, \mathsf{env}_{TS}; \mathsf{env}_V, x : v \rangle \end{aligned}$$

  can be concluded by rule [R-DECV]. By Theorem 26, the transition can be concluded for any env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59. There are no other possible transitions. Finally, since

  $$\begin{aligned} &(S; \bot, (\Delta, x : \mathsf{var}(B_{s_1})), s) \in \mathcal{R}_{\Sigma, \Gamma, \mathsf{env}_T} \\ &\textsc{fin}(S; \bot, (\Delta, x : \mathsf{var}(B_{s_1})), s) = ((\Delta, x : \mathsf{var}(B_{s_1})), s) \end{aligned}$$

  we have by the construction of $\mathcal{R}$ that $(S; \mathsf{del}(x); \bot, (\Delta, x : \mathsf{var}(B_{s_1})), s) \in \mathcal{R}$.

Next we consider an arbitrary triplet $(q'; \mathsf{del}(x); \bot, \Delta', s')$ from the set

$$\left\{ (q'; \mathsf{del}(x); \bot, \Delta', s') \;\middle|\; \begin{array}{c} (q'; \bot, \Delta', s') \in \mathcal{R}_{\Sigma, \Gamma, \mathsf{env}_T} \\ \wedge\ \textsc{fin}(q'; \bot, \Delta', s') = ((\Delta, x : \mathsf{var}(B_{s_1})), s) \end{array} \right\}$$

As q' can be empty, we have three cases to consider:

- If q' is empty, then it must have come from the triplet $(\bot, (\Delta, x : \mathsf{var}(B_{s_1})), s) \in \mathcal{R}_{\Sigma, \Gamma, \mathsf{env}_T}$, since $\textsc{fin}(\bot, (\Delta, x : \mathsf{var}(B_{s_1})), s) = ((\Delta, x : \mathsf{var}(B_{s_1})), s)$. Thus, the triplet in $\mathcal{R}$ is of the form $(\mathsf{del}(x); \bot, (\Delta, x : \mathsf{var}(B_{s_1})), s)$. The transition

  $$\begin{aligned} &\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s'}) \vDash_s \langle \mathsf{del}(x); \bot, \mathsf{env}_{TS}; \mathsf{env}_V, x : v \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle \bot, \mathsf{env}_{TSV} \rangle \end{aligned}$$

  can be concluded by that rule for any appropriately shaped $\mathsf{env}_S; \mathsf{env}_V, x : v$ by [R-DELV], and, as argued above, $(\bot, \Delta, s) \in \mathcal{R}$ by construction.
- If q' is of the form $\texttt{throw}; q''$, then this satisfies Case 1 of Definition 64.

- If q' is non-empty and not of the form $\texttt{throw}; q''$, then all we know is that $(q'; \bot, \Delta', s') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ and $\textsc{fin}(q'; \bot, \Delta', s') = ((\Delta, x : \mathsf{var}(B_{s_1})), s)$, and there exists at least one transition

$$\Sigma; \Gamma; \Delta' \vDash_{s'} \langle q'; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta'' \vDash_{s''} \left\langle q''; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \right\rangle$$

  for all appropriately shaped env_{SV}, and $(q''; \bot, \Delta'', s'') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. There are now two possibilities, depending on the shape of q'':

  - If q'' is empty, then the transition above terminates normally. Hence, the reduct is of the form $\Sigma; \Gamma; \Delta \vDash_s \left\langle \bot, \mathsf{env}_T; \mathsf{env}'_S; \mathsf{env}'_V, x : v'' \right\rangle$, and Δ' must have contained the entry $x : \mathsf{var}(B_{s_1})$ as well; i.e. $\Delta' = \Delta'', x : \mathsf{var}(B_{s_1})$ for some Δ''. Thus, the transition is of the form

$$\begin{aligned} &\Sigma; \Gamma; \Delta'', x : \mathsf{var}(B_{s_1}) \vDash_{s'} \langle q'; \bot, \mathsf{env}_{TS}; \mathsf{env}_V, x : v' \rangle \\ \to\ &\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s_1}) \vDash_s \left\langle \bot, \mathsf{env}_T; \mathsf{env}'_S; \mathsf{env}'_V, x : v'' \right\rangle \end{aligned}$$

    for some Δ'', s', env'_S, env'_V and v'', since we cannot know which, if any, of these components were affected by the transition. Then, the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta'', x : \mathsf{var}(B_{s_1}) \vDash_{s'} \langle q'; \mathsf{del}(x); \bot, \mathsf{env}_{TS}; \mathsf{env}_V, x : v' \rangle \\ \to\ &\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s_1}) \vDash_s \left\langle \mathsf{del}(x); \bot, \mathsf{env}_T; \mathsf{env}'_S; \mathsf{env}'_V, x : v'' \right\rangle \end{aligned}$$

    can likewise be concluded, and $(\mathsf{del}(x); \bot, (\Delta, x : \mathsf{var}(B_{s_1})), s) \in \mathcal{R}$ by construction, as argued above.
  - Otherwise, if q'' is non-empty, then the reduct is instead of the form $\Sigma; \Gamma; \Delta'' \vDash_{s''} \left\langle q''; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \right\rangle$ and $(q''; \bot, \Delta'', s'') \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. Then, the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta' \vDash_{s'} \langle q'; \mathsf{del}(x); \bot, \mathsf{env}_{TS}; \mathsf{env}_V, x : v' \rangle \\ \to\ &\Sigma; \Gamma; \Delta'' \vDash_{s''} \left\langle q''; \mathsf{del}(x); \bot, \mathsf{env}_T; \mathsf{env}'_S; \mathsf{env}'_V, x : v'' \right\rangle \end{aligned}$$

    can likewise be concluded. Finally, by Theorem 27 (Terminal type level) we have that $\textsc{fin}(q''; \bot, \Delta'', s'') = ((\Delta, x : \mathsf{var}(B_{s_1})), s)$, so by construction we also have that $(q''; \mathsf{del}(x); \bot, \Delta'', s'') \in \mathcal{R}$, as required.

Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation containing the triplet $(\mathsf{var}(B_{s_1})\ \ x\ \texttt{:=}\ e\ \texttt{in}\ S; \bot, \Delta, s)$, as required. □

Lemma 71 (Variable assignment). *If*

- $\Sigma; \Gamma; \Delta \vDash e : B_s$, *and*

- $\Delta(x) = \mathsf{var}(B_s)$

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash x \ := \ e : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash x \ := \ e : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet $(x \ := \ e; \bot, \Delta, s)$. We construct a candidate typing interpretation as follows:

$$\mathcal{R} \triangleq \{ (x \ := \ e; \bot, \Delta, s), (\bot, \Delta, s) \}$$

and clearly, $(x \ := \ e; \bot, \Delta, s) \in \mathcal{R}$.

We must now show that $\mathcal{R}$ indeed is a typing interpretation. We consider each of the triplets we have added:

- $(\bot, \Delta, s) \in \mathcal{R}$ by Case 1 of Definition 64.
- From $\Sigma; \Gamma; \Delta \vDash e : B_s$ and Definition 60, we know that

  $$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV} \rangle \to v$$

  where v is a value of type B and level s for any env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59. Furthermore, from $\Delta(x) = \mathsf{var}(B_s)$, we know that $\Delta = \Delta', x : \mathsf{var}(B_s)$ for some Δ'. By the construction in Definition 59, we therefore also have that env_V is such that $\mathsf{env}_V = \mathsf{env}'_V, x : v'$ for some env'_V and some v' of type B_s. We can then conclude the transition

  $$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle x \ := \ e; \bot, \mathsf{env}_{TS}; \mathsf{env}'_V, x : v' \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle \bot, \mathsf{env}_{TS}; \mathsf{env}_V, x : v \rangle \end{aligned}$$

  by rule [R-ASSV]. By Theorem 26, the transition can then be concluded for any env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59. There are no other possible transitions.

  Finally, $(\bot, \Delta, s) \in \mathcal{R}$ by construction.

Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation for $(x \ := \ e; \bot, \Delta, s)$, as required. □

Lemma 72 (Field assignment)**.** *If*

- $\Sigma; \Gamma; \Delta \vDash e : B_s$, *and*
- $\Delta(\texttt{this}) = \mathsf{var}(I_{s_1})$, *and*
- $\Gamma(I)(p) = \mathsf{var}(B_s)$, *and*

- $s_1 \sqsubseteq s$,

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vdash \texttt{this}.p \ := \ e : \mathsf{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{this}.p \ := \ e : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing the triplet $(\texttt{this}.p \ := \ e; \bot, \Delta, s)$. We construct a candidate typing interpretation as follows:

$$\mathcal{R} \triangleq \{(\texttt{this}.p \ := \ e; \bot, \Delta, s), (\bot, \Delta, s)\}$$

and clearly, $(\texttt{this}.p \ := \ e; \bot, \Delta, s) \in \mathcal{R}$.

We must now show that $\mathcal{R}$ indeed is a typing interpretation. We consider each of the triplets we have added:

- $(\bot, \Delta, s) \in \mathcal{R}$ by Case 1 of Definition 64.
- From $\Sigma; \Gamma; \Delta \vDash e : B_s$ and Definition 60, we know that

  $$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV} \rangle \to v$$

  where v is a value of type B and level s for any env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59.

  From $\Delta(\texttt{this}) = \mathsf{var}(I_{s_1})$, we know that $\Delta = \Delta', \texttt{this} : \mathsf{var}(I_{s_1})$ for some Δ', and $s_1 \sqsubseteq s$. Likewise, from $\Gamma(I)(p) = \mathsf{var}(B_s)$, we know that the field p exist on the interface I and is declared with type $\mathsf{var}(B_s)$. By the construction in Definition 59, we therefore also have that the environments are such that

  $$\begin{aligned} \mathsf{env}_V &= \mathsf{env}'_V, \texttt{this} : X \\ \mathsf{env}_S &= \mathsf{env}'_S, X : \mathsf{env}_F \\ \mathsf{env}_F &= \mathsf{env}'_F, p : v' \end{aligned}$$

  for some env'_V, env'_S, env'_F, and some address X of type I_{s_1}; and some value v' of type B_s. We can then conclude the transition

  $$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{this}.p \ := \ e; \bot, \mathsf{env}_T; \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v']]; \mathsf{env}_V \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle \bot, \mathsf{env}_T; \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]; \mathsf{env}_V \rangle \end{aligned}$$

  by rule [R-ASSF]. By Theorem 26, the transition can then be concluded for all env_{SV} built from $\Sigma; \Gamma; \Delta$ according to Definition 59. There are no other possible transitions. Finally, $(\bot, \Delta, s) \in \mathcal{R}$ by construction.

Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation for $(x \ := \ e; \bot, \Delta, s)$, as required. □

$$[\text{ST-ENVT}_1]\frac{}{\Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}_T^{\varnothing}}$$

$$[\text{ST-ENVT}_2]\frac{\Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}'_T \quad \Sigma;\Gamma;\mathsf{env}_T \vDash_X \mathsf{env}_M}{\Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}'_T, X : \mathsf{env}_M}$$

$$[\text{ST-ENVM}_1]\frac{}{\Sigma;\Gamma;\mathsf{env}_T \vDash_X \mathsf{env}_M^{\varnothing}}$$

$$[\text{ST-ENVM}_2]\frac{\Sigma;\Gamma;\mathsf{env}_T \vDash_X \mathsf{env}_M \quad \Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s)}{\Sigma;\Gamma;\mathsf{env}_T \vDash_X \mathsf{env}_M, f : (x_1,\ldots,x_n,S)} \ (s \sqsubseteq s_1)$$

where:

$$I_{s_1} = \Gamma(X)$$
$$\mathsf{var}(\mathsf{int}_{s_2}) = \Gamma(I)(\texttt{balance})$$
$$\Gamma(I)(f) = \mathsf{proc}((B_1,s'_1),\ldots,(B_n,s'_n)) : s$$
$$\Delta = \texttt{this} : \mathsf{var}(I_{s_1}), \texttt{value} : \mathsf{var}(\mathsf{int}_{s_2}), \texttt{sender} : \mathsf{var}(I^{\mathsf{T}}_{s_{\mathsf{T}}}), x_1 : \mathsf{var}(B_1,s'_1),\ldots,x_n : \mathsf{var}(B_n,s'_n)$$

Figure 8.30: Semantic type rules for code environments env_T.

Before proceeding with the final compatibility lemmas for method calls, we shall need to define a semantic notion of type safety for the code environment env_T, using Definition 64. It is given by the rules in Figure 8.30. As can be seen, the definition closely follows the syntactic type rules from Figure 8.18, except that we now use the double turnstile, and we also have env_T on the left-hand side. Hence, as expected, env_T is judged type-safe relative to *itself*, as well as the type constraints in Σ and Γ. Thus, the predicate expresses that there exists a typing interpretation for each method body S, with Δ and s derived from the method signature in Γ.

Lemma 73 (Method call)**.** *If*

- $\Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}_T$, *and*
- $\Sigma;\Gamma;\Delta \vDash e_1 : I^Y_s$, *and*
- $\Sigma;\Gamma;\Delta \vDash e_2 : \mathsf{int}_s$, *and*
- $\Sigma;\Gamma;\Delta \vDash e'_1 : (B'_1,s'_1) \wedge \ldots \wedge \Sigma;\Gamma;\Delta \vDash e'_k : (B'_k,s'_k)$, *and*
- $\Delta(\texttt{this}) = I^X_{s_1}$, *and*
- $\Gamma(I^X)(\texttt{balance}) = \mathsf{var}(\mathsf{int}_{s_3})$, *and*
- $\Gamma(I^Y)(\texttt{balance}) = \mathsf{var}(\mathsf{int}_{s_4})$, *and*

- $\Gamma(I^Y)(f) = \text{proc}((B'_1, s'_1), \ldots, (B'_k, s'_k)) : s$, *and*
- $s_1 \sqsubseteq s \sqsubseteq s_3, s_4$, *and*

then $\Sigma; \Gamma; \Delta; \text{env}_T \vDash \texttt{call}\ \ e_1 . f(e'_1, \ldots, e'_k) \$ e_2 : \text{cmd}(s)$.

Proof. By the definition of $\Sigma; \Gamma; \Delta; \text{env}_T \vDash \texttt{call}\ \ e_1 . f(e'_1, \ldots, e'_k) \$ e_2 : \text{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\text{env}_T}$ which is such that it contains the triplet $(\texttt{call}\ \ e_1 . f(e'_1, \ldots, e'_k) \$ e_2; \bot, \Delta, s)$.

From $\Sigma; \Gamma; \Delta \vDash e_1 : I_s^Y$ and Theorem 21 we know that e_1 evaluates to an address Y of type I' such that $\Sigma \vdash I' <: I^Y$, and of a level s' such that $s' \sqsubseteq s$. Furthermore, from $\Gamma(I^Y)(f) = \text{proc}((B'_1, s'_1), \ldots, (B'_k, s'_k)) : s$ we know that every such interface I' declares a method f with this signature, or a subtype thereof, since every subtype of I^Y must declare at least the same methods.

Finally, from $\Sigma; \Gamma; \text{env}_T \vDash \text{env}_T$ (Figure 8.30), we know that for each such declaration of the method f in env_T, it holds that $\Sigma; \Gamma; \Delta_i; \text{env}_T \vDash S_i : \text{cmd}(s_i)$, where S_i is the method body, and Δ_i and s_i are obtained from the method signature in Γ. Since both env_T and Γ are finite, the number of interfaces in Γ which are subtypes of I^Y is also finite, and the number of implementations of I^Y and any of its subtypes in env_T is also finite. We use i to index this collection in the following. Thus, we know there exists a finite collection of typing interpretations $\mathcal{R}^i_{\Sigma,\Gamma,\text{env}_T}$ containing the triplets (S_i, Δ_i, s_i) for each declaration of f.[13] We then construct a candidate typing interpretation as follows:

$$
\begin{aligned}
\mathcal{R} \triangleq\ & \left\{ (\texttt{call}\ \ e_1 . f(e'_1, \ldots, e'_k) \$ e_2; \bot, \Delta, s) \right\} \\
& \cup \left\{ (q'_i; (\text{env}_V, \Delta); s; \bot, \Delta'_i, s'_i) \;\middle|\; \begin{array}{c} (q'_i; \bot, \Delta'_i, s'_i) \in \mathcal{R}^i_{\Sigma,\Gamma,\text{env}_T} \\ \wedge\ \text{FIN}(q'_i; \bot, \Delta'_i, s'_i) = (\Delta_i, s_i) \end{array} \right\}_i \\
& \cup \left\{ (s; \bot, \Delta, s_i) \right\}_i \\
& \cup \left\{ (\bot, \Delta, s) \right\}
\end{aligned}
$$

where env_V is constructed from Σ, Γ and Δ according to the method in Definition 59.

We must now show that $\mathcal{R}$ indeed is a typing interpretation. First, we consider the three singleton sets we have added:

- Consider the triplet $(\bot, \Delta, s)$. As $\bot$ has no transitions, the triplet $(\bot, \Delta, s)$ satisfies Case 1 of Definition 64.
- Consider the triplets $(s; \bot, \Delta, s_i)$ for each i. For each s_i we know that $s_i \sqsupseteq s$ from the definition of subtyping of method declarations in the consistency rules

[13] Note: The Δ_i and s_i may differ because of subtyping of the method signature, and because of different types for the magic variable `this`. Each of the types in any Δ_i may be supertypes of the types from the signature of f on I^Y, and $s_i \sqsupseteq s$, by contravariance of the type constructor.

for Σ (Figure 8.11). Thus, this satisfies the side condition of [R-RESTORE], so the transition

$$\Sigma;\Gamma;\Delta \vDash_{s_i} \langle s;\bot,\mathsf{env}_{TSV}\rangle \to \Sigma;\Gamma;\Delta \vDash_{s} \langle \bot,\mathsf{env}_{TSV}\rangle$$

can be concluded by that rule for any appropriately shaped env_{SV}, and, by construction, $(\bot,\Delta,s) \in \mathcal{R}$.

- Consider the triplet $(\texttt{call}\ \ e_1\,.\,f\,(e'_1,\ldots,e'_k)\,\$e_2;\bot,\Delta,s)$. Pick any env_{SV} built from Σ, Γ and Δ according to Definition 59. From

$$\Sigma;\Gamma;\Delta \vDash e'_1 : (B'_1,s'_1) \wedge \ldots \wedge \Sigma;\Gamma;\Delta \vDash e'_k : (B'_k,s'_k)$$

and $\Sigma;\Gamma;\Delta \vDash e_2 : \mathsf{int}_s$ and Definition 60 and Theorem 21, we know that each of the expressions evaluates to a value of the appropriate type (B'_i,s'_i). From this, and the remaining premises of the lemma, we know that the transition

$$\begin{aligned}&\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{call}\ \ e_1\,.\,f\,(\tilde{e})\,\$e_2;\bot,\mathsf{env}_{TSV}\rangle\\ \to\ &\Sigma;\Gamma;\Delta_i \vDash_{s_i} \langle S_i;(\mathsf{env}_V,\Delta);s;\bot,\mathsf{env}_T;\mathsf{env}'_{SV}\rangle\end{aligned}$$

can be concluded by [R-CALL], and as we know that $f \in \mathrm{dom}\,(\mathsf{env}_T(Y))$ the rule [R-FCALL] cannot be used. Thus, no other transitions can be concluded, and $(S_i;(\mathsf{env}_V,\Delta);s;\bot,\Delta_i,s_i) \in \mathcal{R}$ by construction.

Next we consider an arbitrary triplet $(q'_i;(\mathsf{env}_V,\Delta);s;\bot,\Delta'_i,s'_i)$ from the sets

$$\left\{(q'_i;(\mathsf{env}_V,\Delta);s;\bot,\Delta'_i,s'_i)\ \middle|\ \begin{array}{c}(q'_i;\bot,\Delta'_i,s'_i) \in \mathcal{R}^i_{\Sigma,\Gamma,\mathsf{env}_T}\\ \wedge\ \textsc{fin}(q'_i;\bot,\Delta'_i,s'_i) = (\Delta_i,s_i)\end{array}\right\}_i$$

As q'_i can be empty, we have three cases to consider:

- If q'_i is empty, then it must have come from the triplet $(\bot,\Delta_i,s_i) \in \mathcal{R}^i_{\Sigma,\Gamma,\mathsf{env}_T}$, since $\textsc{fin}(\bot,\Delta_i,s_i) = (\Delta_i,s_i)$. Thus by construction, the triplet in $\mathcal{R}$ is of the form $((\mathsf{env}_V,\Delta);s;\bot,\Delta_i,s_i)$, and we know that $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V$. The transition

$$\Sigma;\Gamma;\Delta_i \vDash_{s_i} \left\langle(\mathsf{env}_V,\Delta);s;\bot,\mathsf{env}_{TS};\mathsf{env}'_V\right\rangle \to \Sigma;\Gamma;\Delta \vDash_{s_i} \langle s;\bot,\mathsf{env}_{TSV}\rangle$$

can therefore be concluded by that rule for any appropriately shaped $\mathsf{env}_S;\mathsf{env}'_V$ by [R-RETURN], and $(s;\bot,\Delta,s_i) \in \mathcal{R}$ by construction.

- If q'_i is of the form $\texttt{throw};q''_i$, then this satisfies Case 1 of Definition 64.

