A Type-Based Locality Analysis for a Functional Distributed Language

Álvaro Freitas Moreira

Doctor of Philosophy
University of Edinburgh
1999

Abstract

In this thesis we give a type-based analysis for an ML-like distributed language that detects references certain not to escape from one processor to another.

We assume a model of distribution based on *distributed shared memory*. From the programmer's viewpoint, the same reference on different machines refers to the same data object in a single *logical* store, but data is in fact distributed among the machines. A coherency protocol is then responsible for determining for each operation with references whether the associated data is available on the current machine, and if not, retrieving it over the network.

The costs of calling a coherency protocol for each store access can be reduced if a *locality analysis* can determine which references refer to local data only. Assignment and dereference operations using these *local references* can then be compiled to specialised versions, usually comprising a few machine instructions to manipulate data in the store.

The locality analysis we propose takes the form of a conservative extension of the Hindley-Milner polymorphic type discipline where reference types are tagged with locality information. We prove type soundness of the type system with respect to an operational semantics, and we also show that the type system soundly describes the locality of references in the sense that a local reference is certain not to escape according to the operational semantics.

This result means that a compiler can safely use the locality information provided by the types to replace assignment and dereferencing operations performed on local references with specialised variants, which are less costly than the originals. In order to illustrate how this can be done we define a target language and we give an operational semantics for it at a level of abstraction that differentiates between local and global versions of operations with the store. We define a translation from well typed source expressions to expressions in the target language induced by the locality information on types, and we prove that this translation preserves the original behaviour.

We then give a sound type reconstruction algorithm and we discuss a restricted form of *best locality* property that we conjecture the algorithm possesses. We also report on experiments showing that detecting local references has a significant impact on the performance of programs.

Acknowledgements

Thanks to Mike Fourman for supervising this thesis and for his words of encouragement along these years. Thanks to Paul Steckler for providing advice, encouragement and criticism. I had useful conversations with David Matthews in the very beginning of this work.

The financial support for this thesis came from a CAPES (Brazilian Postgraduate Agency) studenship.

The company of Graham Clark and Mark Hartswood, my office mates, has provided a welcome diversion from work.

I'm grateful to my long time Brazilian friends Cristian Koliver, Débora Abdala, Heitor Strogulsky, Rafael Bordini, Renata Vieira, Ricardo Diefenthaeler and Vanderlei Rodrigues. Their daily messages kept me going along these years. I'm specially grateful to Renata and Camila Vieira for their friendship, generosity and for being a significant part of all these years in Edinburgh. In Scotland, thanks to Elena Juarez, Ewen Denney, Barbara Gebhardt, Birgit Jenschen, Norma and Alastair Martin and Peter Lietz for their friendship.

Thanks to my father, my mother, sisters and brother (and his big family!) for their emotional support.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Álvaro Freitas Moreira*)

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Memory management for distributed systems has been an active area of research for several years. Two distinct models for building parallel machines and for parallel programming are known as *shared memory* systems and *distributed memory* systems.

The architecture for shared memory is more costly due to the complications of connecting more than one processor to a single shared memory. On the other hand it makes the programming task easier as there is a single address space.

Most distributed systems are however, based on architectures with *distributed memories* where each processor has its own store. This architecture is less expensive and more scalable but programming is made more difficult. If the same data is to be used by different machines the programmer should be aware of that and control the transmission of data from one machine to another.

Distributed Shared Memory (DSM) systems combine the advantages of shared memory multiprocessors and distributed memory systems. They provide the shared memory programming model in systems with physically distributed memories. Because of that, the concept of DSM is recognised as one of the most attractive approaches for building large-scale, high-performance multiprocessor systems. There is a vast literature about the subject, for a recent and comprehensive survey see [21].

But everything comes at a cost. Distributed copies of mutable objects must be kept consistent. This is performed by a *coherency protocol*. The coherency protocol is responsible for ensuring that each read operation on mutable shared data operates on the latest version of it.

There are several approaches to the design of coherency protocols ([8], [23]) all aiming at reducing the overhead caused by the traffic of data among processors.

This overhead can be further reduced if it can be determined which mutable data is *local* to a single processor. Operations to access or modify this *local mutable data* will usually comprise only a few machine instructions to manipulate data in the store, avoiding the need to call a protocol.

Parallel to the research effort on DSM and on coherency protocols there has been also an interest in *functional distributed languages* ([17],[1],[6]). These languages encourage a mostly functional programming style and also provide facilities for creating threads of execution on possibly different processors. Most data in a typical functional program is *immutable*. For this reason, using such languages potentially reduces the need for a coherency protocol. Recent research [15] shows that functional languages compare favourably to non-functional languages not only for implementation and maintenance but also in the execution performance of distributed applications.

In trying to reduce costs even more while maintaining data consistency one can think of using *static program analysis* to collect *locality information* about mutable data. Static analysis techniques [2] have been extensively used in several compilers to collect information about programs. This information is then used by the compiler to transform the original program into one with equivalent behaviour but with better performance and/or better resource utilisation.

