Security Made Easy
- Proving Security using Type Inference

Morten Dahl
Department of Computer Science
Aalborg University

June 6, 2008

Abstract

In this paper we give a type inference algorithm for a type system for a pi calculus. The effect type system with dependent types guarantees that well-typed processes respect an authenticity property expressed as correspondences. Given a process and a type context we generate constraints that are satisfiable if and only if the process can be made well-typed under the context. Unification is used to solve type constraints and a non-deterministic algorithm is used to solve effect constraints. Effects capture correspondences and effect solving is essentially correspondence matching. The output of the algorithm can be used to efficiently construct a proof of safety in the form of a derivation tree. Experiments with a Caml implementation have shown that the algorithm performs well in praxis.

Preface

The following work constitutes the authors master thesis as part of the Masters Degree in Computer Science at Aalborg University. The thesis is largely self contained but the reader may want to be familiar with the pi calculus and type systems beforehand. The only real prerequisites are familiarity and comfort with mathematical formulation and presentation. Specifically, graduate students of computer science should have no trouble picking up this work.

Over the years I have developed a strong dislike to the fact that friends and family quite naturally have no clue about what I am working on. As a consequence, Chapter 1 accompanied by Appendix B hopefully provide a soft introduction to basic concepts.

Preparation for this work began while visiting the Laboratory for Foundations of Computer Science at the University of Edinburgh in fall 2007. The experience exceeded my expectation by any measure and my gratitude goes to Ian Stark for making it possible and having me as a visitor, for helping me make the most out of the stay, and for motivation and supervision.

This work was done at Aalborg University in spring 2008 under supervision of Hans Hüttel. I am grateful to Hans for his impressive dedication, motivation, and help in regards to the thesis, and for suggesting and making possible my visit to Scotland.

Finally, friends and family have made the less exciting experiences in the past five years much easier and enjoyable. My greatest gratitude goes to my parents for helping me in every way possible and for their endless support and understanding; it has without a doubt made a difference.

Contents

1 Introduction **7**
- 1.1 Type Systems . . . 8
- 1.2 Security Protocols . . . 9
- 1.3 Verifying Security Protocols . . . 11
- 1.4 Automated Proof Generation . . . 12

2 Typed Calculus **13**
- 2.1 A Pi Calculus with Correspondences . . . 13
- 2.2 Correspondence Certifying Type System . . . 18
- 2.3 Name-less Formulation of the Type System . . . 24

3 Type Inference **29**
- 3.1 Introducing Variables . . . 29
- 3.2 Constraint Generation . . . 34
- 3.3 Solving Type Constraints . . . 39
- 3.4 Solving Effect Constraints . . . 45
- 3.5 The Type Inference Algorithm . . . 52
- 3.6 Example . . . 53

4 Conclusion **57**
- 4.1 Summary . . . 57
- 4.2 Related Work . . . 58
- 4.3 Future Work . . . 60

A Proofs **63**
- A.1 Message Constraints . . . 63
- A.2 Process Constraints . . . 65
- A.3 Solving . . . 68

B Basic Type System Concepts **73**
- B.1 Simple Type System: Arithmetic Expressions . . . 73
- B.2 Type Inference: The Simply Typed Lambda Calculus . . . 75

Chapter 1

Introduction

In many areas, not limited to the world of computer science, we are interested in making sure our creations behave as we intend them to i.e. that they are correct and satisfy certain correctness properties. In computer science such a property could be that an algorithm produces the right output or that a program never enters a state of error. For many software programs, ensuring correctness is mostly done in the head of the programmer using informal methods, perhaps aided by a simple type system. Obviously, this leads to false proofs of correctness and has resulted in several formal methods being suggested over the years with the hope that they clarify and leave less space for hidden errors, or in some cases even give way to automated proof validation. Some of the formal methods are logics, denotational semantics, (bisimulation) equivalence, and model checking.

This paper will concentrate on another formal method, namely statically enforced type systems. In contrast to dynamically enforced methods, static methods determine if a system is well-behaved by examining the syntax without ever running it. In this paper we will use a type system to show how to statically ensure that security protocols do not go wrong. Our main result shows how type inference can be performed for this type system, effectively giving an algorithm for automated generation of proof of authenticity.

The aim of this chapter is to get things off the ground by providing a context for the discussion in the following chapters. We keep a focus on background ideas, principles, and intuition instead of technical details and formal descriptions. Concepts dealt with in the rest of this work are presented in a simple manner (see also Appendix B).

Chapter 2 introduces the calculus used for expressing protocols and the type system used to certify correspondences. Chapter 3 gives a type inference algorithm for the type system. Chapter 4 summaries our results, discusses related work and gives ideas for future work.

For this chapter we first introduce the general concept of a *type system* and a *security protocol*. In Section 1.3 and 1.4 we briefly discuss ways of verifying security protocols.

1.1 Type Systems

Type systems [10, 37] can be used as a way of restricting a set of expressions to a fragment or subset that is in some specific sense well-behaved, or equivalently, as a way of classifying data. We say that expressions allowed by the type system are well-typed, and type systems can be are used to guarantee that if an expression is well-typed then it has a certain desired property, e.g. no error state is reached during execution. Type systems typically also classify data by assigning the same type to similar expressions. A classical example is found in programming languages where type systems are used to reject expressions adding anything but numbers as well as to separate numbers from e.g. strings by giving numbers one type and strings another. Examples of other properties expressible by type systems are termination and containment within array bounds. Also, type systems can provide explicit documentation in the form of annotations, showing the classification of and relation between expressions as in "they are both integers" or "they are both objects exposing an operation with signature X".

Besides containing types some type systems also include effects [31] in order to capture what is perhaps not naturally expressed by a type. For instance, effects can be used to describe the intensional information about what takes place during evaluation of the program by e.g. collecting the set of storage cells written or read during execution [39], and collecting the regions in which evaluation takes place [40]; we see that effects in these cases are used to capture the *side effects* of the evaluation. Effects are the central idea in our type system so we shall see plenty more of them later.

From a more concrete point of view, a type system consist of a set of types (and effects) and a set of rules identifying just the set of well-typed expressions, and how types (and effects) can be assigned to these expressions. In this paper we will focus on formal type systems taking the form of a formal proof system. As we will see, automated type checking and type inference are possible in some systems.

An often important and desirable property of type systems is soundness (or safety). Soundness is defined upon evaluation rules and state two properties in regard to these: progress and preservation. Progress ensures that a well-typed expression is not in a deadlocked or error state ("finished" states are not deadlocked). Preservation (or subject reduction) ensures that a well-typed expression never evolves or take steps to a non-well-typed state.

Because of well-known undecidability results it is in general impossible for a computer to determine anything interesting about programs in finite time and without ever lying*. For instance, no computer can in the general case determine if a program terminates on not. For this reason no automated method can never be both sound and complete, and in particular no machine verifiable type system can be both sound and complete. This implies that sound type systems will always be incomplete and reject some well-behaved expressions (but also ensuring an endless stream of research in making methods less incomplete). This is known as the *slack* of a type system. Figure 1.1 show the relationship between well-behaved and

*A requirement is that Turing complete programs can be expressed in the analysed language.

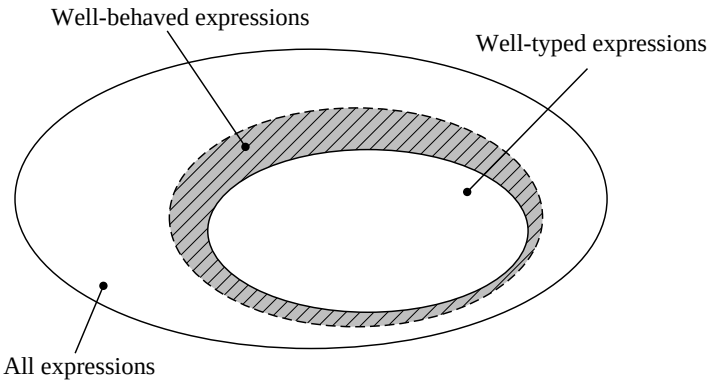

Figure 1.1: Relationship between well-behaved and well-typed expressions

well-typed expressions; the grey area is the slack of the type system, i.e. the wrongly rejected expressions.

1.2 Security Protocols

A *protocol* is an algorithm for two or more parties to perform a task in collaboration including a description of the communication between the parties. Whenever we set to download a webpage from the Internet the *HTTP* protocol is at work dictating how our computer and the server hosting the webpage should work together to perform the transfer. In the context of this paper, a more relevant protocol would be one that dictates how Alice can transfer an amount of currency, say £10, to Bob using a digital network. One way to build such a protocol would be to have Alice simply send a message to her bank telling them to "transfer £10 from Alice's account to Bob's". We write this as

$$\begin{aligned} A \to T: &\quad \text{Transfer } £10 \text{ to } B \\ T \to B: &\quad \text{Aquired } £10 \text{ from } A \end{aligned}$$

with the meaning that Alice (A) sends a message "Transfer £10 to B" to her bank T to transfer the amount to Bob (B). The last message is a notification from the bank to Bob to let him know about his newly gained credit. While this protocol can be used to settle debts it has a serious flaw: if Bob is dishonest he will record Alice's message on its way to the bank, so that whenever he is in need of money he simply replays the recorded message thereby gaining £10 each time.

This ability of Bob's to record messages sent from Alice to the bank is a typical assumption made by the security community known as the Dolev-Yao assumption [12]. Our threat model allows an attacker to not only listen in on conversations taking place between other parties but also to synthesise any message as will, limited only by the constraints of the cryptographic methods used. In other words, the attacker is the network so the protocol must alone provide the required security.

Returning to the sample protocol, we see that what the bank is lacking is a way to tell if it is Bob replaying Alice's old messages or if it really is Alice sending a transfer message, a property known as *authenticity*. One way to fix the above protocol would be to introduce a unique value N different for each message sent and whose only purpose is to ensure freshness of messages. Using such a *nonce* ("number used once") we get

$$\begin{array}{ll} T \to A: & N \\ A \to T: & \{\text{Transfer } £10 \text{ to } B, N\}_K \\ T \to B: & \text{Aquired } £10 \text{ from } A \end{array}$$

where $\{\dots\}_A$ means that Alice encrypts the message using a key K shared only by the bank and Alice. Upon receiving a transfer-message the bank verifies it by decrypting it using K and continues with the transfer (and the notification) only if verification succeeds and the nonce N matched what the bank sent in the first step. Note that encryption alone is not enough nor is the use of only a nonce: the nonce has to be attached to the message in a way that associates it with Alice.

Other interesting properties besides authenticity are *secrecy* (preventing eavesdropping), *integrity* (preventing modification), and *authorisation* (controlling access). Protocols whose main focus is on these properties are *security protocols* and typically employ some form of cryptography such as encryption or commitment. The main focus of this work is on ensuring authenticity.

The notation used above for specifying protocols are standard for informal specifications. As our work focuses on formal verification we need a formal specification. A language for this is the pi calculus [32] and its descendant the spi calculus [3, 4]. While the pi calculus can be used for expressing protocols in general, the spi calculus is designed specifically for the purpose of expressing security protocols by providing operations modelling cryptography such as encryption and decryption. Many details of cryptography are abstracted away. For instance, a message M is encrypted under key K by $\{M\}_K$ without giving any attention to the actually encryption function. Both calculi come in many different versions and both aim to be an executable specification fairly close to implementation. In Chapter 2 we present the variant of the pi calculus used for our analysis.

Expressed in the spi calculus the protocol from above could look like

$$\begin{array}{ll} A\ (Alice) = & T\ (Bank) = \\ \quad \mathsf{in}\ net\ nonce; & \quad \mathsf{new}\ nonce; \\ \quad \mathsf{new}\ transfer; & \quad \mathsf{out}\ net\ nonce; \\ \quad \mathsf{out}\ net\ \{transfer, nonce\}_A & \quad \mathsf{in}\ net\ ctext; \\ & \quad \mathsf{decrypt}\ ctext \text{ using } K \text{ as } msg; \\ & \quad \text{perform checks on } msg \dots \end{array}$$

where the system is $A \mid T$, i.e. A running in parallel with T.

1.3 Verifying Security Protocols

If the first step in building a security protocol is to design it to ensure e.g. authenticity, then the next step is to make sure it really satisfies this property, i.e. that the protocol does not contain a design flaw. An often used approach to ensure no such flaw exists is a thorough argumentation but with no formal definition of the protocol nor of the desired properties. In these cases both definitions and statements can be thought valid for an extended period of time and still turn out to be insufficient or wrong. One example is the Needham-Schroeder public-key authentication protocol [33] which was thought safe for several years before discovering that it contained a security flaw [30]. More serious methods turn to mathematics and give somewhat formal definitions but informal proofs. Examples of these include a method based on belief logic [9] and the notion of zero-knowledge [11].

Highly formal methods model the protocol in a formal language with a formal semantics, and give formal definitions of security. One common method is to model the protocol in a process calculus such as pi, spi, or applied pi [2, 15], and define, say, secrecy using *may testing* or other forms of behavioural equivalence. More relevant to this work, authenticity can be specified and verified using *correspondences* in the form of *begin-* and *end-events* as suggested by Woo and Lam [41]. End-events capture the idea of asserting that we believe an action from other part of the protocol has occurred. By proving that each end-event has a matching begin-event we prove that the action was in fact caused by the protocol. Consider the simple protocol

$$\begin{array}{l|l} \mathsf{new}\ msg; & \mathsf{in}\ net\ x; \\ \quad \mathsf{begin}\ sent(msg); & \quad \mathsf{end}\ sent(x) \\ \quad \mathsf{out}\ net\ msg & \end{array}$$

with a sender (left) and a receiver (right) running in parallel. With the begin- and end-events annotations we have expressed the authenticity property that when the receiver receives a message it should have been sent by the sender. For the protocol to satisfy the authenticity property it must ensure that for every run of the protocol the correspondence property is respected, i.e. whenever an end-event is encountered there must have been a matching and preceeding begin-event. In the example, if *net* is a public network then the property is not satisfied since anyone could have sent the message and hence in some runs the $\mathsf{end}\ sent(x)$ is not preceded by a begin-event. If, however, the network were shared only by the sender and the receiver the property would be satisfied.

Correspondences can be injective or non-injective [21]. Authenticity properties expressed using injective correspondences state that each begin-event can only be used to match one end-event, whereas non-injective correspondences allow a begin-event to match and hence satisfy one or more end-event.

No matter how formal the definitions of security are, informal proofs can still turn out to be wrong. To combat this, formal methods allowing computer verification of proofs have been suggested, some of which are mentioned in Section 4.2. As we shall see, type systems can be used to provide a highly formal method that in some cases allow for automated proof

verification in the form of type checking. Gordon and Jeffrey [18] pioneered the verification of correspondences using type systems and we return to this Section 4.2 on related work.

1.4 Automated Proof Generation

While some type systems allow for automated verification in the form of automated type checking, producing the proof in form of a type assignment can be time consuming and clutter the language with annotations. In programming languages the ML family is a prime example of avoiding annotations: while the language has a statically enforced type system, no type annotations are needed. Instead the compiler inferes a type assignment during compilation.

A common way of doing type inference (i.e. proof generation) is via constraint generation and solving. For constraint generation the type checking algorithm is run in a reversed order recording type checks as constraints instead of doing actual checks. The goal is for the generated constraints to have a solution if and only if a type assignment exists. A solving algorithm is then invoked on the generated constraint with the goal of yielding a solution if such exist. In the best case a principal solution, i.e. a most general solution satisfying the type rules, is produced. As we shall see, unification in the form of a rewrite system can in some cases be used for solving constraints.

As mentioned above, the main contribution of this paper is how to perform type inference for a type system ensuring authenticity. In our method, constraints either express relationship between types or between effects. Unification is used for solving type constraints and a nondeterministic method is used for solving effect constraints, giving a sound and complete type inference algorithm. Ultimately we provide a method rejecting all flawed protocol (and some safe protocols as well due to incompleteness in the type system). Chapter 2 and 3 goes into more detail.

Chapter 2

Typed Calculus

This chapter presents our calculus and type system of choice. Both are taken from [17] with a few minor modifications. Though not formally proved we conjecture that the results, in particular the safety results hold for this modified version as well.

The calculus in Section 2.1 is a variant of the pi calculus with non-injective (one-to-many) correspondences. The type system in Section 2.2 includes name binding pair types and *ok* types with effects and messages. Section 2.3 gives an equivalent but more manageable formulation of the type system.

On a technical note, we follow standard terminology and say that a relation $\sim$ is an *equivalence* if it is reflexive ($x \sim x$), symmetric (if $x \sim y$ then $y \sim x$), and transitive (if $x \sim y$ and $y \sim z$ then $x \sim z$). Furthermore, an equivalence $\sim$ is a *congruence* on a set if it is closed under the contexts of the set.

2.1 A Pi Calculus with Correspondences

In this section we introduce the calculus for expressing protocols. We start by defining messages, followed by effects used in correspondence annotations. The section ends by defining processes and process safety.

2.1.1 Messages

The most primitive part of the calculus is that of a message. We assume two countable disjoint sets of names and variables and let the set of *messages* be given by:

$M ::=$	message
n, m	name
x, y	variable
$\mathsf{fst}\ M$	projection of first component
$\mathsf{snd}\ M$	projection of second component
(M_1, M_2)	message pair
ok	ok annotation

Besides the standard names n and variables x we have pairs (M_1, M_2) and operators $\mathsf{fst}\ M$ and $\mathsf{snd}\ M$ for projecting the first and second component, respectively, of a pair. In addition we also have a special ok message. As in previous systems, the only purpose of this construct is as annotation for the type system, allowing the transfer of begin correspondence events (by populating $\mathsf{Ok}(S)$ types as introduced in Section 2.2). As no types are specified on ok messages it is not a type annotation in the traditional sense, and since the type system is statically enforced ok messages have no influence on the evaluation of a process and can be ignored after type checking.

Note, that only in this section do we make a distinction between names and variables (unless otherwise stated). For our analysis message names and variables are treated the same so in the rest of this paper we use names to mean both names and variables.

To match messages, i.e. determine if two messages are equivalent, we need an equivalence relation that, besides respecting the standard equivalence rules, also captures the semantics of the projection messages; for instance, we should have $\mathsf{fst}\ (M_1, M_2) \equiv M_1$.

Definition 1 (**Message Equivalence:** $M_1 \equiv M_2$)**.** Define *message equivalence* $\equiv$ to be the least congruence on messages closed under projection rewrite rules

$$\frac{}{\mathsf{fst}\ (M_1, M_2) > M_1}\ \text{MR-FST} \qquad \frac{}{\mathsf{snd}\ (M_1, M_2) > M_2}\ \text{MR-SND}$$

We say messages M_1 and M_2 are *equivalent* if $M_1 \equiv M_2$.

As we shall see messages equivalence is used both for matching channel names as well as for matching correspondence events. Note that the rewrite rules are strongly normalising so $\equiv$ is decidable.

For the type system introduced in the next section we need a way to substitute message variables inside types that contain messages. We introduce a *message instantiation* operation $\rightarrow$ that recursively applies a *message substitution* $\langle M/x \rangle$ to a message M' with the indend of replacing all occurences of x in M' with M:

$$\begin{array}{rcl}
n\langle M/x\rangle & \rightarrow & n \\
x'\langle M/x\rangle & \rightarrow & \begin{cases} M & \text{if } x = x' \\ x' & \text{otherwise} \end{cases} \\
\mathsf{ok}\langle M/x\rangle & \rightarrow & \mathsf{ok} \\
(\mathsf{fst}\ M')\langle M/x\rangle & \rightarrow & \mathsf{fst}\ (M'\langle M/x\rangle) \\
(\mathsf{snd}\ M')\langle M/x\rangle & \rightarrow & \mathsf{snd}\ (M'\langle M/x\rangle) \\
(M_1, M_2)\langle M/x\rangle & \rightarrow & (M_1\langle M/x\rangle, M_2\langle M/x\rangle)
\end{array}$$

Messages have no binding constructs so the set of free names and variables $fn(M')$ in a

message M' is trivially defined as:

$$\begin{aligned} fn(n) &= \{n\} \\ fn(x) &= \{x\} \\ fn(\mathsf{ok}) &= \emptyset \\ fn(\mathsf{fst}\ M) &= fn(M) \\ fn(\mathsf{snd}\ M) &= fn(M) \\ fn((M_1, M_2)) &= fn(M_1) \cup fn(M_2) \end{aligned}$$

2.1.2 Effects

Next we define the objects to be used in correspondence events. We assume a countable set of labels distinct from the set of messages, and denoted members by l. We also assume a decidable equivalence relation $=$ on labels. Let an *atomic effect* be a pair (l, M) written as $l(M)$ and let an *effect* S be a set of atomic effects $\{l_1(M_1), \ldots, l_n(M_n)\}$. We shall sometimes refer to effects as *credit*.

We say atomic effect $l_1(M_1)$ and $l_2(M_2)$ are equivalent if $l_1 = l_2$ and $M_1 \equiv M_2$ and denote this $l_1(M_1) \equiv l_2(M_2)$. This is the central relation in corresponding matching. We extend equivalence to effects by the normal rules of set equality and write $S_1 \equiv S_2$.

Message instantiation is extended to atomic effects in the obvious ways and to effects pointwise. Similar for $fn(l(M))$ and $fn(S)$.

2.1.3 Processes

Using messages and effects we can define processes. Let the set of *processes* be given by:

$P ::=$	process
$\mathsf{in}\ M\ x;\ P$	input of n on M
$!\mathsf{in}\ M\ x;\ P$	repeated input
$\mathsf{out}\ M_1\ M_2$	output of M_2 on M_1
$\mathsf{new}\ n;\ P$	restriction of n
$P_1 \mid P_2$	parallel composition
nil	inactivity
$\mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$	conditional
$\mathsf{exercise}\ M; P$	exercise an ok
$\mathsf{begin}\ l(M)$	begin-event
$\mathsf{end}\ l(M)$	end-event

Most of the constructs are well-known [32] perhaps with the exception of $\mathsf{exercise}\ M; P$ and the correspondence events $\mathsf{begin}\ l(M)$ and $\mathsf{end}\ l(M)$. Neither the exercise construct nor the events have any influence during process evaluation. Events are used as annotations in specifying the authenticity property and exercise is an annotation used by the type system to verify the property. Note that replicated input $!\mathsf{in}\ M\ x;\ P$ is enough to encode general replication.

Consider process

$$\begin{array}{l|l} \mathsf{new}\ n; & \mathsf{in}\ \mathit{net}\ x; \\ \quad \mathsf{begin}\ \mathit{sent}(n)\ | & \quad \mathsf{exercise\ snd}\ x; \\ \quad \mathsf{out}\ \mathit{net}\ (n, \mathsf{ok}) & \quad \mathsf{end}\ \mathit{sent}(\mathsf{fst}\ x) \end{array}$$

consisting of a sender (left) and a receiver (right) running in parallel. The sender generates a message n and outputs this on the shared network *net*. The receiver simply inputs from the shared network. The end-event at the receiver is matched by the begin-event at the sender. Because of the ok message the type of *net* will dictate that it "costs" $\mathit{sent}(n)$ to send on the network, which the sender can pay for by begin $\mathit{sent}(n)$. The exercise snd x in the receiver uses the credit paid for by the sender to match end-event end $\mathit{sent}(\mathsf{fst}\ x)$. We shall return to this in Section 2.2.

We give a formal definition of the evaluation semantics of processes as a standard reduction semantics. To do this we first introduce the notion of free names and variables in a process and a equivalence relation on processes.

Let the free names and variables $\mathit{fn}(P)$ of a process P be defined by

$$\begin{array}{rcl} \mathit{fn}(\mathsf{in}\ M\ x;\ P) & = & \mathit{fn}(M) \cup (\mathit{fn}(P) - \{x\}) \\ \mathit{fn}(!\mathsf{in}\ M\ x;\ P) & = & \mathit{fn}(M) \cup (\mathit{fn}(P) - \{x\}) \\ \mathit{fn}(\mathsf{out}\ M_1\ M_2) & = & \mathit{fn}(M_1) \cup \mathit{fn}(M_2) \\ \mathit{fn}(\mathsf{new}\ n;\ P) & = & \mathit{fn}(P) - \{n\} \\ \mathit{fn}(P_1 \mid P_2) & = & \mathit{fn}(P_1) \cup \mathit{fn}(P_2) \\ \mathit{fn}(\mathsf{nil}) & = & \emptyset \\ \mathit{fn}(\mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2) & = & \mathit{fn}(M_1) \cup \mathit{fn}(M_2) \cup \mathit{fn}(P_1) \cup \mathit{fn}(P_2) \\ \mathit{fn}(\mathsf{exercise}\ M; P) & = & \mathit{fn}(M) \cup \mathit{fn}(P) \\ \mathit{fn}(\mathsf{begin}\ l(M)) & = & \mathit{fn}(M) \\ \mathit{fn}(\mathsf{end}\ l(M)) & = & \mathit{fn}(M) \end{array}$$

where we see that only input and restriction bind variables and names; we say that a *bound name* in a process is a name or variable bound by input or restriction.