- If q_i' is non-empty and not of the form $\texttt{throw}; q_i''$, then all we know is that $(q_i'; \bot, \Delta_i', s_i') \in \mathcal{R}^i_{\Sigma,\Gamma,\mathsf{env}_T}$ and $\textsc{fin}(q_i'; \bot, \Delta_i', s_i') = (\Delta_i, s_i)$, and there exists at least one transition

$$\Sigma; \Gamma; \Delta_i' \vDash_{s_i'} \langle q_i'; \bot, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta_i'' \vDash_{s_i''} \langle q_i''; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle$$

  for all appropriately shaped env_{SV} built from $\Sigma; \Gamma; \Delta_i'$ according to Definition 59, and $(q_i''; \bot, \Delta_i'', s_i'') \in \mathcal{R}^i_{\Sigma,\Gamma,\mathsf{env}_T}$. There are now two possibilities, depending on the shape of q_i'':

  - If q_i'' is empty, then the transition above terminates normally. Hence, the reduct is of the form $\Sigma; \Gamma; \Delta_i \vDash_{s_i} \langle \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle$. Then, the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta_i' \vDash_{s_i'} \langle q_i'; (\mathsf{env}_V, \Delta); s; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta_i \vDash_{s_i} \langle (\mathsf{env}_V, \Delta); s; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle \end{aligned}$$

    can likewise be concluded, and $((\mathsf{env}_V, \Delta); s; \bot, \Delta_i, s_i) \in \mathcal{R}$ by construction, as argued above.

  - Otherwise, if q_i'' is non-empty, then the reduct is instead of the form $\Sigma; \Gamma; \Delta_i'' \vDash_{s_i''} \langle q_i''; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle$ and $(q_i''; \bot, \Delta_i'', s_i'') \in \mathcal{R}^i_{\Sigma,\Gamma,\mathsf{env}_T}$. Then, the transition

$$\begin{aligned} &\Sigma; \Gamma; \Delta' \vDash_{s_i'} \langle q_i'; (\mathsf{env}_V, \Delta); s; \bot, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta'' \vDash_{s_i''} \langle q_i''; (\mathsf{env}_V, \Delta); s; \bot, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle \end{aligned}$$

    can likewise be concluded. Finally, by Theorem 27 (Terminal type level) we have that $\textsc{fin}(q_i''; \bot, \Delta_i'', s_i'') = (\Delta_i, s_i)$, so by construction we also have that $(q_i''; (\mathsf{env}_V, \Delta); s; \bot, \Delta_i'', s_i'') \in \mathcal{R}$, as required.

Thus we conclude that $\mathcal{R}$ indeed is a typing interpretation containing the triplet $(\texttt{call}\ e_1.f(e_1', \ldots, e_k')\$e_2; \bot, \Delta, s)$, as required. □

Lemma 74 (Delegate call)**.** *If*

- $\Sigma; \Gamma; \mathsf{env}_T \vDash \mathsf{env}_T$, *and*
- $\Sigma; \Gamma; \Delta \vDash e : I_s^Y$, *and*
- $\Sigma; \Gamma; \Delta \vDash e_1' : (B_1', s_1') \wedge \ldots \wedge \Sigma; \Gamma; \Delta \vDash e_k' : (B_k', s_k')$, *and*
- $\Delta(\texttt{this}) = I_{s_1}^X$, *and*
- $\Sigma \vdash I^X <: I^Y$, *and*

- $\Gamma(I^Y)(f) = \mathsf{proc}((B'_1, s'_1), \ldots, (B'_k, s'_k)) : s$, *and*
- $s_1 \sqsubseteq s$,

then $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vdash \texttt{dcall}\ \ e.f(e'_1, \ldots, e'_k)\ : \mathsf{cmd}(s)$

Proof sketch. By the definition of $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{dcall}\ \ e.f(e'_1, \ldots, e'_k)\ : \mathsf{cmd}(s)$, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ which is such that it contains the triplet $(\texttt{dcall}\ \ e.f(e'_1, \ldots, e'_k); \bot, \Delta, s)$.

The proof is almost exactly the same as the proof for Lemma 73 (Method call) above. The only differences are that we here do not have a value parameter e_2, and the magic variables `this`, `sender` and `value` are not rebound in the new type environments Δ_i. However, neither of these differences affect the argument in the proof. Hence, we shall not repeat it here. □

Both of the preceding Lemmas 73–74 included a premise $\Sigma; \Gamma; \mathsf{env}_T \vDash \mathsf{env}_T$ requiring that *every* method declaration in env_T (except fallback functions) must be semantically type-safe. This is of course not an unreasonable requirement, since a standard Subject-Reduction theorem also would require well-typedness of all the environments (cf. Chapter 6). However, it is worth pointing out that this is an over-approximation, since we actually only need the method f (and its possible subtypes) declared on the interface I^Y (and its possible subtypes) to be type-safe, in order for the proof to work. This restriction can thus be weakened without affecting the proof. We merely preferred to use the more general requirement $\Sigma; \Gamma; \mathsf{env}_T \vDash \mathsf{env}_T$ to keep the premises of the lemmas simpler.

As with the compatibility lemmas for stacks, the preceding Lemmas 65–74 can of course also be written as semantic type rules. These are given in Figure 8.31. Again, the reader may notice the general similarity to the syntactic type rules from Figure 8.16, but also a few differences. Most notably, the side conditions of the form $s \sqsubseteq s'$ are now gone, since, as previously mentioned, the purpose of these side conditions was to ensure that the coercion property holds, for which we now instead have the coercion rule in Figure 8.28. Otherwise, the rules have similar premises, apart from the aforementioned premise $\Sigma; \Gamma; \mathsf{env}_T \vDash \mathsf{env}_T$ in [ST-CALL] and [ST-DCALL].

The similarity between the two sets of rules might seem to indicate that we have achieved nothing more than the syntactic type system from Section 8.4; i.e. the rules in Figure 8.28 provide only an *approximation* to the set of type-safe programs, just as the syntactic type rules only define well-typedness as an approximation to type safety. Both must necessarily induce *slack* in the form of statements S for which safety cannot be concluded by the rules, but which nevertheless *behave* in a type safe way at runtime; i.e. their typed transitions would not be stuck. For example, the rules [ST-CALL] and [ST-DCALL] require method calls to use a method name f, and thus these rules still cannot be used to handle cases where the magic variable `id` is used to issue a call. However, the important thing to note here is that using the

$$[\textsc{st-skip}]\frac{}{\Sigma;\Gamma;\Delta \vDash \texttt{skip} : \mathsf{cmd}(s)}$$

$$[\textsc{st-throw}]\frac{}{\Sigma;\Gamma;\Delta \vDash \texttt{throw} : \mathsf{cmd}(s)}$$

$$[\textsc{st-seq}]\frac{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S_1 : \mathsf{cmd}(s) \quad \Sigma;\Gamma;\Delta;\mathsf{env}_T \vdash S_2 : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S_1; S_2 : \mathsf{cmd}(s)}$$

$$[\textsc{st-if}]\frac{\Sigma;\Gamma;\Delta \vDash e : \mathsf{bool}_s \qquad \begin{array}{c}\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S_\mathsf{T} : \mathsf{cmd}(s)\\ \Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S_\mathsf{F} : \mathsf{cmd}(s)\end{array}}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F} : \mathsf{cmd}(s)}$$

$$[\textsc{st-while}]\frac{\Sigma;\Gamma;\Delta \vDash e : \mathsf{bool}_s \quad \Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash \texttt{while}\ e\ \texttt{do}\ S : \mathsf{cmd}(s)}$$

$$[\textsc{st-decv}]\frac{\Sigma;\Gamma;\Delta \vDash e : B_{s_1} \quad \Sigma;\Gamma;\Delta, x : \mathsf{var}(B_{s_1});\mathsf{env}_T \vDash S : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash \mathsf{var}(B_{s_1})\ x\ \texttt{:=}\ e\ \texttt{in}\ S : \mathsf{cmd}(s)}$$

$$[\textsc{st-assv}]\frac{\Sigma;\Gamma;\Delta \vDash e : B_s}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash x\ \texttt{:=}\ e : \mathsf{cmd}(s)}\ (\Delta(x) = \mathsf{var}(B_s))$$

$$[\textsc{st-assf}]\frac{\Sigma;\Gamma;\Delta \vDash e : B_s}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vdash \texttt{this}.p\ \texttt{:=}\ e : \mathsf{cmd}(s)}\ \left(\begin{array}{c}\Delta(\texttt{this}) = \mathsf{var}(I_{s_1})\\ \Gamma(I)(p) = \mathsf{var}(B_s)\end{array}\right)$$

$$[\textsc{st-call}]\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vDash e_1 : I^Y_s \qquad \Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}_T\\ \Sigma;\Gamma;\Delta \vDash e_2 : \mathsf{int}_s \quad \Sigma;\Gamma;\Delta \vDash \tilde{e} : \widetilde{B}_{\tilde{s}}\end{array}}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash \texttt{call}\ e_1.f(\tilde{e})\$e_2 : \mathsf{cmd}(s)}\ (s_1 \sqsubseteq s \sqsubseteq s_3, s_4)$$

where:

$$\mathsf{var}(I^X_{s_1}) = \Delta(\texttt{this})$$
$$\mathsf{var}(\mathsf{int}_{s_3}) = \Gamma(I^X)(\texttt{balance})$$
$$\mathsf{var}(\mathsf{int}_{s_4}) = \Gamma(I^Y)(\texttt{balance})$$
$$\mathsf{proc}(\widetilde{B}_{\tilde{s}}) : s = \Gamma(I^Y)(f)$$

$$[\textsc{st-dcall}]\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vDash e : I^Y_s \quad \Sigma \vdash I^X <: I^Y\\ \Sigma;\Gamma;\Delta \vDash \tilde{e} : \widetilde{B}_{\tilde{s}} \quad \Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}_T\end{array}}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vdash \texttt{dcall}\ e.f(\tilde{e}) : \mathsf{cmd}(s)}\ (s_1 \sqsubseteq s)$$

where:

$$\mathsf{var}(I^X_{s_1}) = \Delta(\texttt{this})$$
$$\mathsf{proc}(\widetilde{B}_{\tilde{s}}) : s = \Gamma(I^Y)(f)$$

Figure 8.31: Semantic type rules for statements

semantic type rules is *optional*. There is always the possibility of showing safety for a statement S by giving an explicit typing interpretation instead, to satisfy a premise anywhere in the system. The syntactic type system left us no way to reason about such statements, e.g. to show that a compound statement S' still is safe, even if it contains S as a sub-statement. The semantic type rules now give us that option.

Finally, we might now also want to show a theorem of compatibility for statements, similar to Theorem 23 for expressions, to relate the syntactic type rules to their semantic counterparts:

Theorem 30 (Compatibility for statements)**.**

$$\begin{aligned} &\Sigma;\Gamma;\varnothing \vdash \mathsf{env}_T \wedge \Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s) \\ \Longrightarrow\ &\Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}_T \wedge \Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash S : \mathsf{cmd}(s). \end{aligned}$$

Proof sketch. By induction on the derivations of the two type judgments

$$\Sigma;\Gamma;\varnothing \vdash \mathsf{env}_T \qquad \text{and} \qquad \Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s),$$

and a case analysis of the last rules used. Both statements have to be shown together, because of their mutual dependency. Each syntactic construct is typed by a particular syntactic type rule, and the case can then be concluded by the corresponding compatibility lemma, and using Lemma 59 for cases where the syntactic type rule has a side condition of the form $s \sqsubseteq s'$.

As an example, suppose $\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s)$ was concluded by [T-IF]. Then the judgment is of the form

$$\Sigma;\Gamma;\Delta \vdash \texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F} : \mathsf{cmd}(s)$$

and from the premise and side condition of that rule we know that

$$\begin{aligned} \Sigma;\Gamma;\Delta &\vdash e : (\mathsf{bool}_{s'}) \\ \Sigma;\Gamma;\Delta &\vdash S_\mathsf{T} : \mathsf{cmd}(s') \\ \Sigma;\Gamma;\Delta &\vdash S_\mathsf{F} : \mathsf{cmd}(s') \\ s &\sqsubseteq s' \end{aligned}$$

By Theorem 23 (Compatibility for expressions) we have that

$$\Sigma;\Gamma;\Delta \vdash e : \mathsf{bool}_{s'} \implies \Sigma;\Gamma;\Delta \vDash e : \mathsf{bool}_{s'}$$

and by the induction hypothesis we have that

$$\begin{aligned} \Sigma;\Gamma;\Delta \vdash S_\mathsf{T} : \mathsf{cmd}(s') &\implies \Sigma;\Gamma;\Delta \vDash S_\mathsf{T} : \mathsf{cmd}(s') \\ \Sigma;\Gamma;\Delta \vdash S_\mathsf{F} : \mathsf{cmd}(s') &\implies \Sigma;\Gamma;\Delta \vDash S_\mathsf{F} : \mathsf{cmd}(s') \end{aligned}$$

and $\Sigma; \Gamma; \varnothing \vdash \mathsf{env}_T \implies \Sigma; \Gamma; \mathsf{env}_T \vDash \mathsf{env}_T$. This satisfies the premises of rule [ST-IF], so by the corresponding compatibility lemma (Lemma 68), we can conclude

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} : \mathsf{cmd}(s')$$

Finally, from $s \sqsubseteq s'$ and rule [ST-STM-SUB] (Lemma 59) we can conclude

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F} : \mathsf{cmd}(s)$$

as required. □

We omit a full proof of the above result, since the syntactic type rules are no longer needed, but the sketch above shows the steps involved. A similar statement can then also be shown for stacks Q, which finally gives us the same guarantees as a Subject-Reduction result; namely that well-typedness indeed implies safety, as expected.[14]

In summary, what we have now achieved, is thus a proof system for type safety, which gives us the same safety guarantees as a proof system for well-typedness (a syntactic type system), but with the additional benefit of having a separate definition of type safety (Definition 64). This gives us more flexibility, because we can now *choose* how to show that a given premise holds; either by using the semantic type rules, or by giving an explicit typing interpretation. We can even use the definition of type safety to prove additional compatibility lemmas for special cases of statements residing in the slack of the rules from Figure 8.31. This, finally, gives us a way to approach safety for even some usages of the fallback function.

8.7 Type safe fallback functions

Armed with our notion of type safety, we can now return to the problem of how to ensure safety of the fallback function. As described in Section 8.1.3, the problem is actually twofold:

- The magic variable `id` can be bound to any method name at runtime, and the magic variable `args` works like a variadic argument list, which makes it impossible to statically assign either of them any type.
- The static type check ensures that all method calls are to *declared* method names, so fallback functions can never be invoked in a well-typed program, which thus makes either the construct, or well-typedness, useless.

[14] There is actually a minor, technical difference: Subject reduction cannot be shown directly, using the untyped semantics, because it does not store information about entering a heightened security context, which we, in the typed semantics, represented by updating the security level on the turnstile $\vDash_{s'}$ and pushing the original security level s onto the stack. For example, in the case of `if-else`, execution of the guarded statement S_b must preserve the judgment $\mathsf{cmd}(s')$, and not merely $\mathsf{cmd}(s)$ for all its execution steps, but the untyped semantics does not take this difference into account.

```
1 contract Y : I_s^Y {
2   ⋮
3   func fallback() {
4     if id = f then dcall X.f() else dcall X.g()
5   }
6 }
```

Figure 8.32: A simple example of a fallback function declaration.

In the present section we shall discuss how both of these problems may be handled using the semantic approach to type soundness.

8.7.1 Type-safe fallback calls

As we also argued in Section 8.1.3, it is possible to write a fallback function body in such a way that there is only a finite number of possible continuations, regardless of the bindings of `id` and `args`. Consider, for example, the body of the fallback function in Figure 8.32. This body is type safe at some level s, if both of the two delegate calls are type safe at level s. In other words, if we know that

- $\Sigma; \Gamma; \Delta \vDash \texttt{id = } f : \mathsf{bool}_s$, and
- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{dcall } X.f() : \mathsf{cmd}(s)$, and
- $\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{dcall } X.g() : \mathsf{cmd}(s)$

hold, then we can also conclude

$$\Sigma; \Gamma; \Delta; \mathsf{env}_T \vDash \texttt{if id = } f \texttt{ then dcall } X.f() \texttt{ else dcall } X.g() : \mathsf{cmd}(s)$$

by [ST-IF] (Lemma 68). Hence, any failed method call, that would lead to the invocation of this fallback function, could also be shown to be type safe at level s. We cannot conclude this using the existing semantic type rules, [ST-CALL] and [ST-DCALL], since both require the method name f to exist on the interface I^Y, but we *can* instead provide a new type rule, which can be applied in cases where $f \notin \mathrm{dom}\left(\Gamma(I^Y)\right)$. This rule is given in Figure 8.33.

The rule can of course also be stated in the form of a lemma and given a formal proof. However, we shall omit this, since the steps are the same as in the proof of Lemma 73 (Method call). The only difference is that we obtain the bodies of the fallback functions, S, and the type environments Δ', from an explicit quantification over addresses Y in the side condition, rather than having the more general statement $\Sigma; \Gamma; \mathsf{env}_T \vDash \mathsf{env}_T$ in the premise.

$$[\textsc{st-fcall}]\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vDash e_1 : I_s^Y \\ \Sigma;\Gamma;\Delta \vDash e_2 : \mathsf{int}_s \quad \Sigma;\Gamma;\Delta \vDash \tilde{e} : \widetilde{B}_{\tilde{s}}\end{array}}{\Sigma;\Gamma;\Delta;\mathsf{env}_T \vDash \texttt{call}\ \ e_1.f(\tilde{e})\$e_2 : \mathsf{cmd}(s)}\ (s_1 \sqsubseteq s \sqsubseteq s_3, s_4)$$

where:

$$\begin{aligned}\mathsf{var}(I_{s_1}^X) &= \Delta(\texttt{this}) \\ \mathsf{var}(\mathsf{int}_{s_3}) &= \Gamma(I^X)(\texttt{balance}) \\ \mathsf{var}(\mathsf{int}_{s_4}) &= \Gamma(I^Y)(\texttt{balance}) \\ f &\notin \mathrm{dom}\left(\Gamma(I^Y)\right) \\ \forall Y \in \mathrm{dom}\left(\mathsf{env}_T\right)\ .\ \Gamma(Y) = I'_{s'} \wedge \Sigma \vdash I' <: I^Y \wedge s' \sqsubseteq s &\implies \\ \Sigma;\Gamma;\Delta';\mathsf{env}_T \vDash S : \mathsf{cmd}(s)&\end{aligned}$$

where:

$$\begin{aligned}(\epsilon, S) &= \mathsf{env}_T(Y)(\texttt{fallback}) \\ \Delta' = \texttt{this} &: \mathsf{var}(I'_{s'}), \texttt{value} : \mathsf{var}(\mathsf{int}_s), \texttt{sender} : \mathsf{var}(I_{s_\top}^\top), \\ \texttt{id} &: \mathsf{var}(\mathsf{idf}_s), \texttt{args} : \mathsf{var}(\widetilde{B}_{\tilde{s}})\end{aligned}$$

Figure 8.33: A semantic type rule for fallback function calls.

The aforementioned difference is unimportant w.r.t. proving admissibility of the rule, but it *does* make a difference w.r.t. *using* the rule in type checking a piece of code: In rules [ST-CALL] and [ST-DCALL], the statement $\Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}_T$ is independent of the other premises. It does not have to be proved for each application of one of the aforementioned rules. In contrast, in the rule [ST-FCALL], the quantification is over all addresses Y satisfying a condition which depends on the type of the expression e_1, and this, in turn, determines the type environment Δ' and the level s for which the body S must be type safe. The truth value of this side condition may thus have to be evaluated anew *each time* the rule is applied, whereas $\Sigma;\Gamma;\mathsf{env}_T \vDash \mathsf{env}_T$ can be shown separately, once and for all.

When spelt out in words, the quantification in the side condition reads as follows: Assume I_s^Y is the runtime type of the object path e_1. For all addresses Y in env_T, if the following three conditions hold for Y:

- $\Gamma(Y) = I'_{s'}$, i.e. the type of Y is $I'_{s'}$, and
- $\Sigma \vdash I' <: I^Y$, i.e. I' is a subtype of I^Y, and
- $s' \sqsubseteq s$, i.e. the level of Y is below or equal to the target level s,

then $\Sigma;\Gamma;\Delta';\mathsf{env}_T \vDash S : \mathsf{cmd}(s)$ must also hold, where S is the body of the fallback function of Y, and Δ' is built in the expected way, in accordance with the semantic rule [R-FCALL]. In other words, we must require that the fallback functions of *every* contract implementing the interface I^Y, *or one of its subtypes*, must be type safe at level s. This extra step is necessary, since I_s^Y is not guaranteed to be the actual type of

```
contract Y : I_s^Y {
  ⋮
  func fallback() {
    if id = f_1 ∨ ... ∨ id = f_n then
      dcall X.id()
    else
      dcall X.f_default()
  }
}
```

Figure 8.34: An example where `id` is used as the method name.

e_1; only its runtime type. Any address having an actual type $I'_{s'}$, where I' is a subtype of I^Y, and $s' \sqsubseteq s$, is also in the set of values inhabiting the type I^Y_s; hence they too could be obtained from the evaluation of e_1, and they must therefore be included as well.

As with the other semantic type rules, rule [ST-FCALL] only requires *that* the premises and side conditions must hold, but it imposes no restrictions on *how* they are shown. A proof of type safety for a fallback function body S, w.r.t. some type environment Δ and security level s, could simply be given in the form of an explicit typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\text{env}_T}$ containing $(S; \bot, \Delta, s)$. Such a typing interpretation is just a set of triplets, and it could therefore in principle be provided by the contract creator and stored together with the code itself, to witness the type safety of the contract's fallback function. This would then allow any *user* of the code, who might be writing either a transaction or another contract containing a method call that will result in an invocation of that fallback function, to use this typing interpretation to satisfy the premise of rule [ST-FCALL] when type checking his own code.