In this thesis we propose a *type-based locality analysis* for an ML-like distributed language. Compared with other kinds of program analysis, type-based analyses have the advantage of being compositional, making the task of proving their correctness simpler. The locality analysis we propose takes the form of a conservative extension of the Hindley-Milner polymorphic type discipline where types of mutable data are tagged with locality information.

In order to evaluate our claims of improved performance we did tests with an experimental version of the Distributed Poly/ML [17]. Distributed Poly/ML is an extension of Poly/ML which supports the creation of process in remote machines. Poly/ML in its turn is an implementation of the Standard ML language [20].

In these tests it was observed that the local versions of dereference and assignment operations are both at least three times as fast as the default, non-local versions.

1.2 Related Work

Locality issues in Distributed Poly/ML are addressed in [25] where a global dataflow analysis to detect local *channels* is presented. The syntactic approach to dataflow analysis used there built upon earlier work for sequential languages, for example, [26] and [27]. Such global analyses, contrary to the type-based analysis designed here, are inherently non-compositional.

That work uses a language that is similar to ours and it does a reachability analysis that detects some cases of mutable objects that go to processors different where they were declared but are not used in these processors. Such cases are not detected by our type-based analysis.

In [24] and [32] type-based approaches using subtypes are used to detect global and local capabilities of channels in distributed calculi based on Milner's π-calculus.

All the analysis cited above were proved to be correct in the sense that they do not infer that an object is local when in fact it escapes. In oder to make a precise comparison between the amount of locality these analysis and ours detect it is necessary to have a common framework in which to represent them. The development of such a common framework is though, outside the scope of this thesis.

To the best of our knowledge the work in this thesis is the first to propose a type-based locality analysis integrated with the Hindley-Milner polymorphic type discipline.

1.3 Contents of the Thesis

An overview of the contents of the thesis chapter by chapter is:

Chapter 2. We describe a simplified version of a functional language with references and with primitives for creating distributed threads. We give an operational semantics for the language at a level of abstraction matching the programmer's view of the store. This view is the one provided by the mechanism of Distributed Shared Memory. We conclude this chapter with a discussion of the aspects of the locality of references that will be detected by our analysis.

Chapter 3. We begin by presenting the language of types annotated with locality information. We discuss how terms can be *locality polymorphic* and why references cannot be polymorphic in their locality. We then define *constraints* over locality labels of types and we explain how these constraints are used to guarantee that references tagged as local do not escape. The type system is finally presented and we give intuitions for the ways the rules assign locality labels to types. This chapter concludes by proving that our type system is a conservative extension of the Hindley-Milner type discipline.

Chapter 4. For clarity of presentation we split the soundness result of the type system with respect to the operational semantics in two parts. In this chapter we prove *type* soundness, not taking into account locality properties. We follow the syntactic approach of Wright and Felleisen [31] and Harper [12],[13] so we first prove type preservation, then we prove that expressions that may lead to execution errors are untypable, and finally we prove type soundness.

Chapter 5. In this chapter we prove that references that are considered local by our type system are certain not to escape according to the operational semantics. This property, which we call *Locality Soundness*, together with the type soundness proved in Chapter 4 completes the result that our type system is sound with respect to the operational semantics.

Chapter 6. We define a translation from well typed source expressions to expressions in a target language which distinguishes local and global dereference and assignment operators. We start by presenting the target language. We then give an operational semantics for it that differentiates between local and non-local operations on the store. We then prove that the translation preserves the behaviour of source programs.

Chapter 7. In this chapter we give a type reconstruction algorithm for our type system and we prove it sound with respect to the type system. This result combined with the result of locality soundness of the type system with respect to the operational semantics, means that the locality analysis performed by the algorithm is safe. We conclude this chapter with a discussion of the restricted form of *best locality* property that we conjecture holds for the algorithm.

Chapter 8. In this chapter we present our conclusions and some further directions for this work.

Appendices. We have two appendices. In Appendix A we have proofs for some auxiliary lemmas. In Appendix B we report on an experiment we did using an experimental version of DP/ML which provides the programmer with local and global operations with references. We compared the execution time of programs written using the local operations with the execution time of the same programs using the global operations.

Chapter 2

The Language and its Operational Semantics

We describe a simplified version of an implicitly typed polymorphic functional language with references and with primitives for creating distributed threads at explicitly identified processors.

In Section 2.2 we give an operational semantics for the language at a level of abstraction matching the programmer's view of the store. This view is provided by the mechanism of Distributed Shared Memory explained in Chapter 1. We conclude this chapter discussing the aspects of the locality behaviour of references that are captured by our analysis.