We want the renaming of bound names and variables to make no difference. For instance, processes

$$\mathsf{new}\ n_1;\ \mathsf{end}\ l(n_1) \qquad \text{and} \qquad \mathsf{new}\ n_2;\ \mathsf{end}\ l(n_2)$$

should be equivalent as the bound names (n_1 and n_2, respectively) are local. For this reason we make process equivalence respect alpha conversion:

Definition 2 (**Structural Congruence:** $P_1 \equiv P_2$)**.** Define *structural congruence* $\equiv$ to be the least congruence on processes closed under alpha conversion and axioms in Table 2.1.

We can now give the process evaluation semantics in the form of a reduction semantics:

Definition 3 (**Process Reduction:** $P_1 \to P_2$)**.** Define *process reduction* $\to$ to be the least relation on processes closed under axioms in Table 2.2.

$$\frac{}{P \equiv P \mid \mathsf{nil}} \quad \text{PE-PAR-NIL}$$

$$\frac{}{P_1 \mid P_2 \equiv P_2 \mid P_1} \quad \text{PE-PAR-COMM}$$

$$\frac{}{P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3} \quad \text{PE-PAR-ASSOC}$$

$$\frac{n \notin fn(P_1)}{\mathsf{new}\ n;\ (P_1 \mid P_2) \equiv P_1 \mid (\mathsf{new}\ n;\ P_2)} \quad \text{PE-RES-PAR}$$

$$\frac{}{\mathsf{new}\ n;\ \mathsf{nil} \equiv \mathsf{nil}} \quad \text{PE-RES-NIL}$$

$$\frac{}{\mathsf{new}\ n_1;\ \mathsf{new}\ n_2;\ P \equiv \mathsf{new}\ n_2;\ \mathsf{new}\ n_1;\ P} \quad \text{PE-RES-ORDER}$$

Table 2.1: Structural congruence rules

$$\frac{P \equiv P' \qquad P' \to P'' \qquad P'' \equiv P'''}{P \to P'''} \quad \text{PR-EQ}$$

$$\frac{}{\mathsf{exercise}\ M; P \to P} \quad \text{PR-EX}$$

$$\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2} \quad \text{PR-PAR}$$

$$\frac{P \to P'}{\mathsf{new}\ n;\ P \to \mathsf{new}\ n;\ P'} \quad \text{PR-RES}$$

$$\frac{M_1 \equiv M_1'}{\mathsf{out}\ M_1\ M_2 \mid \mathsf{in}\ M_1'\ x;\ P \to P\{M_2/x\}} \quad \text{PR-COMM}$$

$$\frac{M_1 \equiv M_1'}{\mathsf{out}\ M_1\ M_2 \mid !\mathsf{in}\ M_1'\ x;\ P \to P\{M_2/x\} \mid !\mathsf{in}\ M_1'\ x;\ P} \quad \text{PR-RCOMM}$$

$$\frac{M \equiv M'}{\mathsf{if}\ M = M'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \to P_1} \quad \text{PR-IF-T}$$

$$\frac{M \not\equiv M'}{\mathsf{if}\ M = M'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \to P_2} \quad \text{PR-IF-F}$$

Table 2.2: Process reduction rules

Note that rule PR-EX simply ignore the exercise process $\mathsf{exercise}\ M; P$ and skips to P. In the next section we shall see that it plays a central part in the type system.

Let $\rightarrow^*$ be the transitive closure of $\rightarrow$ and write $P \rightarrow^*_{\equiv} P'$ if $P \rightarrow^* P'$ or $P \equiv P'$. We define safety as

Definition 4 (**Safety**). A process P is *safe* if whenever $P \rightarrow^*_{\equiv} \mathsf{new}\ \tilde{n};\ (\mathsf{end}\ l(M) \mid P')$ we have M' and P'' such that $P' \equiv \mathsf{begin}\ l(M') \mid P''$ and $M \equiv M'$.

where $\mathsf{new}\ \tilde{n};\ P$ is short for zero or more restrictions in front of P and part of the definition in order to allow events to contain restricted names.

2.2 Correspondence Certifying Type System

In this section we present the type system for statically verifying that a process is safe with respect to the correspondence events occuring in it. We first define types and type equivalence. Then we define typing contexts used during type checking to provide a type for free names as well as to store effects. All judgments are relative to a typing context. After presenting the typing rules we state the main property of the type system, namely that well-typed processes are safe. In the next section we provide an alternative but equivalent definition of the type system avoiding the need for alpha conversion in type equivalence.

Starting from this section we assume the uniqueness of name declarations, e.g. no occurrences of $\mathsf{new}\ n;\ \mathsf{new}\ n;\ P$ or $(\mathsf{new}\ n;\ P_1) \mid (\mathsf{new}\ n;\ P_2)$. This is of no loss of generality as any process can be transformed to one with unique names using alpha conversion.

2.2.1 Types

Let the set of *types* be defined by:

$T ::=$	type
A	base type
$\mathsf{Ch}(T)$	channel for messages of type T
$\mathsf{Pair}(x : T_1, T_2)$	pair type
$\mathsf{Ok}(S)$	effect carrier type

Types A and $\mathsf{Ch}(T)$ are standard: A is an element from a set of base or atomic types, i.e. a set of unspecified types with no operations of importance in the context of this work. Obvious members of this set could be Un (i.e. "untrusted" from [1]), Data, or Int. $\mathsf{Ch}(T)$ denotes a channel type for sending messages of type T.

Type $\mathsf{Ok}(S)$ is a dependent type containing effects and in turn messages. It is inhabited only by the special ok message and a popular phrasing of its purpose is "to transfer credit gained from begin-events (from one place to another)". For instance, type $\mathsf{Ok}(\{l(M)\})$ can be assigned to message ok only if the process containing ok has a $l(M)$ at its disposal. In other terms, it "costs" $l(M)$ to type the message. By exercising a typed ok message we gain the credit stored in the type and paid for when typing the message. Note that this use of effects differs from how they are typically employed in type systems, where they are not contained in the types (as they are in the ok types of our system) but appear outside and

in addition to the types.

Type $\mathsf{Pair}(x : T_1, T_2)$ is a pair type. It binds a message variable x and is used for typing message pairs (M_1, M_2) with the first projection M_1 having type T_1 and the second projection M_2 having type T_2 with x replaced by M_1. As always with binding we want to be able to rename bound variables without it making any difference. For instance, type

$$\mathsf{Pair}(x : \mathsf{A}, \mathsf{Ok}(\{l(x)\})) \qquad \text{and type} \qquad \mathsf{Pair}(y : \mathsf{A}, \mathsf{Ok}(\{l(y)\}))$$

should be equivalent. This is reflected in the definition of type equivalence below by the fact that alpha conversion is respected*.

We extend the message instantiation operation $\rightarrow$ to types:

$$\begin{array}{rcl} \mathsf{A}\langle M/x\rangle & \rightarrow & \mathsf{A} \\ \mathsf{Ch}(T)\langle M/x\rangle & \rightarrow & \mathsf{Ch}(T\langle M/x\rangle) \\ \mathsf{Pair}(y : T_1, T_2)\langle M/x\rangle & \rightarrow & \mathsf{Pair}(y : T_1\langle M/x\rangle, T_2\langle M/x\rangle) \text{ where } x \not\equiv y \\ \mathsf{Ok}(S)\langle M/x\rangle & \rightarrow & \mathsf{Ok}(S\langle M/x\rangle) \end{array}$$

Note that since pair types bind variables we may need to alpha convert the bound variable in the pair type before applying the substitution to the second component in order to avoid capture. This is no restriction since the bound variable can always be renamed using alpha conversion of types.

Let us look at an example. Recall that we chose to only consider processes with unique name declarations. We did this as to not confuse messages with each other during the static analysis. Furthermore, note that event matching relies on message equivalence and is hence highly syntax oriented (a begin event $l(M_1)$ matches an end event $l(M_2)$ if and only if $M_1 \equiv M_2$). This is also because we are doing a static analysis. Now, having messages insides types imposes problems. One is that these messages may use local names which would be unbound if the containing type is used outside the scope of the names. For instance, in the protocol from earlier

$$\begin{array}{l|l} \mathsf{new}\ n; & \mathsf{in}\ \mathit{net}\ x; \\ \quad \mathsf{begin}\ \mathit{sent}(n)\ | & \quad \mathsf{exercise\ snd}\ x; \\ \quad \mathsf{out}\ \mathit{net}\ (n, \mathsf{ok}) & \quad \mathsf{end}\ \mathit{sent}(\mathsf{fst}\ x) \end{array}$$

the first projection of the message pair sent across the network is known syntactically as n to the sender while it is known as $\mathsf{fst}\ x$ to the receiver. In typing *net* we cannot use n nor x as both are local. And even if we could, we would have a problem matching end event $\mathit{sent}(\mathsf{fst}\ x)$ with $\mathit{sent}(n)$. To remedy this we need a mechanism for temporarily abstracting out of types the static identity of messages inside effects so that they can later be given an identity suitable for the scope in which they are used. This is done using pair types.

*The additional β and η rules found in the definition of λ-term equivalence are not needed since we cannot apply a type to a type.

Using pair type we can give name *net* type $\mathsf{Ch}(\mathsf{Pair}(y : \mathsf{A}, \mathsf{Ok}(\{sent(y)\})))$. Then (n, ok) and x can both be typed $\mathsf{Pair}(y : \mathsf{A}, \mathsf{Ok}(\{sent(y)\}))$. However, in "building up" and "breaking down" the type of the message pair, the message in the first component of the pair is respectively abstracted out of and instantiated into the type of the second component. Concretely, we have that ok on the side of the sender can be typed as $\mathsf{Ok}(\{sent(y)\})\langle n/y\rangle \rightarrow \mathsf{Ok}(\{sent(n)\})$ and $\mathsf{snd}\ x$ on the side of the receiver as $\mathsf{Ok}(\{sent(y)\})\langle \mathsf{fst}\ x/y\rangle \rightarrow \mathsf{Ok}(\{sent(\mathsf{fst}\ x)\})$. This is done in the typing rules for pairs and second projections respectively.

Since types can contain messages which in turn have free names we define the free names $fn(T)$ of a type T by:

$$\begin{aligned} fn(\mathsf{A}) &= \emptyset \\ fn(\mathsf{Ch}(T)) &= fn(T) \\ fn(\mathsf{Pair}(x : T_1, T_2)) &= fn(T_1) \cup (fn(T_2) - \{x\}) \\ fn(\mathsf{Ok}(S)) &= fn(S) \end{aligned}$$

where we see that pair types bind a variable in the second component.

Type equivalence is the relation used for type checking and hence the relation type inference aim to agree with. We define:

Definition 5 (Type Equivalence: $T_1 \equiv T_2$). Let *type equivalence* be the least congruence on types closed under alpha conversion and axiom

$$\frac{S_1 \equiv S_2}{\mathsf{Ok}(S_1) \equiv \mathsf{Ok}(S_2)}\ \text{TE-OK}$$

Intuitively, type equivalence relates identitical types up to message equivalence. All relations, in particular matching of events, respect message equivalence and hence equivalent types can be interchanged in all important aspects.

2.2.2 Typing Contexts

All judgments are relative to a typing context providing types for free names as well as keeping account of effects. Let a typing context Γ be defined by:

$\Gamma ::=$	typing context
$\emptyset$	empty context
$\Gamma, n : T$	name typing
Γ, S	effect

and let the domain of a typing context $dom(\Gamma)$ be defined by:

$$\begin{aligned} dom(\emptyset) &= \emptyset \\ dom(\Gamma, n : T) &= dom(\Gamma) \cup \{n\} \\ dom(\Gamma, S) &= dom(\Gamma) \end{aligned}$$

We use a function $\mathit{effects}(\Gamma)$ to collect effects stored in a typing context Γ:

$$\begin{aligned} \mathit{effects}(\emptyset) &= \emptyset \\ \mathit{effects}(\Gamma, n : T) &= \mathit{effects}(\Gamma) \\ \mathit{effects}(\Gamma, S) &= \mathit{effects}(\Gamma) \cup S \end{aligned}$$

Finally, we say that a context Γ is *well-formed* if $\Gamma \vdash \diamond$ can be inferred from the rules in Table 2.3. Rule WE-TY requires that well-formed contexts only have one binding for each name ($n \notin \mathit{dom}(\Gamma)$). This is no restriction as we have assumed uniqueness of bound names. Well-formed contexts furthermore ensure that any free names occuring in types and effects in the context are accounted for in earlier bindings.

$$\frac{}{\emptyset \vdash \diamond} \quad \text{WF-EMP}$$

$$\frac{\Gamma \vdash \diamond \qquad \mathit{fn}(S) \subseteq \mathit{dom}(\Gamma)}{\Gamma, S \vdash \diamond} \quad \text{WF-EF}$$

$$\frac{\Gamma \vdash \diamond \qquad n \notin \mathit{dom}(\Gamma) \qquad \mathit{fn}(T) \subseteq \mathit{dom}(\Gamma)}{\Gamma, n : T \vdash \diamond} \quad \text{WF-TY}$$

Table 2.3: Rules for well-formed typing contexts

2.2.3 Typing Rules for Messages

The first part of the typing rules is concerned with assigning types to messages. We type messages according to rules in Table 2.4 and say that a message M is well-typed relative to Γ if a derivation tree with root $\Gamma \vdash M : T$ can be inferred for some type T using the rules.

$$\frac{\Gamma \vdash \diamond \qquad n : T \in \Gamma}{\Gamma \vdash n : T} \quad \text{MT-NAME}$$

$$\frac{\Gamma \vdash \diamond \qquad S \subseteq \mathit{effects}(\Gamma)}{\Gamma \vdash \mathsf{ok} : \mathsf{Ok}(S)} \quad \text{MT-OK}$$

$$\frac{\Gamma \vdash M : \mathsf{Pair}(x : T_1, T_2)}{\Gamma \vdash \mathsf{fst}\ M : T_1} \quad \text{MT-FST}$$

$$\frac{\Gamma \vdash M : \mathsf{Pair}(x : T_1, T_2)}{\Gamma \vdash \mathsf{snd}\ M : T_2\langle \mathsf{fst}\ M/x\rangle} \quad \text{MT-SND}$$

$$\frac{\Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash M_2 : T_2\langle M_1/x\rangle}{\Gamma \vdash (M_1, M_2) : \mathsf{Pair}(x : T_1, T_2)} \quad \text{MT-PAIR}$$

Table 2.4: Typing rules for messages

Rule MT-PAIR allows a message M_1 to be abstracted out of type T_2 in the sense that message (M_1, M_2) is typable with type $\mathsf{Pair}(x : T_1, T_2)$ if M_2 is typable with T_2 instantiated with M_1. Dually, in rule MT-SND message $\mathsf{snd}\ M$ is typable with type T_2 instantiated with message $\mathsf{fst}\ M$ (for x). Rule MT-OK allows an ok to be typed with an $\mathsf{Ok}(S)$ only if all of S can be found in the context; intuitively, if the context can pay for S.

Observe that we used application to describe both an application (MT-SND) and abstraction (MT-PAIR) relation between types. Alternatively we could have used an abstraction operation $(x)_M$ and expressed the type equivalence in rule MT-PAIR as $(M_1, M_2) : \mathsf{Pair}(x : T_1, T_2(x)_M)$ for $M_1 : T_1$ and $M_2 : T_2$. Contrary to application, the abstraction operation is non-deterministic in the general case since it may choose to abstract out only some occurrences of the message. For instance, to type $(n, (n, \mathsf{ok}))$ we would have

$$\mathsf{Pair}\Big(x : N, \mathsf{Pair}\big(y : N, O(y)_n\ \big)(x)_n\Big)$$

where O is $\mathsf{Ok}(\{l((n, n))\})$. Both abstraction operations abstract away the same message n so we would have to non-deterministically guess what the resulting atomic effect in the type is: $l((x, y))$, $l((y, x))$, $l((y, y))$, etc. are all possibilities. To remedy this we could assume the *principle of maximal abstraction* where the abstraction operation abstracts as much as possible by replacing all occurences of the message with the variable. Abstraction becomes deterministic and we have that the only option for $\big(O(y)_n\big)(x)_n$ is $\mathsf{Ok}(\{l((y, y))\})$. For describing the type system and for type checking we stick to only using application. For type inference we need to also perform abstraction, and we provide a general algorithm supporting both non-deterministic and deterministic abstraction. We return to this in the next chapter.

2.2.4 Typing Rules for Processes

For the typing of processes we replace construct $\mathsf{new}\ n;\ P$ with an annotated version $\mathsf{new}\ n : T;\ P$. Input processes get the type of the input from the type of the channel so no annotated version is needed here.

Before giving the process typing rules we need a function to collect credit available for a process. Let the accessible credit $\mathit{begins}(P)$ for a process P be defined by:

$$\begin{aligned}
\mathit{begins}(\mathsf{begin}\ l(M)) &= \{l(M)\} \\
\mathit{begins}(\mathsf{end}\ l(M)) &= \emptyset \\
\mathit{begins}(\mathsf{new}\ n : T;\ P) &= \{l(M) \in \mathit{begins}(P) \mid n \notin \mathit{fn}(M)\} \\
\mathit{begins}(P_1 \mid P_2) &= \mathit{begins}(P_1) \cup \mathit{begins}(P_2) \\
\mathit{begins}(P) &= \emptyset \text{ for any other } P
\end{aligned}$$

Let a type T be *generative* if and only if it is a channel type, i.e. $T = \mathsf{Ch}(T')$ for some type T'. We can now present the typing rules for processes in Table 2.5. Rule PT-PAR is seen to collect credit in parallel processes. Rule PT-EX makes the credit stored in the type of M available to P, and rule PT-END requires that $l(M)$ is among the available credit.

$$\frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma, n : T \vdash P}{\Gamma \vdash \mathsf{in}\ M\ n;\ P} \quad \text{PT-IN}$$

$$\frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma, n : T \vdash P}{\Gamma \vdash !\mathsf{in}\ M\ n;\ P} \quad \text{PT-REIN}$$

$$\frac{\Gamma \vdash M_1 : \mathsf{Ch}(T) \qquad \Gamma \vdash M_2 : T}{\Gamma \vdash \mathsf{out}\ M_1\ M_2} \quad \text{PT-OUT}$$

$$\frac{\Gamma, n : T \vdash P \qquad T \text{ generative}}{\Gamma \vdash \mathsf{new}\ n : T;\ P} \quad \text{PT-NEW}$$

$$\frac{\Gamma, \mathit{begins}(P_2) \vdash P_1 \qquad \Gamma, \mathit{begins}(P_1) \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \quad \text{PT-PAR}$$

$$\frac{\Gamma \vdash M : \mathsf{Ok}(S) \qquad \Gamma, S \vdash P}{\Gamma \vdash \mathsf{exercise}\ M; P} \quad \text{PT-EX}$$

$$\frac{\Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash M_2 : T_2 \qquad \Gamma \vdash P_3 \qquad \Gamma \vdash P_4}{\Gamma \vdash \mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_3\ \mathsf{else}\ P_4} \quad \text{PT-IF}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{nil}} \quad \text{PT-NIL}$$

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash \mathsf{begin}\ l(M)} \quad \text{PT-BEGIN}$$

$$\frac{\Gamma \vdash M : T \qquad l(M) \in \mathit{effects}(\Gamma)}{\Gamma \vdash \mathsf{end}\ l(M)} \quad \text{PT-END}$$

Table 2.5: Typing rules for processes

With these rules we can now state the main result of the type system, that well-typed processes are safe. Let a context Γ be generative if all types T such that $x : T \in \Gamma$ are generative.

Theorem 6. Assume typing context Γ is generative and $\mathit{effect}(\Gamma) = \emptyset$. If $\Gamma \vdash P$ then P is safe.

The intuition of the proof is that the typability of any end-event ensures that it is matched by a parallel begin event. Also, having effects in types allows for the transfer of credit in the sense that channel types can now state a price for what it costs to send on the channel. Typability ensures that a process can afford to send on a channel. We refer to [17] for a detailed proof of the theorem.

2.3 Name-less Formulation of the Type System

In order to avoid having to deal with alpha conversion in type equivalence we now give an alternative but equivalent definition of the type system based on the well-known *de Bruijn* formulation of the λ-calculus [37]. The main difference is that pair types now bind *holes* in the form of natural numbers instead of message variables. This has an impact on effects and messages since these occur inside pair types. Also, the typing rules for messages need to reflect the modification as well.

Despite great similarity with previous definitions we give the new definitions with holes in full for completeness and for showing the differences. Important changes are highlighted.

2.3.1 Messages and Effects with Holes

Holes can only occur in messages within dependent types so we need to maintain two sets of messages: one without holes for use in specifying the protocol (as defined in the previous section), and one for use in describing dependent types. Messages with holes are a proper superset of messages without so it is no loss of generality to assume messages with holes are always used in types.

We let the set of *message with holes* be identical to the set of message from the previous section but extended with holes drawn from the set of natural numbers. We denote a message from this new set of messages by $\dot{M}$ and denote a hole by ω. Equality of messages with holes is similar to that for messages without, the only addition being equality of holes which is the standard equality of natural numbers.

The message instantiation operation $\rightarrow$ now recursively applies a message substitution $\langle M/\omega \rangle$ consisting of a message M and a hole ω to a message $\dot{M}$:

$$\begin{array}{rcl}
n\langle M/\omega\rangle & \rightarrow & n \\
x\langle M/\omega\rangle & \rightarrow & x \\
\omega'\langle M/\omega\rangle & \rightarrow & \begin{cases} M & \text{if } \omega = \omega' \\ \omega' & \text{otherwise} \end{cases} \\
\mathsf{ok}\langle M/\omega\rangle & \rightarrow & \mathsf{ok} \\
(\mathsf{fst}\ \dot{M})\langle M/\omega\rangle & \rightarrow & \mathsf{fst}\ (\dot{M}\langle M/\omega\rangle) \\
(\mathsf{snd}\ \dot{M})\langle M/\omega\rangle & \rightarrow & \mathsf{snd}\ (\dot{M}\langle M/\omega\rangle) \\
(\dot{M}_1, \dot{M}_2)\langle M/\omega\rangle & \rightarrow & (\dot{M}_1\langle M/\omega\rangle, \dot{M}_2\langle M/\omega\rangle)
\end{array}$$

Note that messages M inside message substitutions cannot themself contain holes as messages with holes only occur in types and cannot be used as messages in the calculus.[†]

Atomic effects with holes are pairs $l(\dot{M})$ (i.e. the same as atomic effects but based on messages with holes instead) and an effect with holes is a set of atomic effects with holes.

[†]Combining that substituted messages have no holes and that messages have no binding constructs we see that *de Bruijn* substitution collapses to the very simple substitution just defined.

We denote an effect with holes as $\dot{S}$. Message application and equivalence are extended to atomic effects and effects with holes in the obvious ways.

Finally, define a function $fi(\dot{M})$ to be the largest index occurring in $\dot{M}$, 0 if it is index free. Extend this to atomic effects in the obvious way and to effects by taking the largest index occuring in the set of atomic effects.

2.3.2 Types with Holes

Let the set of *types* be defined by:

$T ::=$	type
A	base type
$\mathsf{Ch}(T)$	channel for messages of type T
$\mathsf{Pair}(T_1, T_2)$	pair type
$\mathsf{Ok}(\dot{S})$	effect carrier type

The important difference in this formulation is evident in the definition of pair types $\mathsf{Pair}(T_1, T_2)$ where a message variable is no longer bound. Instead, pair types can be said to "bind" holes or indices in the form of numbers. The innermost pair type binds 1, the second inner most 2, and so on. For instance, $\mathsf{Pair}(x : \mathsf{A}, \mathsf{Pair}(y : \mathsf{A}, \mathsf{Ok}(\{l((x, y))\})))$ is represented as $\mathsf{Pair}(\mathsf{A}, \mathsf{Pair}(\mathsf{A}, \mathsf{Ok}(\{l((2, 1))\})))$. To keep with tradition we call this the *name-less* (or index-based) presentation.

The main advantage of the name-less representation is that alpha conversion becomes unnecessary. Recall types $\mathsf{Pair}(x : \mathsf{A}, \mathsf{Ok}(\{l(x)\}))$ and $\mathsf{Pair}(y : \mathsf{A}, \mathsf{Ok}(\{l(y)\}))$ from earlier. We have that both are now represented as $\mathsf{Pair}(\mathsf{A}, \mathsf{Ok}(\{l(1)\})))$ and hence clearly equivalent. It is easy to convert back and forth between the two representations and we have:

Proposition 7. Types T_1 and T_2 and alpha equivalent if and only if their name-less presentation is the same.