In the case of Figure 8.32, showing type safety of the body is actually easy, since this can be done by the semantic type rules directly, because the magic variable `id` is not used directly to issue the delegate call. Its value usage is fully encapsulated by the `if-else` construct and its value is only used in the guard expression e, effectively reducing the infinite number of possible values (method names) to a finite number of possible continuations. However, we can also construct a slightly more elaborate example, where calls are issued via `id` directly, such as in Figure 8.34. Here, calls to certain known method names $f_1, \ldots, f_n$, are forwarded to their respective implementations on X, and all other calls are passed to a default handler $f_{\texttt{default}}$. This approaches the pointer-to-implementation pattern from Section 8.1.2. Clearly, this example could also be rewritten as a chain of `if-else` constructs, as in Figure 8.32, and thus be directly typable by the semantic type rules, but if we prefer this more succinct notation then safety will have to be shown via an explicit typing interpretation.

```
1  func fallback() {
2    if len(args) = 1 then
3      if typeof(args[0]) = type(int,H) then call X.f_1(args)$value
4      else if ...
5      ⋮
6    else
7      call X.f_default()$value
8  }
```

Figure 8.35: An example with safe encapsulation of variadic arguments.

8.7.2 Variadic arguments

Another issue pertains to the magic variable `args`, which we have hitherto avoided using. It acts as a variadic argument list, i.e. an array of unknown length with values of mixed types. This can easily cause transitions to be stuck, due to a wrong number of arguments or mismatched types in method calls, if a method call is issued with `args` in place of an ordinary argument list. Yet `args` too may be encapsulated in a safe manner, albeit this will require some extensions to the type language, which we have preferred to avoid in order not to complicate the presentation. It is beyond the scope of this chapter to include the necessary developments to handle this construct, but we can at least sketch the steps involved.

As a first step, suppose we admit operators such as `len`, `typeof` and array indexing into the language of expressions, which can be used to extract information from the `args` object at runtime. These operators have their expected meaning, so `len(args)` would return the number of elements in the array, and `args[`i`]` would allow us to *read* the i'th element of the array. Note: expressions cannot occur on the LHS of an assignment, so we still assume that `args` is immutable. Finally, `typeof` returns a value representing the *type* of its argument, so we will have to further assume that such type objects are added to the set of values and that they can be created by an operator `type`, i.e. `type(int,` s`)`, `type(bool,` s`)` for basic values; and `type(`I`,` s`)` for an arbitrary interface name I and security level s.

Consider now the code in Figure 8.35. By using these constructs, we can meticulously check the number of arguments present in `args` *and* their actual types, and then pass control to various handlers accordingly. Again, we ensure safety by reducing the infinite number of possible values of `args` to yield only a finite number of possible continuations.

The problem here is the operator `typeof`. We have hitherto assumed that all expression operators `op` can be given a type $\widetilde{B} \to B$, but the `typeof` operator must be able to operate on a value of *any* type. Hence, it is polymorphic, and the language of types for operators must therefore be extended with polymorphic types, and we

must furthermore extend the language of base types B with a type for type objects, e.g. type. Thus, the type of `typeof` must be

$$\vdash \texttt{typeof} : \forall a \,.\, a \to \texttt{type}$$

where a is a type variable. These are technical details that are, in a sense, orthogonal to the issue of semantic typing, so we shall not pursue them further in the present chapter. However, extending the type language with these features would be a relevant avenue for future work.

8.7.3 Up-to techniques for semantic typing

To manually prove type safety for a stack Q w.r.t. some type environment Δ and security level s, we must exhibit a typing interpretation $\mathcal{R}_{\Sigma,\Gamma,\text{env}_T}$ containing the triplet (Q, Δ, s). A major disadvantage of this approach is that it requires us to check the conditions of a typing interpretation for all the states reachable from (Q, Δ, s) in the type-lifted transition system (Definition 63). This poses some practical problems in light of our proposal to store explicit typing interpretations beside contract code to witness type safety of fallback functions. In particular, from an implementation perspective, we obviously cannot store an infinite object on the blockchain, unless it can be given a finite representation. There are also two other, related problems in this regard:

- A contract creator, wishing to demonstrate type safety of his untypable code, must be able to *build* a typing interpretation containing the code.
- A user, wishing to invoke a piece of untypable code and ensure that the invocation is type safe, must be able to *verify* that a given set of triplets indeed is a typing interpretation. This should preferably be doable by an automatic decision procedure, which can be called by the type checker itself, rather than requiring manual intervention.

Not only may these tasks be laborious to do manually; they may even be impossible, since typing interpretations can contain infinitely many triplets. Furthermore, to be practically useful, it must be decidable whether a given set $\mathcal{R}$ in fact is a typing interpretation, which is not always the case.

The problem is quite similar to the problem of proving bisimilarity between two programs P_1, P_2, by exhibiting a bisimulation relation $\mathcal{R}$ containing the pair (P_1, P_2). In that case, we would also have to check that the conditions of a bisimulation hold for every pair in the candidate relation. This problem is particularly well-known in the field of process calculi, such as the π-calculus [83; 86; 96], where program equivalences such as bisimulation are studied extensively; see e.g. [113; 112].

To alleviate this burden of proof, Pous and Sangiorgi [106] present a number of *up-to techniques* for proving bisimilarity. The idea is that it may not be necessary

to find a relation $\mathcal{R}$ containing (P_1, P_2), that *is* a bisimulation; it may be enough that $\mathcal{R}$ is *contained in* a bisimulation, as long as the missing pairs can be inferred from those in $\mathcal{R}$. Pous and Sangiorgi present a general theory of how to prove soundness of an up-to technique, but the theory does not actually depend on the definition of a bisimulation: it is formulated in terms of greatest fixed-points on complete lattices, and it can therefore also be adapted to work with other coinductively defined objects. In particular, it may also be made to work with typing interpretations.

8.7.3.1 A motivating example

Consider the body of the fallback function in Figure 8.34. Calling methods by using the magic variables `id` makes the body untypable by the semantic type rules, as we have previously argued. However, the body itself consists of just a single `if-else` statement and a delegate method call in each branch. Thus, after just two transition steps, the body on the stack will have reduced to the body of one of the methods $f_1, \ldots, f_n$ or $f_{\texttt{default}}$. In terms of the type-lifted transition system, this body will yield a transition sequence of the form

$$\Sigma; \Gamma; \mathsf{env}_T \vDash (\texttt{if } \ldots \texttt{ ;} \bot, \Delta, s_1) \Rightarrow (\texttt{dcall } \ldots \texttt{ ;} s_1; \bot, \Delta, s_2)$$
$$\Sigma; \Gamma; \mathsf{env}_T \vDash (\texttt{dcall } \ldots \texttt{ ;} s_1; \bot, \Delta, s_2) \Rightarrow (S_i; (\mathsf{env}_V, \Delta); s_2; s_1; \bot, \Delta_i, s_i)$$

where S_i is the body of the called handler method f_i; and Δ_i is the constructed type environment; and $s_i \sqsupseteq s_2 \sqsupseteq s_1$. If we further assume that each of the handler methods f_i in fact are semantically typable, then we know there exists a typing interpretation $\mathcal{R}^i_{\Sigma,\Gamma,\mathsf{env}_T}$ containing $(S_i; \bot, \Delta_i, s_i)$. From this and the above, we can also conclude that

$$\Sigma; \Gamma; \Delta_i; \mathsf{env}_T \vDash S_i; (\mathsf{env}_V, \Delta); s_2; s_1; \bot \,:\, \mathsf{cmd}(s_i)$$

by rules [ST-RES], [ST-RET] and [ST-STM] in Figure 8.29. Thus, we *also* know that there exists a typing interpretation $\mathcal{R}^{i'}_{\Sigma,\Gamma,\mathsf{env}_T}$ containing $(S_i; (\mathsf{env}_V, \Delta); s_2; s_1; \bot, \Delta_i, s_i)$.

A typing interpretation for the fallback function body must contain the three triplets mentioned above for each of the possible other method calls f_i, *and* the union of all the typing interpretations $\mathcal{R}^{i'}_{\Sigma,\Gamma,\mathsf{env}_T}$. The union of two typing interpretations is itself a typing interpretation, so we can let

$$\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T} \triangleq \mathcal{R}^{1'}_{\Sigma,\Gamma,\mathsf{env}_T} \cup \ldots \cup \mathcal{R}^{n'}_{\Sigma,\Gamma,\mathsf{env}_T}$$

for each of the $1, \ldots, n$ possible method calls and thus obtain a typing interpretation containing all the *continuations* after the method call is issued. However, the problem is that the set

$$\mathcal{S} \triangleq \bigcup_{i \in 1,\ldots n} \left\{ \begin{array}{l} (\texttt{if } \ldots \texttt{ ;} \bot, \Delta, s_1), \\ (\texttt{dcall } \ldots \texttt{ ;} s_1; \bot, \Delta, s_2), \\ (S_i; (\mathsf{env}_V, \Delta); s_2; s_1; \bot, \Delta_i, s_i) \end{array} \right\}_i$$

consisting of the union of the three triplets for each of the f_i handler methods, is *not* on its own a typing interpretation. Hence, we cannot just discard $\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ and only take $\mathcal{S}$ as our witness of type safety of the fallback function. We need to take the whole union of $\mathcal{S}$ and $\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$, and likewise, any user wishing to ascertain type safety of a call, that would invoke this fallback function, would therefore also need to *check* this whole union. This is impractical, since we can clearly see that if $(Q', \Delta', s') \in \mathcal{S}$ then

$$\Sigma; \Gamma; \mathsf{env}_T \vDash (Q', \Delta', s') \Rrightarrow (Q'', \Delta'', s'') \wedge (Q'', \Delta'', s'') \in \mathcal{S} \cup \mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}.$$

Thus, although $\mathcal{S}$ is not itself a typing interpretation, it *is* a typing interpretation *up-to union* with $\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$. Thus, we can use it in place of the full typing interpretation, and let the remainder of the stack, $S_i; (\mathsf{env}_V, \Delta); s_2; s_1; \bot$, be typed by the ordinary semantic type rules. Using $\mathcal{S}$ in place of $\mathcal{S} \cup \mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ will thus drastically reduce the number of triplets that must be checked, and make it easier for both the contract creator to build the set $\mathcal{S}$ and for the user to verify it.

8.7.3.2 Typing interpretations up-to union

Following Pous and Sangiorgi [106], we shall formulate up-to techniques in terms of progressions, which we define as follows:

Definition 65 (Progression)**.** For a fixed set of environments Σ, Γ and env_T, and two sets of stack type triplets $\mathcal{S}, \mathcal{R} \subseteq \mathcal{P}$, we say that $\mathcal{S}$ *progresses* to $\mathcal{R}$, written $\mathcal{S} \rightarrowtail \mathcal{R}$, if $(Q, \Delta, s) \in \mathcal{S}$ implies

1. $Q = \bot$ or $Q = \texttt{throw}; Q'$, or
2. $\exists P'_1$ such that $\Sigma; \Gamma; \mathsf{env}_T \vDash (Q, \Delta, s) \Rrightarrow P'_1$, and
   $\forall P'_2$ such that $\Sigma; \Gamma; \mathsf{env}_T \vDash (Q, \Delta, s) \Rrightarrow P'_2$ it holds that $P'_2 \in \mathcal{R}$. ■

Note the similarity to the definition of typing interpretations (Definition 64). When $\mathcal{S}$ and $\mathcal{R}$ coincide, we have the ordinary definition of a typing interpretation; thus $\mathcal{S} \rightarrowtail \mathcal{S}$ if and only if $\mathcal{S}$ is a typing interpretation. The purpose of an up-to technique is to allow us to infer $\mathcal{S} \subseteq \nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ from a progression $\mathcal{S} \rightarrowtail \mathcal{S} \cup \mathcal{R}$, where $\mathcal{R}$ is a set of redundant triplets. If this is possible, then the up-to technique is *valid* (sound).

Using Definition 65, we can now define the up-to union technique and prove its validity as follows:

Definition 66 (Typing interpretation up-to union)**.** Let $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ be a typing interpretation. A set of stack type triplets $\mathcal{S}$ is a *typing interpretation up-to union* w.r.t. environments Σ, Γ and env_T if $\mathcal{S} \rightarrowtail \mathcal{S} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. ■

Lemma 75 (Validity of up-to union). *Let $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ be a typing interpretation, and assume $\mathcal{S}$ is a typing interpretation up-to union with $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. Then*

$$\mathcal{S} \rightarrowtail \mathcal{S} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T} \implies \mathcal{S} \subseteq \nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}.$$

Proof. We proceed by showing that $\mathcal{S} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ is a typing interpretation according to Definition 64.

By assumption, we know that $\mathcal{S} \rightarrowtail \mathcal{S} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ where $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ is a typing interpretation. Now consider an arbitrary triplet $P \in \mathcal{S} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. If $P \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$, then we already know that P satisfies the requirements of Definition 64, since $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ is a typing interpretation. Otherwise, if $P \in \mathcal{S}$, there are two possibilities:

- If $P \not\Mapsto$ then P must satisfy Case 1 of Definition 65. But then P also satisfies Case 1 of Definition 64, since the requirements are the same.
- If $P \Mapsto P_1'$ then we know by Case 2 of Definition 65 that for all P_2' such that $P \Mapsto P_2'$ it holds that $P_2' \in \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. Thus $P_2' \in \mathcal{S} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ as well. Therefore P satisfies Case 2 of Definition 64.

Thus we conclude that $\mathcal{S} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ indeed is a typing interpretation. By Definition 64, it therefore also holds that

$$\mathcal{S} \cup \mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T} \subseteq \nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$$

since $\nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ is the union of all typing interpretations. Thus we conclude that $\mathcal{S} \subseteq \nu\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$ holds as well, as required. □

Lemma 75 assures us that the up-to union technique is valid. Going back to the example in the previous section, this means that we indeed can use the set

$$\mathcal{S} \triangleq \bigcup_{i \in 1,\ldots n} \left\{ \begin{array}{l} (\texttt{if } \ldots \texttt{ ;} \bot, \Delta, s_1), \\ (\texttt{dcall } \ldots \texttt{ ;} s_1; \bot, \Delta, s_2), \\ (S_i; (\mathsf{env}_V, \Delta); s_2; s_1; \bot, \Delta_i, s_i) \end{array} \right\}_i$$

in place of the full typing interpretation $\mathcal{S} \cup \mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ to witness type-safety of the body of the fallback function, because, as we argued there,

$$\mathcal{S} \rightarrowtail \mathcal{S} \cup \mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$$

where $\mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ is the typing interpretation for the continuations after the initial three transitions. Instead of storing $\mathcal{S} \cup \mathcal{R}'_{\Sigma,\Gamma,\mathsf{env}_T}$ on the blockchain, it will therefore suffice to store just $\mathcal{S}$, together with an indication that $\mathcal{S}$ is a typing interpretation up-to union with the typing interpretation corresponding to the semantic type judgments

$$\Sigma; \Gamma; \Delta_i; \mathsf{env}_T \vDash S_i; (\mathsf{env}_V, \Delta); s_2; s_1; \bot \,:\, \mathsf{cmd}(s_i)$$

which can be concluded by the ordinary semantic type rules. This greatly reduces the number of triplets that must be checked by the type checker of the user.

Besides up-to union, Pous and Sangiorgi [106] also define a number of other up-to techniques for bisimulations, such as up-to context, and up-to union-and-context, which is a combined technique. They also define a more abstract framework for formulating and proving the validity of up-to techniques, and for showing when technique can safely be combined. Some of these techniques might also be relevant as up-to techniques for typing interpretations; e.g. up-to context for contexts that preserve type-safety. However, it is beyond the scope of this chapter to develop a further collection of up-to techniques, but we mention it here as a relevant area for future work.

8.8 Conclusions and future work

The fallback function is a particularly problematic programming construct, both in terms of the security breaches it has given rise to, and the challenges it poses for a type theoretic approach to ensuring program safety. In the present chapter, we have studied this construct in detail, and proposed a solution to the problem of how to recover type safety of programs with fallback functions, without simply rendering the construct useless by preventing its execution altogether.

As we have seen, the difficulty with the fallback function construct derives from the two magic variables `id` and `args`, which are bound to the method name and actual parameter list of a failed method call at runtime. This, in effect, admits introspection (a limited form of reflection) into the language, which cannot be handled by the usual syntactic approach to type soundness, that we have used in previous chapters, because these two variables cannot be assigned a meaningful, static type. To be precise, they *can* be assigned a type, such as `idf` for `id`, but this type is too general to say anything about how `id` may be used safely at runtime.

The problem with the syntactic approach is that it tends to identify 'well-typedness' (a syntactic property) with 'safety' (a semantic, or behavioural, property). Specifically, in the syntactic approach, we define a syntactic predicate, well-typedness, by means of syntactic type rules. These rules are defined *a priori* and then subsequently shown to *imply* safety, but they are not *derived* from the notion of safety itself. This renders the syntactic approach incapable of reasoning about programs, or parts of programs, that *behave* safely, regardless of their syntactical structure. In the case of the fallback function, it makes the syntactic approach unsuitable, because the distinction between safe and unsafe usages of this construct is not a purely syntactic feature.

Instead, we follow a different approach: the *semantic* approach to type safety. We define the meaning of types and type judgments in terms of a typed semantics, which gives us a definition of type safety that does not depend on syntactic type rules. The typed semantics is simply a restricted version of the untyped semantics,

in the sense that every transition, that can be concluded in the typed semantics, has a corresponding transition in the untyped semantics (shown in Theorem 24); and, furthermore, typed transitions preserve type safety (Theorem 28). Programs are type safe, if they have a typed transition, and, conversely, a struck program configuration corresponds to a violation of type safety.

We should note that using a typed semantics is not the only possible way of formulating a semantics of types. For example, in [23], Caires gives a *logical* characterisation of various notions of type safety in the π-calculus, using the language of spatial logics [24; 25; 22], by expressing types as formulae in this language. This allows him to directly define the safe sets of programs (processes) w.r.t. a given type formula, corresponding to our typing interpretations, as simply those processes that satisfy the formula. We are not aware of a program logic that is suitable for describing secure data flows as in the Volpano, Smith and Irvine type system [130], so we preferred to use a typed semantics, since this appeared more straightforward. The downside of this choice was that we needed to show several theorems regarding properties of typed transitions to ensure that the typed semantics in fact captures the intended notions of type safety, and to be able to abstract away from details such as the specific values stored in the environments env_{SV}. In that light, a logical characterisation might therefore have been preferable. In the words of Caires [23, p. 8]: *any type system should be seen as a compositional, decidable, and (usually incomplete) proof system for a specialized logic*, which naturally poses the question of what kind of logic might be appropriate for describing secure data flows. Developing a such logic, with a corresponding logical characterisation of the present notion of safety, is therefore a relevant avenue for future work.

Based on the typed semantics, we proceed to define the notion of a *typing interpretation*, which is a set of programs, given in the form of stack type triplets (stacks equipped with a security level and a local type environment), satisfying one of two conditions: either the top of the stack is a terminal symbol ($\bot$ or `throw`); or the stack triplet has at least one transition, and the reducts of all transitions are again contained in the typing interpretation. The definition is coinductive, with the largest typing interpretation, *typing*, defined as the union of all typing interpretations. Using the familiar coinductive proof technique, we then prove a collection of lemmas (Lemmas 58–74) stating conditions under which type safety, witnessed by the existence of a typing interpretation containing the program, is preserved by the syntactic constructs of the language. These lemmas closely correspond to our syntactic type rules, and indeed they can also be written in such a form (here called *semantic* type rules), but with two important differences: Firstly, the rules are derived from the notion of safety, rather than the other way around; and secondly, the premises of the rules only require that the constituents of the program must *be* type safe; not that type safety necessarily must be *provable* by the semantic type rules. To give it a slogan, one might say that semantic typing is a *type-safety first* approach.

These two features finally give us a way to recover type safety for programs

containing fallback functions. However, in our proposed scheme, the obligation to prove type safety rests on the contract creator, but the user (or rather, the type checker on the user's system) must still be able to *verify* that a given 'witness set' of stack type triplets in fact *is* a typing interpretation. Both of these tasks may be difficult, time-consuming, resource-costly or outright impossible in some cases, because typing interpretations can be infinite. Finally, we therefore discuss a way to make these tasks more manageable by introducing techniques to allow type safety to be proved (and witnessed) by sets that are smaller than a full-blown typing interpretation. By once again making use of the fact that our semantic typing approach is based on coinduction, we introduce the concept of *up-to techniques for typing interpretations*, drawing on the work of Pous and Sangiorgi [106] on enhancements of the bisimulation proof method. Specifically, we define the *up-to union* technique for typing interpretations and prove its validity in Lemma 75. Using this technique, it is sufficient to show that all reducts of triplets in a witness set are *contained in* a typing interpretation, but this need not necessarily be the same as the witness set itself. This can drastically decrease the number of stack type triplets that must be checked; and, from an implementation perspective, it can drastically reduce the size of the witness set to be stored on the blockchain, thereby making our proposed scheme more viable.

The adaptation of techniques from the field of process calculi and bisimulation is interesting, as it also opens new avenues of future work. For example, Pous and Sangiorgi also discuss several other up-to techniques besides up-to union, such as up-to context and also methods for combining up-to techniques. These techniques should also be transferable to our setting, since they pertain to the coinduction proof method, and not only to bisimulations. We do not attempt to formulate equivalent techniques for typing interpretations in the present chapter, but they too might be relevant to help making explicit typing interpretations both easier to find and to verify.

8.9 Proofs

8.9.1 Proof of Theorem 19

Proof. By induction on the rules of the typed semantics of expressions (Figure 8.22).

- Suppose [R-VAL] was used. Then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle v, \mathsf{env}_{SV} \rangle \to v$$

  As there are no premises or side conditions, $\langle v, \mathsf{env}_{SV} \rangle \to v$ can then be concluded by [EXP-VAL].

- Suppose [R-VAR] was used. Then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle x, \mathsf{env}_{SV} \rangle \to v$$

  and from the side condition of this rule we know that $\mathsf{env}_V(x) = v$. This satisfies the side condition of [EXP-VAR], which then allows us to conclude $\langle x, \mathsf{env}_{SV} \rangle \to v$.