2.1 The Language

In this section we present the language which will be the object of our locality analysis. One of its key aspects is that it supports the creation of remote threads and that the processors where these remote threads will be executed should be explicitly specified. The language is implicitly typed and it supports polymorphic types. The terms of the language are given by the following grammar

$$
\begin{array}{rcl}
M & ::= & x \\
 & | & c \\
 & | & \texttt{fn}\, x.M \\
 & | & \texttt{rec}\, f\, x.M \\
 & | & M\,M' \\
 & | & \texttt{(}M, M'\texttt{)} \\
 & | & \texttt{if}\ M\ \texttt{then}\ M'\ \texttt{else}\ M'' \\
 & | & \texttt{let}\ x = M\ \texttt{in}\ M'\ \texttt{end}
\end{array}
$$

where c ranges over the constants given by the grammar

$$
\begin{array}{rclllll}
c & ::= & n & & & & \\
& | & r & & & & \\
& | & \texttt{unit} & & & & \\
& | & \texttt{zero?} & & & & \\
& | & \texttt{true} & | & \texttt{false} & & \\
& | & \texttt{fst} & | & \texttt{snd} & & \\
& | & \texttt{succ} & | & \texttt{pred} & & \\
& | & \texttt{fork} & | & \texttt{rfork}_\rho & & \\
& | & \texttt{ref} & | & \texttt{deref} & | & :=\,.
\end{array}
$$

Variables are binded by functions, let expressions and recursive functions. In a recursive function $\texttt{rec}\, f\, x.M$ both variables f and x are binded. We give an example of how $\texttt{rec}\, f\, x.M$ is used to define a recursive function in Section 2.2, after we give the rules of the operational semantics. Let-expressions do not allow recursive definitions: in an expression like $\texttt{let}\ x = M\ \texttt{in}\ N\ \texttt{end}$, the variable x cannot occur free in M. The language supports *value* let-polymorphism [30]. Variables binded to a value by a let expression can then have polymorphic types. The language also has pair and conditional expressions.

In the grammar for c,

- n ranges over integer constants,
- r ranges over an infinite set of reference constants, which denote store locations,
- `fork` is a *fork constant*, used to create new local threads,
- $\texttt{rfork}_\rho$ ranges over a finite set of constants, one for each processor ρ in a fixed network of processors. Each $\texttt{rfork}_\rho$ is used to create new threads on processor ρ,
- `ref` is a constant used to create references, and `deref` and $:=$ are the constants used to read and modify them, respectively.

The other constants of the grammar above have their usual meaning. The reason for having one constant $\texttt{rfork}_\rho$ for each processor ρ, instead of having a single constant receiving a processor identifier as one of its arguments, is to simplify the formal presentation. Otherwise we would have to assume the existence of a set of processor identifiers in the language and to introduce a type for them.

We refer to this language as the source language. In Chapter 6, we shall define a target language for a translation of the source language.

The set of free variables in a term M is given by $FV(M)$. The set of reference constants of a term M is given by $Refs(M)$. A term M is a *program* if $FV(M) = \emptyset$ and $Refs(M) = \emptyset$.

We call *syntactic values* or *values* for short, the irreducible terms of the language. The values of the language are given by the following grammar

$$\begin{array}{rcl} V & ::= & x \\ & | & c \\ & | & \texttt{fn}\,x.M \\ & | & \texttt{rec}\,f\,x.M \\ & | & \texttt{(}V, V'\texttt{)}. \end{array}$$

We use V as a metavariable ranging over values. We also write $M := N$ for the application of the assignment operator $:=$ to the pair $\texttt{(}M, N\texttt{)}$. We sometimes write n_{i+1} and n_{i-1} for the successor and predecessor, respectively, of a integer constant n_i. Application is associative to the left, and parentheses are used to change the associativity.

We follow the variable convention of Barendregt [4, section 2.1.13], that the bound variables are distinct from the free variables when discussing one or more terms in a given context.

One would naturally expect such a language to have channels for inter-thread synchronous communication. Channels are mutable objects and it has been shown in [5] that channels and threads can be used to define stores and to simulate the high level execution behaviour of operations with references. Hence, for conceptual simplicity, we could dispense with references in our study language and adopt channels instead.

However, the locality analysis we propose should work for any mutable object. In a distributed program channels can, of course, be used both for communication between threads running on the same processor or in different processors. But we believe that in a language with channels, references are more likely to be local to a single processor. The payoff of detecting local references is then potentially greater than of detecting local channels. We then deliberately omit channels from our treatment and adopt references as our mutable objects of study.

Because shared memory (in the form of references) will be used as a means of communication between concurrent threads, one would also expect the language to have constructs for thread synchronization. We claim that the introduction of primitives for synchronization would have no effect on the key aspects of our type-based analysis, because of that, in order to keep the formal treatment simpler, we omit them from the language.

2.2 Operational Semantics

We give a small step operational semantics for the language in a style similar to the semantics of concurrent ML given in [5], [22], and [25].

We use evaluation contexts in order to reduce the number of rules in the operational semantics and consequently reduce the number of cases in proofs relying on the semantics.