The message instantiation operation $\rightarrow$ for types becomes:

$$\begin{array}{rcl}
\mathsf{A}\langle M/\omega\rangle & \rightarrow & \mathsf{A} \\
\mathsf{Ch}(T)\langle M/\omega\rangle & \rightarrow & \mathsf{Ch}(T\langle M/\omega\rangle) \\
\mathsf{Pair}(T_1, T_2)\langle M/\omega\rangle & \rightarrow & \mathsf{Pair}(T_1\langle M/\omega\rangle, T_2\langle M/\omega + 1\rangle) \\
\mathsf{Ok}(\dot{S})\langle M/\omega\rangle & \rightarrow & \mathsf{Ok}(\dot{S}\langle M/\omega\rangle)
\end{array}$$

Note that since pair types bind a hole we have to increase the index of the application by one when applying it to the second type of a pair. As the substituted message M cannot contain holes we need not shift any holes in it (as done in the standard formulation of *de Bruijn* indices).

The only difference in the definition of type equivalence is that alpha conversion is no longer necessary:

Definition 8 (**Type Equivalence:** $T_1 \equiv T_2$)**.** Let *type equivalence* be the least congruence on types closed under axiom

$$\frac{\dot{S}_1 \equiv \dot{S}_2}{\mathsf{Ok}(\dot{S}_1) \equiv \mathsf{Ok}(\dot{S}_2)} \text{ TE-OK}$$

2.3.3 Typing Rules

The introduction of holes means that the set of free names $fn(T)$ of a type T is now defined as:

$$\begin{aligned} fn(\mathsf{A}) &= \emptyset \\ fn(\mathsf{Ch}(T)) &= fn(T) \\ fn(\mathsf{Pair}(T_1, T_2)) &= fn(T_1) \cup fn(T_2) \\ fn(\mathsf{Ok}(S)) &= fn(S) \end{aligned}$$

where we see that pair types no longer bind a variable in the second component. Having removed this we need to introduce another function in order to express well-formed typing contexts. Intuitively, function $fi(T)$ expresses the number of encapsulating pair types a type T must be inside in order to have no free holes. We have:

$$\begin{aligned} fi(\mathsf{A}) &= 0 \\ fi(\mathsf{Ch}(T)) &= fi(T) \\ fi(\mathsf{Pair}(T_1, T_2)) &= max\Big(fi(T_1), fi(T_2) - 1\Big) \\ fi(\mathsf{Ok}(S)) &= fi(S) \end{aligned}$$

where we see that the number is decreased by one in the second projecting for pair types.

The definition of well-formed typing contexts $\Gamma \vdash \diamond$ is now given by the rules in Table 2.6 where the $fi(\cdot) = 0$ intuitively means that all holes are accounted for.

$$\frac{}{\emptyset \vdash \diamond} \text{ WF-EMP}$$

$$\frac{\Gamma \vdash \diamond \qquad fn(S) \subseteq dom(\Gamma) \qquad fi(S) = 0}{\Gamma, S \vdash \diamond} \text{ WF-EF}$$

$$\frac{\Gamma \vdash \diamond \qquad n \notin dom(\Gamma) \qquad fn(T) \subseteq dom(\Gamma) \qquad fi(T) = 0}{\Gamma, n : T \vdash \diamond} \text{ WF-TY}$$

Table 2.6: Rules for well-formed typing contexts

Only the typing rules for messages are changed and only in rules MT-SND and MT-PAIR. Table 2.7 lists all rules. We see that rules T-SND and T-PAIR now substitute a hole 1 instead of a message variable.

$$\frac{\Gamma \vdash \diamond \qquad n : T \in \Gamma}{\Gamma \vdash n : T} \quad \text{MT-NAME}$$

$$\frac{\Gamma \vdash \diamond \qquad S \subseteq \mathit{effects}(\Gamma)}{\Gamma \vdash \mathsf{ok} : \mathsf{Ok}(S)} \quad \text{MT-OK}$$

$$\frac{\Gamma \vdash M : \mathsf{Pair}(T_1, T_2)}{\Gamma \vdash \mathsf{fst}\ M : T_1} \quad \text{MT-FST}$$

$$\frac{\Gamma \vdash M : \mathsf{Pair}(T_1, T_2)}{\Gamma \vdash \mathsf{snd}\ M : T_2\langle \mathsf{fst}\ M/1\rangle} \quad \text{MT-SND}$$

$$\frac{\Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash M_2 : T_2\langle M_1/1\rangle}{\Gamma \vdash (M_1, M_2) : \mathsf{Pair}(T_1, T_2)} \quad \text{MT-PAIR}$$

Table 2.7: Typing rules for messages

2.3.4 Important Properties

The most important property is that the new formulation of the type system matches the old. Since process typing rules are the same it suffices to have that any typing context is well-formed in the old formulation exactly when it is well-formed in the new, and that any message is well-typed in the old formulation exactly when it is in the new:

Lemma 9. For any typing context Γ we have $\Gamma \vdash_{old} \diamond$ if and only if $\Gamma \vdash_{new} \diamond$. For any message M we have $M \vdash_{old} : T_{old}$ if and only if $M \vdash_{new} : T_{new}$.

Proof. In the old formulation, all variables bound by pair types are instantiated when "breaking down" a pair type and hence the typing context never holds variables coming from a pair type. This is only made explicit in the new formulation where a distinction is made between variables bound by processes and variables bound by pair types. As for the message typing rules, we have that the old and new derivation trees are isomorphic with the only difference being pair types. Since it is easy to switch back and forth between a pair type without holes and one with, any derivation tree in one formulation can easily be turned into a derivation tree in the other. □

An important part of being able to do type checking and type inference is being able to do type equivalence. We have

Lemma 10. Type equivalence is decidable.

Proof. All relations used by the type equivalence relation are easily seen to be decidable. □

Theorem 11. Type checking is decidable.

Proof. Given a type derivation tree, a type checking algorithm need only do instantiation and type equivalence check. Since we assumed an decidable equality on labels we have that all relations are decidable. □

Furthermore, if label equivalence is decidable in polynomial time we have

Remark 12. Type checking can be performed in polynomial time (in the size of the type derivation tree).

Chapter 3

Type Inference

In this chapter we answer the question posed in the previous chapter: *is process P typable?* If this is the case then Theorem 6 yields that P is safe. Doing one better, we not only give an algorithm that can answer the question with *yes* or *no*, but an algorithm that gives a machine checkable proof in the form of a type derivation tree for P if the answer is *yes*.

For type inference (or type reconstruction) we turn to the well-known praxis of generating and solving constraints. Constraint generation is done by more or less "running the typing rules backwards" and constraint solving is done via unification for type contraints and via propagation for effect constraints.

In Section 3.1 we introduce *type variables* as placeholders for yet-unknown types and type schemes, and *effect variables* as placeholders for yet-unknown effects. To capture the relationship between types in the presence of message instantiation we need a new formulation of these with *explicit substitutions*. In Section 3.2 we discuss constraint generation, and constraint solving in Section 3.3 and 3.4. Section 3.5 gives the combined type inference algorithm and Section 3.6 shows an example inference.

Note that this chapter is based on the formulation of the type system using name-less pair types.

3.1 Introducing Variables

The naive mixture of (meta) type and effect variables with message instantiation does not work since it is not well-defined to instantiate a variable with a message substitution: instantiation depends on the kind of the type or effect and whether or not it has a matching hole.

To remedy this we give a slightly different formulation of types using *explicit substitutions* inspired by [13]. For variable-free types and effects this new formulation coincides with the one from Chapter 2. Furthermore, explicit substitutions allow us to express a minimal type scheme.

Effect (meta) variables cannot be used in process annotations but only in contexts or types. Also, note that there are no (meta) message variables so we adapt the definition of messages $\dot{M}$ from Chapter 2 without modification.

3.1.1 Effects with Variables

Define *effects with holes* $\dot{S}$ by

$\dot{S} ::=$	effect with holes
$\{l_1(\dot{M}_1), \ldots, l_n(\dot{M}_n)\}$	set of atomic effects with holes
E	effect variable
$\dot{S}\langle M/\omega\rangle$	explicit substitution

where E is from a distinct set *EVars* of (meta) effect variables. By the *underlying effect* of an effect under explicit substitution $\dot{S}\langle M/\omega\rangle$ we mean the maximal constituent of $\dot{S}$ that is not an effect under explicit substitution.

Note that we can form effects $\dot{S}\langle M_1/\omega_1\rangle\langle M_2/\omega_2\rangle\ldots$ for any finite number of substitutions. As a notational convenience we shall sometimes denote a list of one or more substitutions as μ. For instance, $\dot{S}\langle M_1/\omega_1\rangle\langle M_2/\omega_2\rangle$ is denoted $\dot{S}\ \mu$ with $\mu = \langle M_1/\omega_1\rangle\langle M_2/\omega_2\rangle$.

A key difference in this new formulation of effects with holes is in the definition of how an effect is instantiated with a message substitution. In the old formulation we defined an operation taking an effect $\dot{S}$ and a substitution $\langle M/\omega\rangle$ as input and yielding a new effect $\dot{S}'$ with the substitution applied as output. In the new formulation we simply form the explicit substitution effect $\dot{S}\langle M/\omega\rangle$ and then have a instantiation relation $\rightarrow$:

Definition 13 (Effect Instantiation: $\dot{S} \rightarrow \dot{S}'$). Let *effect instantiation* $\rightarrow$ be the least relation on effects closed under transitivity and rule

$$\frac{\dot{M}_i\langle M/\omega\rangle \rightarrow \dot{M}'_i \text{ for all } i}{\{l_1(\dot{M}_1), \ldots, l_n(\dot{M}_n)\}\langle M/\omega\rangle \rightarrow \{l_1(\dot{M}'_1), \ldots, l_n(\dot{M}'_n)\}}\ \text{EI-AE}$$

We say $\dot{S}$ *instantiates to* $\dot{S}'$ if $\dot{S} \rightarrow \dot{S}'$.

By this definition we see that the order of explicit substitutions in a list μ is important. Indeed, for $M_1 \not\equiv M_2$ we see that

$$\{l(\omega)\}\langle M_1/\omega\rangle\langle M_2/\omega\rangle$$

is not the same effect as

$$\{l(\omega)\}\langle M_2/\omega\rangle\langle M_1/\omega\rangle$$

since the first reduces to $\{l(M_1)\}$ whereas the second reduces to $\{l(M_2)\}$. By the deterministic nature of application this also shows that any substitution for a hole ω behind another substitution for ω has no impact and can be removed. In the above example $\langle M_2/\omega\rangle$ has no impact in the first case and $\langle M_1/\omega\rangle$ no impact in the second.

Furthermore, note that no effect variable under substitution can be instantiated (as the relation is not closed under reflexivity). Intuitively $\dot{S}\langle M/\omega\rangle$ can be though of as a delayed substitution waiting to be applied: if $\dot{S}$ is a variable (with or without other substitutions) then nothing can be instantiated; on the other hand, if $\dot{S}$ is something else then the substitutions can be applied by the instantiation rules. We see that $\dot{S}\langle M/\omega\rangle$ captures the effect $\dot{S}'$ obtained by applying $\langle M/\omega\rangle$ to $\dot{S}$ in accordance with the old definition.

Definition 14 (Effect Equivalence: $\dot{S}_1 \equiv \dot{S}_2$). Let *effect equivalence* $\equiv$ be the least equivalence on effects closed under effect instantiation.

By this definition we have e.g. $\{l(1)\}\langle M/1\rangle \equiv \{l(M)\}$ and $\{l(1)\}\langle M/1\rangle \equiv \{l(2)\}\langle M/2\rangle$ even though $\{l(1)\} \not\equiv \{l(2)\}$.

Denote by $FEV(\dot{S})$ the set of effect variables in effect $\dot{S}$:

$$\begin{aligned} FEV(E) &= \{E\} \\ FEV(\{l_1(\dot{M}_1), \ldots, l_n(\dot{M}_n)\}) &= \emptyset \\ FEV(\dot{S}\langle M/\omega\rangle) &= FEV(\dot{S}) \end{aligned}$$

A *ground effect* is an effect $\dot{S}$ with no effect variables, i.e. $FEV(\dot{S}) = \emptyset$.

Let the *opening* of an effect $\dot{S}$ be a similar effect but with fresh variables and no substitutions:

$$\begin{aligned} open(E) &= \text{fresh variable } E' \\ open(\dot{S}\langle M/\omega\rangle) &= open(\dot{S}) \\ open(\{l_1(\dot{M}_1), \ldots, l_n(\dot{M}_n)\}) &= (\{l_1(\dot{M}_1), \ldots, l_n(\dot{M}_n)\} \end{aligned}$$

In the later context where we need to open an effect, the effect is instantiated as much as possible before opening. This means that substitutions are only thrown away when we are opening a variable under explicit substitutions.

3.1.2 Types with Variables

Types are extended with type variables and explicit substitutions:

$T ::=$	type
X	type variable
A	base type
$\mathsf{Ch}(T)$	channel for messages of type T
$\mathsf{Pair}(T_1, T_2)$	pair type
$\mathsf{Ok}(\dot{S})$	effect carrier type
$T\langle M/\omega\rangle$	explicit substitution

where X is from a distinct set *TVars* of (meta) type variables. By the *underlying type* of a type under explicit substitution $T\langle M/\omega\rangle$ we mean the maximal constituent of T that is not an type under explicit substitution. As for effects we shall sometimes denote a list of

one or more substitutions as μ.

Denote by $FV(T)$ the set of type and effect variables in type T:

$$\begin{aligned} FV(X) &= \{X\} \\ FV(\mathsf{A}) &= \emptyset \\ FV(\mathsf{Ch}(T)) &= FV(T) \\ FV(\mathsf{Pair}(T_1, T_2)) &= FV(T_1) \cup FV(T_2) \\ FV(\mathsf{Ok}(\dot{S})) &= FEV(\dot{S}) \\ FV(T\langle M/\omega\rangle) &= FV(T) \end{aligned}$$

and by $FTV(T)$ the set of type variables in type T, i.e. $FTV(T) = FV(T) \cap TVars$. A *ground type* is a type T with no type nor effect variables, i.e. $FV(T) = \emptyset$.

As we did with effects we defined a type instantiation relation:

Definition 15 (**Type Instantiation:** $T_1 \to T_2$)**.** Let *type instantiation* $\to$ be the least relation on types closed under transitivity and rules of congruence for type constructors, along with axioms in Table 3.1. We say T *instantiates to* T' if $T \to T'$.

$$\frac{}{\mathsf{A}\langle M/\omega\rangle \to \mathsf{A}} \quad \text{TI-BASE}$$

$$\frac{}{\mathsf{Ch}(T)\langle M/\omega\rangle \to \mathsf{Ch}(T\langle M/\omega\rangle)} \quad \text{TI-BASE}$$

$$\frac{}{\mathsf{Ok}(\dot{S})\langle M/\omega\rangle \to \mathsf{Ok}(\dot{S}\langle M/\omega\rangle)} \quad \text{TI-OK}$$

$$\frac{}{\mathsf{Pair}(T_1, T_2)\langle M/\omega\rangle \to \mathsf{Pair}(T_1\langle M/\omega\rangle, T_2\langle M/\omega+1\rangle)} \quad \text{TI-PAIR}$$

Table 3.1: Type instantiation rules

As in the old formulation of type instantiation, rule TI-PAIR increases the index of the hole by one in the second component. Otherwise the intuition is the same as for effect instantiation.

Lemma 16. The instantiation relation $\to$ is strongly normalising.

To control explicit substitutions we define a normal form for types where explicit substitutions are either applied if possible thereby disappering, or pushed as far down into types as possible (before hitting a type or effect variable). Formally we have

Definition 17 (**Type Normal Form**)**.** The *normal form* of a type T is the type T' obtained by instantiating T so that no further instantiation is possible, i.e. $T \to T'$ and $T' \not\to$. Denote by $nf(T)$ the normal form of T. We shall sometimes write $\widehat{T}$ for a type T on normal form.

As an example, the normal form of

$$T_1 = \mathsf{Ch}(\mathsf{Pair}(\mathsf{Ch}(X), Y)\langle M/\omega\rangle)$$

is

$$T_1' = \mathsf{Ch}(\mathsf{Pair}(\mathsf{Ch}(X\langle M/\omega\rangle), Y\langle M/\omega+1\rangle))$$

since $T_1 \to T_1'$ and no further instantiation is possible.

We see that the set of normalised types coincides with the set of types generated by rules

$$\begin{array}{l}\widehat{T} ::= \\ \quad \mathsf{A} \\ \quad \mathsf{Ch}(\widehat{T}) \\ \quad \mathsf{Pair}(\widehat{T}_1, \widehat{T}_2) \\ \quad \mathsf{Ok}(\dot{S}) \\ \quad X \\ \quad X\ \mu\end{array}$$

where the underlying type of explicit substitutions is always a type variable. With this definition of normalised types it is easy to see the relationship with the old formulation of types

Remark 18. The set of normalised ground types coincides with the old definition of the set of types (i.e. without variables).

Definition 19 (**Type Equivalence:** $T_1 \equiv T_2$)**.** Let *type equivalence* $\equiv$ be the least congruence on types closed under type instantiation and axiom

$$\frac{\dot{S}_1 \equiv \dot{S}_2}{\mathsf{Ok}(\dot{S}_1) \equiv \mathsf{Ok}(\dot{S}_2)}\ \text{TE-OK}$$

Note that we can have

$$T_1 = \mathsf{Ch}(\mathsf{Ok}(\{(l,(1,2))\})) \qquad \text{and} \qquad T_2 = \mathsf{Ch}(\mathsf{Ok}(\{(l,(2,1))\}))$$

such that $T_1 \not\equiv T_2$ but $T_1\mu_1 \equiv T_2\mu_2$ for $\mu_1 = \langle M_1/1\rangle\langle M_2/2\rangle$ and $\mu_2 = \langle M_1/2\rangle\langle M_2/1\rangle$.

Not surprisingly we have that type equivalence is preserved by normal forms:

Lemma 20. $T_1 \equiv T_2$ if and only if $nf(T_1) \equiv nf(T_2)$.

Proof. By Definition 17 and since $T \to T'$ implies $T \equiv T'$. Desired result follows by transitivity of $\equiv$. □

In order to compare types that may have different effects but same type constructor structure, we define a relation that ignores all effects and in turn messages occurring in types:

Definition 21 (**Structural Constructor Equivalence:** $T_1 \simeq T_2$)**.** Let *structural constructor equivalence* $\simeq$ be the least congruence on types closed under axioms

$$\frac{}{X_1 \simeq X_2}\ \text{TC-VAR} \qquad \frac{}{\mathsf{Ok}(\dot{S}_1) \simeq \mathsf{Ok}(\dot{S}_2)}\ \text{TC-OK} \qquad \frac{}{T_1\langle M/\omega\rangle \simeq T_1}\ \text{TC-SUBST}$$

We say that T_1 and T_2 have *equivalent constructor structure* if $T_1 \simeq T_2$.

Since structural constructor equivalence clearly ignores any effects and messages, we see that instantiation cannot change the constructor structure of a type. Formally we have

Lemma 22. If $T\langle M/\omega\rangle \to T'$ then $T \simeq T'$

Proof. Induction in the derivation of $\to$. □

The main motivation for structural constructor equivalence is to justify the *opening* operation of a type. Define function $open(T)$ by:

$$\begin{aligned}
open(X) &= \text{fresh variable } X' \\
open(\mathsf{A}) &= \mathsf{A} \\
open(\mathsf{Ch}(T_1)) &= \mathsf{Ch}(open(T_1)) \\
open(\mathsf{Pair}(T_1, T_2)) &= \mathsf{Pair}(open(T_1), open(T_2)) \\
open(\mathsf{Ok}(\dot{S})) &= \mathsf{Ok}(open(\dot{S})) \\
open(T\langle M/\omega\rangle) &= open(T)
\end{aligned}$$

which follows $\simeq$ in the sense that a type and its opening have equivalent constructor structure ($T \simeq open(T)$), the difference being that the opening has a fresh set of variables and no explicit substitutions. Intuitively we have that the opening of a type is the minimally constrained type (relative to a set of constraints) with the same structure. As is also the case for effects, when we in a later context use opening of types, these types are always instantiated as much as possible beforehand. For this reason substitutions are only discarded if the underlying type is a variable.

The relationship between type equivalence and structural constructor equivalence is evident. We have that two types cannot be equivalent and have different constructor structure so when testing if $T_1 \equiv T_2$ we can stop if $T_1 \not\simeq T_2$.

Lemma 23. If $T_1 \equiv T_2$ then $T_1 \simeq T_2$

Proof. Induction in the derivation of $T_1 \equiv T_2$. □

3.2 Constraint Generation

The purpose of this section is to give an algorithm that on input Γ and P generates a set of constraints which is satisfiable if and only if $\Gamma \vdash P$. The algorithm takes the form of a set of rules matching closely the typing rules of the type system. In the next section on constraint solving we shall see how to determine if a set of constraints are satisfiable.

Definition 24 (**Type and Effect Substitution**)**.** A (partial) *substitution* σ is a finite map from type and effect variables to types and effects, i.e. $\sigma : TVars \cup EVars \to T \cup \dot{S}$. Let $dom(\sigma)$ be the set of type and effect variables assigned to by σ and let $ran(\sigma)$ be the set of types and effects assigned by σ. Let $\sigma\backslash_V$ be the substitution σ' undefined for all variables in V but otherwise matching σ, and let $\sigma|_V$ be σ restricted to V. A substitution σ is a *ground substitution* if $ran(\sigma)$ only contains ground types and ground effects. The composition $\sigma_1 \circ \sigma_2$ of substitutions σ_1 and σ_2 forms a new substitution as in normal function composition.

Definition 25 (**Application of Type and Effect Substitution**)**.** Application of a type and effect substitution is done simultaneously and according to rules:

$$
\begin{aligned}
\sigma X &= \begin{cases} T & \text{if } X \mapsto T \in \sigma \\ X & \text{otherwise} \end{cases} \\
\sigma \mathsf{A} &= \mathsf{A} \\
\sigma \mathsf{Ch}(T) &= \mathsf{Ch}(\sigma T) \\
\sigma \mathsf{Pair}(T_1, T_2) &= \mathsf{Pair}(\sigma T_1, \sigma T_2) \\
\sigma \mathsf{Ok}(E) &= \mathsf{Ok}(\sigma E) \\
\sigma(T\langle M/\omega \rangle) &= (\sigma T)\langle M/\omega \rangle \\
\\
\sigma E &= \begin{cases} \dot{S} & \text{if } E \mapsto \dot{S} \in \sigma \\ E & \text{otherwise} \end{cases} \\
\sigma\{l_1(M_1), \cdots, l_n(M_n)\} &= \{l_1(M_1), \ldots, l_n(M_n)\} \\
\sigma(\dot{S}\langle M/\omega \rangle) &= (\sigma \dot{S})\langle M/\omega \rangle \\
\\
\sigma(\mathsf{new}\ n : T;\ P) &= \mathsf{new}\ n : \sigma T;\ \sigma P \\
\sigma(\mathsf{in}\ M\ n;\ P) &= \mathsf{in}\ M\ n;\ \sigma P \\
\sigma(!\mathsf{in}\ M\ n;\ P) &= !\mathsf{in}\ M\ n;\ \sigma P \\
\sigma(\mathsf{exercise}\ M; P) &= \mathsf{exercise}\ M; \sigma P \\
\sigma(\mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_3\ \mathsf{else}\ P_4) &= \mathsf{if}\ M_1 = M_2\ \mathsf{then}\ \sigma P_3\ \mathsf{else}\ \sigma P_4 \\
\sigma(P_1 \mid P_2) &= \sigma P_1 \mid \sigma P_2 \\
\sigma P &= P \text{ for any other } P
\end{aligned}
$$

Application is extended to a typing context Γ by point-wise application.

For ensuring well-formedness of typing contexts we introduce *boundaries*. Intuitively, a set of boundaries is respected if and only if the typing context from which they are derived is well-formed.

Definition 26 (**Boundaries**)**.** A *boundary* is of the following kind:

- *name boundary*: $T \preceq \mathcal{N}$ or $\dot{S} \preceq \mathcal{N}$, where $\mathcal{N}$ is a set of message names
- *index boundary*: $T \leq \omega$ or $\dot{S} \leq \omega$

We use B to denote a set of boundaries.

We then define when a substitution stays within a set of boundaries. Note that this is only defined for ground substitutions since *fn* and *fi* is not defined for type and effect variables:

Definition 27 (**Staying within Boundaries**)**.** A ground substitution σ *stays within* a boundary:

- name boundary: σ stays within $T \preceq \mathcal{N}$ if $fn(\sigma T) \subseteq \mathcal{N}$, similar for $\dot{S} \preceq \mathcal{N}$
- index boundary: σ stays within $T \leq \omega$ if $fi(\sigma T) \leq \omega$, similar for $\dot{S} \leq \omega$

Substitution σ stays within a boundary set B if it stays within all boundaries in B.