- Suppose [R-FIELD] was used. Then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e.p, \mathsf{env}_{SV} \rangle \to v$$

  and from the premise and side conditions we know that

$$\frac{\Sigma; \Gamma; \Delta \vDash_{I_s} \langle e, \mathsf{env}_{SV} \rangle \to X}{\mathsf{env}_S(X)(p) = v}$$

  Instantiating the induction hypothesis then yields

$$\Sigma; \Gamma; \Delta \vDash_{I_s} \langle e, \mathsf{env}_{SV} \rangle \to X \implies \langle e, \mathsf{env}_{SV} \rangle \to X$$

  and from $\langle e, \mathsf{env}_{SV} \rangle \to X$ and $\mathsf{env}_S(X)(p) = v$ and rule [EXP-FIELD] we can then conclude $\langle e.p, \mathsf{env}_{SV} \rangle \to v$.

- Suppose [R-OP] was used. Then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle \mathrm{op}(\tilde{e}), \mathsf{env}_{SV} \rangle \to v$$

  and from the premises we have that

$$\frac{\begin{array}{c}\Sigma; \Gamma; \Delta \vDash_{(B_1, s)} \langle e_1, \mathsf{env}_{SV} \rangle \to v_1 \\ \vdots \\ \Sigma; \Gamma; \Delta \vDash_{(B_n, s)} \langle e_n, \mathsf{env}_{SV} \rangle \to v_n \end{array}}{\mathrm{op}(v_1, \ldots, v_n) \to_{\mathrm{op}} v}$$

Instantiating the induction hypothesis for each of the n transitions then yields

$$\begin{aligned}
&\Sigma;\Gamma;\Delta \vDash_{(B_1,s)} \langle e_1, \mathsf{env}_{SV}\rangle \to v_1 \implies \langle e_1, \mathsf{env}_{SV}\rangle \to v_1\\
&\qquad\vdots\\
&\Sigma;\Gamma;\Delta \vDash_{(B_n,s)} \langle e_n, \mathsf{env}_{SV}\rangle \to v_n \implies \langle e_n, \mathsf{env}_{SV}\rangle \to v_n
\end{aligned}$$

From $\langle e_1, \mathsf{env}_{SV}\rangle \to v_1$, ..., $\langle e_n, \mathsf{env}_{SV}\rangle \to v_n$ and $\mathrm{op}(v_1, \ldots, v_n) \to_{\mathrm{op}} v$ and rule [EXP-OP] we can then conclude $\langle \mathrm{op}(\vec{e}), \mathsf{env}_{SV}\rangle \to v$. □

8.9.2 Proof of Theorem 20

Proof. By induction on the rules of the typed semantics (Figure 8.22).

- If [R-VAL] was used, then the transition is of the form

$$\Sigma;\Gamma;\Delta \vDash_{(B_1,s_1)} \langle v, \mathsf{env}_{SV}\rangle \to v$$

and from the premise and side condition we know that

$$\begin{aligned}
\textsc{TypeOf}_\Gamma(v) &= (B', s')\\
\Sigma &\vdash B' <: B_1\\
s' &\sqsubseteq s_1
\end{aligned}$$

By assumption, $\Sigma \vdash B_1 <: B_2$ and $s_1 \sqsubseteq s_2$, so by transitivity of the subtyping relation and the ordering relation we have that

$$\begin{aligned}
\Sigma &\vdash B' <: B_2\\
s' &\sqsubseteq s_2
\end{aligned}$$

This satisfies the premises of [R-VAL], so

$$\Sigma;\Gamma;\Delta \vDash_{(B_2,s_2)} \langle v, \mathsf{env}_{SV}\rangle \to v$$

can be concluded as well.

- If [R-VAR] was used, then the transition is of the form

$$\Sigma;\Gamma;\Delta \vDash_{(B_1,s_1)} \langle x, \mathsf{env}_{SV}\rangle \to v$$

and from the premise and side condition we know that

$$\begin{aligned}\mathsf{env}_V(x) &= v\\ \textsc{TypeOf}_\Gamma(v) &= (B_1', s_1')\\ \Delta(x) &= \mathsf{var}(B_2', s_2')\\ \Sigma &\vdash B_1' <: B_2' <: B_1\\ s_1' &\sqsubseteq s_2' \sqsubseteq s_1\end{aligned}$$

By assumption, $\Sigma \vdash B_1 <: B_2$ and $s_1 \sqsubseteq s_2$, so by transitivity of the subtyping relation and the ordering relation we have that

$$\begin{aligned}\Sigma &\vdash B_1' <: B_2' <: B_2\\ s_1' &\sqsubseteq s_2' \sqsubseteq s_2\end{aligned}$$

This satisfies the premises of [R-VAR], so

$$\Sigma; \Gamma; \Delta \vDash_{(B_2, s_2)} \langle x, \mathsf{env}_{SV}\rangle \to v$$

can be concluded as well.

- If [R-FIELD] was used, then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_{(B_1, s_1)} \langle e.p, \mathsf{env}_{SV}\rangle \to v$$

and from the premise and side condition we know that

$$\begin{aligned}\Sigma; \Gamma; \Delta &\vDash_{(I, s_1)} \langle e, \mathsf{env}_{SV}\rangle \to X\\ \mathsf{env}_S(X)(p) &= v\\ \textsc{TypeOf}_\Gamma(v) &= (B_1', s_1')\\ \Gamma(I)(p) &= \mathsf{var}(B_2', s_2')\\ \Sigma &\vdash B_1' <: B_2' <: B_1\\ s_1' &\sqsubseteq s_2' \sqsubseteq s_1\end{aligned}$$

By assumption, $\Sigma \vdash B_1 <: B_2$ and $s_1 \sqsubseteq s_2$, so by transitivity of the subtyping relation and the ordering relation we have that

$$\begin{aligned}\Sigma &\vdash B_2' <: B_2\\ s_2' &\sqsubseteq s_2\end{aligned}$$

By the induction hypothesis,

$$\Sigma; \Gamma; \Delta \vDash_{(I, s_2)} \langle e, \mathsf{env}_{SV}\rangle \to X$$

can be concluded as well. This satisfies the premises of [R-FIELD], so

$$\Sigma; \Gamma; \Delta \vDash_{(B_2, s_2)} \langle e.p, \mathsf{env}_{SV} \rangle \to v$$

can be concluded as well.

- If [R-OP] was used, then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_{(B_1, s_1)} \langle \mathrm{op}(e_1, \ldots, e_n), \mathsf{env}_{SV} \rangle \to v$$

and from the premise and side condition we know that

$$\begin{array}{l} \Sigma; \Gamma; \Delta \vDash_{(B_1', s_1)} \langle e_1, \mathsf{env}_{SV} \rangle \to v_1 \\ \qquad \vdots \\ \Sigma; \Gamma; \Delta \vDash_{(B_n', s_1)} \langle e_n, \mathsf{env}_{SV} \rangle \to v_n \\ \qquad \mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v \\ \qquad \vdash \mathrm{op} : B_1', \ldots B_n' \to B_1 \end{array}$$

By assumption, $s_1 \sqsubseteq s_2$. Thus, by the induction hypothesis,

$$\begin{array}{l} \Sigma; \Gamma; \Delta \vDash_{(B_1', s_2)} \langle e_1, \mathsf{env}_{SV} \rangle \to v_1 \\ \qquad \vdots \\ \Sigma; \Gamma; \Delta \vDash_{(B_n', s_2)} \langle e_n, \mathsf{env}_{SV} \rangle \to v_n \end{array}$$

can be concluded as well.

Furthermore, by requirement, no operation op can yield any other type of value than bool or int, for which the only form of subtyping, that can be concluded, is by reflexivity (rule [SUB-REFL]). Thus, since $\Sigma \vdash B_1 <: B_2$ by assumption, we know that $B_1 = B_2$. Therefore,

$$\vdash \mathrm{op} : B_1', \ldots B_n' \to B_2$$

also holds. This satisfies the premises of [R-OP], so

$$\Sigma; \Gamma; \Delta \vDash_{(B_2, s_2)} \langle \mathrm{op}(e_1, \ldots, e_n), \mathsf{env}_{SV} \rangle \to v$$

can be concluded as well. □

8.9.3 Proof of Theorem 21

Proof. By induction on the rules of the typed semantics (Figure 8.22).

- Suppose the transition was concluded by [R-VAL]. Then the transition is of the form
$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle v, \mathsf{env}_{SV} \rangle \to v$$
so as no variables or fields are evaluated, the requirement on their security level is vacuously satisfied. From the premise and side condition we have directly that
$$\begin{aligned} \textsc{TypeOf}_\Gamma(v) &= (B_1, s_1) \\ \Sigma &\vdash B_1 <: B \\ s_1 &\sqsubseteq s \end{aligned}$$
as desired.

- Suppose the transition was concluded by [R-VAR]. Then the transition is of the form
$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle x, \mathsf{env}_{SV} \rangle \to v$$
and from the premise and side condition we have that
$$\begin{aligned} \mathsf{env}_V(x) &= v \\ \textsc{TypeOf}_\Gamma(v) &= (B_1, s_1) \\ \Delta(x) &= \mathsf{var}(B_2, s_2) \\ \Sigma &\vdash B_1 <: B_2 <: B \\ s_1 &\sqsubseteq s_2 \sqsubseteq s \end{aligned}$$
We read from a container, which as type $\mathsf{var}(B_2, s_2)$, and $s_2 \sqsubseteq s$ as required. Furthermore, we can conclude $s_1 \sqsubseteq s$ and $\Sigma \vdash B_1 <: B$ as well by transitivity of the two relations.

- Suppose the transition was concluded by [R-FIELD]. Then the transition is of the form
$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e.p, \mathsf{env}_{SV} \rangle \to v$$

and from the premise and side condition we have that

$$\begin{aligned}
\Sigma; \Gamma; \Delta \vDash_{I_s} \langle e, \mathsf{env}_{SV} \rangle \to X \\
\mathsf{env}_S(X)(p) = v \\
\textsc{TypeOf}_\Gamma(v) = (B_1, s_1) \\
\Gamma(I)(p) = \mathsf{var}(B_2, s_2) \\
\Sigma \vdash B_1 <: B_2 <: B \\
s_1 \sqsubseteq s_2 \sqsubseteq s
\end{aligned}$$

By the induction hypothesis, the statement holds for the transition

$$\Sigma; \Gamma; \Delta \vDash_{I_s} \langle e, \mathsf{env}_{SV} \rangle \to X$$

so all containers read from in the evaluation of e are of level s or lower. The field p has type $\mathsf{var}(B_2, s_2)$, and the requirement $s_2 \sqsubseteq s$ holds. Finally, we can conclude $s_1 \sqsubseteq s$ and $\Sigma \vdash B_1 <: B$ as well by transitivity of the two relations.

- Suppose the transition was concluded by [R-OP]. Then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle \mathrm{op}(e_1, \ldots, e_n), \mathsf{env}_{SV} \rangle \to v$$

and from the premise and side condition we have that

$$\begin{aligned}
\Sigma; \Gamma; \Delta \vDash_{(B_1, s)} \langle e_1, \mathsf{env}_{SV} \rangle \to v_1 \\
\vdots \\
\Sigma; \Gamma; \Delta \vDash_{(B_n, s)} \langle e_n, \mathsf{env}_{SV} \rangle \to v_n \\
\mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v \\
\vdash \mathrm{op} : \widetilde{B} \to B
\end{aligned}$$

By n applications of the induction hypothesis, we conclude that the statement holds for the transitions

$$\begin{aligned}
\Sigma; \Gamma; \Delta \vDash_{(B_1, s)} \langle e_1, \mathsf{env}_{SV} \rangle \to v_1 \\
\vdots \\
\Sigma; \Gamma; \Delta \vDash_{(B_n, s)} \langle e_n, \mathsf{env}_{SV} \rangle \to v_n
\end{aligned}$$

so no container of a level higher than s is read from in the evaluations of e_1, ..., e_n. Finally, we know that $\textsc{TypeOf}_\Gamma(v) = (B, s_\perp)$ by the safety requirement for operations (Definition 57), and $s_\perp \sqsubseteq s$ by definition. □

8.9.4 Proof of Theorem 23

Proof. We must show that $\Sigma; \Gamma; \Delta \vdash e : B_s$ implies that $\Sigma; \Gamma; \Delta \vDash e : B_s$. Unfolding the definition, this means that we must show that there exists a typing interpretation $\mathcal{R}^{B_s}_{\Sigma;\Gamma;\Delta}$ containing e. The simplest such typing interpretation is just the singleton set $\{ e \}$, which by Definition 60 is a typing interpretation w.r.t. $\Sigma; \Gamma; \Delta$ and B_s *if* the transition

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV} \rangle \to v$$

can be concluded by the typed semantics for some state env_{SV} built according to Definition 59, and which by Lemma 57 therefore is ensured to be well-typed and consistent w.r.t. $\Sigma; \Gamma; \Delta$.

Furthermore, from $\Sigma; \Gamma; \Delta \vdash e : B_s$, we know that all variable names, addresses and field names in e must exist in Γ and Δ, so by the construction of env_{SV} we therefore also know that

- $\mathcal{F}_A(e) \subseteq \mathrm{dom}\,(\mathsf{env}_S)$, and
- $\mathcal{F}_V(e) \subseteq \mathrm{dom}\,(\mathsf{env}_V)$.

The statement to be shown then simplifies to

$$\Sigma; \Gamma; \Delta \vdash e : B_s \implies \Sigma; \Gamma; \Delta \vDash_{B_s} \langle e, \mathsf{env}_{SV} \rangle \to v$$

We show this by induction on the type rules (Figure 8.15):

- Suppose the type judgment was concluded by [T-VAL]. Then the conclusion is of the form

  $$\Sigma; \Gamma; \Delta \vdash v : B_s$$

  and from the premise and the side condition we know that

  $$\begin{gathered} \textsc{TypeOf}_\Gamma(v) = (B', s') \\ s' \sqsubseteq s \\ \Sigma \vdash B' <: B \end{gathered}$$

  This satisfies the side conditions of [R-VAL], so the transition

  $$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle v, \mathsf{env}_{SV} \rangle \to v$$

  can be concluded by [R-VAL].

- Suppose the type judgment was concluded by [T-VAR]. Then the conclusion is of the form

$$\Gamma; \Delta \vdash x : B_s$$

and from the premise and side condition we know that

$$\begin{gathered}\Delta(x) = \mathsf{var}(B_2, s_2) \\ s_2 \sqsubseteq s \\ \Sigma \vdash B_2 <: B\end{gathered}$$

By assumption, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_V$, which was concluded by [T-ENVV$_u$]. We know $x \in \mathrm{dom}(\mathsf{env}_V)$ so $\mathsf{env}_V(x) = v$ by the assumption $\mathcal{F}_V(e) \subseteq \mathrm{dom}(\mathsf{env}_V)$. Therefore, let $\mathsf{env}_V = \mathsf{env}'_V, (x, v)$.

From the premise and side condition of the aforementioned rule, we then know that

$$\begin{gathered}\Sigma; \Gamma; \Delta \vdash v : (B_2, s_2) \\ \Delta(x) = \mathsf{var}(B_2, s_2)\end{gathered}$$

Now, $\Gamma; \Delta \vdash v : (B_2, s_2)$ must have been concluded by [T-VAL], and from its side condition we know that

$$\begin{gathered}\textsc{TypeOf}_\Gamma(v) = (B_1, s_1) \\ s_1 \sqsubseteq s_2 \\ \Sigma \vdash B_1 <: B_2\end{gathered}$$

This satisfies all the side conditions of [R-VAR], so the transition

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle x, \mathsf{env}_{SV} \rangle \to v$$

can be concluded by [R-VAR].

- Suppose the type judgment was concluded by [T-FIELD]. Then the conclusion is of the form

$$\Sigma; \Gamma; \Delta \vdash e.p : B_s$$

and from the premise and side condition we know that

$$\begin{gathered}\Gamma; \Delta \vdash e : I_s \\ \Gamma(I)(p) = \mathsf{var}(B_2, s_2) \\ s_2 \sqsubseteq s \\ \Sigma \vdash B_2 <: B\end{gathered}$$

From $\Sigma; \Gamma; \Delta \vdash e : I_s$ we instantiate the induction hypothesis and obtain that

$$\Sigma; \Gamma; \Delta \vdash e : I_s \implies \Gamma; \Delta \vDash_{I_s} \langle e, \mathsf{env}_{SV} \rangle \to X$$

for some address X, since only addresses are given a type I by the definition of $\textsc{TypeOf}_\Gamma(\cdot)$ (Definition 54).

By assumption, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S$ holds, which was concluded by [T-ENVS]. We know that $X \in \mathrm{dom}(\mathsf{env}_S)$, so $\mathsf{env}_S(X) = \mathsf{env}_F$ by $\mathcal{F}_A(e) \subseteq \mathrm{dom}(\mathsf{env}_S)$ and consistency. Therefore, let $\mathsf{env}_S = \mathsf{env}'_S, (X, \mathsf{env}_F)$.

From the premise and side condition of this rule, we know that $\Sigma; \Gamma; \Delta \vdash_X \mathsf{env}_F$, which must have been concluded by [T-ENVF]. We know that $p \in \mathrm{dom}(\mathsf{env}_F)$, so $\mathsf{env}_F(p) = v$ by consistency and well-typedness of env_{SV}. This establishes that $\mathsf{env}_S(X)(p) = v$.

Now, let $\mathsf{env}_F = \mathsf{env}'_F, (p, v)$. From the premise and side condition of [T-FIELD] we know that

$$\begin{gathered} \Sigma; \Gamma; \Delta \vdash v : (B_1, s_1) \\ \Gamma(X) = (I, s)^{15} \\ \Gamma(I)(p) = \mathsf{var}(B_2, s_2) \end{gathered}$$

Now, $\Sigma; \Gamma; \Delta \vdash v : (B_1, s_1)$ must have been concluded by [T-VAL], and from its side condition we know that

$$\begin{gathered} \textsc{TypeOf}_\Gamma(v) = (B_1, s_1) \\ s_1 \sqsubseteq s_2 \\ \Sigma \vdash B_1 <: B_2 \end{gathered}$$

This satisfies all the side conditions of [R-FIELD], so the transition

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle e.p, \mathsf{env}_{SV} \rangle \to v$$

can be concluded by [R-FIELD].

- Suppose the type judgment was concluded by [T-OP]. Then the conclusion is of the form

$$\Sigma; \Gamma; \Delta \vdash \mathrm{op}(\tilde{e}) : B_s$$

[15] Or, in principle $\Gamma(X) = (I', s')$, but by the coercion property (Theorem 20) we then have that $\Sigma \vdash I' <: I$ and $s' \sqsubseteq s$, so we omit this extra complication.

and from the premise we know that

$$\vdash \mathrm{op} : \widetilde{B} \to B$$
$$\Sigma; \Gamma; \Delta \vdash \tilde{e} : \widetilde{B}_{\tilde{s}}$$

For each e_i in $\tilde{e}$ and (B_i, s) we instantiate the induction hypothesis and obtain that

$$\Sigma; \Gamma; \Delta \vdash e_i : (B_i, s) \implies \Sigma; \Gamma; \Delta \vDash_{(B_i,s)} \langle e_i, \mathsf{env}_{SV} \rangle \to v_i$$

Now, by our safety requirement for operations (Definition 57), we know that it must be the case that $\mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v$ and $\textsc{TypeOf}_\Gamma(v) = (B, s_\perp)$. Furthermore, $s_\perp \sqsubseteq s$ holds by definition of $s_\perp$. Thus, this satisfies all the premises and side conditions of [R-OP], so the transition

$$\Sigma; \Gamma; \Delta \vDash_{B_s} \langle \mathrm{op}(\tilde{e}), \mathsf{env}_{SV} \rangle \to v$$

can be concluded by [R-OP]. □

8.9.5 Proof of Theorem 27

Proof. We first show a simpler statement, namely that the terminal type environment and security level are preserved by a single transition step: Thus, assume $\textsc{fin}(Q, \Delta, s) = (\Delta', s')$. If

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta'' \vDash_{s''} \left\langle Q'', \mathsf{env}_T, \mathsf{env}''_{SV} \right\rangle$$

then $\textsc{fin}(Q'', \Delta'', s'') = (\Delta', s')$.

We show this by case analysis of the semantic rules used to conclude the transition:

- The cases for [R-SKIP], [R-WHILE$_\mathsf{F}$], [R-ASSV], [R-ASSF] are immediately seen to hold, since neither Δ nor s are modified.
- The cases for [R-IF] and [R-WHILE$_\mathsf{T}$] are similar, so we shall only consider the case for [R-IF]. We have that

  $$\textsc{fin}(\texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; Q', \Delta, s) = \textsc{fin}(Q', \Delta, s)$$

  by the clauses of Definition 62. After the transition, the reduct is of the following form:

  $$\Sigma; \Gamma; \Delta \vDash_{s''} \langle S_b; s; Q', \mathsf{env}_{TSV} \rangle$$

  and

  $$\textsc{fin}(S; s; Q', \Delta, s'') = \textsc{fin}(s; Q', \Delta, s'') = \textsc{fin}(Q', \Delta, s)$$

  by the clauses of Definition 62.