Evaluation contexts are terms with a "hole" in them, given by the grammar:

$$\begin{array}{rcl} E[\,] & ::= & [\,] \\ & | & E[\,]\ M \\ & | & V E[\,] \\ & | & (E[\,], M) \\ & | & (V, E[\,]) \\ & | & \mathtt{if}\ E[\,]\ \mathtt{then}\ M\ \mathtt{else}\ M' \\ & | & \mathtt{let}\ x = E[\,]\ \mathtt{in}\ M\ \mathtt{end}\ . \end{array}$$

The notation $E[M]$ means that the hole in the context $E[\,]$ is filled by the term M. The holes are used to indicate the syntactic location of the next reduction step [10]. By the grammar above, in applications, the operator is reduced before the operand; in pairs, the first element is reduced before the second. Note that context holes can occur neither in the bodies of functions nor in the bodies of `let` expressions.

As mentioned before the constants `fork` and $\mathtt{rfork}_\rho$ are used to create new threads of execution. At each step of evaluation a thread of execution is represented by what we call a *thread component*. A thread component is a pair (ρ, M) where ρ identifies the processor where the thread runs and M is the term associated to the thread at that point of execution.

A *thread component map*, or *thread map* for short, is a finite, non-empty map from *thread identifiers* to thread components. We assume that there are infinitely many thread identifiers available and we write p and q as typical thread identifiers. We write Π for a thread map. The thread map $\Pi[p : \rho, M]$ is the same as Π, except that the thread of execution identified by p is running on processor ρ and M is its associated term.

A *store* is a finite, possibly empty map from store locations to values. We write S as a metavariable for stores. The store $S[r \mapsto V]$ is the same as S, except that the store location r is mapped to the value V.

The operational semantics for the language is divided in two parts. The sequential part of the semantics is shown in Figure 2.1 and the concurrent part in Figure 2.2. In the sequential part, an evaluation step takes a store and a term

Succ:

$$S, E[\,\texttt{succ}\ n_i] \underset{\textsf{seq}}{\longrightarrow} S, E[n_{i+1}]$$

Pred:

$$S, E[\,\texttt{pred}\ n_i] \underset{\textsf{seq}}{\longrightarrow} S, E[n_{i-1}]$$

Zero-True:

$$S, E[\,\texttt{zero?}\ 0] \underset{\textsf{seq}}{\longrightarrow} S, E[\texttt{true}]$$

Zero-False:

$$S, E[\,\texttt{zero?}\ n] \underset{\textsf{seq}}{\longrightarrow} S, E[\texttt{false}] \quad n \neq 0$$

Fst:

$$S, E[\,\texttt{fst}\ \texttt{(}V, V'\texttt{)}] \underset{\textsf{seq}}{\longrightarrow} S, E[V]$$

Snd:

$$S, E[\,\texttt{snd}\ \texttt{(}V, V'\texttt{)}] \underset{\textsf{seq}}{\longrightarrow} S, E[V']$$

β:

$$S, E[(\texttt{fn}\, x.M)\ V] \underset{\textsf{seq}}{\longrightarrow} S, E[M[V/x]]$$

Rec-Unroll:

$$S, E[(\texttt{rec}\, f\, x.M)\ V] \underset{\textsf{seq}}{\longrightarrow} S, E[(\texttt{fn}\, x.M)[\texttt{rec}\, f\, x.M/f]\ V]$$

Cond-True:

$$S, E[\,\texttt{if true then}\ M\ \texttt{else}\ N] \underset{\textsf{seq}}{\longrightarrow} S, E[M]$$

Cond-False:

$$S, E[\,\texttt{if false then}\ M\ \texttt{else}\ N] \underset{\textsf{seq}}{\longrightarrow} S, E[N]$$

Ref:

$$S, E[\,\texttt{ref}\ \ V] \underset{\textsf{seq}}{\longrightarrow} S[r \mapsto V], E[r] \qquad r \notin Dom(S)$$

Deref:

$$S, E[\,\texttt{deref}\ \ r] \underset{\textsf{seq}}{\longrightarrow} S, E[V] \qquad S(r) = V$$

Assign:

$$S, E[r := V] \underset{\textsf{seq}}{\longrightarrow} S[r \mapsto V], E[\texttt{unit}] \qquad r \in Dom(S)$$

Let:

$$S, E[\,\texttt{let}\ x = V\ \texttt{in}\ M\ \texttt{end}\ ] \underset{\textsf{seq}}{\longrightarrow} S, E[M[V/x]]$$

Figure 2.1: Sequential evaluation.