Using boundaries we can characterise a well-formed typing context Γ:

Definition 28 (**Well-formed Boundaries**)**.** Let the well-formed boundaries for a typing context Γ be defined by rules in Table 3.2.

$$\frac{}{\emptyset \vdash \diamond \rightsquigarrow \emptyset} \quad \text{WB-EMP}$$

$$\frac{\Gamma \vdash \diamond \rightsquigarrow B_1 \qquad B = \{T \preceq dom(\Gamma), T \leq 0\} \cup B_1}{\Gamma, n : T \vdash \diamond \rightsquigarrow B} \quad \text{WB-TY}$$

$$\frac{\Gamma \vdash \diamond \rightsquigarrow B_1 \qquad B = \{\dot{S} \preceq dom(\Gamma), \dot{S} \leq 0\} \cup B_1}{\Gamma, \dot{S} \vdash \diamond \rightsquigarrow B} \quad \text{WB-EF}$$

Table 3.2: Boundary generation rules

Not surprisingly we have that these rules characterise the rules of well-formed typing contexts:

Lemma 29. Suppose $\Gamma \vdash \diamond \rightsquigarrow B$. Ground substitution σ stays within boundary set B if and only if $\sigma\Gamma \vdash \diamond$.

Given a boundary set, say, $\{\mathsf{Pair}(T_1, T_2) \preceq \mathcal{N}\}$, we can equate this with bound set $\{T_1 \preceq \mathcal{N}, T_2 \preceq \mathcal{N}\}$ due to the definition of *fn* and *fi*. Also, $\{\mathsf{Pair}(T_1, T_2) \leq \omega\}$ is equivalent to bound set $\{T_1 \leq \omega, T_2 \leq \omega + 1\}$. Formally we have

Definition 30 (Boundary Reduction)**.** Let *boundary reduction* $\rightarrow$ be the least relation on boundary sets closed under transitivity and type and effect instantiation, along with axioms in Table 3.3. We say B *reduces to* B' if $B \rightarrow B'$.

As a simple consequence of boundary reduction we have

Lemma 31. Suppose $B \rightarrow B'$. Ground substitution σ stays within B if and only if it stays within B'.

With this in mind we notice that it is safe to assume that no type constructors occurs in a boundary set, i.e. any boundary set contains only type variables X or type variables under explicit substitutions $X\mu$ along with effects $\dot{S}$.

Having captured well-formed typing contexts we next focus on constraints for characterising the typing rules for messages and processes. We start with a definition of constraints on types and effects:

$$\frac{}{\{\mathsf{Ch}(T) \preceq \mathcal{N}\} \cup B \to \{T \preceq \mathcal{N}\} \cup B} \quad \text{BR-CH-N}$$

$$\frac{}{\{\mathsf{Ch}(T) \leq \omega\} \cup B \to \{T \leq \omega\} \cup B} \quad \text{BR-CH-I}$$

$$\frac{}{\{\mathsf{Pair}(T_1, T_2) \preceq \mathcal{N}\} \cup B \to \{T_1 \preceq \mathcal{N}, T_2 \preceq \mathcal{N}\} \cup B} \quad \text{BR-P-N}$$

$$\frac{}{\{\mathsf{Pair}(T_1, T_2) \leq \omega\} \cup B \to \{T_1 \leq \omega, T_1 \leq \omega + 1\} \cup B} \quad \text{BR-P-I}$$

$$\frac{}{\{\mathsf{Ok}(\dot{S}) \preceq \mathcal{N}\} \cup B \to \{\dot{S} \preceq \mathcal{N}\} \cup B} \quad \text{BR-OK-N}$$

$$\frac{}{\{\mathsf{Ok}(\dot{S}) \leq \omega\} \cup B \to \{\dot{S} \leq \omega\} \cup B} \quad \text{BR-OK-I}$$

Table 3.3: Bound reduction rules

Definition 32 (**Constraints**)**.** A *constraint* is of the following kind:

- *type equality*: $T_1 \doteq T_2$
- *type requirement*: T $\mathsf{generative}$
- *effect equality*: $\dot{S}_1 \doteq \dot{S}_2$
- *effect requirement*: $l(M) \in \dot{S}_1, \ldots, \dot{S}_n$
- *effect bound*: $\dot{S} \sqsubseteq \dot{S}_1, \ldots, \dot{S}_n$

A *constraint set* C is a set of constraints.

In the current type system a type is generative if and only if it is a channel type so we can express a type requirement on, say type T, by a type equality constraint $T \doteq \mathsf{Ch}(X)$ for a fresh variable X.

Definition 33 (**Constraint Satisfaction**)**.** A substitution σ *satisfies* a constraint as follows:

- type equality: $T_1 \doteq T_2$ is satisfied if $\sigma T_1 \equiv \sigma T_2$
- type requirement: T $\mathsf{generative}$ is satisfied if σT is generative
- effect equality: $\dot{S}_1 \doteq \dot{S}_2$ is satisfied if $\sigma \dot{S}_1 \equiv \sigma \dot{S}_2$
- effect requirement: $l(M) \in \dot{S}_1, \ldots, \dot{S}_n$ is satisfied if $l(M) \in \sigma \dot{S}_1 \cup \cdots \cup \sigma \dot{S}_n$
- effect bound: $\dot{S} \sqsubseteq \dot{S}_1, \ldots, \dot{S}_n$ is satisfied if $\sigma \dot{S} \subseteq \sigma \dot{S}_1 \cup \cdots \cup \sigma \dot{S}_n$

Substitution σ satisfies constraint set C if σ satisfies all constraints in C.

We first represent constraint generation rules for messages which are later used in the constraint generation rules for processes. The rules can be interpreted as an algorithm taking (Γ, M) as input and yielding (T, C, V, B) as output where T is the type (or type variable) of M, C is the generated constraint set, B is a set of boundaries needed for ensuring well-formed typing contexts, and the set V is used during constraint generation to keep account of fresh variables. It is implicitly understood that when choosing a new variable it has to be fresh in respect to the set V. Also, when the union of two sets V_1 and V_2 is formed, we assume that the sets are disjoint.

Definition 34 (**Message Constraints**)**.** Let the *constraint derivation tree* $\Gamma \vdash M \rightsquigarrow T, C, B, V$ for a message M under a context Γ be defined by rules in Table 3.4.

$$\frac{n : T \in \Gamma \qquad \Gamma \vdash \diamond \rightsquigarrow B_1}{\Gamma \vdash n \rightsquigarrow T, \emptyset, B_1, \emptyset} \text{ MC-NAME}$$

$$\frac{C = \{X \doteq \mathsf{Ok}(E), E \sqsubseteq \mathit{effects}(\Gamma)\} \qquad \Gamma \vdash \diamond \rightsquigarrow B_1 \qquad V = \{X, E\}}{\Gamma \vdash \mathsf{ok} \rightsquigarrow X, C, B_1, V} \text{ MC-OK}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1 \qquad V = \{X_1, X_2\} \uplus V_1 \qquad C = \{T_1 \doteq \mathsf{Pair}(X_1, X_2)\} \cup C_1}{\Gamma \vdash \mathsf{fst}\ M_1 \rightsquigarrow X_1, C, B_1, V} \text{ MC-FST}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1 \qquad V = \{X_1, X_2, X_2'\} \uplus V_1 \qquad C = \{T_1 \doteq \mathsf{Pair}(X_1, X_2),\ X_2' \doteq X_2\langle \mathsf{fst}\ M_1/1\rangle\} \cup C_1}{\Gamma \vdash \mathsf{snd}\ M_1 \rightsquigarrow X_2', C, B_1, V} \text{ MC-SND}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1 \qquad \Gamma \vdash M_2 \rightsquigarrow T_2, C_2, B_2, V_2 \qquad C = \{X \doteq \mathsf{Pair}(T_1, X_2'),\ T_2 \doteq X_2'\langle M_1/1\rangle\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = \{X, X_2'\} \uplus V_1 \uplus V_2}{\Gamma \vdash (M_1, M_2) \rightsquigarrow X, C, B, V} \text{ MC-PAIR}$$

Table 3.4: Message constraint generation rules

Note that *effects*(Γ) now returns a list of effects with holes $\dot{S}_1, \ldots, \dot{S}_n$ instead of a set of atomic effects.

Intuitively, the rules generate a constraint derivation tree collecting constraints to be satisfied in C. It is not hard to see that this constraint derivation tree is isomorphic to any type derivation tree there might exists for M under Γ. Due to the high resemblance to the typing rules it should come as no surprise that the constraint rules are sound and complete in respect to the typing rules.

Lemma 35 (**Soundness of constraints for messages**)**.** Suppose we have constraint derivation tree $\Gamma \vdash M \rightsquigarrow T, C, B, V$. If a ground substitution σ satisfies C and stays within B then there exist a type derivation tree with root $\sigma\Gamma \vdash M : \sigma T$. Furthermore, the type derivation tree can be efficiently constructed.

Proof. The proof is by induction in the constraint derivation tree of $\Gamma \vdash M \rightsquigarrow T, C, B, V$. Using that the constraint derivation tree is isomorphic to any type derivation tree we built a type derivation tree using assignments found in σ. Since σ is a ground substitution we have that all variables are assigned a ground type as required by the type system. See Section A.1 for full proof. □

Notice that since we are building a full-typed type derivation tree we can say more than just "yes, the message is typable" and actually provide a proof of this in the form of a type derivation tree checkable by the type checker.

Lemma 36 (**Completeness of constraints for messages**)**.** Suppose we have constraint derivation tree $\Gamma \vdash M \rightsquigarrow T, C, B, V$. If there exist a type derivation tree with root $\sigma\Gamma \vdash M : U$ for some type U and substitution σ with $dom(\sigma) \cap V = \emptyset$ then there exist a substitution σ' satisfying C and staying within B and where $\sigma' T \equiv U$ and $\sigma'\backslash_V = \sigma$.

Proof. Induction in the derivation of $\Gamma \vdash M \rightsquigarrow T, C, B, V$. Using the assignments found in σ we construct the extended substitution σ' satisfying C. See Section A.1 for full proof. □

Definition 37 (**Process Constraints**)**.** Let the *constraint derivation tree* for a process P under context Γ be defined by rules in Table 3.5.

Not surprisingly we have that these rules are sound and complete with respect to the process typing rules.

Lemma 38 (**Soundness of constraints for processes**)**.** Suppose we have constraint derivation tree $\Gamma \vdash P \rightsquigarrow C, B, V$. If a ground substitution σ satisfies C and stays within B then there exists a type derivation tree with root $\sigma\Gamma \vdash \sigma P$. Furthermore, the type derivation tree can be efficiently constructed.

Proof. The proof is by induction in the derivation of $\Gamma \vdash P \rightsquigarrow C, B, V$. Using that the constraint derivation tree is isomorphic to any type derivation tree we built a type derivation tree using assignments found in σ and Lemma 35. Since σ is a ground substitution we have that all variables are assigned a ground type as required by the type system. See section A.2 for full proof. □

Lemma 39 (**Completeness of constraints for processes**)**.** Suppose we have constraint derivation tree $\Gamma \vdash P \rightsquigarrow C, B, V$. If there exists a type derivation tree with root $\sigma\Gamma \vdash \sigma P$ for some substitution σ with $dom(\sigma) \cap V = \emptyset$ then there exists a substitution σ' satisfying C and staying within B and with $\sigma'\backslash_V = \sigma$.

Proof. Induction in the derivation of $\Gamma \vdash P \rightsquigarrow C, B, V$. Using the assignments found in σ we construct the extended substitution σ' satisfying C. See Section A.2 for full proof. □

Theorem 40 (**Completeness and soundness of constraints**)**.** Suppose we have constraint derivation tree $\Gamma \vdash P \rightsquigarrow C, B, V$. Process P has a type derivation tree under Γ if and only if there exists a ground substitution σ satisfying C and staying within B. A type derivation tree is efficiently constructable given the substitution.

Proof. Follows at once by Lemma 38 and 39. □

3.3 Solving Type Constraints

Having seen in the previous section that the generation of a type derivation tree for a message or process can be reduced to the generation of a substitution σ satisfying a constraint set, we now consider how to find such a substitution. We split this substitution generation into two steps, dealing with type constraints in this section and with effect constraints in Section 3.4. The two algorithms are combined in Section 3.5 to form the type inference algorithm.

$$\frac{\Gamma \vdash M \rightsquigarrow T, C_1, B_1, V_1 \qquad \Gamma, n : X \vdash P \rightsquigarrow C_2, B_2, V_2 \qquad C = \{T \doteq \mathsf{Ch}(X)\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = \{X\} \uplus V_1 \uplus V_2}{\Gamma \vdash \mathsf{in}\ M\ n;\ P \rightsquigarrow C, B, V} \quad \text{PC-IN}$$

$$\frac{\Gamma \vdash M \rightsquigarrow T, C_1, B_1, V_1 \qquad \Gamma, n : X \vdash P \rightsquigarrow C_2, B_2, V_2 \qquad C = \{T \doteq \mathsf{Ch}(X)\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = \{X\} \uplus V_1 \uplus V_2}{\Gamma \vdash !\mathsf{in}\ M\ n;\ P \rightsquigarrow C, B, V} \quad \text{PC-REIN}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1 \qquad \Gamma \vdash M_2 \rightsquigarrow T_2, C_2, B_2, V_2 \qquad C = \{T_1 \doteq \mathsf{Ch}(T_2)\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = V_1 \uplus V_2}{\Gamma \vdash \mathsf{out}\ M_1\ M_2 \rightsquigarrow C, B, V} \quad \text{PC-OUT}$$

$$\frac{\Gamma, n : T \vdash P \rightsquigarrow C_1, B_1, V_1 \qquad C = \{T\ \mathsf{generative}\} \cup C_1}{\Gamma \vdash \mathsf{new}\ n : T;\ P \rightsquigarrow C, B_1, V_1} \quad \text{PC-NEW}$$

$$\frac{\Gamma, \mathit{begins}(P_2) \vdash P_1 \rightsquigarrow C_1, B_1, V_1 \qquad \Gamma, \mathit{begins}(P_1) \vdash P_2 \rightsquigarrow C_2, B_2, V_2 \qquad C = C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = V_1 \uplus V_2}{\Gamma \vdash P_1 \mid P_2 \rightsquigarrow C, B, V} \quad \text{PC-PAR}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1 \qquad \Gamma \vdash M_2 \rightsquigarrow T_2, C_2, B_2, V_2 \qquad \Gamma \vdash P_3 \rightsquigarrow C_3, B_3, V_3 \qquad \Gamma \vdash P_4 \rightsquigarrow C_4, B_4, V_4 \qquad C = C_1 \cup \cdots \cup C_4 \qquad B = B_1 \cup \cdots \cup B_4 \qquad V = V_1 \uplus \cdots \uplus V_4}{\Gamma \vdash \mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_3\ \mathsf{else}\ P_4 \rightsquigarrow C, B, V} \quad \text{PC-IF}$$

$$\frac{\Gamma \vdash M \rightsquigarrow T, C_1, B_1, V_1 \qquad \Gamma, E \vdash P \rightsquigarrow C_2, B_2, V_2 \qquad C = \{T \doteq \mathsf{Ok}(E)\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = \{E\} \uplus V_1 \uplus V_2}{\Gamma \vdash \mathsf{exercise}\ M; P \rightsquigarrow C, B, V} \quad \text{PC-EX}$$

$$\frac{\Gamma \vdash M \rightsquigarrow T, C, B, V}{\Gamma \vdash \mathsf{begin}\ l(M) \rightsquigarrow C, B, V} \quad \text{PC-BEGIN}$$

$$\frac{\Gamma \vdash M \rightsquigarrow T, C_1, B_1, V_1 \qquad C = \{l(M) \in \mathit{effects}(\Gamma)\} \cup C_1}{\Gamma \vdash \mathsf{end}\ l(M) \rightsquigarrow C, B_1, V_1} \quad \text{PC-END}$$

$$\frac{\Gamma \vdash \diamond \rightsquigarrow B}{\Gamma \vdash \mathsf{nil} \rightsquigarrow \emptyset, B, \emptyset} \quad \text{PC-NIL}$$

Table 3.5: Process constraint generation rules

We first give an informal presentation of the algorithm for solving type constraints and the principles behind it. We then give a formal definition and proofs of correctness. In the informal description we ignore boundaries. Note that by Lemma 20 we have $T_1 \equiv T_2$ if and only if $\mathit{nf}(T_1) \equiv \mathit{nf}(T_2)$, and in the following we assume all types to be on normal form.

Given a constraint set C the algorithm either fails or outputs a substitution σ and a constraint set C_{ef} containing only effect constraints. σ partially satisfies C in the sense that if another substitution satisfies C_{ef} then the two substitutions can be combined to satisfy C. Effect constraints in C are ignored during type solving and end up in C_{ef} together with any additional effect constraints generated during solving. Furthermore, in the interest of generating a minimal substitution the algorithm does not always create a full substitution in that σ might not assign to all type variables in C. Since there can be constraints on these variables the algorithm also returns a set C_{ty} of type constraints that must be satisfied by any substitution assigning to the left-over type variables. However, for any C_{ty} there exists a trivial extension to σ assigning types to all type variables and satisfying C_{ty}. We shall return to this briefly. The two constraint sets can be combined so the algorithm returns a substitution and a single constraint set.

The algorithm can be seen as standard unification up to a set of ignored constraints. In outline, given a constraint set C the recursive algorithm proceeds with steps

1. Normalise all types in C (using type instantiation)
2. Pick a type constraint $T_1 \doteq T_2$ not on form $X \doteq X\mu$ nor $X\mu \doteq X'\mu'$; if there are none then return the empty substitution along with C; otherwise let C' be the set of remaining constraints and continue to the next step
3. Apply the rule matching the picked type constraint; fail in no rule applies
   - If T_1 and T_2 have the same top-most type constructor then break them down and recursively solve C' unioned with the immediate constituents. For instance, if $T_1 = \mathsf{Ch}(T_1')$ and $T_2 = \mathsf{Ch}(T_2')$ then recursively solve $\{T_1' \doteq T_2'\} \cup C'$.
   - If T_1 is X then recursively solve C' with X replaced by T_2. The recursive call returns a substitution σ which is extended with $X \mapsto T_2$ by forming $\sigma \circ [X \mapsto T_2]$. Note the usage of explicit message substitution in types; having them allows us to define the variable to be whatever is on the right hand side, even if it is a type under message substitution. An occur-check is performed to make sure X does not occur in T_2.
   - If T_1 is $X\mu$ then the constraint can only be satisfied if T_1 have the same construction as T_2 since message substitutions cannot change the construction of a type. This is captured by stating that X must match the opening of T_2, i.e. recursively solve C (and not C' as in the previous rule) but with X replaced by $\mathit{open}(T_2)$. The recursive call returns a substitution σ which is extended with $X \mapsto \mathit{open}(T_2)$ by forming $\sigma \circ [X \mapsto \mathit{open}(T_2)]$. Note that T_2 cannot be a variable nor a variable under substitution but must consist of at least one type constructor; if T_2 were a variable the opening would also be a variable and the algorithm would not terminate. An occur-check is performed to make sure X does not occur in T_2.
4. Return the (possibly extended) substitution from the recursive call along with the set of effect and ignored constraints also produced by the recursive call.

The algorithm only fails if one of the following type constraints occur

- $X \doteq T$ and $X \in FV(T)$ (with $T \neq X$ and $T \neq X\mu$)
- $X\mu \doteq T$ and $X \in FV(T)$ (with $T \neq X'\mu'$)
- $T_1 \doteq T_2$ and $T_1 \not\approx T_2$

The occur-check $X \in FV(T)$ causes the first two cases to fail since no type assignment to X would satisfy the equation*. The last case causes failure since no types of different construction can be equivalent by Lemma 23.

We see that the algorithm ignores constraints on form $X \doteq X\mu$. As mentioned above this is to keep the substitution minimal. To satisfy the constraint, any type substituted for X must clearly ignore message substitutions μ by not containing any holes occurring in μ. Also, as nothing is known about X there are clearly many correct assignments, one such being $X \mapsto \mathsf{A}$ for any base type A. However, to keep the substitution minimal the constraint is ignored when solving but kept as a requirement that must be satisfied by any extending substitution assigning to X, but at this point assign nothing to X. Note that it has to be the same variable on both sides of the equality; for $X \doteq X'\mu$ with $X \neq X'$ the algorithm proceed as normal.

Constraints on form $X\mu \doteq X'\mu'$ are also ignored during solving but kept in the constraint set. By the same argument as above the substitution is kept minimal by not assigning to X nor X'. The variables do not have to be different (in the special case where the constraint is on identical types we can remove the constraint).

Note that the ignored type equality constraint cannot be discarded completely. For instance, in constraint set

$$C = \left\{ \begin{array}{l} X\langle M/\omega\rangle \doteq X'\langle M'/\omega'\rangle \\ X \doteq \mathsf{Ch}(T_1) \\ X' \doteq \mathsf{Pair}(T_2, T_3) \end{array} \right\}$$

we cannot simply throw away $X\langle M/\omega\rangle \doteq X'\langle M'/\omega'\rangle$ as this is the equation relating X and X', in turn making C unsatisfiable. Instead, the equation is ignored until one of the variables is assigned to. As noted, if there are only ignored equations left in the constraint set C the algorithm returns them as the set of constraints that must be satisfied by any extending substitutions along with the produced substitution.

Moving on to the formal definition, we say that a type equality constraint $T_1 \doteq T_2$ is *obviously unsatisfiable* if T_1 and T_2 have different type constructors or if T_1 is a variable X and $X \in FV(T_2)$ when T_2 is not a variable nor a variable under substitution; similar for the symmetric case.

Formally we present the algorithm as a reduction relation on a triple $(C, \sigma)_B$ consisting of a constraint set C, a substitution σ, and a boundary set B. It is assumed that all types in constraint set C are normalised before rule application and that B contain no type constructors. To solve a constraint set C simply reduce $(C, [\,])_B$ using the rules until no more reduction is possible (where $[\,]$ is the empty substitution). The resulting triple

*A recursive type would satisfy the equation but the type system allows no such.

$(C', \sigma')_{B'}$ has the property that if C' does not contain an obviously unsatisfiable constraint then a solution for C depends only on the satisfiability of the remaining effect constraints: if the effect constraints are satisfiable under B' then there exists a substitution satisfying C.

Definition 41 (**Type Solving Rules**)**.** Let $\xrightarrow{T}$ be the least transitive relation on triples $(C, \sigma)_B$ closed under the rules in Table 3.6. We say $(C, \sigma)_B$ *reduces to* $(C', \sigma')_{B'}$ if $(C, \sigma)_B \xrightarrow{T} (C', \sigma')_{B'}$.

$$\frac{T_1 \equiv T_2}{\big(\{C, T_1 \doteq T_2\}, \sigma\big)_B \xrightarrow{T} \big(C, \sigma\big)_B} \quad \text{TS-TRIV}$$

$$\frac{}{\big(\{C, \mathsf{Ch}(T_1) \doteq \mathsf{Ch}(T_2)\}, \sigma\big)_B \xrightarrow{T} \big(\{C, T_1 \doteq T_2\}, \sigma\big)_B} \quad \text{TS-CH}$$

$$\frac{}{\big(\{C, \mathsf{Pair}(T_1, T_2) \doteq \mathsf{Pair}(T_1', T_2')\}, \sigma\big)_B \xrightarrow{T} \big(\{C, T_1 \doteq T_1', T_2 \doteq T_2'\}, \sigma\big)_B} \quad \text{TS-PAIR}$$

$$\frac{}{\big(\{C, \mathsf{Ok}(\dot{S}_1) \doteq \mathsf{Ok}(\dot{S}_2)\}, \sigma\big)_B \xrightarrow{T} \big(\{C, \dot{S}_1 \doteq \dot{S}_2\}, \sigma\big)_B} \quad \text{TS-OK}$$

$$\frac{X \notin FV(T)}{\big(\{C, X \doteq T\}, \sigma\big)_B \xrightarrow{T} \big([X \mapsto T]C, [X \mapsto T] \circ \sigma\big)_{[X \mapsto T]B}} \quad \text{TS-VAR1}$$

$$\frac{T \neq X' \qquad T \neq X'\mu' \qquad X \notin FV(T) \qquad T' = \mathit{open}(T)}{\big(\{C, X\mu \doteq T\}, \sigma\big)_B \xrightarrow{T} \big([X \mapsto T']\{C, X\mu \doteq T\}, [X \mapsto T'] \circ \sigma\big)_{[X \mapsto T']B}} \quad \text{TS-VAR2}$$

Table 3.6: Type constraint solving rules

$X \notin FV(T)$ in rules TS-VAR1 and TS-VAR2 are the occur-checks and also prevent solving equations on form $X \doteq X\mu$. Condition $T \neq Y$ and $T \neq Y\mu'$ prevent TS-VAR2 from being applied if TS-VAR1 can be applied, as well as prevent solving equations on form $X\mu \doteq X'\mu'$. The extension of σ in TS-VAR1 and TS-VAR2 is by composition, and the update of B is by replacement. We write $\{C, T_1 \doteq T_2\}$ in place of $C \cup \{T_1 \doteq T_2\}$.