- If [R-DECV] was used, then we have that

$$\textsc{fin}(\mathsf{var}(B_{s1})\ \ x\ \ \texttt{:=}\ \ e\ \ \texttt{in}\ \ S; Q', \Delta, s) = \textsc{fin}(Q', \Delta, s)$$

  and after the transition, the reduct is of the form

$$\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s1}) \vDash_s \langle S; \mathsf{del}(x); Q', \mathsf{env}_{TS}; \mathsf{env}_V, x : v \rangle$$

  and by the clauses of Definition 62 we then have that

$$\begin{aligned} &\textsc{fin}(S; \mathsf{del}(x); Q', (\Delta, x : \mathsf{var}(B_{s1})), s) \\ &= \textsc{fin}(\mathsf{del}(x); Q', (\Delta, x : \mathsf{var}(B_{s1})), s) \\ &= \textsc{fin}(Q', \Delta, s) \end{aligned}$$

- The cases for [R-CALL], [R-DCALL] and [R-FCALL] are similar, since the reduct has the same form in all three cases. We shall only consider the case for [R-CALL]. We have that

$$\textsc{fin}(\texttt{call}\ \ e_1 . m(\tilde{e}) \$ e_2; Q', \Delta, s) = \textsc{fin}(Q', \Delta, s)$$

  and after the transition, the reduct is of the form

$$\Sigma; \Gamma; \Delta'' \vDash_{s''} \left\langle S; (\mathsf{env}_V, \Delta); s; Q', \mathsf{env}_T; \mathsf{env}'_{SV} \right\rangle$$

  and by the clauses of Definition 62 we then have that

$$\begin{aligned} &\textsc{fin}(S; (\mathsf{env}_V, \Delta); s; Q', \Delta'', s'') \\ &= \textsc{fin}((\mathsf{env}_V, \Delta); s; Q', \Delta'', s'') \\ &= \textsc{fin}(s; Q', \Delta, s'') = \textsc{fin}(Q', \Delta, s) \end{aligned}$$

- If [R-DELV] was used, then we have that

$$\textsc{fin}(\mathsf{del}(x); Q', (\Delta, x : \mathsf{var}(B_{s1})), s) = \textsc{fin}(Q', \Delta, s)$$

  and after the transition, the reduct is of the form

$$\Sigma; \Gamma; \Delta \vDash_s \langle Q', \mathsf{env}_{TSV} \rangle$$

  and $\textsc{fin}(Q', \Delta, s) = (\Delta', s')$ by assumption.

- If [R-RETURN] was used, then we have that

$$\textsc{fin}((\mathsf{env}''_V, \Delta''); Q', \Delta, s) = (Q', \Delta'', s)$$

  and after the transition, the reduct is of the form

$$\Sigma; \Gamma; \Delta'' \vDash_s \langle Q', \mathsf{env}_{TS}; \mathsf{env}''_V \rangle$$

  and $\textsc{fin}(Q', \Delta'', s) = (\Delta', s')$ by assumption.

- If [R-RESTORE] was used, then we have that

$$\textsc{fin}(s''; Q', \Delta, s) = \textsc{fin}(Q', \Delta, s'')$$

  and after the transition, the reduct is of the form

$$\Sigma; \Gamma; \Delta \vDash_{s''} \langle Q, \mathsf{env}_{TSV} \rangle$$

  and $\textsc{fin}(Q', \Delta, s'') = (\Delta', s')$ by assumption.

In all cases we obtain the same result by $\textsc{fin}(\cdot)$ from the redex and the reduct. Finally, we generalise to the case of $\rightarrow^*$ by induction on the length of the transition sequence with repeated application of the statement shown above. □

8.9.6 Auxiliary lemmas for the proof of Theorem 28

Lemma 76 (Weakening of Δ). *If* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_V$, *and* $x \notin \mathrm{dom}(\mathsf{env}_V)$ *then* $\Sigma; \Gamma; \Delta, x : T \vdash \mathsf{env}_V$.

Lemma 77 (Strengthening of Δ). *If* $\Sigma; \Gamma; \Delta, x : T \vdash \mathsf{env}_V$, *and* $x \notin \mathrm{dom}(\mathsf{env}_V)$ *then* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_V$.

Lemma 78 (Substitution in env_V). *If*

- $\Gamma \vdash \mathsf{env}_{SV}$ *(consistency), and*
- $\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_s) \vdash \mathsf{env}_V$ *(well-typedness), and*
- $x \in \mathrm{dom}(\mathsf{env}_V)$, *and*
- $\Sigma; \Gamma; \varnothing \vdash v : B_s$ *and* $v \in \mathsf{ANames} \implies v \in \mathrm{dom}(\mathsf{env}_S)$,

then $\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_s) \vdash \mathsf{env}'_V$ *and* $\Gamma \vdash \mathsf{env}_S; \mathsf{env}'_V$, *where* $\mathsf{env}'_V = \mathsf{env}_V[x \mapsto v]$.

Lemma 79 (Substitution in env_S)**.** *If*

- $\Gamma; \mathsf{env}_S \vdash \mathsf{env}_S$ *(consistency), and*
- $\Sigma; \Gamma; \varnothing \vdash \mathsf{env}_S$ *(well-typedness), and*
- $X \in \mathrm{dom}(\mathsf{env}_S)$*, and*
- $\Gamma(X) = (I, s'')$*, and*
- $\Gamma(I)(p) = \mathsf{var}(B_s)$*, and*
- $\Sigma; \Gamma; \varnothing \vdash v : (B_s)$ *and* $v \in \mathsf{ANames} \implies v \in \mathrm{dom}(\mathsf{env}_S)$*,*

then $\Sigma; \Gamma; \varnothing \vdash \mathsf{env}'_S$*, and* $\Gamma; \mathsf{env}'_S \vdash \mathsf{env}'_S$*, where* $\mathsf{env}'_S = \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]$.

Lemma 80 (Properties of well-formedness)**.**

- *If* $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$ *then* $s \sqsupseteq \textsc{fst}(Q)$.
- *If* $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$ *and* $s' \sqsupseteq s$ *then* $\Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash s; Q$.
- *If* $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$ *and* $x \in \mathrm{dom}(\mathsf{env}_V)$ *then* $\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V[x \mapsto v]; s \vdash Q$ *for any* v.
- *If* $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$ *and* $X \in \mathrm{dom}(\mathsf{env}_S)$ *and* $p \in \mathrm{dom}(\mathsf{env}_S(X))$ *then* $\Sigma; \Gamma; \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]; \mathsf{env}_V; s \vdash Q$ *for any* v.

The lemmas are shown by induction on the type rules (Figure 8.18) and consistency rules (Figure 8.21) and well-formedness rules (Figure 8.27). In the substitution lemmas (Lemmas 78–79) we incorporate consistency in addition to well-typedness. They could equally well be shown separately, but since we shall always require both properties to hold for the env_{SV} we shall consider, we prefer to combine them to avoid repeating essentially the same premises.

8.9.7 Proof of Theorem 28

Proof. By case analysis of the rules used to conclude the transition (Figures 8.24–8.26).

If [R-SKIP] was used, then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_s \langle \mathtt{skip}; Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle$$

and the properties are immediately seen to hold as follows:

- Well-typedness: $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$ holds by assumption.
- Consistency: $\Gamma \vdash \mathsf{env}_{SV}$ holds by assumption.

- Well-formedness: $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$ follows from the assumption

$$\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash \texttt{skip}; Q$$

  and rule [WF-STM].

- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal, so they agree on *all* fields, regardless of their security level.

This concludes the case for [R-SKIP].

If [R-IF] was used, then the transition is of the form

$$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; Q, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_{s'} \langle S_b; s; Q, \mathsf{env}_{TSV} \rangle \end{aligned}$$

and from the premise and side condition we have that

$$\begin{aligned} s &\sqsubseteq s' \\ \Sigma; \Gamma; \Delta &\vDash_{(\mathsf{bool}, s')} \langle e, \mathsf{env}_{SV} \rangle \to b \in \mathbb{B}. \end{aligned}$$

The properties are now seen to hold as follows:

- Well-typedness: $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$ holds by assumption.
- Consistency: $\Gamma \vdash \mathsf{env}_{SV}$ holds by assumption.
- Well-formedness: $\Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash S_b; s; Q$:
  By assumption

$$\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash \texttt{if } e \texttt{ then } S_\mathsf{T} \texttt{ else } S_\mathsf{F}; Q$$

  holds. This was concluded by [WF-STM], and from the premise we have that $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$ holds. From $s' \sqsupseteq s$ and Lemma 80 we then have that $\Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash s; Q$ holds as well. Then $\Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash S_b; s; Q$ can be concluded by [WF-STM].

- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal.

This concludes the case for [R-IF].

If [R-WHILE$_\mathsf{T}$] was used, then the transition is of the form

$$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{while } e \texttt{ do } S; Q, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_{s'} \langle S; s; \texttt{while } e \texttt{ do } S; Q, \mathsf{env}_{TSV} \rangle \end{aligned}$$

and from the side condition we have that $s \sqsubseteq s'$. The properties are now seen to hold as follows:

- Well-typedness: $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$ holds by assumption.
- Consistency: $\Gamma \vdash \mathsf{env}_{SV}$ holds by assumption.
- Well-formedness: $\Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash S; s; \texttt{while}\ e\ \texttt{do}\ S; Q$:
  By assumption
  $$\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash \texttt{while}\ e\ \texttt{do}\ S; Q$$
  holds. From $s' \sqsupseteq s$ and Lemma 80 we then have that
  $$\Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash s; \texttt{while}\ e\ \texttt{do}\ S; Q$$
  holds as well. Then $\Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash S; s; \texttt{while}\ e\ \texttt{do}\ S; Q$ can be concluded by [WF-STM].
- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal.

This concludes the case for [R-WHILE$_\mathrm{T}$].

If [R-WHILE$_\mathrm{F}$] was used, then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{while}\ e\ \texttt{do}\ S; Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle$$

and the properties are now seen to hold as follows:

- Well-typedness: $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$ holds by assumption.
- Consistency: $\Gamma \vdash \mathsf{env}_{SV}$ holds by assumption.
- Well-formedness: $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$:
  By assumption
  $$\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash \texttt{while}\ e\ \texttt{do}\ S; Q$$
  holds. This was concluded by [WF-STM], and from the premise we have that $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$ holds as well.
- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal.

This concludes the case for [R-WHILE$_F$].

If [R-DECV] was used, then the transition is of the form

$$\begin{aligned}&\Sigma; \Gamma; \Delta \vDash_s \langle \mathsf{var}(B, s')\ \ x\ \ \texttt{:=}\ \ e\ \ \texttt{in}\ \ S; Q, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta, x : \mathsf{var}(B, s') \vDash_s \langle S; \mathsf{del}(x); Q, \mathsf{env}_{TS}; \mathsf{env}_V, x : v \rangle\end{aligned}$$

and from the premise and side condition we know that

$$\begin{gathered}x \notin \mathrm{dom}\,(\mathsf{env}_V) \\ x \notin \mathrm{dom}\,(\Delta) \\ \Sigma; \Gamma; \Delta \vDash_{(B,s')} \langle e, \mathsf{env}_{SV} \rangle \to v.\end{gathered}$$

We now show how each of the properties hold, in turn:

- Well-typedness: The goal is to show $\Sigma; \Gamma; \Delta, x : \mathsf{var}(B, s') \vdash \mathsf{env}_S; \mathsf{env}_V, x : v$.

  We know $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S$ by assumption. From the premise and side condition of [R-DECV] we can then conclude the following:

  $$\begin{aligned}\Sigma; \Gamma; \Delta \vdash v : (B, s') &\qquad \text{by Theorem 21,} \\ \Sigma; \Gamma; \Delta, x : \mathsf{var}(B, s') \vdash \mathsf{env}_V &\qquad \text{by Lemma 76,} \\ \Sigma; \Gamma; \Delta, x : \mathsf{var}(B, s') \vdash \mathsf{env}_V, x : v &\qquad \text{by rule [T-ENVV}_u\text{],} \\ \Sigma; \Gamma; \Delta, x : \mathsf{var}(B, s') \vdash \mathsf{env}_S; \mathsf{env}_V, x : v &\qquad \text{by rule [T-ENVSV],}\end{aligned}$$

  where the final inference is the desired result.

- Consistency: The goal is to show $\Gamma \vdash \mathsf{env}_S; \mathsf{env}_V, x : v$.

  We know $\Gamma \vdash \mathsf{env}_{SV}$ by assumption. From the premise and side condition of [R-DECV] we can then conclude the following:

  $$\begin{aligned}\Sigma; \Gamma; \Delta \vdash v : (B, s') &\qquad \text{by Theorem 21.} \\ v \in \mathsf{ANames} \implies v \in \mathrm{dom}\,(\mathsf{env}_S) &\qquad \text{by Corollary 13.} \\ \mathsf{env}_S \vdash \mathsf{env}_V, x : v &\qquad \text{by rule [C-ENVV}_2\text{].} \\ \Gamma \vdash \mathsf{env}_S; \mathsf{env}_V, x : v &\qquad \text{by rule [C-ENVSV].}\end{aligned}$$

  where the final inference is the desired result.

- Well-formedness: The goal is to show $\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V, x : v; s \vdash S; \mathsf{del}(x); Q$.

  By assumption, we know that

  $$\Sigma; \Gamma; \mathsf{env}_{SV}, s \vdash \mathsf{var}(B, s')\ \ x\ \ \texttt{:=}\ \ e\ \ \texttt{in}\ \ S; Q$$

which was concluded by by [WF-STM]. From the premise of that rule we have that $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$ holds. We can then conclude as follows:

$$\begin{aligned} &\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V, x : v; s \vdash \mathsf{del}(x); Q && \text{by rule [WF-DEL].} \\ &\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V, x : v; s \vdash S; \mathsf{del}(x); Q && \text{by rule [WF-STM].} \end{aligned}$$

where the final inference is the desired result.

- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal.

This concludes the case for [R-DECV].

If [R-ASSV] was used, then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_s \langle x := e; Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TS}; \mathsf{env}_V[x \mapsto v] \rangle$$

and from the premise and side condition of [R-ASSV] we know that

$$\begin{aligned} x &\in \mathrm{dom}\,(\mathsf{env}_V) \\ \Delta(x) &= \mathsf{var}(B, s') \\ s &\sqsubseteq s' \\ \Sigma; \Gamma; \Delta &\vDash_{(B,s')} \langle e, \mathsf{env}_{SV} \rangle \to v. \end{aligned}$$

We now show how each of the properties hold, in turn:

- Well-typedness and consistency: The goal is to show
  - $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S; \mathsf{env}_V[x \mapsto v]$,
  - $\Gamma \vdash \mathsf{env}_S; \mathsf{env}_V[x \mapsto v]$.

  We have the following:

$$\begin{aligned} &\Sigma; \Gamma; \varnothing \vdash v : (B, s') && \text{from premises of [R-ASSV] and Thm. 21,} \\ &v \in \mathsf{ANames} \implies v \in \mathrm{dom}\,(\mathsf{env}_S) && \text{by Corollary 13.} \end{aligned}$$

  By Lemma 78 we can then conclude

  - Well-typedness: $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S; \mathsf{env}_V[x \mapsto v]$.
  - Consistency: $\Gamma \vdash \mathsf{env}_S; \mathsf{env}_V[x \mapsto v]$.

- Well-formedness: The goal is to show $\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V[x \mapsto v]; s \vdash Q$. We know that
$$\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V; s \vdash x \ := \ e; Q$$
which was concluded by [WF-STM], and from the premise of this rule we have that $\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V; s \vdash Q$. Then $\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V[x \mapsto v]; s \vdash Q$ can be concluded by Lemma 80.
- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal.

This concludes the case for [R-ASSV].

If [R-ASSF] was used, then the transition is of the form

$$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{this}\,.\,p \ := \ e; Q, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]; \mathsf{env}_V \rangle \end{aligned}$$

and from the premises and side condition of [R-ASSF] we know that

$$\begin{aligned} X &= \mathsf{env}_V(\texttt{this}) \\ \mathsf{env}_F &= \mathsf{env}_S(X) \\ p &\in \mathrm{dom}\,(\mathsf{env}_F) \\ \mathsf{var}(I, s_1) &= \Delta(\texttt{this}) \\ \mathsf{var}(B, s') &= \Gamma(I)(p) \\ \Sigma; \Gamma; \Delta &\vDash_{(B,s')} \langle e, \mathsf{env}_{SV} \rangle \to v \\ s_1 &\sqsubseteq s' \\ s &\sqsubseteq s'. \end{aligned}$$

We now show how each of the properties hold, in turn:

- Well-typedness and consistency: The goals are to show
  - $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]; \mathsf{env}_V$.
  - $\Gamma \vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]; \mathsf{env}_V$.

  From $\mathsf{env}_V(\texttt{this}) = X$ and consistency of env_{SV} we infer that $X \in \mathrm{dom}\,(\mathsf{env}_S)$. From the premises and Theorem 21 we know that $\Sigma; \Gamma; \varnothing \vdash v : (B, s')$, and by Corollary 13 that $v \in \mathsf{ANames} \implies v \in \mathrm{dom}\,(\mathsf{env}_S)$. By Lemma 79 we can then conclude
  - Well-typedness: $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]; \mathsf{env}_V$, and

  - Consistency: $\Gamma \vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]$.

- Well-formedness: The goal is to show

$$\Sigma; \Gamma; \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]; \mathsf{env}_V; s \vdash Q.$$

  We know that

$$\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V; s \vdash \texttt{this}.p \texttt{ := } e; Q$$

  which was concluded by [WF-STM], and from the premise of this rule we have that $\Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V; s \vdash Q$. Then

$$\Sigma; \Gamma; \mathsf{env}_S\mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]; \mathsf{env}_V; s \vdash Q$$

  can be concluded by Lemma 80.

- Equivalence: The goal is to show

$$\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]\,.$$

  The only difference between the two environments is in the value stored at p, and from the premise and side conditions we know that $\Gamma(I)(p) = \mathsf{var}(B, s')$ and $s \sqsubseteq s'$. Hence, the field p is not amongst those for which the equivalence must hold.

This concludes the case for [R-ASSF].

Suppose one of [R-CALL], [R-DCALL], [R-FCALL] was used. We consider only the case for [R-CALL], since the two other cases are similar or simpler. If [R-CALL] was used, then the transition is of the form

$$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_s \langle \texttt{call } e_1.m(\tilde{e})\$e_2; Q, \mathsf{env}_{TSV} \rangle \\ \to\ &\Sigma; \Gamma; \Delta' \vDash_{s'} \langle S; (\mathsf{env}_V, \Delta); s; Q, \mathsf{env}_T; \mathsf{env}'_{SV} \rangle \end{aligned}$$

where S is the method body; env'_S is modified because of the update of the `balance` fields; and env'_V is the new variable environment containing bindings for the formal parameters of the method. As there are many side conditions, we shall only mention those that are relevant for each property below. We now show how each of the properties hold, in turn:

- Well-typedness and consistency: The goal is to show:

$$\begin{aligned} \Sigma; \Gamma; \Delta' &\vdash \mathsf{env}'_{SV} \\ \Gamma &\vdash \mathsf{env}'_{SV} \end{aligned}$$

  We treat each environment separately:

– From the premise of [R-CALL], we have that

$$\Sigma; \Gamma; \Delta \vDash_{(\mathsf{int}, s')} \langle e_2, \mathsf{env}_{SV} \rangle \to n$$

and by Theorem 21, we know that n is an integer, and all variables read in the evaluation are of level s and lower. From the side condition $s' \sqsubseteq s_3, s_4$, we have that the levels of the two `balance` fields, s_3, s_4 are higher than, or equal to n. From another side condition, we have that

$$\mathsf{env}'_S = \mathsf{env}_S\big[X \mapsto \mathsf{env}_F^X[\texttt{balance -= } n]\big]\big[Y \mapsto \mathsf{env}_F^Y[\texttt{balance += } n]\big]$$

where X is obtained from `this`, and Y is obtained from e_1. This update is thus equivalent to executing the statements

$$\begin{aligned} \texttt{this.balance := this.balance - } e_2 \\ e_1\texttt{.balance := } e_1\texttt{.balance + } e_2 \end{aligned}$$

at level s'. We see that the transition in both cases can be concluded by rule [R-ASSF]. Hence, well-typedness and consistency of env'_S holds by the case for [R-ASSF] above.

– From the side condition $\mathsf{env}_V(\texttt{this}) = X$, and the (expanded) premise

$$\begin{aligned} &\Sigma; \Gamma; \Delta \vDash_{(I^Y, s')} \langle e_1, \mathsf{env}_{SV} \rangle \to Y \\ &\Sigma; \Gamma; \Delta \vDash_{(\mathsf{int}, s')} \langle e_2, \mathsf{env}_{SV} \rangle \to n \\ &\Sigma; \Gamma; \Delta \vDash_{(B_1, s'_1)} \langle e'_1, \mathsf{env}_{SV} \rangle \to v_1 \\ &\qquad\vdots \\ &\Sigma; \Gamma; \Delta \vDash_{(B_k, s'_k)} \langle e'_k, \mathsf{env}_{SV} \rangle \to v_k \end{aligned}$$

and Theorem 21, we know that the bindings of the formal parameters in the new env'_V, and the type bindings in the new Δ' will correspond, as we in the side condition have

$$\begin{aligned} \mathsf{env}'_V &= \texttt{this} : Y, \texttt{sender} : X, \texttt{value} : n, x_1 : v_1, \dots, x_k : v_k \\ \Delta' &= \texttt{this} : \mathsf{var}(I^Y, s_2), \texttt{sender} : \mathsf{var}(I^X, s_1), \texttt{value} : \mathsf{var}(\mathsf{int}, s'), \\ &\qquad x_1 : \mathsf{var}(B_1, s'_1), \dots, x_k : \mathsf{var}(B_k, s'_k) \end{aligned}$$

and the types set of `this` and `sender` are obtained from the actual types of X and Y in Γ. Hence $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_V$ (well-typedness) holds.

Furthermore, we know directly from the side conditions that X and Y, which appear as values in env_V, have types in Γ, so by the assumption

of consistency of env_S, we also know that X and Y must appear in the domain of env_S, and therefore also in the domain of env'_S, since the domain is static. Thus, consistency for these entries holds.