Seq

$$\frac{S, E[M] \xrightarrow[\mathsf{seq}]{} S', E[M']}{S, \Pi[\,p : \rho, E[M]] \xrightarrow[\mathsf{con}]{} S', \Pi[\,p : \rho, E[M']]}$$

Fork

$$\frac{q \notin Dom(\Pi) \cup \{p\}}{S, \Pi[\,p : \rho, E[\texttt{fork fn}\,x.M]] \xrightarrow[\mathsf{con}]{} S, \Pi[\,p : \rho, E[\texttt{unit}]][\,q : \rho, M[\texttt{unit}/x]]}$$

Rfork

$$\frac{q \notin Dom(\Pi) \cup \{p\}}{S, \Pi[\,p : \rho, E[\texttt{rfork}_{\rho'}\ \texttt{fn}\,x.M]] \xrightarrow[\mathsf{con}]{} S, \Pi[\,p : \rho, E[\texttt{unit}]][\,q : \rho', M[\texttt{unit}/x]]}$$

Figure 2.2: Thread map evaluation.

and produces another, possibly different, store and a term. The relation $\xrightarrow[\mathsf{seq}]{}$ is thus a binary relation on store and term pairs and we write $\xrightarrow[\mathsf{seq}]{}^{*}$ for its reflexive and transitive closure.

We call a pair consisting of store and thread component map a *configuration.* In the concurrent part of the semantics, a configuration on the left-hand side produces a different configuration on the right-hand side. The relation $\xrightarrow[\mathsf{con}]{}$ is thus a binary relation on configurations and $\xrightarrow[\mathsf{con}]{}^{*}$ is its reflexive and transitive closure. We also say that $S, \Pi \xrightarrow[\mathsf{con}]{}^{n} S', \Pi'$ iff $S, \Pi \xrightarrow[\mathsf{con}]{}^{*} S', \Pi'$ in n steps.

A *redex* is either an application of the form $V\ V'$, a conditional expression $\texttt{if}\ V\ \texttt{then}\ N\ \texttt{else}\ M$ or a let expression $\texttt{let}\ x = V\ \texttt{in}\ M\ \texttt{end}$. The redexes filling the holes in the evaluation contexts appearing in the left side of $\xrightarrow[\mathsf{seq}]{}$ in Figure 2.1, and those filling the holes in evaluation contexts in the left side of $\xrightarrow[\mathsf{con}]{}$ in Figure 2.2, are called *proper redexes.* Any other redex is called *improper.*

According to the lemma below any non-value term can be uniquely represented as a redex inside an evaluation context.

Lemma 2.1 *Every non-value of the language can be uniquely decomposed as $E[M]$ for some evaluation context $E[]$ and redex M.*

Proof. By induction on the structure of non-value terms. ∎

One consequence of the previous lemma is that sequential evaluation given in Figure 2.1 is deterministic.

Note that the operational semantics is given at a level of abstraction that reflects the *programmer's view* of the store. The fact that store is physically distributed and that a coherency protocol is needed to keep distributed copies of mutable data consistent is therefore not expressed by the semantic rules.

We call $p \in Dom(\Pi)$ the *selected thread* in a transition $S, \Pi \underset{\mathsf{con}}{\longrightarrow} S', \Pi'$ if its associated term is evaluated by the transition. Because the rules also do not express which thread of a configuration will be *selected*, if the configuration has more than one, concurrent evaluation of threads is non-deterministic.

Below we give some examples to illustrate how the semantic rules can be used in the evaluation of expressions. For clarity, we underline the expression filling the hole in evaluation contexts instead of enclosing it with square brackets. The evaluation context itself surrounds the underlined expression. In the right side we write the names of the operational rules used in the derivation. For the concurrent transitions using a sequential transition as premise we indicate between parentheses the name of the sequential rule used.

Example 2.1 *The Minus function can be defined recursively in the following way:*

$$Minus \;\equiv\; \mathtt{rec}\, f\, x.M$$

where

$$M \;\equiv\; \mathtt{fn}\, y.\, \mathtt{if\ zero?}\ y\, \mathtt{then}\ x\, \mathtt{else}\ f\,(\mathtt{pred}\ x)(\mathtt{pred}\ y).$$

Here is how the rule for recursive functions can be used to prove that $2 - 1 = 1$.