To each constraint set C we can associate a pair of natural numbers (n_v, n_s) where n_v is the number of distinct type variables occuring in C, and n_s is the sum of the number of type constructors used in types in C. For all cases but TS-VAR2 this pair is seen to decrease lexicographically by each rule. However, for TS-VAR2 the opening T' may introduce new variables (and increase the sum of type constructors) thereby breaking the lexicographical descend. We might argue that since we never open just a variable but only types with at least one type constructor, any variables occurring in T' are in some sense placeholders for types with a smaller "height" than the variable X being substituted. Since all types have a finite "height" it can only be decreased a finite number of times, and since X is substituted immediately after opening it can only be opened once. Unfortunately, we do

not know if this can be turned into a proof and can only conjecture that the type solving rules terminates. We return briefly to this in Section 4.3.

Conjecture 42 (**Termination**)**.** A triple $(C, \sigma)_B$ can be reduced only a finite number of times.

Lemma 43. If $(C, \sigma)_B$ cannot be reduced then any constraint in C is either on the form $X \doteq X\mu$, $X\mu \doteq X'\mu'$, an effect constraint, or an obvious non-satisfiable type constraint.

Proof. By case analysis of the types T_1 and T_2 occurring in any type constraint $T_1 \doteq T_2$ in C. See Section A.3 for proof details. □

Lemma 44. Assume constraint set C only contains type constraints on form $X \doteq X\mu$ or $X\mu \doteq X'\mu'$. Then there exist a substitution σ satisfying C. In particular, for any boundary set B containing only boundaries on type variables there exists a ground substitution σ satisfying C and staying within B.

Proof. If we assign only types unaffected by any explicit substitution occurring in any type in C we can forget any explicit substitution occurring in C. Having only simple type equations on form $X \doteq X'$ left we can easily produce a substitution σ satisfying C. Note that it is easy to find types unaffected by any explicit substitution, for instance base types or ok types with no holes. In particular, it is easy to produce a ground substitution that also stays within B. In the simplest case the same base type A is assigned to all variables. □

Lemma 45. If $(C, \sigma)_B \xrightarrow{T} (C', \sigma')_{B'}$ then there exists a substitution σ'' such that $\sigma' = \sigma'' \circ \sigma$ and for all substitutions σ''' satisfying C' we have that $\sigma''' \circ \sigma''$ satisfying C. Furthermore, if σ''' is a ground substitution staying within B' then $\sigma''' \circ \sigma''$ is a ground substitution staying within B.

Proof. The proof is by induction in the length of the derivation of $\xrightarrow{T}$. See Section A.3 for proof details. □

Lemma 46 (**Soundness**)**.** Assume $(C, \sigma)_B \xrightarrow{T} (C', \sigma')_{B'}$ where no further reduction is possible. If C' is not obviously unsatisfiable and there exists σ_E satisfying all effect constraints in C' then there exists a substitution σ''' satisfying C. Furthermore, if σ_E is a ground substitution staying within effect boundaries in B' then there exists a ground substitution σ''' satisfying C and staying within B.

Proof. We only show the case with ground substitutions. Since C' is not obviously unsatisfiable Lemma 43 gives that C' only contains type constraint on form $X \doteq X\mu$ or $X\mu \doteq X'\mu'$ together with any effect constraints. Partition B' into a set B'_T of boundaries on types and a set B'_E of boundaries on effects so that $B' = B'_T \cup B'_E$. By Lemma 44 there exists a ground substitution σ_T satisfying all type constraints in C' and staying within B'_T. Also, by assumption there exists a ground substitution σ_E satisfying all effect constraints in C' and staying within B'_E. We can assume without loss of generality that σ_E only assign to effect variables, i.e. that $dom(\sigma_E) \subseteq EVar$. Now let $\sigma_{triv} = \sigma_T \circ \sigma_E$ which is clearly a ground substitution satisfying C' and staying within B'. By Lemma 45 there exists σ'' such that $\sigma' = \sigma'' \circ \sigma$. Furthermore, $\sigma_{triv} \circ \sigma''$ is a ground substitution satisfying C and staying within B. □

Lemma 47. If there exists a substitution σ satisfying $\{X\mu \doteq T\} \cup C$ then there exists σ' satisfying $\{X \doteq T', X\mu \doteq T\} \cup C$ for $T' = \mathit{open}(T)$. Furthermore, for any boundary set B, if σ stays within B then σ' also stays within B.

Proof. The opening of a type respects the structure and introduces fresh variables, so since σ satisfies $X\mu \doteq T$ we can use type σX to construct σ' such that $\sigma' \backslash_{FV(T')} = \sigma$ and $\sigma' T' \equiv \sigma' X$. σ' clearly satisfies $\{X \doteq T', X\mu \doteq T\} \cup C$. Assume σ stays within B. Since σ' is σ extended with values for fresh variables $FV(T')$ we have that σ' also stays within B. □

Lemma 48. Assume $(C, \sigma)_B \xrightarrow{T} (C', \sigma')_{B'}$. If substitution δ satisfies C then there exists a substitution δ' satisfying C'. Furthermore, if δ stays within B then δ' stays within B'.

Proof. By induction in the length of the derivation of $\xrightarrow{T}$. The only interesting case is TS-VAR2 where we apply Lemma 47. See Section A.3 for proof details. □

Lemma 49 (Completeness). Assume $(C, \sigma)_B \xrightarrow{T} (C', \sigma')_{B'}$ where no further reduction is possible. If substitution δ satisfyies C then there exists a substitution δ' satisfying C'. Also, C' contains only type constraints on form $X \doteq X\mu$, $X\mu \doteq X'\mu$, or effect constraints. Furthermore, if δ stays within B then δ' stays within B'.

Proof. Follows at once by Lemma 48 and Lemma 43. □

3.4 Solving Effect Constraints

Since the previous sections showed that the problem of type inference can be reduced to the problem of solving effect constraints we next consider how to do this.

We start with a general algorithm suitable for the non-deterministic abstraction used in our type system and show that it is sound and complete with respect to the typing rules. The algorithm needs to make choices or guesses during computation and, as is often the case with choices, this leads to a high (computational) complexity. The general algorithm can be adapted to support different abstraction strategies and we next discuss how it can be made more efficient by assuming the principle of maximal abstraction thereby making abstraction deterministic. We end the section by mentioning an alternative approach based on a least fixed-point logic.

3.4.1 The General Algorithm

The main idea behind the algorithm is simply to recursively pick an effect constraint not satisfied by the current effect substitution and update the substitution to satisfy the constraint. The update may give raise to new unsatisfied constraints so the algorithm proceeds recursively until either a substitution satisfying all constraints is produced or it finds that the constraints cannot be satisfied. The algorithm is very simple and very brute force, consisting fundamentally of a fixed-point algorithm with choices.

For instance, to solve constraint set

$$C = \left\{ \begin{array}{l} l(n) \in E_1 \\ E_1 \doteq E_2 \end{array} \right\}$$

with the empty effect currently assigned to E_1 and E_2, we update the substitution such that E_1 contains $l(n)$. To satisfy the constraints under the updated substitution $l(n) \in E_2$ must further be satisfied and hence the substitution is updated such that both E_1 and E_2 contains $l(n)$. Had the constraint set furthermore contained constraint

$$E_2 \sqsubseteq \emptyset$$

we see that $l(n) \in E_2$ cannot be satisfied and hence no solution exist for the constraint set.

Note that requirements can be solved independently of each other. For instance, in constraint set

$$C = \left\{ \begin{array}{l} l(M) \in E \\ l'(M') \in E' \end{array} \right\}$$

it makes no difference if we make sure the former requirement is satisfied before satisfying the latter or vice versa[†].

All but very few simple constraint sets contain one or more constraints with explicit substituions or unions. In these cases there may be several possible updates satisfying a requirement and some opdates may lead to unsatisfiability. In the general algorithm we non-deterministically "guess" which update leads to a solution (if one exist) by simply trying all possibilities. Since the failure of one choice does not imply no good choice exists we cannot report failure if one choice turns out to be unsatisfyable but may have to backtrack and try another option.

For instance, in solving constraint set

$$C = \left\{ \begin{array}{l} l(n) \in E_1 \cup E_2\langle n/1 \rangle \\ E_1 \sqsubseteq \emptyset \end{array} \right\}$$

we might first try to satisfy $l(n) \in E_1 \cup E_2\langle n/1 \rangle$ by updating E_1 to contain $l(n)$. This leads to the unsatisfiable $l(n) \sqsubseteq \emptyset$ and hence we have to backtrack and try to satisfy the requirement by updating E_2 to contain $l(n)$. Since E_2 has no upper bound this leads to a solution. Note that there were two updates of E_2 that would satisfy the requirement: $l(n)$ and $l(1)$. Had C also contained

$$\begin{array}{l} E_3 \doteq E_2\langle m/1 \rangle \\ E_3 \sqsubseteq \{l(m)\} \end{array}$$

we get that only update $l(1)$ leads to a solution since $l(n)$ in E_2 would propagate to E_3 and $n \not\equiv m$.

[†]It would make a difference though if effects could have size limits in which case one atomic effect could block for another.

The substitution we are looking for must also stay within the boundaries given. For instance, process

$$\begin{array}{l}\mathsf{new}\ n : N;\\ \quad \mathsf{new}\ m : M; \quad \left|\ \begin{array}{l}\mathsf{in}\ n\ x;\\ \quad \mathsf{exercise\ snd}\ x;\\ \quad \mathsf{end}\ l(\mathsf{fst}\ x)\end{array}\right.\\ \qquad \mathsf{begin}\ l(m)\ |\\ \qquad \mathsf{out}\ n\ (m, \mathsf{ok})\end{array}$$

under the empty typing context yields effect constraints C and boundaries B

$$C = \left\{ \begin{array}{l} E_0 \doteq E_2\langle m/1\rangle \\ E_1 \doteq E_2\langle \mathsf{fst}\ x/1\rangle \\ E_0 \sqsubseteq \{l(m)\} \\ l(\mathsf{fst}\ x) \in E_1 \end{array} \right\} \qquad B = \left\{ \begin{array}{l} E_2 \preceq \emptyset \\ E_2 \leq 1 \end{array} \right\}$$

Assume σ assigns the empty effect to E_0 and E_2, and $l(\mathsf{fst}\ x)$ to E_1 in order to satisfy $l(\mathsf{fst}\ x) \in E_1$. For σ to satisfy $E_1 \doteq E_2\langle \mathsf{fst}\ x/1\rangle$ we must extend E_2. However, because of the boundaries, the only option is to let E_2 include $l(1)$ as $l(\mathsf{fst}\ x)$ would not stay within the boundary $E_2 \preceq \emptyset$.

In the spirit of this paper let us formally define the effect solving algorithm as a relation between triples consisting of a constraint set C, a boundary set B, and a substitution σ. Since C and B remain fixed during the computation we denote relationship between two triples as $C, B \vdash \sigma \xrightarrow{S} \sigma'$ instead of $(C, \sigma)_B \xrightarrow{S} (C, \sigma')_B$.

Let a ground substitution σ be *too big* relative to a number n if σ maps some effect variable E to an effect with cardinality greater than n, i.e. $|\sigma E| > n$ for some $E \in dom(\sigma)$.

Definition 50 (Effect Solving Rules). Let $\xrightarrow{S}$ be the least transitive relation on triples $(C, \sigma)_B$ closed under the rules in Table 3.7. We say there is a *path* from σ to σ' under constraint set C and boundary set B if $C, B \vdash \sigma \xrightarrow{S} \sigma'$. We assume the rules are parameterised by a number n and have as a general premise (not mentioned in the rules) that σ' must not be too big relative to n. Also, σ' must stay within B. If σ' violates either general premise no rule applies.

Note that, contrary to the type solving rules, substitutions are extended by updates and not composition. Function $UV(\dot{S})$ gives the underlying type of $\dot{S}$ if it is a variable or a variable under explicit substitution and is undefined otherwise. We write $\{e, C\}$ in place of $\{e\} \cup C$ for effect constraint e.

Intuitively, the rules can be seen as propagation of requirements or as saturation of the substitution. This is closely related to fixed-point theory, the only difference being that backtracking can occur due to the non-deterministic guesses that must be made. More precisely, the rules can be interpreted as a function taking as input a substitution and extending this substitution if needed to "better" satisfy C. For instance, rule ES-EQ1 extend σ to "better" match the requirement that $\dot{S}_1$ and $\dot{S}_2$ must contain the same atomic effects; which is not currently satisfied since $\dot{S}_2$ contains an atomic effect $l(\dot{M})$ not found in $\dot{S}_1$. A fixed-point of the function is a substitution requiring more no updates, i.e. a substitution

$$\frac{\begin{array}{c} l(\dot{M}) \in \sigma\dot{S}_2 \qquad l(\dot{M}) \notin \sigma\dot{S}_1 \qquad UV(\dot{S}_1) = E_1 \\ l(\dot{M}') \in E_1 \Rightarrow l(\dot{M}) \in \sigma\dot{S}_1 \qquad \sigma' = [E_1 \mapsto \{l(\dot{M}')\} \cup \sigma E_1]\sigma \end{array}}{\{\dot{S}_1 \doteq \dot{S}_2, C\}, B \vdash \sigma \xrightarrow{S} \sigma'} \text{ES-EQ1}$$

$$\frac{\begin{array}{c} l(\dot{M}) \in \sigma\dot{S}_1 \qquad l(\dot{M}) \notin \sigma\dot{S}_2 \qquad UV(\dot{S}_2) = E_2 \\ l(\dot{M}') \in E_2 \Rightarrow l(\dot{M}) \in \sigma\dot{S}_2 \qquad \sigma' = [E_2 \mapsto \{l(\dot{M}')\} \cup \sigma E_2]\sigma \end{array}}{\{\dot{S}_1 \doteq \dot{S}_2, C\}, B \vdash \sigma \xrightarrow{S} \sigma'} \text{ES-EQ2}$$

$$\frac{\begin{array}{c} l(M) \notin \bigcup \sigma\dot{S}_i \qquad UV(\dot{S}_j) = E_j \\ l(\dot{M}') \in E_j \Rightarrow l(M) \in \sigma\dot{S}_j \qquad \sigma' = [E_j \mapsto \{l(\dot{M}')\} \cup \sigma E_j]\sigma \end{array}}{\{l(M) \in \dot{S}_1, \ldots, \dot{S}_n, C\}, B \vdash \sigma \xrightarrow{S} \sigma'} \text{ES-IN}$$

$$\frac{\begin{array}{c} l(\dot{M}) \in \sigma\dot{S} \qquad l(\dot{M}) \notin \bigcup \sigma\dot{S}_i \qquad UV(\dot{S}_j) = E_j \\ l(\dot{M}') \in E_j \Rightarrow l(\dot{M}) \in \sigma\dot{S}_j \qquad \sigma' = [E_j \mapsto \{l(\dot{M}')\} \cup \sigma E_j]\sigma \end{array}}{\{\dot{S} \sqsubseteq \dot{S}_1, \ldots, \dot{S}_n, C\}, B \vdash \sigma \xrightarrow{S} \sigma'} \text{ES-SUB}$$

Table 3.7: Effect constraint solving rules

satisfying all constraints in C.

Non-deterministic choices occur in all rules either in the form of guessing which $\dot{M}'$ to extend the assignment with or in the form of guessing which effect E_j in a union to extend. As argued, picking which rule to apply next can be done independently and hence cannot require backtracking. In praxis it would make the algorithm faster if each effect requirement $l(M) \in \dot{S}_1, \ldots, \dot{S}_n$ (corresponding to each end event) is solved separately one at a time. Doing this decreases the amount of work wasted if backtracking is required. Also, to improve the expected running time the algorithm could try most likely paths first; one such heuristic would be to make maximal abstraction the first choice.

For any set of effect variables let $[\emptyset]$ be the substitution assigning the empty effect to all. To compute a substitution satisfying C and staying within B the algorithm proceeds with steps

1. instantiate the rules with the number of effect requirements $l(M) \in \dot{S}_1, \ldots, \dot{S}_n$ occurring in C
2. test if $[\emptyset]$ stays within B; fail if not
3. look for a path from $[\emptyset]$ to a substitution σ satisfying C; fail is none found

A path exists in step 3 if and only if there exists a substitution satisfying C and staying with B. We have that the algorithm is sound and complete and terminates:

Lemma 51. If there exists a substitution staying within boundary set B then $[\emptyset]$ stays within B.

Lemma 52. If σ stays within boundary set B and $C, B \vdash \sigma \xrightarrow{S} \sigma'$ then σ' stays within B.

Lemma 53 (**Soundness**)**.** Assume the algorithm produces a substitution σ. Then σ satisfies C and stays within B.

Proof. By definition of the algorithm we have $C, B \vdash [\emptyset] \xrightarrow{S} \sigma$ where σ satisfies C. Since the algorithm does not fail we have that $[\emptyset]$ stays within B and the result follows by Lemma 51 and Lemma 52. □

Lemma 54. If there exists a substitution σ satisfying C and staying within B then there exists a substitution σ' satisfying C and staying within B and which is not too big relative to the number of effect requirements in C.

Proof. Follows from the fact that if the process P from which C and B are generated is typable, then the type derivation tree does not need to include any effects whose cardinality is larger than the number of end-events occurring in the process as only one begin-event is needed to match an end-event. In the worst case, each end-event requires a distinct begin event and they all need to be transfered by the same channel. In this case the effect of the channel is the number of end-events in the process. Finally, the constraint generation rules ensures that the number of effect requirements in C is the same as the number of end-events. □

Completeness is a consequence of the previous lemma and the systematic exploration of all possible substitutions performed by the algorithm:

Lemma 55 (**Completeness**)**.** Assume there exists a substitution satisfying C and staying within B. Then the algorithm does not fail.

Proof. By Lemma 51 the first test will not fail, and by Lemma 54 there is a satisfying substitution not too big relative to the limit. That the satisfying substituion is in the search space of the algorithm is implied by the fact that the algorithm tries all possible ways to satisfy the effect requirements in the constraint set. □

Due to the size limit termination is an easy consequence:

Lemma 56 (**Termination**)**.** There are only finitely many paths from $[\emptyset]$.

Proof. Every application of a rule extends the substitution so since there are only finitely many effect variables we have that only finitely long paths exists. The number of effects in unions is finite, and the number of possible messages $\dot{M}'$ to extend an effect with is finite; the only slightly non-trivial case is when abstraction is performed but here we have that there are only finitely many ways to abstract a message out of another message. Combined we get the desired result. Relating to the algorithm we have that it will always find a satisfying substitution, run out of choices, or hit the size limit; in the latter two cases it fails. □

Let us compute a limit on the size of the search space, i.e. the set of all substitutions not too big relative to the limit. Note that this can be larger than the search space since the algorithm will not consider substitutions which does not help in satisfying the constraints.

We first need to introduce a few functions. Let the *height* $h(M)$ of a message M be the number of message pair constructors occuring in M:

$$\begin{aligned} h(n) &= 0 \\ h(\mathsf{ok}) &= 0 \\ h(\mathsf{fst}\ M) &= h(M) \\ h(\mathsf{snd}\ M) &= h(M) \\ h((M_1, M_2)) &= max\big(h(M_1), h(M_2)\big) + 1 \end{aligned}$$

and let the *type height* $i(M)$ of message M be the number of pair types indicated by M:

$$\begin{aligned} i(n) &= 0 \\ i(\mathsf{ok}) &= 0 \\ i(\mathsf{fst}\ M) &= i(M) + 1 \\ i(\mathsf{snd}\ M) &= i(M) + 1 \\ i((M_1, M_2)) &= max\big(i(M_1), i(M_2)\big) + 1 \end{aligned}$$

Lemma 57. The number of different messages to consider in effect assignments is limited by $2p^{\mathcal{O}(log\ p)}$ where p is the size of the process from which the constraint set is generated.

Proof. At most p messages appear in the process. Message instantiation can only increase the height of a message as a hole is filled by a message and messages occuring in events are fully instantiated since they cannot contain holes. These facts imply that any message in an effect higher than a maximal height message occuring in an event makes no difference in matching events and hence the height h_m of a maximal height message occuring in an event provides an upper limit on the height of messages in effects. Function i gives an upper limit i_m on the maximal index used in effects by taking the maximum over all messages appearing in events. We see that the number of possible effect messages is limited by $\Sigma_{0 \leq j \leq h_m}(p + i_m)^{j+1} \leq (p + i_m)^{h_m+1} \cdot h_m$. Furthermore, since $h_m \leq \mathcal{O}(log\ p)$ and $i_m \leq p$ the number of possible effect messages is limited by $2p^{\mathcal{O}(log\ p)}$. □

Lemma 58. For any number n and finite set V of effect variables there are only finitely many substitutions σ with $dom(\sigma) = V$ such that σ is not too big relative to n. Moreover, this number is limited by $2^{\mathcal{O}(log^2 p) \cdot n \cdot v}$ where p is the size of the process from which constraints are generated and v is the cardinality of V.

Proof. By Lemma 57 the number of different messages to consider is limited by $2p^{\mathcal{O}(log\ p)}$. Then the number of different substitutions to consider is limited by $2p^{\mathcal{O}(log\ p) \cdot v \cdot n} \leq 2^{\mathcal{O}(log^2 p) \cdot n \cdot v}$. □

Each effect requirement in C corresponds exactly to one end event in the process from which C is generated, so since we chose the limit n to be the number of effect requirements in C we have that n is limited by p: $n \leq p$. By the above lemma we have that the number of substitutions in the search space is limited by $2^{\mathcal{O}(log^2 p) \cdot p \cdot v} \leq 2^{\mathcal{O}(p^3) \cdot v}$.

Besides being non-deterministic the algorithm also has the disadvantage that a produced assignment is not necessarily the smallest nor a minimal solution. Indeed, for

$$C = \left\{ \begin{array}{l} l(n) \in E_1 \\ l(m) \in E_2 \\ E_1 \doteq E_3\langle n/1 \rangle \\ E_2 \doteq E_3\langle m/1 \rangle \end{array} \right\}$$

and an empty bound set the algorithm may output either

$$\begin{array}{lll} E_1: & l(n) & \\ E_2: & l(m) & \text{or} \\ E_3: & l(1) & \end{array} \qquad \begin{array}{lll} E_1: & l(n),\ l(m) & \\ E_2: & l(n),\ l(m) & \\ E_3: & l(n),\ l(m) & \end{array}$$

which are both solutions.

3.4.2 Maximal Abstraction Variant

The problem with the general algorithm is that a triple (C, B, σ) may have several next steps according to $\xrightarrow{S}$ and some of these may lead to failure while others do not; moreover, we have to find a successful next step before declaring victory and we have to try all next steps before accepting (overall) defeat. Without any guidance on which union to choose or which abstraction to perform this path exploration becomes expensive.

The algorithm can be made more efficient if we can remove guessing, effectively making the algorithm capable of generating its own guidance. This means the algorithm must deterministically determine which effect in a union to update as well as which message $\dot{M}'$ to update the effect with in order to satisfy a constraint. In this configuration there is no need for the algorithm to backtrack so after at most $\mathcal{O}(p \cdot v)$ updates, where v is the number of effect variables, the algorithm terminates as each effect is limited in size by the size p of the process. We have that by removing the choices we obtain a simpler fixed-point-only algorithm running in low polynomial time (relative to the number of effect variables).

One way to make the choice of message to update with deterministic would be to assume the principle of maximal abstraction. For instance, to satisfy requirement $l(n) \in E\langle n/1 \rangle$ we only consider the option of adding $l(1)$ to E (and not the option of adding $l(n)$). More generally, we assume that there is only one option for $\dot{M}'$ thereby making an exponential reduction of the number of paths. While it is easy to construct a constraint set solvable by the general algorithm but not by this adaption, it remains an open problem whether or not the adaption is incompleteness relative to the typing rules and the constraints generated by processes. At any rate, making the maximal abstraction the first choice for the general algorithm to try might turn out to match the expected behaviour and reduce the expected running time.

We leave deterministic union choice for future work.

3.4.3 Alternating Least Fixed-Point Logic

To end the section let us mention an alternative method for solving effect constraints that could turn out to be fruitful. We leave the investigation of this for future work.

The method based on Alternating Least Fixed-Point (ALFP) logic [34] expresses constraints as formulae over a proper subset of first-order logic which are then solved using e.g. the Succinct Solver [34]. This is the approach used in [17]. A model of a formula is an assignment to the relations used in the formula, and the advantage of ALFP is that if a formula is satisfiable then a minimal model of it can be constructed in low polynomial time; if the formula is not satisfiable then failure is reported. Expressing effects as relations and effect constraints as formulae between the relations, a minimal effect assignment could be constructed in low polynomial time. However, at the moment there seems to be some obstacles down this road.