For the values $v_1, \ldots, v_k$, we use Corollary 13 to conclude that

$$v_i \in \mathsf{ANames} \implies v_i \in \mathrm{dom}\,(\mathsf{env}_S)$$

which then again implies $v_i \in \mathrm{dom}\,(\mathsf{env}'_S)$, since the domain is static. Hence, $\mathsf{env}'_S \vdash \mathsf{env}'_V$ (consistency) holds as well.

Combining the four results above gives us that

$$\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_{SV}$$
$$\Gamma \vdash \mathsf{env}'_{SV}$$

both hold, as required.

- Well-formedness: The goal is to show $\Sigma; \Gamma; \mathsf{env}'_{SV}; s' \vdash S; (\mathsf{env}_V, \Delta); s; Q$. By assumption,

$$\Sigma; \Gamma; \mathsf{env}_{SV}, s \vdash \texttt{call}\ e_1.m(\vec{e})\$e_2; Q$$

holds, which was concluded by [WF-STM]. Let

$$\mathsf{env}'_S = \mathsf{env}_S[X \mapsto \mathsf{env}_F^X[\texttt{balance -= } n]][Y \mapsto \mathsf{env}_F^Y[\texttt{balance += } n]]$$

for ease of notation. We know that $s \sqsubseteq s'$ from the side condition of [R-CALL]. We then conclude the following:

$$\begin{aligned} \Sigma; \Gamma; \mathsf{env}_{SV}, s &\vdash Q && \text{from premise of [WF-STM]},\\ \Sigma; \Gamma; \mathsf{env}'_S; \mathsf{env}_V, s &\vdash Q && \text{by Lemma 80},\\ \Sigma; \Gamma; \mathsf{env}'_S; \mathsf{env}_V, s' &\vdash s; Q && \text{by rule [WF-SEC]}. \end{aligned}$$

We also know the following:

$$\begin{aligned} \Sigma; \Gamma; \Delta &\vdash \mathsf{env}_V && \text{by assumption}\\ \mathsf{env}'_S &\vdash \mathsf{env}_V && \text{by assumption, since } \mathrm{dom}\,(\mathsf{env}_S) = \mathrm{dom}\,(\mathsf{env}'_S). \end{aligned}$$

Using this, and the above, we can then conclude

$$\Sigma; \Gamma; \mathsf{env}'_{SV}; s' \vdash S; (\mathsf{env}_V, \Delta); s; Q$$

by [WF-RET], as required.

- Equality: The goal is to show $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}'_S$.

  We know that the only difference between env_S and env'_S is that the fields $X\,.\,\texttt{balance}$ and $Y\,.\,\texttt{balance}$ may have been modified. From the side conditions we have that

  $$\begin{aligned} \mathsf{var}(\mathsf{int}, s_3) &= \Gamma(I^X)(\texttt{balance}) \\ \mathsf{var}(\mathsf{int}, s_4) &= \Gamma(I^Y)(\texttt{balance}) \\ s &\sqsubseteq s' \sqsubseteq s_3, s_4 \end{aligned}$$

  so by transitivity $s \sqsubseteq s_3, s_4$. As neither field is strictly lower than, or unrelated to, s, we therefore have that $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}'_S$ holds.

This concludes the case for [R-CALL], [R-DCALL] and [R-FCALL].

If [R-DELV] was used, then the transition is of the form

$$\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s'}) \vDash_s \langle \mathsf{del}(x); Q, \mathsf{env}_{TS}; \mathsf{env}_V, x : v \rangle \to \Sigma; \Gamma; \Delta \vDash_s \langle Q, \mathsf{env}_{TSV} \rangle$$

We now show how each of the properties hold, in turn:

- Well-typedness: The goal is to show $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$. We have that

  $$\begin{aligned} \Sigma; \Gamma; \varnothing &\vdash \mathsf{env}_S && \text{by assumption,} \\ \Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s'}) &\vdash \mathsf{env}_V, x : v && \text{by assumption,} \\ \Sigma; \Gamma; \Delta, x : \mathsf{var}(B_{s'}) &\vdash \mathsf{env}_V && \text{from premise of } [\text{T-ENVV}_u], \\ \Sigma; \Gamma; \Delta &\vdash \mathsf{env}_V && \text{by Lemma 77,} \\ \Sigma; \Gamma; \Delta &\vdash \mathsf{env}_{SV} && \text{by } [\text{T-ENVSV}]. \end{aligned}$$

  The final inference is the desired result.

- Consistency: The goal is to show $\Gamma \vdash \mathsf{env}_{SV}$.

  $$\begin{aligned} \Gamma; \mathsf{env}_S &\vdash \mathsf{env}_S && \text{by assumption.} \\ \Gamma; \mathsf{env}_S &\vdash \mathsf{env}_V, x : v && \text{by assumption.} \\ \Gamma; \mathsf{env}_S &\vdash \mathsf{env}_V && \text{from premise of } [\text{C-ENVV}_2]. \\ \Gamma &\vdash \mathsf{env}_{SV} && \text{by } [\text{C-ENVSV}]. \end{aligned}$$

  The final inference is the desired result.

- Well-formedness: The goal is to show $\Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash Q$.

$$\begin{array}{rr} \Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V, x : vs \vdash \mathsf{del}(x); Q & \text{by assumption.} \\ \Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}_V, s \vdash Q & \text{from premise of [WF-DEL].} \end{array}$$

  The final inference is the desired result.

- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal.

This concludes the case for [R-DELV].

If [R-RETURN] was used, then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_s \langle (\mathsf{env}'_V, \Delta'); Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta' \vDash_s \langle Q, \mathsf{env}_{TS}; \mathsf{env}'_V \rangle .$$

We now show how each of the properties hold, in turn:

- Well-typedness: The goal is to show $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S; \mathsf{env}'_V$. We have that

$$\begin{array}{rr} \Sigma; \Gamma; \varnothing \vdash \mathsf{env}_S & \text{by assumption,} \\ \Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash (\mathsf{env}'_V, \Delta'); Q & \text{by assumption,} \\ \Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_V & \text{from premise of [WF-RET],} \\ \Sigma; \Gamma; \Delta' \vdash \mathsf{env}_S; \mathsf{env}'_V & \text{by [T-ENVSV].} \end{array}$$

  The final inference is the desired result.

- Consistency: The goal is to show $\Gamma \vdash \mathsf{env}_{SV}$. We have that

$$\begin{array}{rr} \Gamma; \mathsf{env}_S \vdash \mathsf{env}_S & \text{by assumption,} \\ \Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash (\mathsf{env}'_V, \Delta'); Q & \text{by assumption,} \\ \Gamma; \mathsf{env}_S \vdash \mathsf{env}'_V & \text{from premise of [WF-RET],} \\ \Gamma \vdash \mathsf{env}_S; \mathsf{env}'_V & \text{by [C-ENVSV].} \end{array}$$

  The final inference is the desired result.

- Well-formedness: The goal is to show $\Sigma; \Gamma; \mathsf{env}_S; s \vdash Q$. We have that

$$\begin{array}{rr} \Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash (\mathsf{env}'_V, \Delta'); Q & \text{by assumption.} \\ \Sigma; \Gamma; \mathsf{env}_S; \mathsf{env}'_V; s \vdash Q & \text{from premise of [WF-RET].} \end{array}$$

  The final inference is the desired result.

- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal.

This concludes the case for [R-RETURN].

If [R-RESTORE] was used, then the transition is of the form

$$\Sigma; \Gamma; \Delta \vDash_s \langle s'; Q, \mathsf{env}_{TSV} \rangle \to \Sigma; \Gamma; \Delta \vDash_{s'} \langle Q, \mathsf{env}_{TSV} \rangle$$

and we know from the side condition that $s \sqsupseteq s'$. We now show how each of the properties hold, in turn:

- Well-typedness: $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$ holds by assumption.
- Consistency: $\Gamma \vdash \mathsf{env}_{SV}$ holds by assumption.
- Well-formedness: The goal is to show $\Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash Q$. We have that

$$\begin{array}{rr} \Sigma; \Gamma; \mathsf{env}_{SV}; s \vdash s'; Q & \text{by assumption,} \\ \Sigma; \Gamma; \mathsf{env}_{SV}; s' \vdash Q & \text{from premise of [WF-SEC].} \end{array}$$

  The final inference is the desired result.

- Equivalence: $\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S =_{s''} \mathsf{env}_S$ holds, since the two environments are equal.

This concludes the case for [R-RESTORE].

We have examined every rule that may have been used to conclude the transition, and shown that the properties all hold in each case. This concludes the proof. □

8.9.8 Proof of Theorem 29

Proof. Firstly, consider the two field environments: We know that

$$\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2.$$

By twice application of Theorem 28, we have that

$$\begin{array}{l} \forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S^1 =_{s''} \mathsf{env}_S^{1'} \\ \forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S^2 =_{s''} \mathsf{env}_S^{2'} \end{array}$$

so the both environments agree before and after the transition on all fields that are *strictly lower* than, or unrelated to, s. Hence, we also have that

$$\forall s'' \,.\, s \not\sqsubseteq s'' \implies \Gamma \vdash \mathsf{env}_S^{1'} =_{s''} \mathsf{env}_S^{2'}$$

Suppose now that the two transitions modified a field that is exactly at level s (the field being modified must be the same in both transitions, since Q is the same). From the semantics (Figures 8.24–8.26) we see that there are only three rules that could have been used to conclude a transition that modifies env_S: [R-ASSF], [R-CALL] and [R-FCALL]. The two call rules can modify the `balance` fields of the caller and callee, and the assignment rule can modify any other field. We consider only [R-ASSF], but the argument is the same for the two other rules: The transitions are of the form

$$\begin{aligned}&\Sigma;\Gamma;\Delta \vDash_s \langle \texttt{this}.p \ := \ e; Q, \mathsf{env}_T; \mathsf{env}^i_{SV}\rangle\\ \to\ &\Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}^i_S[X \mapsto \mathsf{env}_F[p \mapsto v_i]]\,; \mathsf{env}^i_V\rangle\end{aligned}$$

for $i \in \{1, 2\}$, and from the premise of that rule we have that

$$\Sigma;\Gamma;\Delta \vDash_{(B,s')} \langle e, \mathsf{env}^i_{SV}\rangle \to v_i.$$

By Corollary 14, we have that if

$$\begin{aligned}\Sigma;\Gamma;\Delta \vDash_{(B,s')} \langle e, \mathsf{env}^1_{SV}\rangle \to v_1\\ \Sigma;\Gamma;\Delta \vDash_{(B,s')} \langle e, \mathsf{env}^2_{SV}\rangle \to v_2\end{aligned}$$

then $v_1 = v_2$, so the expression in the premise yields the same value v in both executions. Hence, the field p at level s may be changed in the two executions, but it will be changed to the *same value*. Thus $\mathsf{env}^{1'}_S$ and $\mathsf{env}^{2'}_S$ also agree at level s.

For env^i_V, we do not have a result similar to Theorem 28, so we must examine all the rules that could have been used to conclude the transition:

- If one of the rules [R-SKIP], [R-IF], [R-WHILE$_\mathrm{T}$], [R-WHILE$_\mathrm{F}$], [R-ASSF], or [R-RESTORE] was used, then the result is immediate, as the transition does not modify env^i_V. Thus, the statement holds by assumption.
- If [R-DELV] was used, the transitions are of the form

  $$\begin{aligned}&\Sigma;\Gamma;\Delta, x : \mathsf{var}(B_{s'}) \vDash_s \langle \mathsf{del}(x); Q, \mathsf{env}_T; \mathsf{env}^i_S; \mathsf{env}^i_V, x : v_i\rangle\\ \to\ &\Sigma;\Gamma;\Delta \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}^i_S; \mathsf{env}^i_V\rangle\end{aligned}$$

  so for both env^1_V and env^2_V we are removing the entry for the variable x. Equality of the remaining entries, for level s and lower, then follows from the initial assumption of $s - \mathit{equality}$ of the environments.
- If [R-RETURN] was used, the transitions are of the form

  $$\begin{aligned}&\Sigma;\Gamma;\Delta \vDash_s \langle(\mathsf{env}'_V, \Delta'); Q, \mathsf{env}_T; \mathsf{env}^i_{SV}\rangle\\ \to\ &\Sigma;\Gamma;\Delta' \vDash_s \langle Q, \mathsf{env}_T; \mathsf{env}^i_S; \mathsf{env}'_V\rangle\end{aligned}$$

  i.e. the same variable environment env'_V is taken off the stack in both transitions, so they obviously agree since they are equal.

- If one of [R-DECV] or [R-ASSV] was used, then the transition either extends the variable environment with a new entry, or modifies an existing entry. In either case, the argument is essentially the same. In the premise, we have an expression evaluation of the form

$$\Sigma; \Gamma; \Delta \vDash_{B_{s'}} \langle e, \mathrm{env}^i_{SV} \rangle \to v_i.$$

  where s' is the level of the variable being assigned to. There are now two cases:

  1. If $s' \not\sqsubseteq s$, i.e. s' is *strictly higher* than s, (or, in the case of [R-DECV], unrelated to s), then the result is immediate, since only variables of level s or lower must agree in the two environments for the result to hold.
  2. If $s' \sqsubseteq s$, we have by Corollary 14 that if

$$\Sigma; \Gamma; \Delta \vDash_{B_{s'}} \langle e, \mathrm{env}^1_{SV} \rangle \to v_1$$
$$\Sigma; \Gamma; \Delta \vDash_{B_{s'}} \langle e, \mathrm{env}^2_{SV} \rangle \to v_2$$

     then $v_1 = v_2$, so the expression in the premise yields the same value v in both executions. Hence, the two environments will contain the same value v for the variable x. Equality for all variables of level s or lower then follows from the initial assumption of s-equality of the two environments.

- Finally, if one of the call rules [R-CALL], [R-DCALL], or [R-FCALL] was used, then the $\mathrm{env}^{i'}_V$ environments will be two *new* environments created in the transition, to contain the bindings for the formal parameters $\tilde{x}$ of the method call, and the magic variables.

  For each variable binding, the argument is then exactly the same as for [R-DECV] and [R-ASSV] above: Either the variable is of a level strictly higher than, or unrelated to, s, in which case its value is allowed to differ in the two executions; or the variable is of level s or lower, in which case the corresponding expression also must be evaluated at level s or lower, and then we obtain the same value v in both executions by Corollary 14. Thus, s-equality of the two environments is again ensured. □

9 Concluding remarks

The main theme of this dissertation has been the development and application of type systems to ensure various safety properties, primarily in the field of smart contract languages, but also with some preliminary investigations carried out in the field of process calculi. These earlier developments then reappeared in various guises in later chapters, culminating in the work in Chapter 8, which directly or indirectly draws on all the previous chapters and thereby forms a natural conclusion of the present work. Some key finding on this path are the following:

- In Chapter 3 we studied *reflection* and reflective programming constructs in the context of the ρ-calculus. One of our results here was a simple, syntactic type system for disciplining channel usage in the ρ-calculus. However, in light of the main theme of this thesis, the most important finding was perhaps not the type system itself, but rather the *restrictions* we had to impose on the shape of generated names, in order to make the syntactic approach work. The need for those restrictions already hinted at the limitations of the syntactic approach, which became important for our work on the fallback function in Chapter 8.

- The contributions in Chapters 4 and 5 should be viewed together. The former chapter concerned a generic type system for a generic process calculus, the Higher-Order Ψ-calculus, which allowed several different safety properties to be treated uniformly; in particular, simple safety of channel usage, termination, and integrity/secrecy. These are the very same safety properties we later explored in the context of TinySol.

  In the latter chapter, we highlighted the connexion between ${}^{e}\pi$ (a particular instance of the Ψ-calculus) and `WC` (a class-based, imperative language similar to TinySol) by means of an encoding. The language ${}^{e}\pi$ was given a type system, which, albeit not stated in the chapter, is an instance of the generic type system from Chapter 4. We also created a type system with interface types for `WC` and an encoding of these types into the type language of ${}^{e}\pi$, so that well-typedness is preserved. This implicitly guided our development of type systems for ensuring these safety properties in TinySol.

- In Chapter 6, we created a type system for ensuring non-interference in TinySol programs, based on the dual safety properties of integrity and secrecy, that we also saw in Chapter 4. This is interesting its own right, but additionally we showed that the type system can also be used to approximate the *call-integrity* property of Grishchenko et al. [51] by setting a particular security policy. This is a surprising result, since non-interference and call-integrity are incomparable properties; one is a property of data flows, and the other is a property of control flows. The result stemmed from the realisation that a currency flow in TinySol is also a data flow, which thus can be disciplined with the security types, and every currency flow in TinySol is also a control flow.

- In Chapter 7 we created a type system for preventing out-of-gas exceptions in TinySol by ensuring termination of transactions. This is a rather crude attempt, since it disallows all forms of recursive method calls, as well as increments/decrements of counter variables used to control loops. As was the case with the type system in Chapter 3, the most interesting finding here was therefore not the type system itself, but rather the restrictions we had to impose via the types to ensure that subject-reduction could be shown.

- Finally, in Chapter 8, we extended TinySol with delegate calls and fallback functions, and created a new type system for ensuring non-interference, similar to the one from Chapter 6, but this time based on a *semantic* approach to type soundness. The fallback function is untypable by syntactic means because of its reflective (introspective) features, but the semantic approach enables us to reason about the safety of constructs, where the difference between safe and unsafe usages is behavioural rather than syntactical. In these cases, a 'manual proof' of safety must be given instead, and we provide a way to state such proofs in the form of an explicit typing interpretation.

  We also define an enhancement of the typing-interpretation proof-method, *typing interpretations up-to union*, which can greatly simplify the work needed to manually show safety of a piece of code for which type safety cannot be inferred by the type rules. To our knowledge, transferring the method of up-to techniques from bisimulations to typing interpretations is novel, and also a useful addition to the semantic approach to type soundness.

9.1 On the significance of our results

Type systems are diverse, even within a single type discipline. The inference rules are syntax-directed, and thus closely tied to the syntax of the particular language. It usually requires a significant amount of work to adapt a type system from one language to another, or even just to extend an existing type system for a language to accommodate the addition of a new syntactic construct. If soundness is shown

by the syntactic (subject-reduction) approach, adding just a single, new construct will require (at least) adding a new case to the safety and subject reduction theorems, as well as to all the auxiliary lemmas. Thus, type system results can be difficult to adapt, extend or generalise, and this is also the case for several of our results. The type systems for termination and call-integrity in TinySol show how these properties may be ensured for this *particular* modelling language, but this does not mean that the type systems can be directly applied to the full Solidity language, or easily generalised to other smart-contract languages, even if they belong to the same programming paradigm. This is not to say that these results are insignificant, since they do demonstrate that these properties indeed may be captured by static analysis techniques for a language containing the characteristic features of Solidity. However, we have only demonstrated the plausibility of the approach. More work is needed for an application to the full Solidity language.

On the other hand, our work in Chapter 8 is of a more general nature. The direct applicability of the type system therein is of course still limited to the particular version of TinySol defined in this chapter, as was the case with our other type systems. However, our most significant finding in this chapter is not the type system itself, but the *method* that we develop to ensure safety of even syntactically untypable constructs, by means of the semantic approach to type soundness. There will always be statements or constructs for which safety cannot be concluded by an incomplete proof system, but which nevertheless *behave* safely at runtime; they reside in the *slack* of the type system. This does not only apply to exotic language constructs such as the fallback function w.r.t. integrity/secrecy of data flows; for example, as we saw in Chapter 7, it also applies to ordinary constructs such as variable increments/decrements and recursive method calls w.r.t. termination. Nor is this only a problem in TinySol, or in smart-contract languages in general. A type system for any Turing complete language will always be an incomplete proof system for all but the most trivial and uninteresting safety properties, so slack is an inherent, unavoidable phenomenon.

The semantic approach gives us a way to attest the safety of *specific* programs or pieces of code residing in the slack of the static type rules, such as specific usages of fallback functions, by providing an explicit typing interpretation to witness safety. This is particularly useful in the case of smart-contract languages, because of the immutability of the code, which is one of the characteristic features of this programming paradigm. We demonstrate the steps involved using TinySol and the fallback function, but our proposed approach itself is in principle applicable to any typed smart-contract language. Thus, we believe that the developments in Chapter 8 are relevant beyond the area of TinySol/Solidity and usages of fallback functions.

9.2 Avenues of future research

We have already pointed out specific avenues of future work in the respective chapters of this thesis. Here we shall just highlight a few that may be of more general interest.

9.2.1 Enhancements of the typing interpretation proof method

In Chapter 8, we define the concept of *up-to techniques* for typing interpretations, following the work of Pous and Sangiorgi in [106] on enhancements of the bisimulation proof method. However, we only defined a single up-to technique, the *typing interpretation up-to union* enhancement, but Pous and Sangiorgi also mention several other techniques that might be interesting and relevant to reformulate as enhancements of the typing interpretation proof method. Some examples could be typing interpretations up-to context, up-to environment extensions, up-to subtyping, to name but a few. Exactly which techniques might be relevant will depend on the features of the specific programming language, and possibly also on the notion of type safety. For example, a technique such as *up-to structural congruence* would allow us to disregard changes introduced by program rewrites under structural congruence, which would be useful for languages such as the ρ-calculus (see Chapter 3) or the π-calculus [83] that make use of this kind of program equivalence.