$$
\begin{array}{lll}
 & \underline{Minus\ 2}\ 1 & \mathsf{Seq\ (Rec\text{-}Unroll)} \\
\underset{\mathsf{seq}}{\longrightarrow} & \mathtt{fn}\, x.M[Minus/f]\ 2\ 1 & \\
\equiv & \underline{\mathtt{fn}\, x.\mathtt{fn}\, y.\, \mathtt{if\ zero?}\ y\, \mathtt{then}\ x\, \mathtt{else}\ Minus\,(\mathtt{pred}\ x)(\mathtt{pred}\ y)\ 2}\ 1 & \mathsf{Seq}(\beta) \\
\underset{\mathsf{seq}}{\longrightarrow} & \underline{\mathtt{fn}\, y.\, \mathtt{if\ zero?}\ y\, \mathtt{then}\ 2\, \mathtt{else}\ Minus\,(\mathtt{pred}\ 2)(\mathtt{pred}\ y)\ 1} & \mathsf{Seq}(\beta) \\
\underset{\mathsf{seq}}{\longrightarrow} & \mathtt{if}\ \underline{\mathtt{zero?}\ 1}\, \mathtt{then}\ 2\, \mathtt{else}\ Minus\,(\mathtt{pred}\ 2)(\mathtt{pred}\ 1) & \mathsf{Seq\ (Zero\text{-}False)} \\
\underset{\mathsf{seq}}{\longrightarrow} & \underline{\mathtt{if\ \ false\ \ then}\ 2\, \mathtt{else}\ Minus\,(\mathtt{pred}\ 2)(\mathtt{pred}\ 1)} & \mathsf{Seq\ (Cond\text{-}False)} \\
\underset{\mathsf{seq}}{\longrightarrow} & Minus\,\underline{(\mathtt{pred}\ 2)}(\mathtt{pred}\ 1) & \mathsf{Seq\ (Pred)} \\
\underset{\mathsf{seq}}{\longrightarrow} & \underline{Minus\ 1}\ (\mathtt{pred}\ 1) & \mathsf{Seq\ (Rec\text{-}Unroll)} \\
\underset{\mathsf{seq}}{\longrightarrow} & \mathtt{fn}\, x.M[Minus/f]\ 1\ (\mathtt{pred}\ 1) & \\
\equiv & \underline{\mathtt{fn}\, x.\mathtt{fn}\, y.\, \mathtt{if\ zero?}\ y\, \mathtt{then}\ x\, \mathtt{else}\ Minus\,(\mathtt{pred}\ x)(\mathtt{pred}\ y)\ 1}\ (\mathtt{pred}\ 1) & \mathsf{Seq}(\beta) \\
\underset{\mathsf{seq}}{\longrightarrow} & \mathtt{fn}\, y.\, \mathtt{if\ zero?}\ y\, \mathtt{then}\ 1\, \mathtt{else}\ Minus\,(\mathtt{pred}\ 1)(\mathtt{pred}\ y)\ \underline{(\mathtt{pred}\ 1)} & \mathsf{Seq\ (Pred)} \\
\underset{\mathsf{seq}}{\longrightarrow} & \underline{\mathtt{fn}\, y.\, \mathtt{if\ zero?}\ y\, \mathtt{then}\ 1\, \mathtt{else}\ Minus\,(\mathtt{pred}\ 1)(\mathtt{pred}\ y)\ 0} & \mathsf{Seq}(\beta) \\
\underset{\mathsf{seq}}{\longrightarrow} & \mathtt{if}\ \underline{\mathtt{zero?}\ 0}\, \mathtt{then}\ 1\, \mathtt{else}\ Minus\,(\mathtt{pred}\ 1)(\mathtt{pred}\ 0) & \mathsf{Seq\ (Zero\text{-}True)} \\
\underset{\mathsf{seq}}{\longrightarrow} & \underline{\mathtt{if\ \ true\ \ then}\ 1\, \mathtt{else}\ Minus\,(\mathtt{pred}\ 1)(\mathtt{pred}\ 0)} & \mathsf{Seq\ (Cond\text{-}True)} \\
\underset{\mathsf{seq}}{\longrightarrow} & 1. &
\end{array}
$$

Example 2.2 *The following expression creates two references and it also creates a new thread of execution using* $\texttt{rfork}_\rho$*:*

$$
\begin{array}{l}
\texttt{let } x = \texttt{ref } 0 \texttt{ in} \\
\quad \texttt{let } y = \texttt{ref fn}\, w.x \texttt{ in } M \texttt{ end} \\
\texttt{end}
\end{array}
$$

where

$$M \equiv \; \texttt{rfork}_{\rho'} \; \texttt{fn}\, z.\,\texttt{snd (deref } y, \texttt{unit)}.$$

We now give a sequence of reduction steps for the expression above.

$$
\begin{array}{lll}
 & [p:\rho,\ \texttt{let } x = \underline{\texttt{ref } 0} \texttt{ in let } y = \texttt{ref fn}\, w.x \texttt{ in } M \texttt{ end end}] & \mathsf{Seq}\ (\mathsf{Ref}) \\
\xrightarrow[\mathsf{con}]{} & [r \mapsto 0], [p:\rho,\ \underline{\texttt{let } x = r \texttt{ in let } y = \texttt{ref fn}\, w.x \texttt{ in } M \texttt{ end end}}] & \mathsf{Seq}\ (\mathsf{Let}) \\
\xrightarrow[\mathsf{con}]{} & [r \mapsto 0], [p:\rho,\ \texttt{let } y = \underline{\texttt{ref fn}\, w.r} \texttt{ in } M \texttt{ end}] & \mathsf{Seq}\ (\mathsf{Ref}) \\
\xrightarrow[\mathsf{con}]{} & [r \mapsto 0, r' \mapsto \texttt{fn}\, w.r], [p:\rho,\ \underline{\texttt{let } y = r' \texttt{ in } M \texttt{ end}}] & \mathsf{Seq}\ (\mathsf{Let}) \\
\xrightarrow[\mathsf{con}]{} & [r \mapsto 0, r' \mapsto \texttt{fn}\, w.r], [p:\rho,\ \underline{\texttt{rfork}_{\rho'}\ \texttt{fn}\, z.\,\texttt{snd (deref } r', \texttt{unit)}}] & \mathsf{Rfork} \\
\xrightarrow[\mathsf{con}]{} & [r \mapsto 0, r' \mapsto \texttt{fn}\, w.r], [p:\ \rho,\ \texttt{unit}]\,[q:\ \rho',\ \texttt{snd (}\underline{\texttt{deref } r'}, \texttt{unit)}] & \mathsf{Seq}\ (\mathsf{Deref}) \\
\xrightarrow[\mathsf{con}]{} & [r \mapsto 0, r' \mapsto \texttt{fn}\, w.r], [p:\ \rho,\ \texttt{unit}]\,[q:\ \rho',\ \underline{\texttt{snd (fn}\, w.r, \texttt{unit)}}] & \mathsf{Seq}\ (\mathsf{Snd}) \\
\xrightarrow[\mathsf{con}]{} & [r \mapsto 0, r' \mapsto \texttt{fn}\, w.r], [p:\ \rho,\ \texttt{unit}]\,[q:\ \rho',\ \texttt{unit}]. &
\end{array}
$$