First off, the Succinct Solver cannot do backtracking but must deal with the union of an effect bound constraint. Semantical restrictions require a ranking of relations dictating in which order the relations are populated by the Succinct Solver. The effect relations on the right hand side of an effect bound constraint could be ranked lower than the effect relation on the left hand side but it might be a problem that a process such as

$$\begin{array}{l}\mathsf{new}\ n : N;\ \mathsf{in}\ n\ x;\\ \quad \mathsf{exercise}\ \mathsf{snd}\ x;\\ \quad \mathsf{begin}\ l(m)\mid \mathsf{out}\ n\ (\mathsf{fst}\ x, \mathsf{ok})\end{array}$$

currently leads to effect constraint set

$$C = \left\{ \begin{array}{l} E_0 \doteq E_2\langle \mathsf{fst}\ x/1\rangle \\ E_1 \doteq E_2\langle \mathsf{fst}\ x/1\rangle \\ E_1 \sqsubseteq \{l(\mathsf{fst}\ x)\}, E_0 \end{array} \right\}$$

containing a cyclic dependency.

Secondly, it may be a problem to express union choices. However, if the relations on the right hand side of an effect bound are fully populated before the left hand side relation then union choice is not an issue since only a check of containment is needed in this case.

Finally, deterministic abstraction seems to be required to avoid backtracking. To satisfy this the principle of maximal abstraction could be employed. However, as mentioned above this might lead to incompleteness (if we insist on expressing type relationship using only application).

3.5 The Type Inference Algorithm

From the algorithms developed in the last sections the full type inference algorithm is defined as:

Definition 59 (**Type Inference Algorithm**)**.** Let the *type inference algorithm* operating on input (Γ, P) where Γ is a typing context and P a process be defined by steps:

1. Generate boundaries and type and effect constraints for (Γ, P) using the rules from Section 3.2. This yields a constraint set C and a boundary set B such that there exists a substitution satisfying C and staying within B if and only if P is typable under Γ.
2. Reduce the triple $(C, [\,])_B$ using type solving rules from Section 3.3 until no further reduction is possible. This gives a triple $(C', \sigma_T)_{B'}$ consisting of unsolved constraints, a boundary set, and a type substitution; formally we have $(C, [\,])_B \xrightarrow{T} (C', \sigma_T)_{B'} \not\xrightarrow{T}$. Fail if C' contain obviously unsatisfiable constraints or B' contain a ground type or effect not staying within the boundaries.
3. Let C_S be the set of effect constraints in C' and let B_S be the set of effect boundaries in B'. Use the effect solving algorithm from Section 3.4 to search for a substitution σ_S such that $C_S, B_S \vdash [\emptyset] \xrightarrow{S} \sigma_S$ and σ_S satisfies C_S. Fail if no such substitution can be found.
4. Form $\sigma = \sigma_S \circ \sigma_T$ and return $(C' - C_S, \sigma)$.

The following results are consequences of the results of this chapter:

Theorem 60 (**Soundness**)**.** If the type inference algorithm on input (Γ, P) gives an output then P is typable under Γ. Moreover, the output of the algorithm can be used to efficiently construct a type derivation tree for P under Γ.

Proof. The results of this chapter show how to find a substitution satisfying the generated constraints and staying within the generated boundaries, and furthermore show how to build a type derivation tree using the substitution: Lemma 53 shows that the effect substitution is sound, Lemma 46 shows how to construct a substitution satisfying the initial constraint set, and Lemma 38 shows how to construct a type derivation tree. □

Theorem 61 (**Completeness**)**.** If process P is not typable under typing context Γ then the type inference algorithm fails.

Proof. The results of this chapter rejects the generated constraints if P is not typable under Γ by Lemma 39, 49, and 55. □

As for termination, it is easy to see that constraint generation terminates and we have by Lemma 56 that effect constraint solving terminates. Unfortunately, we lack a proof that type constraint solving terminates and have only conjectured that it does in Conjecture 42. Hence, we can only conjecture that the type inference algorithm terminates:

Conjecture 62 (**Termination**)**.** The type inference algorithm terminates.

3.6 Example

To give an example of the type inference algorithm we have a process P consisting of three parallel processes sharing two networks:

$$
\begin{array}{l}
\mathsf{new}\ \mathit{opennet} : O; \\
\quad \mathsf{new}\ \mathit{safenet} : S; \\
\qquad \left(\begin{array}{l|l|l}
\mathsf{new}\ m : M; & !\mathsf{in}\ \mathit{opennet}\ y; & !\mathsf{in}\ \mathit{safenet}\ z; \\
\quad \mathsf{out}\ \mathit{opennet}\ m & \quad \mathsf{begin}\ \mathit{confirmed}(y) \mid & \quad \mathsf{exercise\ snd}\ z; \\
 & \quad \mathsf{out}\ \mathit{safenet}\ (y, \mathsf{ok}) & \quad \mathsf{end}\ \mathit{confirmed}(\mathsf{fst}\ z)
\end{array} \right)
\end{array}
$$

The left process is the *client*, the middle process a *proxy*, and the right process the *server*. The client wants to send a message m to the server but cannot do so directly. This would be the case for instance if the server needs to be protected from bad messages. Instead, the client must first send m to the proxy which will relay good message to the server (code performing a check on messages is not included). This is modelled by the client sending on *opennet* while the proxy is communicating with the server on *safenet*. To verify the authenticity property the process is annotated with *confirmed* events requiring any message received by the server to be confirmed. We see intuitively that the protocol satisfies the correspondence property and that it would not if the client were to send on *safenet* instead of *opennet*.

Running the constraint generation rules on the empty typing context and P we get a constraint set C which includes the following constrains (and some not shown):

$$C = \left\{ \begin{array}{lll} S \doteq \mathsf{Ch}(Z) & X_1 \doteq \mathsf{Pair}(Y, X_2) & Z \doteq \mathsf{Pair}(X_3, X_4) \\ S \doteq \mathsf{Ch}(X_1) & X_2' \doteq X_2\langle y/1\rangle & X_4' \doteq X_4\langle \mathsf{fst}\ z/1\rangle \\ O \doteq \mathsf{Ch}(Y) & X_2' \doteq \mathsf{Ok}(E_0) & X_4' \doteq \mathsf{Ok}(E_1) \\ O \doteq \mathsf{Ch}(M) & E_0 \sqsubseteq \{\mathit{confirmed}(y)\} & \mathit{confirmed}(\mathsf{fst}\ z) \in E_1 \\ & \vdots & \end{array} \right\}$$

where Y is the type of y, X_1 the type of (y, ok), $\mathsf{Ok}(E_0)$ the type of ok, Z the type of z, and X_4' the type of $\mathsf{snd}\ z$. We also get boundary set B:

$$B = \left\{ \begin{array}{lcll} O & \preceq & \emptyset & O \leq 0 \\ S & \preceq & \{\mathit{opennet}\} & S \leq 0 \\ M & \preceq & \{\mathit{opennet}, \mathit{safenet}\} & M \leq 0 \\ Y & \preceq & \{\mathit{opennet}, \mathit{safenet}\} & Y \leq 0 \\ Z & \preceq & \{\mathit{opennet}, \mathit{safenet}\} & Z \leq 0 \\ E_1 & \preceq & \{\mathit{opennet}, \mathit{safenet}, z\} & E_1 \leq 0 \end{array} \right\}$$

Reducing the triple $(C, [\,], B)$ using the type solving rules we get a triple (C', B', σ') where σ' includes

$$\sigma' = \left\{ \begin{array}{ll} O \mapsto \mathsf{Ch}(M) & X_2' \mapsto \mathsf{Ok}(E_0) \\ S \mapsto \mathsf{Ch}(\mathsf{Pair}(M, \mathsf{Ok}(E_2))) & X_2 \mapsto \mathsf{Ok}(E_2) \\ Y \mapsto M & X_4' \mapsto \mathsf{Ok}(E_1) \\ Z \mapsto \mathsf{Pair}(M, \mathsf{Ok}(E_2)) & X_4 \mapsto \mathsf{Ok}(E_2) \\ X_1 \mapsto \mathsf{Pair}(M, \mathsf{Ok}(E_2)) & X_3 \mapsto M \\ \vdots & \end{array} \right\}$$

C' contains the remaining (effect) constrains

$$C' = \left\{ \begin{array}{l} E_0 \doteq E_2\langle y/1\rangle \\ E_1 \doteq E_2\langle \mathsf{fst}\ z/1\rangle \\ E_0 \sqsubseteq \{\mathit{confirmed}(y)\} \\ \mathit{confirmed}(\mathsf{fst}\ z) \in E_1 \end{array} \right\}$$

and B' the updated boundary set

$$B' = \left\{ \begin{array}{lcll} M & \preceq & \emptyset & M \leq 0 \\ E_2 & \preceq & \{\mathit{opennet}\} & E_2 \leq 1 \\ M & \preceq & \{\mathit{opennet}, \mathit{safenet}\} & M \leq 0 \\ E_1 & \preceq & \{\mathit{opennet}, \mathit{safenet}, z\} & E_1 \leq 0 \\ & & \vdots & \end{array} \right\}$$

Note that a fresh effect variable E_2 was introduced because an opening was performed during the reduction, and that it is allowed to have one hole in it. Note also that B' contain more than one name boundary entry for M (in this case the later less restrictive entry can be removed).

Solving C' using the effect solving algorithm we get a substitution σ_E

$$\sigma_E = \left\{ \begin{array}{lcl} E_0 & : & \mathit{confirmed}(\mathsf{fst}\ z) \\ E_1 & : & \mathit{confirmed}(y) \\ E_2 & : & \mathit{confirmed}(1) \end{array} \right\}$$

We end up with a substitution satisfying C and staying within B, and hence ensuring that P is well-typed. Had the client tried to send (m, ok) on *safenet*, there would be an unsatisfiable effect bound constraint

$$E \sqsubseteq \emptyset$$

since no begin events occur in the client, in turn making the algorithm report failure.

Chapter 4

Conclusion

The goal of this thesis has been to research type inference for a complex type system. This has been done in a context of verification of security protocols. Specifically, the type system ensures that well-typed protocols in the form of pi calculus processes always respect an authenticity property expressed by correspondence events. Our approach is the product of a desire to make a simple and intuitive type inference algorithm.

Section 4.1 summarises our results, Section 4.2 discuss related work, and Section 4.3 lists some ideas for future work.

4.1 Summary

We have presented a correspondence certifying type system with dependent types and effects. Our main contribution is a constraint based type inference algorithm for the type system, using a unification algorithm to solve type constraints, and a non-deterministic algorithm for solving effect constraints. The output of the algorithm can be used to create a derivation tree for the analysed process, i.e. a proof that the process respects the specified authenticity property. We have implemented the algorithm in Caml and found that it performs efficiently in praxis; specifically, we experienced that backtracking is rarely required.

Besides showing that type inference is possible, our work also illustrates how to do type inference in the presence of bound names and dependent types with application and abstraction. Using explicit substitutions and *de Bruijn* indices we get that only the effect solving algorithm needs to deal with application and abstraction. Also, separation of type and effect constraint solving allow the search for more efficient inference algorithms to be focused on more efficient algorithms for solving effect constraints. We have furthermore presented a general effect solving algorithm supporting various abstraction strategies. Our constraints are expressed in a language very similar to the checks performed by the type checker, and being intuitive they allow for easy proofs of correctness. The same holds for boundaries for managing bound names, and it is our hope that this approach will make extending the type system and the inference algorithm easier.

Of concrete theoretical results, we have proved that our constraints and boundaries are

sound and complete relative to the type system; that our algorithms for solving type and effect constraint are sound and complete relative to the constraints; and that our algorithm for solving effect constraint terminates. We trust that the algorithm for solving type constraints also terminates but lack a formal proof (we elaborate on this below).

4.2 Related Work

For the pi and spi calculus there exist type systems for secrecy, authentication, and authorisation. Common for these type systems is that they are all sound but incomplete. Furthermore, while a type checker may verify the proof, they require manual proof generation in the form of a type assignment as no type inference algorithm is provided.

Type systems for secrecy [1, 24] classify data based on different privacy levels. For instance, for the spi calculus the type system in [1] will classify messages as either *public*, *secret* or *any*. Public messages can be communicated to anyone while secret messages should not be leaked. Given a type assignment the type checker certifies that this is ensured by the protocol. The type system in [24] for the pi calculus extends this, allowing for multiple and dynamically-generated security levels.

There are several type system for authentication [26, 22, 19, 20, 21, 23] ranging from being based on the pi calculus to the spi calculus, from injective to non-injective correspondences, and from using symmetric to asymmetric encryption. All use effects for collecting and matching correspondence events. As for secrecy, options exist for assigning types to messages, ordered under a subtype relation. Most notable is the *Un* (untrusted) type which the typing rules always permits and which is used for typing an adversary and unsafe messages. The Cryptyc tool [25] is an implementation of a type checker for some of these type systems based on the spi calculus. The tool verify safety via type checking but relies on the user to create the type annotation.

For authorisation, the type system in [16] verifies if a protocol correctly implements a security policy. Annotations marking the grant and the requirement of an access right can be checked by ensuring that each required right can be inferred from the granted rights. These statements and expectations generalise the begin- and end-events of correspondences.

Of course, there exist methods not based on type systems, one of which is ProVerif [7]. ProVerif allows for automated (proof generation and) verification of authenticity and secrecy properties via correspondences. Based on applied pi, it covers various cryptographic primitives including shared-key and public-key encryption, signatures, one-way hash functions, and Diffie-Hellman key agreements. A process is translated into a set of Horn clauses which is solved. It is a sound but incomplete method and without guarantee of termination. An implementation is available from [14].

Bodei et al. [8] developed a static analysis for a pi calculus that shows how names will be bound to actual channels at run time. The analysis establishes an upper bound on the set of channels to which a given name may be bound and on the set of channels that

may be sent along a given channel. The analysis runs in low polynomial time and can be used for establishing two simple security properties: that a process do not leak its secret channels and that a process preserves security clearance levels. Although their analysis is not formulated as a type system there are strong similarities, both in solution verification and construction. Solution construction is based on constraint generation and fixed-point solving.

The method proposal by our type system is inferior in expressive power to the methods mentioned above. First, ProVerif allow more properties and more operations to be modelled. However, ProVerif is not guaranteed to terminate; a property we believe our inference algorithm has although lacking a formal proof. Compared to the other type systems for authenticity, ours is weaker in that it does not support modelling of any form of cryptography as well as not guaranteeing robust safety, i.e. safety in the presence of any adversary. We believe our system can be extended to support these but leave this for future work.

The motivation for our "weak" type system, of course, is that it allows us to do sound and complete type inference. In this field of type reconstruction, related work includes the classical algorithm for type inference for the simply typed lambda calculus and the ML programming language [37]. While our algorithm is inspired by these type systems, they do not include, and hence not address, dependent types or effect types as do ours.

The work by Lhoussaine [29] on type inference for a type system with dependent types (similar to record types) for the distributed pi calculus [27] resembles our work but differs in that the dependent types do not bind names and have no instantiation of types. Our system furthermore contains effects for matching of begin- and end-events. Because of these differences it appears that solving for our effect types is more involved than solving for record types. Our formal presentation of algorithms as reduction relations is inspired by [29].

Closer to our work is that of Kobayashi et al. [28] describing type inference for a correspondence type system for a polyadic pi calculus. Based on an early type system by Gordon et al. their type system contains only dependent channel types with latent effects. While we consider non-injective correspondences theirs are injective with a twist: effects are rational numbers yielding constraints in the form of inequalities over rational numbers, thereby making the inference algorithm fundamentally different from ours. They obtain polynomial running time due to the fractional effects and show that the same type inference problem for natural numbers is NP-hard. As in our work, the type system only guarantees safety and not robust safety.

Type inference for a correspondence certifying type system with dependent types and effects is described in [17]. The type system and all constraints are expressed as formulae in the Alternating Least Fixed-Point (ALFP) logic and solved using the Succinct Solver [34]. Properties of the logic and solver ensure that a solution is produced in low polynomial time and with a minimal effect assignment. Since our work is based on [17] there is of course a strong relationship between the two. In particular, our pi calculus and type system is borrowed from [17] along with some of the insights used for type inference. We believe our constraints are easier to reason about since they are not expressed in the ALFP logic. Also, the more

natural constraint formulation may ease the task of extending the inference algorithm to more expressive type systems including for instance, type systems with injective correspondences (where effect are multi-sets and not sets) and systems with several possible types for messages (as used for typing robust safety). In general, not relying on the ALFP logic may allow for greater flexibility as we are not restricted to problems solvable by fixed-point computations. Contrary to [17], our method is not guaranteed to give a minimal solution and the running time is not guaranteed to be low polynomial. However, our algorithm is parameterisable and we believe it can be instantiated to match the low polynomial running time (at least for effect solving). It may also be possible to use the ALFP logic to solve effect constraints and gain the desirable properties but reducing the use and need of the logic.

As a final, more general note, observe that dependent types occur in many type systems, sometimes breaking decidability of type checking. Strong dependent types allow for e.g. lambda terms or first order predicate formulae inside types, leading to undecidability of type checking [6, 36, 38]. Other type systems with dependent types (such as [42]) stay within the decidability boundary by restricting the terms inside types. Our dependent types fit in the latter decidable category since only message terms occur inside dependent types and equivalence of message terms is decidable.

4.3 Future Work

A next step would be to find a better effect solving algorithm and investigate the impact of maximal abstraction. A deeper study of the ALFP logic and its solver might also prove useful, as could a study in the possibility of expressing effect constraints in the logic. An analysis of worst case computational complexity would also be beneficial. Based on experience with the implementation the average case complexity of type solving (and of effect solving) appears to be low. However, it also seems that processes such as

$$
\begin{array}{l}
\mathsf{new}\ n_0; \\
\quad \mathsf{new}\ n_1; \\
\quad\quad \ldots \\
\quad\quad \mathsf{new}\ n_k; \\
\quad\quad\quad \mathsf{out}\ n_0\ (n_1, n_1)\ | \\
\quad\quad\quad \mathsf{out}\ n_1\ (n_2, n_2)\ | \\
\quad\quad\quad \ldots \\
\quad\quad\quad \mathsf{out}\ n_{k-1}\ (n_k, n_k)
\end{array}
$$

leads to type solving taking exponential time (in the size of the process). Though this might imply that overall exponential time worst case complexity is unavoidable it would still be beneficial with low worst case complexity algorithms for effect solving.

While we cannot currently express principal types for the type system due to the effects inside ok types, the type solving rules actually produce "principal types up to effects". By finding a notion of principal effect the inference algorithm could perhaps be modified to produce a principal substitution.

One natural next step is to extend the type system and the inference algorithm to support robust safety. Here, safety is guaranteed in the presence of any adversary process running in parallel. Existing type systems for robust safety use an untrusted type Un to type adversaries and public data. An additional set of typing rules is also introduced allowing for different typings of messages and processes. The type inference algorithm would have to be extended to cope with these additions. One options might be to have "may" and "must" constraints along with a notion of types in an uncertain states; encountering a must constraint collapsed any uncertain state and makes a type concrete.

The type system could also be extended to support cryptographic operations such as symmetric and asymmetric. Some existing system model the latter using a subtyping relation on types. Another approach is to encode asymmetric encryption in a polarised version of the pi calculus. In either case, the type inference algorithm would have to be extended.

On a more fundamental note, our model is based on the Woo-Lam notion of correspondences for security. However, many definitions of security exist and it would be interesting to see if some of these other forms of security can be expressed and proven using type systems. Considering the gap between the symbolic and the computational view of cryptography [5] would also be interesting.

Ending our ideas for future work we have the most important one: to show that the type solving rules terminate. For Conjecture 42 we argued that associating a pair of natural numbers (n_v, n_s) does not seem to be strong enough (n_v is the number of distinct type variables occurring in constraint set C and n_s is the number of type constructors in types in C). One idea is to partition the variables by some notion of "height potential". More concretely, suppose we had a height potential p_X in the form for a number for each variable X in C. There must be a maximal height potential m among these and we can form a tuple

$$\left(\#_m, \#_{m-1}, \ldots \#_0, n_s\right)$$

where $\#_i$ is the number of variables with height potential i. If the height potential furthermore respects that the variables in an opening should be lesser height potential-wise than the variable being substitution, we get that the tuple decreases lexicographically by an opening (and by any of the other rules). We must be able to compute the p_X values in a deterministic manner so that e.g. variables unmodified by a transition obtain the same height potential. Section A.3.1 briefly returns to this.

Appendix A

Proofs

A.1 Message Constraints

In the proofs we shall sometimes denote a type by U in order to keep confusion to a minimal.

Lemma 63 (**Soundness of constraints for messages**)**.** Suppose we have constraint derivation tree $\Gamma \vdash M \rightsquigarrow T, C, B, V$. If a ground substitution σ satisfies C and stays within B then there exists a type derivation tree with root $\sigma\Gamma \vdash M : \sigma T$. Furthermore, the type derivation tree can be efficiently constructed.

Proof. The proof is by induction in the derivation of $\Gamma \vdash M \rightsquigarrow T, C, B, V$.

case MC-NAME: $M = n \quad C = \emptyset \quad B = B_1$
By premise $n : T \in \Gamma$ we have $\Gamma = \Gamma', n : T, \Gamma''$ and $\sigma\Gamma = \sigma\Gamma', n : \sigma T, \sigma\Gamma''$ and hence $n : \sigma T \in \Gamma$. Since σ stays within B we have by Lemma 29 that $\sigma\Gamma \vdash \diamond$. We then get $\sigma\Gamma \vdash n : \sigma T$ by rule MT-NAME as required.

case MC-OK: $M = \mathsf{ok} \quad C = \{X \doteq \mathsf{Ok}(E), E \sqsubseteq \mathit{effects}(\Gamma)\} \quad B = B_1$
Since σ satisfies C we have $\sigma X \equiv \mathsf{Ok}(\sigma E)$ and $\sigma E \subseteq \mathit{effects}(\sigma\Gamma)$. Also, since σ stays within B Lemma 29 tells us that $\sigma\Gamma \vdash \diamond$. Combined we can type ok as $\sigma\Gamma \vdash \mathsf{ok} : \sigma X$ by rule MT-OK as required.

case MC-FST: $M = \mathsf{fst}\ M_1 \quad C = \{T_1 \doteq \mathsf{Pair}(X_1, X_2)\} \cup C_1 \quad B = B_1$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$. Since σ satisfies C_1 and stays within B_1 we have by the induction hypothesis that there exist a type derivation tree with root $\sigma\Gamma \vdash M_1 : \sigma T_1$. Since σ furthermore satisfies $T_1 \doteq \mathsf{Pair}(X_1, X_2)$ we get $\sigma T_1 \equiv \mathsf{Pair}(\sigma X_1, \sigma X_2)$. We have $\sigma\Gamma \vdash \mathsf{fst}\ M_1 : \sigma X_1$ by MT-FST as required.

case MC-SND: $M = \mathsf{snd}\ M_1 \quad C = \{T_1 \doteq \mathsf{Pair}(X_1, X_2),\ X_2' \doteq X_2\langle \mathsf{fst}\ M/1\rangle\} \cup C_1 \quad B = B_1$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$. Since σ satisfies C_1 and stays within B_1 we have by the induction hypothesis that there exist a type derivation tree with root $\sigma\Gamma \vdash M_1 : \sigma T_1$. Since σ furthermore satisfies $T_1 \doteq \mathsf{Pair}(X_1, X_2)$ we get $\sigma T_1 \equiv \mathsf{Pair}(\sigma X_1, \sigma X_2)$. As σ also satisfies $X_2' \doteq X_2\langle \mathsf{fst}\ M/1\rangle$ we have $\sigma X_2' \equiv (\sigma X_2)\langle \mathsf{fst}\ M/1\rangle$. We can then apply rule MT-SND to get $\sigma\Gamma \vdash \mathsf{snd}\ M_1 : \sigma X_2'$ as required.

case MC-PAIR: $M = (M_1, M_2)$ $C = \{X \doteq \mathsf{Pair}(T_1, X_2'),\ T_2 \doteq X_2'\langle M_1/1\rangle\} \cup C_1 \cup C_2$ $B = B_1 \cup B_2$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$ and $\Gamma \vdash M_2 \rightsquigarrow T_2, C_2, B_2, V_2$. Since σ satisfies C_1 and C_2 and stays within B_1 and B_2 we have by induction hypothesis that there exist type derivation trees with roots $\sigma\Gamma \vdash M_1 : \sigma T_1$ and $\sigma\Gamma \vdash M_2 : \sigma T_2$. Since σ satisfies $T_2 \doteq X_2'\langle M_1/1\rangle$ we get $\sigma T_2 \equiv (\sigma X_2')\langle M_1/1\rangle$. Furthermore, σ satisfies $X \doteq \mathsf{Pair}(T_1, X_2')$ so $\sigma X \equiv \mathsf{Pair}(\sigma T_1, \sigma X_2')$. We can then type (M_1, M_2) by rule TM-PAIR as required.