A disadvantage of our definition in Chapter 8 is that it relies on the notion of a progression $\mathcal{S} \rightarrowtail \mathcal{R}$ (Definition 65) which builds on the specific definition of a typing interpretation that we use in this chapter (Definition 64). It would be useful to be able to consider up-to techniques in a more abstract sense, which may make them easier to apply in different settings. In [106], Pous and Sangiorgi also formulate a more general theory of enhancements of the bisimulation proof method, based on greatest fixed-points of monotonic functions on a complete lattice. Again, it seems that these concepts can be adapted to the setting of typing interpretations in a quite straightforward manner. Following their work, we propose the following generic definitions:

- Let $(\mathcal{P}, \Mapsto)$ be a transition system with $P \in \mathcal{P}$ being a representation of programs, such as the stack type triplets in Definition 63.
- Let Ξ be a type environment, or collection of environments, such as $\Sigma; \Gamma; \mathsf{env}_T$ in Definition 64.
- Let $\mathcal{F}_\Xi$ be a monotonic function defined such that $\mathcal{R}_\Xi$ is a typing interpretation of Ξ when $\mathcal{R}_\Xi \subseteq \mathcal{F}_\Xi(\mathcal{R}_\Xi)$. We write νf for the greatest fixed-point of a function f. Thus
$$\nu\mathcal{R}_\Xi \triangleq \nu\mathcal{F}_\Xi = \bigcup \{ \mathcal{R} \mid \mathcal{R} \subseteq \mathcal{F}_\Xi(\mathcal{R}) \}$$
is the largest typing interpretation for Ξ.

As indicated above, this generic notation can be instantiated with the definitions of type environments, type-lifted transitions and typing interpretations that we use in Chapter 8. With this notation, for any fixed typing interpretation $\mathcal{R}_\Xi$, the up-to union technique can then be expressed as the function $u : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ defined as follows:

$$u : \mathcal{S} \mapsto \mathcal{S} \cup \mathcal{R}_\Xi.$$

Pous and Sangiorgi then provide two definitions: validity (soundness) and compatibility. They require no adaptation, apart from a change of symbols:

Definition 67 (Valid up-to technique)**.** Let $u : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ be a monotone function. u is a *valid* up-to technique, if $\nu(\mathcal{F}_\Xi \circ u) \subseteq \nu\mathcal{F}_\Xi$. ■

Definition 68 (Compatibility)**.** Let $u : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ be a monotone function. u is *$\mathcal{F}_\Xi$-compatible* if $u \circ \mathcal{F}_\Xi \subseteq \mathcal{F}_\Xi \circ u$. ■

Compatibility implies validity: this is proved in [106, p. 254] (Theorem 6.3.9). It is a stronger requirement than validity and not strictly necessary, but it helps to ensure that up-to techniques may be combined. To help prove compatibility, Pous and Sangiorgi then state and prove a lemma (Lemma 6.3.12 in [106, p. 255]) which, when adapted to our notation, takes the following form:

Lemma 81. *Let $u : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ be a monotone function. u is $\mathcal{F}_\Xi$-compatible iff for all $\mathcal{S}, \mathcal{R} \subseteq \mathcal{P}$ it holds that $\mathcal{S} \subseteq \mathcal{F}_\Xi(\mathcal{R})$ entails $u(\mathcal{S}) \subseteq \mathcal{F}_\Xi\big(u(\mathcal{R})\big)$.*

Using this generic formulation may not only make it easier to adapt up-to techniques from bisimulations to typing interpretations; it may also make the formulation of the techniques less dependent on the specific definition of a typing interpretation, and thus make them more easily reusable with different languages and types. We believe this is an important area to explore further, with a general application to the field of semantic typing.

9.2.2 Modelling contract creation

The versions of TinySol presented in Chapters 6–8 assume that all contracts are created at the Genesis event, and all subsequent transactions only invoke the existing functionality. Thus, the code environment, env_T, is static; and likewise, the set of field names declared in the state environment env_S does not change. However, this is clearly an oversimplification: actual blockchain architectures also allow users to schedule transactions that will place a *new* contract declaration on the blockchain at any time, e.g. via the `CREATE` and `CREATE2` instructions in EVM [119]. This is a form of *higher-order behaviour*, similar to what we have seen in the ρ-calculus and the Higher-Order Ψ-calculus in Chapters 3–4 where data is instantiated as executable code. However, as we have also seen in these chapters, higher-order behaviour can

be tricky to handle with type systems. Essentially, it must be the case that type-safety (and well-typedness, in the syntactic approach) must be compositional, so that *if* the existing code is type safe, *and* the new code is also type safe, *then* the resulting composition will also be type safe.

In Chapter 8, we assume that env_T and the type environments Σ and Γ are static, and they therefore appear as indices on typing interpretations $\mathcal{R}_{\Sigma,\Gamma,\mathsf{env}_T}$. However, in light of the above, it would be relevant to extend this work to also allow these environments to change at runtime. Special consideration should also be given to the fact that our proposed method for ensuring safety of untypable constructs requires the contract creator to supply an explicit typing interpretation to witness type safety of such constructs in his code. Thus, it will be important to ensure that extensions of Σ, Γ and env_T will not invalidate any existing witnesses on the blockchain. Again, it seems that a possible solution may be to formulate an appropriate up-to technique, such that a piece of code can be shown to be type safe *up-to extensions* of Σ, Γ and env_T. Extending our work in this direction will therefore be important to further strengthen the argument for the applicability of our proposed method.

9.2.3 A logical characterisation of secure data flows

As mentioned in Chapter 8, rather than using a typed semantics to define the semantics of the secure flow types, we might instead have used a form of logic to express the meaning of each type as formulae φ. For example, in [23], Caires uses a type language based on spatial logics [24; 25] to create a type system for the π-calculus, following the semantic approach to type soundness. Adapting spatial logics to TinySol might be interesting in its own right, since it for example would allow us to express more fine-grained properties of data flows than the invariants denoted by types $\mathsf{var}(B_s)$; but it would additionally have the benefit of leading to a cleaner presentation, since we would not need to mix the semantics with checks of the type constraints. We shall here sketch some of the necessary adaptations.

As a first step, we will need a labelled semantics for TinySol. It would take us too far afield to create one here, but let us, for the sake of the presentation, assume a labelled transition system $(\mathcal{P}, \xrightarrow{\alpha} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P})$, where $\mathcal{P}$ is a set of states (i.e. stack configurations), ranged over by P; and $\mathcal{A}$ is a set of labels, ranged over by α, and containing labels of the form

$$\alpha \in \mathcal{A} ::= x\,!\,v \mid X.p\,!\,v \mid \ldots$$

denoting that the value v is written into the variable x, resp. the field $X.p$. Other labels would likely also be needed, but we shall ignore that here.

To express the dual properties of integrity and secrecy, we would need operators that speak of the data flows into, and out of, variables and fields of a particular security level s. This can be captured with the modal 'box' and 'diamond' operators $[a]\varphi$ and

$\langle a \rangle \varphi$, denoting universal, resp. existential, quantification over labelled transitions, with labels satisfying the formula a. Assuming a set of types B_s for values, as in the type system from Chapter 8, formulae a would be of the form

$$a ::= n!B_s \mid X.p!B_s \mid \ldots$$

matching the labels from the semantics. We use n here, rather than x, to allow quantification over bound variable names.[1] Formulae a are then, in a sense, the logical equivalent of type rules for reading and writing, and the semantics of a must therefore depend on type environments $\Sigma; \Gamma$ like the type rule for values. To avoid complicating the matter needlessly, we shall just consider the semantics for variable writes, which can be given in denotational style as

$$[\![x!B_s]\!]_{\Sigma;\Gamma} = \left\{ x!v \mid \textsc{TypeOf}_\Gamma(v) = B'_{s'} \wedge \Sigma \vdash B' <: B \wedge s' \sqsubseteq s \right\}$$

with $\textsc{TypeOf}_\Gamma(\cdot)$ and $\Sigma \vdash B' <: B$ defined as in Chapter 8. The semantics of the box and diamond operators can then be given in the usual way:

$$[\![[a]\varphi]\!]_{\Sigma;\Gamma} = \left\{ P \mid \forall P', \alpha \,.\, P \xrightarrow{\alpha} P' \implies \alpha \in [\![a]\!]_{\Sigma;\Gamma} \wedge P' \in [\![\varphi]\!]_\Gamma \right\}$$
$$[\![\langle a \rangle \varphi]\!]_{\Sigma;\Gamma} = \left\{ P \mid \exists P', \alpha \,.\, P \xrightarrow{\alpha} P' \wedge \alpha \in [\![a]\!]_{\Sigma;\Gamma} \wedge P' \in [\![\varphi]\!]_\Gamma \right\}$$

Using the box operator, a type assignment $x : \mathrm{var}(B_s)$ can then be expressed as an invariant, stating that all $x!v$ labelled transitions must satisfy the formula $x!B_s$. However, unlike the type $\mathrm{var}(B_s)$, the logic does not *require* formulae to be invariants. Using the same language of formulae, we can also express types that *evolve* during the execution of the program, and thereby allow variables to have different capabilities at different points of interaction. This would be approaching the type discipline known as *behavioural types*, which is another area that may be relevant to explore.

[1] This would be used by the spatial *hiding* operator $\mathsf{H}n.\varphi$, which detects the declaration of a new, locally scoped name/variable. We shall not describe it further here, but see [23] for details regarding its meaning.

Bibliography

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1998.2740.

[2] Luca Aceto, Daniele Gorla, and Stian Lybech. A sound type system for secure currency flow, 2024. URL https://arxiv.org/abs/2405.12976.

[3] Luca Aceto, Daniele Gorla, and Stian Lybech. Typing composite subjects, 2024. URL https://arxiv.org/abs/2411.13732.

[4] Luca Aceto, Daniele Gorla, and Stian Lybech. A sound type system for secure currency flow. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:27, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-341-6. doi: 10.4230/LIPIcs.ECOOP.2024.1. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.1.

[5] Luca Aceto, Daniele Gorla, Stian Lybech, and Mohammad Hamdaqa. Preventing out-of-gas exceptions by typing, 2024. URL https://arxiv.org/abs/2407.15676.

[6] Luca Aceto, Daniele Gorla, Stian Lybech, and Mohammad Hamdaqa. Preventing out-of-gas exceptions by typing. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. REoCAS Colloquium in Honor of Rocco De Nicola*, pages 409–426, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-73709-1. doi: 10.1007/978-3-031-73709-1_25.

[7] Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004. URL http://www.ccs.neu.edu/home/amal/ahmedsthesis.pdf.

[8] Andrew W. Appel. Foundational proof-carrying code. *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–256, 2001. doi: 10.1145/363911.363923.

[9] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5): 657–683, September 2001. ISSN 0164-0925. doi: 10.1145/504709.504712.

[10] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Proc. of POST*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017. doi: 10.1007/978-3-662-54455-6_8.

[11] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

[12] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A minimal core calculus for solidity contracts. In Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 233–243, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31500-9. doi: 10.1007/978-3-030-31500-9_15.

[13] Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. Verifying liquidity of recursive bitcoin contracts. *Logical Methods in Computer Science*, Volume 18, Issue 1:22, Feb 2022. ISSN 1860-5974. doi: 10.46298/lmcs-18(1:22)2022. URL https://lmcs.episciences.org/6943.

[14] Massimo Bartoletti, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Giacomo Dal Sasso, Roberto Pettinau, Andrea Pinna, Mattia Piras, Sabina Rossi, Stefano Salis, Alvise Spanò, Viacheslav Tkachenko, Roberto Tonelli, and Roberto Zunino. Smart contract languages: a comparative analysis, 2024. URL https://arxiv.org/abs/2404.04129.

[15] Alex Rønning Bendixen, Bjarke Bredow Bojesen, Hans Hüttel, and Stian Lybech. A generic type system for higher-order ψ-calculi. In Valentina Castiglioni and Claudio A. Mezzina, editors, *Proceedings Combined 29th International Workshop on Expressiveness in Concurrency and 19th Workshop on Structural Operational Semantics, Warsaw, Poland, 12th September 2022*, volume 368 of *Electronic Proceedings in Theoretical Computer Science*, pages 43–59. Open Publishing Association, 2022. doi: 10.4204/EPTCS.368.3.

[16] Alexander R. Bendixen, Bjarke B. Bojesen, and Stian Lybech. Typing reflection in higher-order psi-calculi. Technical report, Department of Computer Science, Aalborg University, June 2021.

[17] Alexander R. Bendixen, Bjarke B. Bojesen, Hans Hüttel, and Stian Lybech. Typing reflection in higher-order psi-calculi. Technical report, Department of Computer Science, Aalborg University, May 2022. URL http://icetcs.ru.is/stian/2022/hopsitypes2022techreport.pdf.

[18] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: Mobile processes, nominal data, and logic. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 39–48. IEEE, 2009. doi: 10.1016/S1571-0661(05)80361-5.

[19] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, Volume 7, Issue 1, March 2011. doi: 10.2168/LMCS-7(1:11)2011. URL https://lmcs.episciences.org/696.

[20] Gérard Boudol. Typing termination in a higher-order concurrent imperative language. *Information and Computation*, 208(6):716–736, 2010. URL https://doi.org/10.1016/j.ic.2009.06.007.

[21] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. Rich specifications for Ethereum smart contract verification. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. URL https://doi.org/10.1145/3485523.

[22] Luís Caires. Behavioral and spatial observations in a logic for the π-calculus. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, pages 72–89, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24727-2. doi: 10.1007/978-3-540-24727-2_7.

[23] Luís Caires. Logical semantics of types for concurrency. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *Algebra and Coalgebra in Computer Science*, pages 16–35, Berlin, Heidelberg, 08 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73857-2. doi: 10.1007/978-3-540-73859-6_2.

[24] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part I). *Information and Computation*, 186(2):194–235, 2003. ISSN 0890-5401. doi: 10.1016/S0890-5401(03)00137-8. Theoretical Aspects of Computer Software (TACS 2001).

[25] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part II). *Theoretical Computer Science*, 322(3):517–565, 2004. ISSN 0304-3975. doi: 10.1016/j.tcs.2003.10.041. Foundations of Wide Area Network Computing.

[26] Marco Carbone. *Trust and Mobility*. PhD thesis, University of Aarhus, 2005. BRICS Dissertation Series Number DS-05-3.

[27] Marco Carbone and Sergio Maffeis. On the expressive power of polyadic synchronisation in pi-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003. ISSN 1236–6064. doi: 10.1016/S1571-0661(05)80361-5.

[28] Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. Formal verification of the Ethereum 2.0 Beacon Chain. In *28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243 of *LNCS*, pages 167–182. Springer, 2022. URL https://doi.org/10.1007/978-3-030-99524-9_9.

[29] Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive verification of smart contracts with Dafny. In *27th International Conference on Formal Methods for Industrial Critical Systems*, volume 13487 of *LNCS*, pages 50–66. Springer, 2022. URL https://doi.org/10.1007/978-3-031-15008-1_5.

[30] Arthur Charguéraud. Pretty-big-step semantics. In *Proc. of ESOP*, volume 7792 of *LNCS*, pages 41–60. Springer, 2013. doi: 10.1007/978-3-642-37036-6_3.

[31] Norine Coenen, Bernd Finkbeiner, Jana Hofmann, and Julia Tillman. Smart contract synthesis modulo hyperproperties, 2022. URL https://arxiv.org/abs/2208.07180.

[32] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011. URL https://doi.org/10.1145/1941487.1941509.

[33] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. Is solidity solid enough? In *Financial Cryptography Workshops*, 2019.

[34] Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In *Proc. of ESOP*, volume 12075 of *LNCS*, pages 169–196. Springer, 2020. doi: 10.1007/978-3-030-44914-8_7.

[35] dao. The dao smart contract. http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code, 2016.

[36] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in higher-order concurrent calculi. *The Journal of Logic and Algebraic Programming*, 79(7):550–577, 2010. ISSN 1567-8326. doi: 10.1016/j.jlap.2010.07.007. The 20th Nordic Workshop on Programming Theory (NWPT 2008).

[37] Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Information and Computation*, 204(7):1045–1082, 2006.

[38] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-carrying smart contracts. In *Financial Cryptography and Data Security: FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*, page 325–338, Berlin, Heidelberg, 2018. Springer-Verlag. ISBN 978-3-662-58819-2. doi: 10.1007/978-3-662-58820-8_22.

[39] Afonso Falcão, Andreia Mordido, and Vasco T. Vasconcelos. Protocol-based smart contract generation, 2021. URL https://arxiv.org/abs/2108.02672.

[40] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 8–15. IEEE / ACM, 2019. URL https://doi.org/10.1109/WETSEB.2019.00008.

[41] Ethereum Foundation. Solidity documentation. https://docs.soliditylang.org/, 2022. Accessed: 2024-01-15.

[42] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *International Summer School on Applied Semantics*, pages 268–332. Springer, 2000. doi: 10.1007/3-540-45699-6_6.

[43] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13:341–363, 07 2002. doi: 10.1007/s001650200016.

[44] Philippa Gardner and Lucian Wischik. Explicit fusions. In *International Symposium on Mathematical Foundations of Computer Science*, pages 373–382. Springer, 2000. doi: 10.1007/3-540-44612-5_33.

[45] Thomas Genet, Thomas P. Jensen, and Justine Sauvage. Termination of Ethereum's smart contracts. In *Proc. of the 17th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT*, pages 39–51. ScitePress, 2020. URL https://doi.org/10.5220/0009564100390051.

[46] Rob van Glabbeek. A theory of encodings and expressiveness. In *International Conference on Foundations of Software Science and Computation Structures*, pages 183–202. Springer, Cham, 2018. doi: 10.1007/978-3-319-89366-2_10.

[47] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982. doi: 10.1109/SP.1982.10014.

[48] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010. ISSN 0890-5401. doi: 10.1016/j.ic.2010.05.002.

[49] Daniele Gorla and Uwe Nestmann. Full abstraction for expressiveness: history, myths and facts. *Mathematical Structures in Computer Science*, 26:639 – 654, 2016. doi: 10.1017/S0960129514000279.

[50] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, 2018. URL https://doi.org/10.1145/3276486.

[51] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 243–269, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89722-6.

[52] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 2nd edition, 2016. ISBN 1107150302.

[53] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002. ISSN 0890-5401. doi: 10.1006/inco.2001.3089.

[54] Mark Hepburn and David Wright. Execution contexts for determining trust in a higher-order π-calculus. Technical Report Technical Report TR-01-2003, School of Computing, University of Tasmania, 2033. https://eprints.utas.edu.au/36/.

[55] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. Kevm: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018. doi: 10.1109/CSF.2018.00022.

[56] Yoichi Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In *Financial Cryptography Workshops*, 2017.

[57] Daniel Hirschkoff, Enguerrand Prebet, and Davide Sangiorgi. On the representation of references in the pi-calculus. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 34:1–34:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPICS.CONCUR.2020.34.

[58] Xinwen Hu, Yi Zhuang, Shang-Wei Lin, Fuyuan Zhang, Shuanglong Kan, and Zining Cao. A security type verifier for smart contracts. *Computers & Security*, 108:102343, 2021. ISSN 0167-4048. doi: 10.1016/j.cose.2021.102343. URL https://doi.org/10.1016/j.cose.2021.102343.

[59] Xinwen Hu, Yi Zhuang, Shangwei Lin, Fuyuan Zhang, Shuanglong Kan, and Zining Cao. A security type verifier for smart contracts. *Comput. Secur.*, 108: 102343, 2021. URL https://doi.org/10.1016/j.cose.2021.102343.

[60] Hans Hüttel. *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010. ISBN 978-0-521-14709-5. doi: 10.1017/CBO9780511840449.

[61] Hans Hüttel. Typed ψ-calculi. In *International Conference on Concurrency Theory*, pages 265–279. Springer, 2011. doi: 10.1007/978-3-642-23217-6_18.

[62] Hans Hüttel. Types for resources in ψ-calculi. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, pages 83–102, Cham, 2014. Springer International Publishing. ISBN 978-3-319-05119-2. doi: 10.1007/978-3-319-05119-2_6.

[63] Hans Hüttel. Binary session types for psi-calculi. In Atsushi Igarashi, editor, *Programming Languages and Systems*, pages 96–115, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47958-3. doi: 10.1007/978-3-319-47958-3_6.

[64] Hans Hüttel, Stian Lybech, Alex R. Bendixen, and Bjarke B. Bojesen. A generic type system for higher-order Ψ-calculi. *Information and Computation*, page 105190, 2024. ISSN 0890-5401. doi: https://doi.org/10.1016/j.ic.2024.105190.

[65] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1):121–163, 2004. ISSN 0304-3975. doi: 10.1016/S0304-3975(03)00325-6.

[66] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1695–1712. IEEE, 2020. doi: 10.1109/SP40000.2020.00066. URL https://doi.org/10.1109/SP40000.2020.00066.

[67] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154.

[68] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium*. The Internet Society, 2018. URL https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf.

[69] Josva Kleist and Davide Sangiorgi. Imperative objects as mobile processes. *Sci. Comput. Program.*, 44(3):293–342, 2002. doi: 10.1016/S0167-6423(02)00034-5.

[70] Barbara König. Analysing input/output-capabilities of mobile processes with a generic type system. *The Journal of Logic and Algebraic Programming*, 63(1): 35–58, 2005. ISSN 1567-8326. doi: 10.1016/j.jlap.2004.01.004. Special issue on The pi-calculus.

[71] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009. doi: 10.1016/J.IC.2007.12.004.

[72] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proc. SIGSAC Conf. on Computer and Communications Security*, page 254–269. ACM, 2016. URL https://doi.org/10.1145/2976749.2978309.

[73] Stian Lybech. Reflection, encodability and separation. NWPT, Nordic Workshop in Programming Technology, 2021. URL http://icetcs.ru.is/nwpt21/abstracts/paper18.pdf.

[74] Stian Lybech. Encodability and separation for a reflective higher-order calculus. In Valentina Castiglioni and Claudio A. Mezzina, editors, *Proceedings Combined 29th International Workshop on Expressiveness in Concurrency and 19th Workshop on Structural Operational Semantics, Warsaw, Poland, 12th September 2022*, volume 368 of *Electronic Proceedings in Theoretical Computer Science*, pages 95–112. Open Publishing Association, 2022. doi: 10.4204/EPTCS.368.6.

[75] Stian Lybech. Encodability and separation for a reflective higher-order calculus. Technical report, Department of Computer Science, Reykjavík University, June 2022. URL http://icetcs.ru.is/stian/2022/reflection_encodability2022techreport.pdf.