In the remainder of this section we give other definitions and a lemma related to the operational semantics. Some of these definitions are used only in Chapter 4 and Chapter 5 in order to state and prove type and locality soundness of programs in a more concise way.

We first define what we mean by a thread to be *faulty in a configuration.*

Definition 2.1 *We say that a thread p is* faulty in a configuration S, Π *if the term M associated to it in Π is not a value and there is no configuration S', Π' such that $S, \Pi \xrightarrow[\mathsf{con}]{} S', \Pi'$, with p the selected thread.*

According to the definition above a thread is faulty in a configuration S, Π if its associated term can be decomposed as $E[R]$, for an evaluation context $E[]$, and an improper redex R. The thread is also faulty if R is of the form $r := V$ or $\texttt{deref}\ r$ but the side condition of the corresponding sequential rule does not hold, that is, $r \notin Dom(S)$.

Definition 2.2 *A* trace T *is a possibly-infinite sequence of configurations*

$$S_0, \Pi_0;\ \ S_1, \Pi_1;\ \ \ldots$$

such that $S_0, \Pi_0 \xrightarrow[\mathsf{con}]{} S_1, \Pi_1 \xrightarrow[\mathsf{con}]{} \ldots.$

Definition 2.3 *The set of threads of a trace T is defined as*

$$Threads(T) = \{\, p \mid \exists\ S_i, \Pi_i \in T \ \textit{with}\ p \in Dom(\Pi_i)\,\}.$$

We define the notions of *convergence* and *divergence* of a thread of execution relative to a particular trace.

Definition 2.4 *A thread p* converges to a value V in T*, written $p \Downarrow_T V$, if there is S, Π in T such that $\Pi(p) = (\rho, V)$. We say that p* diverges in T*, written $p \Uparrow_T$, if for every $S, \Pi \in T$, the term associated to p in Π is not a value and p is not faulty in S, Π.*

A thread may diverge if it is not selected enough times to be evaluated even if its associated term is reducible to a value. It will also diverge of course if it enters in an infinite loop. Note that if T is $S, [p : \rho, \texttt{succ}\ n_i]$ for example, according to the definition above the thread p diverges in T: the term $\texttt{succ}\ n_i$ is not a value and p clearly is not faulty in the configuration. We will be interested only in notions of convergence and divergence in *maximal* traces, which we call *computations.*

Definition 2.5 *A* computation *is an infinite trace or it is a finite trace ending in a configuration S, Π such that for all $p \in Dom(\Pi)$, either p is faulty in S, Π, or there is V such that $\Pi(p) = (\rho, V)$. If P is a program then we define the computations of P to be*

$$Comp(P) = \{\, T \,/\, T \textit{ is a computation starting with configuration } \emptyset, [p_0 : (\rho_0, P)]\,\}$$

where p_0 and ρ_0 are, by convention, initial thread and processor identifiers respectively.

The following lemma will be used only in Chapter 4 where we prove type soundness.

Lemma 2.2 (Uniform Evaluation) *Let P be a program, let T be any trace in $Comp(P)$, and p be any thread in $Threads(T)$, then exactly one of the following holds*

- $p \Uparrow_T$,
- $p \Downarrow_T V$ *or*
- *p is faulty in some configuration of T.*

Proof. From the definitions 2.1 and 2.4. ∎

2.3 Escaping References

We say that a reference *escapes according to the operational semantics* or simply that a reference *escapes*, if it goes to a processor different than the one where it was created; and we say that a reference is *local according to the operational semantics* or *local* for short, if it does not escape.

The references r and r' in Example 2.2 of the previous section, for instance, are created on processor ρ. The reference r' escapes to processor ρ' in the body of the *rforked* function. When the new *rforked* thread is selected for execution the reference r' is dereferenced. The function $\texttt{fn}\, w.r$ then appears on processor ρ', causing the reference r that was created on processor ρ to escape from ρ to ρ'.