□

Lemma 64 (**Completeness of constraints for messages**)**.** Suppose we have constraint derivation tree $\Gamma \vdash M \rightsquigarrow T, C, B, V$. If there exist a type derivation tree with root $\sigma\Gamma \vdash M : U$ for some substitution σ with $dom(\sigma) \cap V = \emptyset$ and type U then there exist a substitution σ' satisfying C and staying within B and where $\sigma' T \equiv U$ and $\sigma'\backslash_V = \sigma$.

Proof. The proof is by induction in the derivation of $\Gamma \vdash M \rightsquigarrow T, C, B, V$.

case MC-NAME: $M = n$ $C = \emptyset$ $V = \emptyset$
Suppose $\Gamma \vdash \diamond \rightsquigarrow B_1$. Since $\sigma\Gamma \vdash \diamond$ we have that σ stays within B_1 by Lemma 29. Since $n : U \in \sigma\Gamma$ we get $\sigma T \equiv U$. By letting $\sigma' = \sigma$ we trivially have that σ' satisfies $C = \emptyset$, stays within B_1, and $\sigma' T \equiv U$ and $\sigma'\backslash_{V=\emptyset} = \sigma$ as required.

case MC-OK: $M = \mathsf{ok}$ $C = \{X \doteq \mathsf{Ok}(E), E \sqsubseteq \mathit{effects}(\Gamma)\}$ $V = \{X, E\}$
Suppose $\Gamma \vdash \diamond \rightsquigarrow B_1$. Only rule MT-OK can be used to type ok so $\sigma\Gamma \vdash \mathsf{ok} : U$ implies $\sigma\Gamma \vdash \diamond$ which in turn implies that σ stays within B_1 by Lemma 29. Also, $U \equiv \mathsf{Ok}(S)$ and $S \subseteq \mathit{effects}(\sigma\Gamma)$. By letting $\sigma' = [X \mapsto U, E \mapsto S] \circ \sigma$ we get that σ' satisfies C, stays within B, and $\sigma' X \equiv U$ and $\sigma'\backslash_V = \sigma$ as required.

case MC-FST: $M = \mathsf{fst}\ M_1$ $C = \{T_1 \doteq \mathsf{Pair}(X_1, X_2)\} \cup C_1$ $V = \{X_1, X_2\} \uplus V_1$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$. Since $\sigma\Gamma \vdash \mathsf{fst}\ M_1 : U$ can only be caused by rule TM-FST we must have $\sigma\Gamma \vdash M_1 : U_1$ where $U_1 \equiv \mathsf{Pair}(U', U'')$ and $U \equiv U'$ for some types U_1, U' and U''. By the induction hypothesis there exist σ_1' satisfying C_1 and staying within B_1 and where $\sigma_1' T_1 \equiv U_1$ and $\sigma_1'\backslash_{V_1} = \sigma$. Let $\sigma' = [X_1 \mapsto U', X_2 \mapsto U''] \circ \sigma_1'$. Then σ' satisfies $T_1 \doteq \mathsf{Pair}(X_1, X_2)$ and hence C. Also, since σ_1' stays within B_1 we have that σ' stays within B_1. Finally, $\sigma' X_1 \equiv U$ and $\sigma'\backslash_V = \sigma$ as required.

case MC-SND: $M = \mathsf{snd}\ M_1$ $C = \{T_1 \doteq \mathsf{Pair}(X_1, X_2),\ X_2' \doteq X_2\langle \mathsf{fst}\ M/1\rangle\} \cup C_1$ $V = \{X_1, X_2, X_2'\} \uplus V_1$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$. Since $\sigma\Gamma \vdash \mathsf{snd}\ M_1 : U$ can only be caused by rule TM-SND we must have $\sigma\Gamma \vdash M_1 : U_1$ where $U_1 \equiv \mathsf{Pair}(U', U'')$ and $U \equiv U''\langle \mathsf{fst}\ M_1/1\rangle$ for some types U_1, U' and U''. By the induction hypothesis there exist σ_1' satisfying C_1 and staying within B_1 and where $\sigma_1' T_1 \equiv U_1$ and $\sigma_1'\backslash_{V_1} = \sigma$. Let $\sigma' = [X_1 \mapsto U', X_2 \mapsto U'', X_2' \mapsto U] \circ \sigma_1'$. Then σ' satisfies $T_1 \doteq \mathsf{Pair}(X_1, X_2)$ and $X_2' \doteq X_2\langle \mathsf{fst}\ M/1\rangle$ and hence C. Also, since σ_1' stays within B_1 we have that σ' stays within B_1. Finally, $\sigma' X_2' \equiv U$ and $\sigma'\backslash_V = \sigma$ as required.

case MC-PAIR: $M = (M_1, M_2)$ $C = \{X \doteq \mathsf{Pair}(T_1, X_2'),\ T_2 \doteq X_2'\langle M_1/1\rangle\} \cup C_1 \cup C_2$ $B = B_1 \cup B_2$ $V = \{X, X_2'\} \uplus V_1 \uplus V_2$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$ and $\Gamma \vdash M_2 \rightsquigarrow T_2, C_2, B_2, V_2$. Only rule TM-PAIR can type (M_1, M_2) so $\sigma\Gamma \vdash M_1 : U_1$ and $\sigma\Gamma \vdash M_2 : U_2$ where $U \equiv \mathsf{Pair}(U_1, U_2)$ and $U_1 \equiv U'$ and $U_2 \equiv U''\langle M_1/1\rangle$. By the induction hypothesis there exist σ_1' satisfying C_1 and staying within B_1 with $\sigma_1' T_1 \equiv U_1$, and there exist σ_2' satisfying C_2 and staying within B_2 with $\sigma_2' T_2 \equiv U_2$. Let $\sigma' = [X \mapsto U, X_2' \mapsto U''] \circ \sigma_1'|_{V_1} \circ \sigma_2'|_{V_2} \circ \sigma$. This is well-defined since $dom(\sigma) \cap V_1 = \emptyset$ and $dom(\sigma) \cap V_2 = \emptyset$, and $\sigma_1' \backslash_{V_1} = \sigma$, $\sigma_2' \backslash_{V_2} = \sigma$, and $V_1 \cap V_2 = \emptyset$. Then σ' satisfies C and stays within B, and $\sigma' X \equiv U$ and $\sigma' \backslash_V = \sigma$ as required.

□

A.2 Process Constraints

For the process constraints proofs it is convenient to consider an annotated version of processes. In particular, we replace process construct $\mathsf{in}\ M_1\ n;\ P_2$ with $\mathsf{in}\ M_1\ n : T;\ P_2$ where T is the type of n, and $\mathsf{exercise}\ M_1; P_2$ with $\mathsf{exercise}\ M_1 : \dot{S}; P_2$ where $\dot{S}$ is the effect hiding inside the type of M_1. Since T and $\dot{S}$ can be chosen to be respectively a type variable and a effect variable, it is straightforward to switch from the unannotated to the annotated version by simply annotating the process with fresh variables. In comparison with the old formulation of constraint generation we now first annotate the process with fresh variables before doing constraint generation.

We redefine the application of a substitution σ to a process to match the annotated version:

$$\begin{aligned}
\sigma(\mathsf{new}\ n : T;\ P) &= \mathsf{new}\ n : \sigma T;\ \sigma P \\
\sigma(\mathsf{in}\ M\ n : T;\ P) &= \mathsf{in}\ M\ n : \sigma T;\ \sigma P \\
\sigma(!\mathsf{in}\ M\ n : T;\ P) &= !\mathsf{in}\ M\ n : \sigma T;\ \sigma P \\
\sigma(\mathsf{exercise}\ M : \dot{S}; P) &= \mathsf{exercise}\ M : \sigma\dot{S}; \sigma P
\end{aligned}$$

and redefine the process constraint generation rules to those in Table A.1.

Note that the annotated version lets us assume we know what to extend the typing context with in rules PC-IN, PC-REIN, PC-EX and PC-NEW. Had we extended the context with fresh variables during constraint generation these variables would be added to V. However, bound generation would obviously mention the variables, in turn causing a conflict in the proof of completeness between our induction assumption that $dom(\sigma) \cap V = \emptyset$ and the assumption that σ respects the bound set.

Lemma 65 (**Soundness of constraints for processes**)**.** Suppose we have constraint derivation tree $\Gamma \vdash P \rightsquigarrow C, B, V$. If a ground substitution σ satisfies C and stays within B then there exists a type derivation tree with root $\sigma\Gamma \vdash \sigma P$. Furthermore, the type derivation tree can be efficiently constructed.

Proof. The proof is by induction in the derivation of $\Gamma \vdash P \rightsquigarrow C, B, V$. We will only give a proof for cases PC-IN, PC-NEW, PC-EX, PC-END, and PC-NIL since the rest are either similar or trivial.

$$\frac{\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1 \qquad \Gamma, n : T \vdash P_2 \rightsquigarrow C_2, B_2, V_2 \qquad C = \{T_1 \doteq \mathsf{Ch}(T)\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = V_1 \uplus V_2}{\Gamma \vdash \mathsf{in}\ M_1\ n : T;\ P_2 \rightsquigarrow C, B, V} \quad \text{PC-IN}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1 \qquad \Gamma, n : T \vdash P_2 \rightsquigarrow C_2, B_2, V_2 \qquad C = \{T_1 \doteq \mathsf{Ch}(T)\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = V_1 \uplus V_2}{\Gamma \vdash !\mathsf{in}\ M_1\ n : T;\ P_2 \rightsquigarrow C, B, V} \quad \text{PC-REIN}$$

$$\frac{\Gamma, n : T \vdash P_1 \rightsquigarrow C_1, B_1, V_1 \qquad C = \{T\ \mathsf{generative}\} \cup C_1}{\Gamma \vdash \mathsf{new}\ n : T;\ P \rightsquigarrow C, B_1, V_1} \quad \text{PC-NEW}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1 \qquad \Gamma, \dot{S} \vdash P_2 \rightsquigarrow C_2, B_2, V_2 \qquad C = \{T_1 \doteq \mathsf{Ok}(\dot{S})\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2 \qquad V = V_1 \uplus V_2}{\Gamma \vdash \mathsf{exercise}\ M_1 : \dot{S}; P_2 \rightsquigarrow C, B, V} \quad \text{PC-EX}$$

Table A.1: Annotated process constraint generation rules

case PC-IN: $P = \mathsf{in}\ M_1\ n : T;\ P_2 \qquad C = \{T_1 \doteq \mathsf{Ch}(T)\} \cup C_1 \cup C_2 \qquad B = B_1 \cup B_2$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$ and $\Gamma, n : T \vdash P_2 \rightsquigarrow C_2, B_2, V_2$. By Lemma 35 we have type derivation tree $\sigma\Gamma \vdash M_1 : \sigma T_1$ and by the induction hypothesis we have type derivation tree $\sigma\Gamma, n : \sigma T \vdash \sigma P_2$. Since σ satisfies $T_1 \doteq \mathsf{Ch}(T)$ we have $\sigma T_1 \equiv \mathsf{Ch}(\sigma T)$ and can apply rule PT-IN.

case PC-NEW: $P = \mathsf{new}\ n : T;\ P_1 \qquad C = \{T\ \mathsf{generative}\} \cup C_1$
Suppose $\Gamma, n : T \vdash P_1 \rightsquigarrow C_1, B_1, V_1$. By the induction hypothesis we have type derivation tree $\sigma\Gamma, n : \sigma T \vdash \sigma P_1$. Since σ satisfies $T\ \mathsf{generative}$ we have that σT is generative and can apply rule PT-NEW.

case PC-EX: $P = \mathsf{exercise}\ M_1 : \dot{S}; P_2 \qquad C = \{T_1 \doteq \mathsf{Ok}(\dot{S})\} \cup C_1 \cup C_2$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$ and $\Gamma, \dot{S} \vdash P_2 \rightsquigarrow C_2, B_2, V_2$. By Lemma 35 we have type derivation tree $\sigma\Gamma \vdash M_1 : \sigma T_1$ and by the induction hypothesis we have type derivation tree $\sigma\Gamma, \sigma\dot{S} \vdash \sigma P_2$. Since σ satisfies $T_1 \doteq \mathsf{Ok}(\dot{S})$ we have $\sigma T_1 \equiv \mathsf{Ok}(\sigma\dot{S})$ and can apply rule PT-EX.

case PC-END: $P = \mathsf{end}\ l(M_1) \qquad C = \{l(M) \in \mathit{effects}(\Gamma)\} \cup C_1$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$. By Lemma 35 we have type derivation tree $\sigma\Gamma \vdash M_1 : \sigma T_1$. Furthermore, since σ satisfies $l(M) \in \mathit{effects}(\Gamma)$ we have $l(M) \in \mathit{effects}(\sigma\Gamma)$. Applying PT-END we get the desired result.

case PC-NIL: $P = \mathsf{nil} \qquad C = \emptyset$
Suppose $\Gamma \vdash \diamond \rightsquigarrow B_1$. Since σ stays within B_1 we have by Lemma 29 that $\sigma\Gamma \vdash \diamond$ and can apply rule PT-NIL.

□

Lemma 66 (**Completeness of constraints for processes**)**.** Suppose we have constraint derivation tree $\Gamma \vdash P \rightsquigarrow C, B, V$. If there exists a type derivation tree with root $\sigma\Gamma \vdash \sigma P$ for some substitution σ with $dom(\sigma) \cap V = \emptyset$ then there exists a substitution σ' satisfying C and staying within B and with $\sigma'\backslash_V = \sigma$.

Proof. The proof is by induction in the derivation of $\Gamma \vdash P \rightsquigarrow C, B, V$. We will only give a proof for cases PC-IN, PC-NEW, PC-EX, PC-END, and PC-NIL since the rest are either similar or trivial.

case PC-IN: $P = \mathsf{in}\ M_1\ n : T;\ P_2 \quad C = \{T_1 \doteq \mathsf{Ch}(T)\} \cup C_1 \cup C_2 \quad B = B_1 \cup B_2 \quad V = V_1 \uplus V_2$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$ and $\Gamma, n : T \vdash P_2 \rightsquigarrow C_2, B_2, V_2$. Since $\sigma\Gamma \vdash \sigma P$ can only be typed by rule TP-IN we must have $\sigma\Gamma \vdash M_1 : U_1$ and $\sigma\Gamma, n : \sigma T \vdash \sigma P_2$ for some type U_1. Furthermore, $U_1 \equiv \mathsf{Ch}(\sigma T)$. By Lemma 36 there exist substitution σ_1' satisfying C_1 and staying within B_1 with $U_1 \equiv \sigma_1' T_1$, and by the induction hypothesis there exist σ_2' satisfying C_2 and staying within B_2. Let $\sigma' = \sigma_1'|_{V_1} \circ \sigma_2'|_{V_2} \circ \sigma$. This is well-defined since $dom(\sigma) \cap V_1 = dom(\sigma) \cap V_2 = \emptyset$, $V_1 \cap V_2 = \emptyset$, and $\sigma_1'\backslash_{V_1} = \sigma_2'\backslash_{V_2} = \sigma$. We have $\sigma' T_1 \equiv U_1 \equiv \mathsf{Ch}(\sigma T) \equiv \mathsf{Ch}(\sigma' T)$ so σ' satisfies C. Also, σ' stays within B and $\sigma'\backslash_V = \sigma$ as required.

case PC-NEW: $P = \mathsf{new}\ n : T;\ P_1 \quad C = \{T\ \mathsf{generative}\} \cup C_1 \quad B = B_1 \quad V = V_1$
Suppose $\Gamma, n : T \vdash P_1 \rightsquigarrow C_1, B_1, V_1$. Since $\sigma\Gamma \vdash \sigma P$ can only be typed by rule TP-NEW we must have $\sigma\Gamma, n : \sigma T \vdash \sigma P_1$ where σT is generative. By the induction hypothesis there exist σ_1' satisfying C_1 and staying within B_1 with $\sigma_1'\backslash_{V_1} = \sigma$. Let $\sigma' = \sigma_1'$. Then σ' satisfy C and stays within B and $\sigma'\backslash_V = \sigma$ as required.

case PC-EX: $P = \mathsf{exercise}\ M_1 : \dot{S}; P_2 \quad C = \{T_1 \doteq \mathsf{Ok}(\dot{S})\} \cup C_1 \cup C_2 \quad B = B_1 \cup B_2 \quad V = V_1 \uplus V_2$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T_1, C_1, B_1, V_1$ and $\Gamma, \dot{S} \vdash P_2 \rightsquigarrow C_2, B_2, V_2$. Since $\sigma\Gamma \vdash \sigma P$ can only be typed by rule TP-EX we must have $\sigma\Gamma \vdash M_1 : U_1$ for some type U_1 and $\sigma\Gamma, \sigma\dot{S} \vdash \sigma P_2$ where $U_1 \equiv \mathsf{Ok}(\sigma\dot{S})$. By Lemma 36 there exist substitution σ_1' satisfying C_1 and staying within B_1 with $\sigma_1' T_1 \equiv U_1$, and by the induction hypothesis there exist σ_2' satisfying C_2 and staying within B_2. Let $\sigma' = \sigma_1'|_{V_1} \circ \sigma_2'|_{V_2} \circ \sigma$. This is well-defined since $dom(\sigma) \cap V_1 = dom(\sigma) \cap V_2 = \emptyset$, $V_1 \cap V_2 = \emptyset$, and $\sigma_1'\backslash_{V_1} = \sigma_2'\backslash_{V_2} = \sigma$. We have $\sigma' T_1 \equiv U_1 \equiv \mathsf{Ok}(\sigma\dot{S}) \equiv \mathsf{Ok}(\sigma'\dot{S})$ so σ' satisfy C. Also, σ' stays within B and $\sigma'\backslash_V = \sigma$ as required.

case PC-END: $P = \mathsf{end}\ l(M_1) \quad C = \{l(M_1) \in \mathit{effects}(\Gamma)\} \cup C_1 \quad B = B_1 \quad V = V_1$
Suppose $\Gamma \vdash M_1 \rightsquigarrow T, C_1, B_1, V_1$. Since $\sigma\Gamma \vdash \sigma P$ can only be typed by rule TP-END we must have $\sigma\Gamma \vdash M_1 : U_1$ and $l(M_1) \in \mathit{effects}(\sigma\Gamma)$. By Lemma 36 there exist substitution σ_1' satisfying C_1 and staying within B_1 with $\sigma_1' T_1 \equiv U_1$ and $\sigma_1'\backslash_{V_1} = \sigma$. Let $\sigma' = \sigma_1'$. Then σ' satisfy C and stays within B and $\sigma'\backslash_V = \sigma$ as required.

case PC-NIL: $P = \mathsf{nil} \quad C = \emptyset \quad B = B_1 \quad V = \emptyset$
Suppose $\Gamma \vdash \diamond \rightsquigarrow B_1$. Since $\sigma\Gamma \vdash \sigma P$ can only be typed by rule TP-NIL we must have $\sigma\Gamma \vdash \diamond$. By Lemma 29 σ stays within $B_1 = B$. By letting $\sigma' = \sigma$ we trivially have that σ' satisfies $C = \emptyset$ and $\sigma'\backslash_{V=\emptyset} = \sigma$ as required.

□

A.3 Solving

Lemma 67. If $(C, \sigma)_B$ cannot be reduced then any constraint in C is either on the form $X \doteq X\mu$, $X\mu \doteq X'\mu'$, an effect constraint, or an obvious non-satisfiable type constraint.

Proof. By case analysis of the types T_1 and T_2 occurring in any type constraint $T_1 \doteq T_2$ in C.

- Suppose that neither T_1 nor T_2 is a variable or a variable under explicit substitution. If rule TS-TRIV, TS-CH, TS-PAIR, or TS-OK applies we reach a contradiction since $(C, \sigma)_B$ can be further reduced. Hence we can only have $T_1 \not\equiv T_2$ which is obvious non-satisfiable.
- Suppose $T_1 = X$. First consider the case where $T_2 = X'$. If $X = X'$ then rule TS-TRIV applies and we reach a contradiction. If $X \neq X'$ then rule TS-VAR1 applies and we reach a contradiction. Secondly, consider the case where $T_2 = X'\mu$. If $X = X'$ then we ignore the constraint. If $X \neq X'$ then rule TS-VAR1 applies and we reach a contradiction. Finally, consider the case where T_2 is neither a variable nor a variable under substitution. If $X \notin FV(T_2)$ then rule TS-VAR1 applies and we have a contradiction. If $X \in FV(T_2)$ we have an obvious non-satisfiable constraint.
- Suppose $T_1 = X\mu$. First consider the case where $T_2 = X'$. If $X = X'$ then we ignore the constraint. If $X \neq X'$ then rule TS-VAR1 applies and we reach a contradiction. Secondly, consider the case where $T_2 = X'\mu'$. In this case the constraint is ignored. Finally, consider the case where T_2 is neither a variable nor a variable under substitution. If $X \notin FV(T_2)$ then rule TS-VAR2 applies and we have a contradiction. If $X \in FV(T_2)$ we have an obvious non-satisfiable constraint.

□

Lemma 68. If substitution σ satisfies $[X \mapsto T]C$ then $\sigma \circ [X \mapsto T]$ satisfies C for any C. If ground substitution σ stays within $[X \mapsto T]B$ then $\sigma \circ [X \mapsto T]$ stays within B for any B.

Lemma 69. If $(C, \sigma)_B \xrightarrow{T} (C', \sigma')_{B'}$ then there exists a substitution σ'' such that $\sigma' = \sigma'' \circ \sigma$ and for all substitutions σ''' satisfying C' we have that $\sigma''' \circ \sigma''$ satisfies C. Furthermore, if σ''' is a ground substitution staying within B' then $\sigma''' \circ \sigma''$ is a ground substitution staying within B.

Proof. The proof is by induction in the length of the derivation of $\xrightarrow{T}$ using Lemma 68.

case TS-TRIV: Trivial since $\sigma'' = [\,]$ and any σ''' satisfies $T_1 \equiv T_2$.

case TS-CH: $\sigma'' = [\,]$ and if σ''' satisfy C' then $\sigma'''T_1 \equiv \sigma'''T_2$ implies that $\sigma'''\mathsf{Ch}(T_1) \equiv \sigma'''\mathsf{Ch}(T_2)$.

case TS-PAIR: Similar to TS-CH.

case TS-OK: Similar to TS-CH.

case TS-VAR1: Let $\sigma'' = [X \mapsto T]$. Since σ''' satisfies $C' = [X \mapsto T]C''$ we have by Lemma 68 that $\sigma''' \circ [X \mapsto T] = \sigma''' \circ \sigma''$ satisfies C''. We get that $\sigma''' \circ \sigma''$ satisfies $\{X \doteq T\} \cup C'' = C$. Furthermore, since σ''' stays within $B' = [X \mapsto T]B$ Lemma 68 shows that $\sigma''' \circ [X \mapsto T] = \sigma''' \circ \sigma''$ stays within B.

case TS-VAR2: Let $\sigma'' = [X \mapsto T']$ and the result follows by Lemma 68.

□

Lemma 70. If σ satisfies C and $\sigma X \equiv \sigma T$ then σ satisfies $[X \mapsto T]C$. If σ stays within B and $\sigma X \equiv \sigma T$ then σ stays within $[X \mapsto T]B$.

Proof. For the first part, assume σ satisfies C and $\sigma X \equiv \sigma T$. Then, for any type T'' we have $\sigma T'' \equiv \sigma([X \mapsto T]T'')$; in particular, since σ satisfies C it must also satisfy $[X \mapsto T]C$. For the second part, first note that $T_1 \equiv T_2$ implies $fn(T_1) = fn(T_2)$ and $fi(T_1) = fi(T_2)$. Assume σ stays within B and $\sigma X \equiv \sigma T$. Then, for any type T'' we have $fn(\sigma T'') = fn(\sigma([X \mapsto T]T''))$ and similar for fi; in particular, since σ stays within B it must also stay within $[X \mapsto T]B$. □

Lemma 71. Assume $(C, \sigma)_B \xrightarrow{T} (C', \sigma')_{B'}$. If δ satisfies C then there is δ' satisfying C'. Furthermore, if δ stays within B then δ' stays within B'.