[76] Stian Lybech. The reflective higher-order calculus: Encodability, typability and separation. *Information and Computation*, 297:105138, 2024. ISSN 0890-5401. doi: 10.1016/j.ic.2024.105138.

[77] Adrian Manning. Solidity security: A comprehensive list of known attack vectors and common anti-patterns, 2018. URL https://blog.sigmaprime.io/solidity-security.html.

[78] Diego Marmsoler, Asad Ahmed, and Achim D. Brucker. Secure smart contracts with isabelle/solidity. In Alexandre Madeira and Alexander Knapp, editors, *Software Engineering and Formal Methods*, pages 162–181, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-77382-2.

[79] Anastasia Mavridou and Aron Laszka. Designing secure Ethereum smart contracts: A finite state machine based approach. In *22nd Conference on Financial Cryptography and Data Security*, volume 10957 of *LNCS*, pages 523–540. Springer, 2018. URL https://doi.org/10.1007/978-3-662-58387-6_28.

[80] L.G. Meredith and Matthias Radestock. A reflective higher-order calculus. *Electronic Notes in Theoretical Computer Science*, 141(5):49 – 67, 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.05.016. Proceedings of the Workshop on the Foundations of Interactive Computation (FInCo 2005).

[81] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. ISSN 0022-0000. doi: 10.1016/0022-0000(78)90014-4. URL https://doi.org/10.1016/0022-0000(78)90014-4.

[82] Robin Milner. Functions as processes. *Mathematical structures in computer science*, 2(2):119–141, 1992.

[83] Robin Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer Berlin Heidelberg, 1993. doi: 10.1007/978-3-642-58041-3_6.

[84] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991. ISSN 0304-3975. doi: 10.1016/0304-3975(91)90033-X. URL https://doi.org/10.1016/0304-3975(91)90033-X.

[85] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992.

[86] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992. ISSN 0890-5401. doi: 10.1016/0890-5401(92)90008-4.

[87] Satoshi Nakamoto. Bitcoin : A peer-to-peer electronic cash system. Technical report, Bitcoin.org, 2009. URL https://bitcoin.org/bitcoin.pdf.

[88] George C. Necula. Proof-carrying code. In *Proc. of POPL*, page 106–119. ACM, 1997. ISBN 0897918533. URL https://doi.org/10.1145/263699.263712.

[89] Uwe Nestmann and António Ravara. Semantics of objects as processes (SOAP). In *ECOOP'99 Workshops*, volume 1743 of *LNCS*, pages 314–325. Springer, 1999.

[90] Hanne Riis Nielson. A Hoare-like proof system for analysing the computation time of programs. *Sci. Comput. Program.*, 9(2):107–136, 1987. URL https://doi.org/10.1016/0167-6423(87)90029-3.

[91] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications - A Formal Introduction*. Wiley Professional Computing. Wiley, 1992. ISBN 978-0-471-92980-2.

[92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag London, 2007. doi: 10.1007/978-1-84628-692-6.

[93] parity a. The parity wallet breach. https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/, 2017.

[94] parity b. The parity wallet vulnerability. https://paritytech.io/blog/security-alert.html, 2017.

[95] Daejun Park, Yi Zhang, and Grigore Rosu. End-to-end formal verification of Ethereum 2.0 Deposit Smart Contract. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2020. URL https://doi.org/10.1007/978-3-030-53288-8_8.

[96] Joachim Parrow. An introduction to the π-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001. doi: 10.1016/B978-044482830-9/50026-6.

[97] Joachim Parrow. Expressiveness of process algebras. In *Emerging Trends in Concurrency Theory*, volume 209 of *ENTCS*, pages 173–186. Elsevier, 2006. doi: 10.1016/J.ENTCS.2008.04.011.

[98] Joachim Parrow, Johannes Borgström, Palle Raabjerg, and Johannes Åman Pohjola. Higher-order psi-calculi. *Mathematical Structures in Computer Science*, 24(2), 2014. doi: 10.1017/S0960129513000170.

[99] Kirstin Peters and Rob J. van Glabbeek. Analysing and comparing encodability criteria. In *Proc. of EXPRESS/SOS*, volume 190 of *EPTCS*, pages 46–60, 2015. doi: 10.4204/EPTCS.190.4.

[100] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 376–385. IEEE, 1993. doi: 10.1109/LICS.1993.287570.

[101] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

[102] Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 455–494. The MIT Press, 2000.

[103] Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. In *Proof, Language, and Interaction*, pages 455–494, 2000. doi: 10.5555/345868.345924.

[104] Gordon Plotkin. Lambda-definability and logical relations, 1973. URL https://www.cl.cam.ac.uk/~nk480/plotkin-logical-relations.pdf.

[105] Casper Bach Poulsen and Peter D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In *Proc. of ESOP*, volume 8410 of *LNCS*, pages 270–289. Springer, 2014. doi: 10.1007/978-3-642-54833-8_15.

[106] Damien Pous and Davide Sangiorgi. Enhancements of the bisimulation proof method. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, Cambridge Tracts in Theoretical Computer Science, pages 233–289. Cambridge University Press, 2011. doi: 10.1017/cbo9780511792588.007.

[107] António Ravara and Vasco Thudichum Vasconcelos. An operational semantics and a type system for GNOME based on a typed calculus of objects. Technical report, 17-96, Dep. of Mathematics, Tech. Univ. of Lisbon, 1996.

[108] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010. URL https://doi.org/10.1016/j.jlap.2010.03.012.

[109] Davide Sangiorgi. From π-calculus to higher-order π-calculus — and back. In M. C. Gaudel and J. P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, pages 151–166, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-47598-9. doi: 10.1007/3-540-56610-4_62.

[110] Davide Sangiorgi. *Expressing mobility in process algebras: first-order and higher-order paradigms*. PhD thesis, University of Edinburgh, 1993. URL http://hdl.handle.net/1842/6569.

[111] Davide Sangiorgi. An interpretation of typed objects into typed pi-calculus. *Inf. Comput.*, 143(1):34–73, 1998. doi: 10.1006/INCO.1998.2711.

[112] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[113] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

[114] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proc. of SIGSAC Conf. on Computer and Communications Security*, pages 621–640. ACM, 2020. URL https://doi.org/10.1145/3372297.3417250.

[115] Clara Schneidewind, Markus Scherer, and Matteo Maffei. The good, the bad and the ugly: Pitfalls and best practices in automated sound static analysis of Ethereum smart contracts. In *Lecture Notes in Computer Science*, pages 212–231. Springer International Publishing, 2020. doi: 10.1007/978-3-030-61467-6_14. URL https://arxiv.org/abs/2101.05735.

[116] Pablo Lamela Seijas, Simon J. Thompson, and Darryl McAdams. Scripting smart contracts for distributed ledger technology. *IACR Cryptol. ePrint Arch.*, 2016:1156, 2016.

[117] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1982. URL http://hdl.handle.net/1721.1/15961.

[118] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of 25th POPL*, pages 355–364. ACM, 1998.

[119] smlxl inc. An Ethereum virtual machine opcodes interactive reference, 2023. URL https://evm.codes/.

[120] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2, 1997.

[121] William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi: 10.2307/2271658. URL https://doi.org/10.2307/2271658.

[122] Bent Thomsen. Plain CHOCS a second generation calculus for higher order processes. *Acta Inf.*, 30(1):1–59, jan 1993. ISSN 0001-5903. doi: 10.1007/BF01200262.

[123] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. *J. ACM*, 71(6), November 2024. ISSN 0004-5411. doi: 10.1145/3676954. URL https://doi.org/10.1145/3676954.

[124] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7):148:1–148:38, 2020. URL https://doi.org/10.1145/3464421.

[125] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *Proc. of SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM, 2018. URL https://doi.org/10.1145/3243734.3243780.

[126] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, UK, 1996. URL https://hdl.handle.net/1842/395.

[127] Rob van Glabbeek. Musings on encodings and expressiveness. In *Proc. of EXPRESS/SOS*, volume 89 of *EPTCS*, pages 81–98, 2012. doi: 10.4204/EPTCS.89.7.

[128] Rob van Glabbeek. A theory of encodings and expressiveness (extended abstract). In *Proc. of FoSSaCS*, volume 10803 of *LNCS*, pages 183–202. Springer, 2018. doi: 10.1007/978-3-319-89366-2_10.

[129] Vasco Thudichum Vasconcelos. An operational semantics and a type system for ABCL/1 based on a calculus of objects. In *Object-Oriented Computing III, Lecture Notes*. Kindai Kagaku Sha, 1995.

[130] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4, 08 2000. doi: 10.3233/JCS-1996-42-304.

[131] David Walker. Objects in the π-calculus. *Information and Computation*, 116(2): 253–271, 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1018.

[132] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. In *Yellow Paper*, 2014. Updated for EIP-150 in 2017.

[133] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[134] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing Ethereum's gas mechanism. In *Proc. of IEEE European Symposium on Security and Privacy Workshops*, pages 310–319. IEEE, 2019. URL https://doi.org/10.1109/EuroSPW.2019.00041.

[135] Nobuko Yoshida. Channel dependent types for higher-order mobile processes. *SIGPLAN Not.*, 39(1):147–160, jan 2004. ISSN 0362-1340. doi: 10.1145/982962.964014. URL https://doi.org/10.1145/982962.964014.

[136] Johannes Åman Pohjola. Psi-calculi revisited: Connectivity and compositionality. In Jorge A. Pérez and Nobuko Yoshida, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 3–20, Cham, 2019. Springer International Publishing. ISBN 978-3-030-21759-4.

Part III

Appendices

A The original formulation of TinySol

In Chapters 6-8 we use a modified version of the language TinySol, which is adapted and extended in various ways to suit the purpose of each study. We shall here give a brief review the original presentation of TinySol, as given in [12], and also add some comments on the changes we have made to the language and the rationale behind them. In the following presentation, we have made some insignificant changes in the syntax to facilitate reading.[1]

A.1 Syntax

The original presentation of TinySol has two kinds of syntactic constructs: *expressions* e, and *statements* S. Expressions can contain values v, atomic variable names x, and atomic contract addresses X, which are distinct from variable names. There is also a concept of keys k, which are values, and function names f, but these are only indirectly defined in the paper. In sum, we thus have the following sets:

- $k \in \mathsf{Keys}$ is a set of atomic *keys* (names), which may be bound to values in a store, associated with each contract. It is assumed that there is a special name `balance` $\in \mathsf{Keys}$, and this name may not appear on the left side in an assignment.
- $X, Y \in \mathsf{ANames}$ is a set of addresses. This set is partitioned into *account addresses* A, B and *contract addresses* C, D. These are distinct from the set of keys (which are local addresses within the scope of each contract).
- $v \in \mathsf{Val} = \mathbb{Z} \cup \mathbb{B} \cup \mathsf{Keys} \cup \mathsf{ANames}$ is a set of *values*, consisting of the set of natural numbers, boolean values, keys and addresses.
- $x, y \in \mathcal{N}$ is a set of atomic names (called constants), which may appear as formal parameters of function declarations. Unlike keys, these names are *not* values, and they cannot appear on the left-hand side in an assignment. There

[1] The original presentation uses differently coloured symbols (red/green) to distinguish between meta-variables ranging over different sets of syntactic constructs, and uses boldface text to denote lists.

are two 'magic names' $\{\texttt{sender}, \texttt{value}\} \subseteq \mathcal{N}$, and it is assumed that they are never used as formal parameters.

- $f, g \in \mathsf{MNames}$ is a set of function/method names. It is assumed that within each contract, all function names are unique.

Definition 69 (Syntax)**.** The syntax of TinySol is as follows:

$$
\begin{aligned}
e \in \mathsf{Exp} &::= v \mid x \mid X \mid \mathrm{op}\,\tilde{e} \mid ?e \mid !e \mid X : e \\
S \in \mathsf{Stm} &::= \texttt{skip} \mid \texttt{throw} \mid e_1 := e_2 \mid S_1;\, S_2 \\
&\quad \mid \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } e \texttt{ do } S \mid e_1 : f(\tilde{e})\,\$\,e_2
\end{aligned}
$$

■

The constructs in expressions e in particular differ markedly from ours, so we give some further explanations:

- $\mathrm{op}\,\tilde{e}$ are the usual unary and binary operators on integers and booleans, including cryptographic operators. Their semantics is well-known and is not described further in the text. It is also implicitly assumed in the text that expressions are well-typed w.r.t. these operators and have the appropriate arity.
- In the expression $?e$, e must evaluate to a key k. This is also implicitly assumed, as a consequence of expressions being well-typed. $?k$ then returns the value bound to k in the store of the present contract.
- Likewise, in $!e$, the expression e must also evaluate to a key k. The expression $!k$ then tests whether k is defined, and returns T (true) if k is bound in the store, and F (false) otherwise.
- $X : e$ evaluates e in the context of the store of the contract residing on address X, rather than the current contract. Specifically for keys, the notation $X.k$ is also used; this is termed a *qualified key*. The main use of this construct seems to be to access the `balance` of a foreign contract.
- In the assignment $e_1 := e_2$, e_1 must evaluate to a *key* k, whilst e_2 must evaluate to a *value* v.
- $e_1 : f(\tilde{e})\,\$\,e_2$ is a function call. Again, there are a number of assumptions on the evaluation of the three expressions:
  - e_1 must evaluate to an *address* X. It denotes the contract containing the definition of the function f to be called.
  - $\tilde{e}$ must evaluate to a list of values, of an arity matching the arity of the list of formal parameters of f.
  - e_2 must evaluate to an integer n. This number represents the amount of a digital asset, which is transferred along with the function call, from the caller to the callee.

A.2 Semantics

A *contract* $C \in \mathcal{C}$ is simply described as a set

$$\{ f_1(\widetilde{x_1})\{S_1\}, \ldots, f_n(\tilde{x}_n)\{S_n\} \}$$

of function names with parameter lists and function bodies. Furthermore, it is assumed that an *account* is simply a contract with a single, parameterless function $\{ f_a()\{\texttt{skip}\} \}$, which does nothing. Hence, calling this standard function allows other accounts, and contracts, to transfer assets to the account, without any other effects.

The semantics uses a number of environments, which are either partial or total functions:[2]

- $\mathsf{env}_C : \mathsf{ANames} \to \mathcal{C}$ is a global environment, which maps every address to a contract (i.e., either an account or an actual contract). As new contracts cannot be declared at runtime, this environment is simply assumed to exist and be populated with contracts prior to any execution.

- $\mathsf{env}_S : \mathsf{ANames} \to (\mathsf{Keys} \rightharpoonup \mathsf{Val})$ is a *state*, which maps every address to a key-value store.[3] If a key k is *not* bound to any value in the store of X, then the result of a lookup is defined to be the special value null, i.e. $\mathsf{env}_S X.k = \mathsf{null}$.[4] The auxiliary operators $\mathsf{env}_S + X : n$ and $\mathsf{env}_S - X : n$ are used to increment/decrement the `balance` of the contract at X by n units.

- $\mathsf{env}_P : \mathcal{N} \rightharpoonup \mathsf{Val}$ is an environment used to map formal parameters, and the two magic names `sender`, `value` to their actual values in a function call.

Lastly, $\sigma : \mathsf{ANames} \rightharpoonup (\mathsf{Keys} \rightharpoonup \texttt{value})$ is a *state update*, which is a substitution from qualified keys to values, written $\{v/X.k\}$.

The semantics of expressions is given as a denotation function $[\![e]\!]^X_{\mathsf{env}_{SP}}$, parametrised with an address X, which is the current evaluation context of e, and the two environments env_S and env_P, which we here abbreviate to env_{SP}. The equations are given in Figure A.1. It is assumed for all operators that if the value of any operand is null then the value of the entire expression is null. Thus, the domain of denotation of this function is the set $\mathsf{Val} \cup \{ \mathsf{null} \}$.

The semantics of statements is also given by a (partial) denotation function, but defined with inference rules as in an operational semantics. The domain of denotation is the set of states, i.e. the function returns an env_S. The rules are given in Figure A.2.

There are a few points to note about the semantics:

[2] We use env_C, env_S here, instead of Γ, σ which were used in the original presentation, to avoid clashes with symbols commonly used in type systems.

[3] In the original presentation, the signature is $\sigma : \mathsf{ANames} \to (\mathsf{Val} \rightharpoonup \mathsf{Val})$, but it is not clear why the set of values, rather than the more specific set of keys, is used.

[4] Again, it is not clear why the key-value store is not defined as a total function then, with null as the value for all keys that are not bound in the store.

$$
\begin{aligned}
[\![v]\!]^X_{\mathsf{env}_{SP}} &= v \\
[\![x]\!]^X_{\mathsf{env}_{SP}} &= \mathsf{env}_P\,(x) \\
[\![\mathrm{op}\,\tilde{e}]\!]^X_{\mathsf{env}_{SP}} &= \mathrm{op}[\![\tilde{e}]\!]^X_{\mathsf{env}_{SP}} \\
[\![Y\,:\,e]\!]^X_{\mathsf{env}_{SP}} &= [\![e]\!]^Y_{\mathsf{env}_{SP}} \\
[\![?e]\!]^X_{\mathsf{env}_{SP}} &= \mathsf{env}_S\left(X.[\![e]\!]^X_{\mathsf{env}_{SP}}\right) \\
[\![!e]\!]^X_{\mathsf{env}_{SP}} &= \begin{cases} \mathsf{T} & \text{if } [\![e]\!]^X_{\mathsf{env}_{SP}} \neq \mathsf{null} \wedge \mathsf{env}_S\left(X.[\![e]\!]^X_{\mathsf{env}_{SP}}\right) \neq \mathsf{null} \\ \mathsf{F} & \text{if } [\![e]\!]^X_{\mathsf{env}_{SP}} \neq \mathsf{null} \wedge \mathsf{env}_S\left(X.[\![e]\!]^X_{\mathsf{env}_{SP}}\right) = \mathsf{null} \end{cases}
\end{aligned}
$$

Figure A.1: Semantics of expressions.

- There is no rule for `throw`, meaning that $[\![\texttt{throw}]\!]^X_{\mathsf{env}_{SP}}$ has no denotation.
- If the loop in [WHILE$_T$] does not terminate, then the function has no denotation. This is used later in the paper [12], to model the effect of an out-of-gas exception (which is to roll back all effects of the call).
- There is no declaration of variables: Values can be stored in a key, via the assignment statement $e_1 := e_2$, if e_1 evaluates to a key k (the rule [ASS]). However, keys are never declared. This makes it difficult to create a typed version of the language, and to ensure that an expression actually will evaluate to a value, rather than to null.
- There is no real difference between a key k, and a name/formal parameter x, but they are nevertheless treated very differently. Formal parameters cannot appear on the left-hand side of an assignment, but keys can. Furthermore, formal parameters are stored in their own environment, env_P, whilst keys are stored in env_S. Not only does this restriction seem unnecessary; it also complicates the semantics by requiring an extra environment.

$$[\textsc{skip}]\ \frac{}{[\![\texttt{skip}]\!]^X_{\mathsf{env}_{SP}} = \mathsf{env}_S}$$

$$[\textsc{seq}]\ \frac{[\![S_1]\!]^X_{\mathsf{env}_{SP}} = \mathsf{env}_{S'}}{[\![S_1;\ S_2]\!]^X_{\mathsf{env}_{SP}} = [\![S_2]\!]^X_{\mathsf{env}_{S'P}}}$$

$$[\textsc{ass}]\ \frac{[\![e_1]\!]^X_{\mathsf{env}_{SP}} = k \quad [\![e_2]\!]^X_{\mathsf{env}_{SP}} = v}{[\![e_1 := e_2]\!]^X_{\mathsf{env}_{SP}} = \mathsf{env}_S\{v/X.k\}}$$

$$[\textsc{if}]\ \frac{[\![e]\!]^X_{\mathsf{env}_{SP}} = b \in \mathbb{B}}{[\![\texttt{if}\ e\ \texttt{then}\ S_\mathsf{T}\ \texttt{else}\ S_\mathsf{F}]\!]^X_{\mathsf{env}_{SP}} = [\![S_b]\!]^X_{\mathsf{env}_{SP}}}$$

$$[\textsc{while}_F]\ \frac{[\![e]\!]^X_{\mathsf{env}_{SP}} = \mathsf{F}}{[\![\texttt{while}\ e\ \texttt{do}\ S]\!]^X_{\mathsf{env}_{SP}} = \mathsf{env}_S}$$

$$[\textsc{while}_T]\ \frac{[\![e]\!]^X_{\mathsf{env}_{SP}} = \mathsf{T} \quad [\![S]\!]^X_{\mathsf{env}_{SP}} = \mathsf{env}_{S'}}{[\![\texttt{while}\ e\ \texttt{do}\ S]\!]^X_{\mathsf{env}_{SP}} = [\![\texttt{while}\ e\ \texttt{do}\ S]\!]^X_{\mathsf{env}_{S'P}}}$$

$$[\textsc{call}]\ \frac{[\![e_1]\!]^X_{\mathsf{env}_{SP}} = Y \quad [\![\widetilde{e_2}]\!]^X_{\mathsf{env}_{SP}} = \tilde{v} \quad [\![e_3]\!]^X_{\mathsf{env}_{SP}} = n}{[\![e_1 : f(\widetilde{e_2})\ \$\ e_3]\!]^X_{\mathsf{env}_{SP}} = [\![S]\!]^Y_{\mathsf{env}_{S'P'}}}$$

where:

$$n \leq \mathsf{env}_S(X.\texttt{balance})$$
$$f(\tilde{x})\{S\} \in \mathsf{env}_C(Y)$$
$$\mathsf{env}_{S'} = \mathsf{env}_S - X : n + Y : n$$
$$\mathsf{env}_{P'} = \{X/\texttt{sender}\}\{n/\texttt{value}\}\{\tilde{v}/\tilde{x}\}$$

Figure A.2: Semantics of statements.