Proof. We prove this by induction in the length of the derivation of $\xrightarrow{T}$.

case TS-TRIV: Suppose substitution δ satisfies $C = \{T_1 \doteq T_2\} \cup C'$ with $T_1 \equiv T_2$. Then it also trivially satisfies C'.

case TS-CH: Suppose substitution δ satisfies $C = \{\mathsf{Ch}(T_1) \doteq \mathsf{Ch}(T_2)\} \cup C''$. But this is the case if and only if δ satisfies $\{T_1 \doteq T_2\} \cup C'' = C'$.

case TS-PAIR: Similar to TS-CH.

case TS-OK: Similar to TS-CH.

case TS-VAR1: Suppose substitution δ satisfies $C = \{X \doteq T\} \cup C''$. Since δ satisfies $X \doteq T$ we have $\delta X \equiv \delta T$. By Lemma 70 δ satisfies $[X \mapsto T]C'' = C'$. Furthermore, if δ stays within B then by Lemma 70 it also stays within $[X \mapsto T]B = B'$.

case TS-VAR2: Suppose substitution δ satisfies $C = \{X\mu \doteq T\} \cup C''$. By Lemma 47 there exist substitution δ' satisfying $\{X \doteq T', X\mu \doteq T\} \cup C''$ for $T' = open(T)$. Since δ' satisfies $X \doteq T'$ we have $\delta' X \equiv \delta' T'$. By Lemma 70 δ' satisfies $[X \mapsto T'](\{X\mu \doteq T\} \cup C'') = C'$. Furthermore, if δ stays within B then δ' stays within B and by Lemma 70 it also stays within $[X \mapsto T']B = B'$.

□

A.3.1 Idea for proving termination of type solving rules

We have argued that associating a pair of natural numbers (n_v, n_s) to a constraint set C does not seem to be strong enough. Rather, we might partition the variables by some notion of "height potential". For instance, for

$$C = \left\{ \begin{array}{l} X\langle n/1\rangle \doteq \mathsf{Ch}(Y) \\ Y \doteq \mathsf{Pair}(Y_1, Y_2) \\ X \doteq Z \end{array} \right\}$$

we have that in any substitution σ satifying C the type σX consist of one more type constructor than σY. Similar, σY have one more type constructor than σY_1 and σY_2. So, if the opening of $\mathsf{Ch}(Y)$ is $\mathsf{Ch}(Y')$ then Y' have lesser "height potential" than X.

More concretely, we could define a *height* function h measuring the number of type constructors in a ground type:

$$\begin{array}{rcl} h(\mathsf{A}) & = & 0 \\ h(\mathsf{Ok}(\dot{S})) & = & 0 \\ h(\mathsf{Ch}(T)) & = & h(T) + 1 \\ h(\mathsf{Pair}(T_1, T_2)) & = & max\big(h(T_1), h(T_2)\big) + 1 \end{array}$$

For types with variables we could introduce a set of *potential* variables p_X and extend h with

$$\begin{array}{rcl} h(X) & = & p_X \\ h(X\mu) & = & p_X \end{array}$$

For $T_1 \equiv T_2$ we must have $h(T_1) = h(T_2)$, so for C from above we could generate a set of equations capturing the relative potential height between the types in the constraints:

$$\begin{array}{rcl} p_X & = & p_Y + 1 \\ p_Y & = & max(p_{Y_1}, p_{Y_2}) + 1 \\ p_X & = & p_Z \end{array}$$

Furthermore, if we can find an assignment to the potential variables there must be a variable with a maximal value. Let m be the maximal value and let $\#_v$ be the number of type variables X with $p_X = v$. We can form a tuple

$$\big(\#_m, \#_{m-1}, \ldots \#_0, n_s\big)$$

which decreases by an opening. For instance, assume $p_{Y_1} = p_{Y_2} = 0, p_Y = 1, p_X = p_Z = 2$. Then we have tuple

$$\big(2, 1, 2, 2\big)$$

for C. If X is substituted with $\mathsf{Ch}(Y')$ we get

$$C' = \left\{ \begin{array}{l} \mathsf{Ch}(Y'\langle n/1\rangle) \doteq \mathsf{Ch}(Y) \\ Y \doteq \mathsf{Pair}(Y_1, Y_2) \\ \mathsf{Ch}(Y') \doteq Z \end{array} \right\}$$

which have the lexicographical lesser tuple

$$\big(1, 2, 2, 4\big)$$

At this point we do not know if termination can be proved based on this idea, nor do we have no algorithm for computing an assignment to the pontential variables. In some cases an assignment does not exists: $X\mu \doteq \mathsf{Ch}(X)$ is an example since no assignment to p_X equates $p_X = p_X + 1$, so we have to show that the algorithm also terminates if no assignment can be found; note that the algorithm would fail because of occur-checks if set to solve $\{X\mu \doteq \mathsf{Ch}(X)\}$.

One possibility for computing the p_X assignment could be to simply remove any explicit substitutions in the constraint set and use the unification algorithm to create a substitution σ. Since we have removed all explicit substitutions no opening will occur during this unification. By assigning 0 to the potential variables of the type variables not assigned to by σ we can use the equations generated by h to get values for all potential variables. A minimal substitution property of unification general could perhaps be used to guarantee uniqueness of the assignment as well as preservation of unmodified constraints. However, it has to be shown that the type solving rules terminate even if unification for potentials fail. We do not know if this leads to a solution.

Appendix B

Basic Type System Concepts

B.1 Simple Type System: Arithmetic Expressions

To get a better understanding of type systems let us start with a very simple one, namely a small type systems for a small language with integers, strings and addition. Besides presenting the idea of a type system it will also allow us to encounter reductions, safety, and soundness.

The syntax of the language is

$$e \;:=\; n \mid s \mid e_1 \circ e_2$$

where n and s are two disjoint sets of *constants* denoting integers and strings respectively, and $e_1 \circ e_2$ an expression denoting an addition. For instance, constants $\mathsf{5}$ and $\mathsf{10}$ are members of n and corresponds to integers 5 and 10 respectively. In s we have constants such as $\mathsf{kibblesworth}$ and botley. Expression $\mathsf{5} \circ \mathsf{10}$ is intuitively meant to evaluate to $\mathsf{15}$ i.e. the integer corresponding to $5 + 10 = 15$.

We want to limit the set of expressions so that only integers are ever added together. For this we introduce a type system with types Int and Str and rules

$$\frac{}{n \;:\; \mathsf{Int}}\ \text{T-INT} \qquad \frac{}{s \;:\; \mathsf{Str}}\ \text{T-STR} \qquad \frac{e_1 \;:\; \mathsf{Int} \qquad e_2 \;:\; \mathsf{Int}}{e_1 \circ e_2 \;:\; \mathsf{Int}}\ \text{T-ADD}$$

As always, the conditions on top of the line are the premises, which, when satisfied, allows us the draw the conclusion below the line. In this particular case, we see that rule T-INT permits us to give type Int to any integer constant n. Similarly, rule T-STR give type Str to any string constant s. The rule T-ADD specifies that an addition expression $e_1 \circ e_2$ is well-typed if each sub-expression e_1 and e_2 is an integer. Also, the result of an addition is an integer.

If we introduce an evaluation strategy we can prove the type system sound relative to this. We say an expression is a *value* if and only if it is a constant, i.e. a number or a string. If an expression is not a value it must be an addition expression $e_1 \circ e_2$. In this case the strategy is to evaluate the two components e_1 and e_2 and add together the result. If we at any point try to add anything but an integer to an integer we say that the evaluation

has reached an *error state* and is not *well-behaved*. With these definitions we want to prove that the evaluation of well-typed expressions never reaches an error state, i.e. that only integers are ever added together. That the set of well-typed expressions is a subset of the set of well-behaved expressions is called the *soundness* of the type system.

For intuition let an evaluation strategy for expression e be defined by the function

$$eval(e) = \begin{cases} v & \text{if } e \text{ is the member of } n \text{ corresponding to integer } v \\ v_1 + v_2 & \text{if } e \text{ is } e_1 \circ e_2 \text{ with } v_1 = eval(e_1) \text{ and } v_2 = eval(e_2) \end{cases}$$

An expression e can then be said to be safe if $eval(e)$ is defined; formally, predicate $safe(e)$ is true if $eval(e)$ is defined, giving a charectarisation of the set of well-behaved expressions as $\{e \mid safe(e)\}$.

In the systems presented in this paper the evaluation strategy is formulated in terms of a *reduction relation*. For this reason we reformulate the strategy *eval* for arithmetic expression as a reduction. Let the relation $e \to e'$ be defined by rules

$$\frac{e_1 \to e_1'}{e_1 \circ e_2 \to e_1' \circ e_2}\ \text{E-LEFT} \qquad \frac{e_2 \to e_2'}{e_1 \circ e_2 \to e_1 \circ e_2'}\ \text{E-RIGHT} \qquad \frac{n = n_1 + n_2}{n_1 \circ n_2 \to n}\ \text{E-ADD}$$

where $n = n_1 + n_2$ in rule E-ADD should be read "n corresponds to integer v where $v = v_1 + v_2$ for n_1 and n_2 corresponding to integers v_1 and v_2 respectively". As an example we have

$$(1 \circ 2) \circ (3 \circ 4) \xrightarrow{\text{E-RIGHT}} (1 \circ 2) \circ 7 \xrightarrow{\text{E-LEFT}} 3 \circ 7 \xrightarrow{\text{E-ADD}} 10$$

since $3 \circ 4 \xrightarrow{\text{E-ADD}} 7$ and $1 \circ 2 \xrightarrow{\text{E-ADD}} 3$. Note that we do normally not label the arrows with the inference rule used.

From the rules we see that values cannot reduce, i.e. for any value n there does not exists e' such that $n \to e'$. Also, bad expressions such as $\mathsf{botley} \circ 10$ cannot be reduced even though they are not values; this is an example of an error state. The safety condition is then that the reduction of an expression e never gets stuck by entering an error state, i.e. an expression is *safe* if it is a value or reduces to a value. The new formulation can be shown to match *eval* in the sense that $eval(e) = v$ if and only if e reduces (perhaps under several reductions) to a value n corresponding to v.

First we prove **progress**: if $e : T$ then e is value or $e \to e'$ for some e'.
We prove this by induction in the derivation of $e : T$. For the base case we consider rule T-INT and T-STR. In both cases we get that e is a value. For the inductive step we consider rule T-ADD. For this rule to apply we must have $e = e_1 \circ e_2$ and $e_1 : T_1$ and $e_2 : T_2$ with $T_1 = T_2 = \mathsf{Int}$. The induction hypothesis then gives that e_1 is a value or can be reduced, in which case rule E-LEFT applies. Similar for e_2 and rule E-RIGHT. If both are values we use $T_1 = T_2 = \mathsf{Int}$ and note that the only values having type Int are integer constants. Hence we get $e_1 = n_1$ and $e_2 = n_2$ for some n_1 and n_2 in which case rule E-ADD applies. In all cases do we have that there exists an e' such that $e \to e'$.

Next we prove **preservation**: if $e : T$ and $e \to e'$ then $e' : T$.
Again we prove this by induction in the derivation of $e : T$. For the base case we consider rule T-INT and T-STR and immediately have that since e is a value there do not exists any e' such that $e \to e'$ and the condition is trivially satisfied. For the inductive step we consider rule T-ADD. For this rule to apply, we must have $e = e_1 \circ e_2$ with $e_1 : \mathsf{Int}$ and $e_2 : \mathsf{Int}$. Since $e \to e'$ we either have $e_1 \to e_1'$, $e_2 \to e_2'$, or e_1 and e_2 are both values. In the first two cases the indution hypothesis yields respectively $e_1' : \mathsf{Int}$ and $e_2' : \mathsf{Int}$ and hence respectively $e_1' \circ e_2 : \mathsf{Int}$ and $e_1 \circ e_2' : \mathsf{Int}$ as needed. In the last case the values are added and we obviously get $n : \mathsf{Int}$ for $n = n_1 + n_2$ as needed.

These results, combined with the fact that any expression e can only be reduced a finite number of times, implies that a well-typed expression is either a value or reduces to a value. Hence safety is guaranteed.

Note that this type system has no slack: the set of well-typed expressions is identical to the set of well-behaved (i.e. error-free) expressions. This is due to the extreme simplicity of this toy example and we will quickly move one to a system with slack: the simply typed lambda calculus.

B.2 Type Inference: The Simply Typed Lambda Calculus

Let us look at a more advanced and more useful type system. We illustrate the idea of type checking and type inference (or type reconstruction). For this we consider the simply typed lambda calculus which has the interesting property that well-typed terms terminate, i.e. terms such as

$$(\lambda x.xx)(\lambda x.xx)$$

where we have a non-terminating reduction

$$(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx)$$

are rejected by the type system. This section is inspired by [37] in which more details and proofs can be found.

Let the set of terms in the language be given by

$$t \ := x \mid \lambda x : T.t \mid t_1 \ t_2$$

where x is a variable, $\lambda x : T.t$ an abstraction (or function) with parameter x of type T and body t, and $t_1 \ t_2$ the application of t_1 on t_2. Assume the standard evaluation rules* and let the set of types be given by

$$T \ := \mathsf{O} \mid T_1 \to T_2$$

*See [37] for details.

where O is some atomic base type and $T_1 \to T_2$ is an arrow type describing functions from T_1 to T_2. Adapt type rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}\ \text{T-ABS}$$

$$\frac{\Gamma \vdash t_1 : T_2 \to T \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T}\ \text{T-APP}$$

As promised above this type system does indeed have slack. While it correctly rejects term $(\lambda x.x\ x)(\lambda x.x\ x)$, it also rejects term

$$(\lambda y.z)(\lambda x.x\ x)(\lambda x.x\ x)$$

despite the fact that it allows only a finite (one) number of reductions

$$(\lambda y.z)(\lambda x.x\ x)(\lambda x.x\ x) \to z$$

This is due to the compositional structure of the type system: a term is well-typed if and only if its immediate constituents are.

Having as safety predicate that terms terminate, i.e. give way to only a finite number of reduction, it is a well known result [37] of this type system that well-typed terms are safe. To see that the previous mentioned term $(\lambda x.x\ x)(\lambda x.x\ x)$ is not well-typed note that for a term to be typable its subterms must be typable. Assume there exists a types T_1 and T such that the first subterm can be typed as $\lambda x : T_1 . x\ x\ :\ T$. According to the type rules we must then have

$$\frac{\dfrac{\Gamma, x : T_1 \vdash x\ :\ T_3 \to T_2 \qquad \Gamma, x : T_1 \vdash x\ :\ T_3}{\Gamma, x : T_1 \vdash x\ x\ :\ T_2}}{\Gamma \vdash \lambda x : T_1 . x\ x\ :\ T = T_1 \to T_2}$$

for some type T_2 and T_3. For x to be well-typed we can only use rule T-VAR yielding $T_1 = T_3 \to T_2$ and $T_1 = T_3$. This implies $T_1 = T_1 \to T_2$ but no type satisfies the recursive occurence required in T_1.

Having seen that well-typed terms are always safe (as illustrated above and proved for the general case in [37]) it is natural to ask how we can check if a term is well-typed or not. More precisely, if we are given a proof in the form of a type derivation tree such as the one above, how can we make sure it is a valid proofs? This is called *type checking* and is the process of recursively going through the derivation tree making sure all steps are valid.

To do type checking we need to compare types. Perhaps surprisingly this can be undecidable, implying that type checking is not always possible by a machine. In the current system however, type checking simply amounts to matching of arrow types and identities. For instance, to validate proof

$$\frac{\dfrac{\Gamma, x : T \to T \vdash x\ :\ T \to T}{\Gamma \vdash (\lambda x : T \to T . x)\ :\ (T \to T) \to (T \to T)} \qquad \dfrac{\Gamma, y : T \vdash y\ :\ T}{\Gamma \vdash (\lambda y : T . y)\ :\ T \to T}}{\Gamma \vdash (\lambda x : T \to T . x)\ (\lambda y : T . y)\ :\ T \to T}$$

for some fixed type T we must compare T with T, $T \to T$ with $T \to T$ and so on. For this system, type comparison is on the border of being too trivial.

Having seen how explicit the proof for type checking has to be we quite naturally wonder if construction of the type derivation tree can be automated. Turning our attention to *type inference* we want to algorithmically map an untyped term into a type derivation tree thereby determining with what types the tree should be annoted. If no such types exists the algorithm should fail, i.e. succeed if and only if the type checker will.

The first step is to introduce type variables, i.e. place holders for the types the algorithm is to infer. Doing this we now get the set of types as

$$T \ := \mathsf{O} \mid T_1 \to T_2 \mid X$$

where X is a type variable. We keep the same terms and typing rules as before, and introduce a set of constraint generation rules. The intuition behind this is to "run the typing rules backwards" recording checks done during type checking as constraints instead of actually performing them. Checking is then done when all constraints are gathered using an unification algorithm *unify*. If the constraints can be satisfied, i.e. if there exists a assignment to the type variables leading to successful type checking, *unify* also yields such an assignment.

Let the constraint generation rules be

$$\frac{x : T \in \Gamma}{\Gamma \vdash x \ \rightsquigarrow \ T, \emptyset} \text{ C-VAR} \qquad \frac{\Gamma, x : T_1 \vdash t_2 : T_2, C}{\Gamma \vdash \lambda x : T_1 . t_2 \ \rightsquigarrow \ T_1 \to T_2, C} \text{ C-ABS}$$

$$\frac{\Gamma \vdash t_1 \ \rightsquigarrow \ T_1, C_1 \qquad \Gamma \vdash t_2 \ \rightsquigarrow \ T_2, C_2 \qquad C = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}}{\Gamma \vdash t_1 \ t_2 \ \rightsquigarrow \ X, C} \text{ C-APP}$$

so that by "running" the generation rules on a term we end up with a type X and a set of constraints C. It is not hard to show that the term can be made to type check if and only if the constraints are satisfied.

As an example let us run constraint generation on term $(\lambda x : T \to T.x)\ (\lambda y : T.y)$ from above. We get

$$\frac{\dfrac{\dfrac{\Gamma, x : X \vdash x \ \rightsquigarrow \ X, \emptyset}{\Gamma \vdash (\lambda x : X.x) \ \rightsquigarrow \ X \to X, \emptyset} \qquad \dfrac{\Gamma, y : Y \vdash y \ \rightsquigarrow \ Y, \emptyset}{\Gamma \vdash (\lambda y : Y.y) \ \rightsquigarrow \ Y \to Y, \emptyset}}{\Gamma \vdash (\lambda x : X.x)\ (\lambda y : Y.y) \ \rightsquigarrow \ Z, \{X \to X = (Y \to Y) \to Z\}}$$

and end up with one constraint, $X \to X = (Y \to Y) \to Z$. The unification algorithm solves this constraint by decomposing the arrow type thereby creating two new constraints equating $X = Y \to Y$ and $X = Z$. The latter constraint is trivially satisfied if Z is replaced by X, the former if X is furthermore replaced by $Y \to Y$. Creating a type derivation tree

isomorphic to the one generated doing constraint generation and substituting our newfound variable replacements we get

$$\dfrac{\dfrac{\Gamma, x : Y \to Y \vdash x \ : \ Y \to Y}{\Gamma \vdash (\lambda x : Y \to Y.x) \ : \ (Y \to Y) \to (Y \to Y)} \qquad \dfrac{\Gamma, y : Y \vdash y \ : \ Y}{\Gamma \vdash (\lambda y : Y.y) \ : \ Y \to Y}}{\Gamma \vdash (\lambda x : Y \to Y.x)\ (\lambda y : Y.y) \ : \ Y \to Y}$$

which is exactly the one from above if Y is replaced by the fixed T. Note that since Y is a type variable, what we have found here is actually a *type scheme* with the property than any typing of the term is an instantiation of the scheme, one possible instantiation being $Y = T$, another being $Y = \mathsf{O} \to \mathsf{O}$ and so one. In this light, we call Y the *principal type* of the term.

In full figure the unification algorithm solving constraints is defined by function *unify*:

```
let rec unify C =
  if C = ∅ then [ ]
  else
    let {S ≐ T} ∪ C' = C in
    if S = T then
      unify C'
    else if S = X and X ∉ FV(T)
      then unify([X ↦ T]C') ∘ [X ↦ T]
    else if T = X and X ∉ FV(S)
      then unify([X ↦ S]C') ∘ [X ↦ S]
    else if S = S1 → S2 and T = T1 → T2
      then unify(C' ∪ {S1 = T1, S2 = T2})
    else
      fail
```

In view of what is done in the rest of this paper we can also give an equivalent definition of the algorithm in terms of a reduction relation $\to$ between pairs (C, σ) as defined in Table B.1.

We compute $\mathit{unify}(C)$ by reducing $(C, [\,])$ as much as possible, i.e. $(C, [\,]) \to^\star (C', \sigma')$ and $(C', \sigma') \not\to$, where $[\,]$ is the empty substitution. The output of the algorithm is σ' if $C' = \emptyset$ and *fail* otherwise. It can be shown [37] that the algorithm is both sound and complete in respect to the constraints, which in turn are sound and complete in respect to the typing rules.

$$\frac{S = T}{\big(\{S \doteq T\} \cup C', \sigma\big) \to \big(C', \sigma\big)} \quad \text{U-EQ}$$

$$\frac{S = X \qquad X \notin FV(T)}{\big(\{S \doteq T\} \cup C', \sigma\big) \to \big([X \mapsto T]C', [X \mapsto T] \circ \sigma\big)} \quad \text{U-VAR1}$$

$$\frac{T = X \qquad X \notin FV(S)}{\big(\{S \doteq T\} \cup C', \sigma\big) \to \big([X \mapsto S]C', [X \mapsto S] \circ \sigma\big)} \quad \text{U-VAR2}$$

$$\frac{S = S_1 \to S_2 \qquad T = T_1 \to T_2}{\big(\{S \doteq T\} \cup C', \sigma\big) \to \big(C' \cup \{S_1 = T_1, S_2 = T_2\}, \sigma\big)} \quad \text{U-ARROW}$$

Table B.1: Unification rules

References

[1] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–115, New York, NY, USA, 2001. ACM.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[4] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. Technical report, Digital Systems Research Center, 1998. 149.

[5] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *IFIP International Conference on Theoretical Computer Science (IFIP TCS2000)*, pages 3–22, Sendai, Japan, 2000. Springer-Verlag, Berlin Germany.

[6] Lennart Augustsson. Cayenne - a language with dependent types. In S. Doaitse Swierstra, Pedro Rangel Henriques, and José N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 240–267. Springer, 1998.

[7] Bruno Blanchet. Automatic verification of correspondences for security protocols, 2008.

[8] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis for the pi-calculus with application to security. *Information and Computation*, 168(1):68–92, 2001.

[9] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems (TOCS)*, 8(1):18–36, 1990.

[10] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[11] Ivan Damgaard and Jesper Buus Nielsen. Commitment schemes and zero-knowledge protocols. Lecture notes.

[12] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[13] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 366–374, San Diego, California, June 1995. IEEE Computer Society Press.

[14] Bruno Blanchet et al. Proverif: Cryptographic protocol verifier in the formal model. http://www.proverif.ens.fr/.

[15] Cédric Fournet and Martín Abadi. Hiding names: Private authentication in the applied pi calculus. In Okada et al. [35], pages 317–338.

[16] Cédric Fournet and Andrew D. Gordon. A type discipline for authorization policies. Technical report, Microsoft Research, 2005. MSR-TR-2005-01.

[17] Andrew D. Gordon, Hans Hüttel, and René Rydhof Hansen. Type inference for correspondence types. Unpublished paper.

[18] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. Technical report, Microsoft Research, 2001. MSR-TR-2001-48.

[19] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *CSFW*, pages 77–91. IEEE Computer Society, 2002.

[20] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. Technical report, Microsoft Research, 2002. MSR-TR-2002-31.

[21] Andrew D. Gordon and Alan Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In Okada et al. [35], pages 263–282.

[22] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–519, 2003.

[23] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300:379–409, 2003.

[24] Andrew D. Gordon and Alan Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In Martín Abadi and Luca de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2005.

[25] Andrew D. Gordon, Alan Jeffrey, and Christian Haack. Cryptyc: Cryptographic protocol type checker. http://www.cryptyc.org/.

[26] Christian Haack and Alan Jeffrey. Pattern-matching spi-calculus. *Information and Computation*, 204(8):1195–1263, 2006.

[27] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, 2007.

[28] Daisuke Kikuchi and Naoki Kobayashi. Type-based verification of correspondence assertions for communication protocols. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2007.

[29] Cédric Lhoussaine. Type inference for a distributed π-calculus. *Science of Computer Programming*, 50(1-3):225–251, 2004.

[30] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[31] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM.

[32] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[33] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[34] Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A succinct solver for ALFP. *Nordic Journal of Computing*, 9(4):335–372, 2002.

[35] Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors. *Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium, ISSS 2002, Tokyo, Japan, November 8-10, 2002, Revised Papers*, volume 2609 of *Lecture Notes in Computer Science*. Springer, 2003.

[36] Sam Owre, John Rushby, and N. Shankar. Integration in PVS: tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems TACAS '97*, number 1217 in Lecture Notes in Computer Science, pages 366–383, Enschede, The Netherlands, apr 1997. Springer-Verlag.

[37] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[38] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[39] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[40] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *POPL*, pages 188–201, 1994.

[41] Thomas Y.C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.

[42] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.