Note that in the following expression no reference escapes to processor ρ

$$\texttt{rfork}_\rho\ \texttt{fn}\, x.\,\texttt{fst (unit, ref unit)}.$$

Although a new thread will be executed on processor ρ, the new reference will not escape to ρ. It will be in fact *created* on that processor.

The goal of our locality analysis is to collect information about programs that can be used to reduce the number of times a coherency protocol has to be used.

Ideally the analysis should be able to identify all `deref` and assignment operations of a program that *do not* operate on references *used* on processors where they escape to when the program is executed. These references are those that do not escape and references that do escape but are not read or modified remotely.

The problem of identifying these references prior to the execution of the program is undecidable, consequently no static analysis can detect all of them.

The alternative we choose is to approximate in a conservative way the set of references that do not escape when executing the program. Our analysis then detects all the references that *can* escape when executing a program. The other references will constitute a safe approximation to the set of references that *do not escape*. Operations `deref` and := with these references can then be safely translated to their specialized versions.

Some opportunities for detecting local references and references that escape but are not used remotely will, of course, be missed. Consider for instance, the following expression running on processor $\rho \neq \rho'$

$$\begin{array}{l}\texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in} \\ \quad (\texttt{fork fn}\, w.\,\texttt{fst (unit, deref}\ x),\ (\texttt{fn}\, z.\,\texttt{rfork}_{\rho'}\ \texttt{fn}\, y.\, x := 3)\ M) \\ \texttt{end}.\end{array}$$

Suppose that M is an expression whose evaluation does not terminate. In this case the $\texttt{rfork}_{\rho'}$ will never be executed and the reference x will remain local to the processor where it was created. The `deref` operation could then be translated to its specialized version. However, as our locality analysis cannot decide about the termination or not of M, it takes a conservative approach and detects that the reference x *can escape*. As a consequence the `deref` will not be specialized.

The next example illustrates the case where a function *is rforked* but no reference is *used* remotely. Suppose the following expression runs on processor $\rho \neq \rho'$, and suppose that M is an expression whose evaluation does not terminate and that $y \notin FV(M)$.

$$\begin{array}{l} \texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in} \\ \qquad (\texttt{fork}\ \texttt{fn}\, w.\ \texttt{fst}\ (\texttt{unit}, \texttt{deref}\ x),\ \ \texttt{rfork}_{\rho'}\ \ \texttt{fn}\, y.(\texttt{fn}\, z.\, x := 3\ \ M)\,) \\ \texttt{end}\,. \end{array}$$

Clearly, the reference x escapes to processor ρ' in the body of the *rforked* function $\texttt{fn}\,y.(\texttt{fn}\,z.x := 3\ \ M)$. But the assignment to x will never be executed. The `deref` operation could then be translated to a version not requiring a coherency protocol. But our analysis detects that the reference x escapes and this will be enough for translating `deref` to its default non-specialized version.

As another example that shows how our analysis misses opportunities for optimization, suppose we start the execution of the expression below on processor ρ'

$$\begin{array}{l} \texttt{let}\ x = \texttt{ref}\ 0\,\texttt{in} \\ \qquad (\texttt{fork}\ \texttt{fn}\, w.\ \texttt{fst}\ (\texttt{unit}, \texttt{deref}\ x),\ \ \texttt{rfork}_{\rho'}\ \ \texttt{fn}\, y.x := 3) \\ \texttt{end}\,. \end{array}$$

Although the reference x in the expression above occurs in the body of an *rforked* function, it does not escape because the function is *rforked* to the same processor where the reference was created. As programs do not have any information of where they will initiate their execution, the reference x will be considered by our analysis as a reference that *can escape*, and the operation `deref` will not be specialized.

In the next chapter we present our type-based locality analysis in the form of a type system that detects references that can escape from the processors where they are created.

Chapter 3

The Type System

In this chapter we begin by presenting the language of types annotated with locality information. In Section 3.2 we discuss how terms can be *locality polymorphic* and why references cannot be polymorphic in their locality. In Section 3.3 we define *constraints* over locality labels of types and we explain how these constraints are used to guarantee that references tagged as local do not escape. The type rules are then finally presented in Section 3.4, and in Section 3.5 we give intuitions for the ways the rules assign locality labels to types. This chapter concludes by proving that our type system is a conservative extension of the Hindley-Milner type discipline. A preliminary version of the type system was presented in [3].

3.1 Labelled Types

The types for our language are similar to those of an ML-like language with references. In addition, some types include a *locality label.*

Definition 3.1 *A* locality label *is an element of the set defined by the grammar*

$$\phi ::= \ell \mid \epsilon \mid \kappa.$$

In the definition above κ ranges over an infinite set *LocVar* of *locality variables.* We call ℓ the *local label* and ϵ the *escape label.* The labelled types are given by the following grammar:

$$\tau \quad ::= \quad \texttt{bool} \mid \texttt{int} \mid \texttt{unit} \mid \alpha^{\phi} \mid \tau\,\texttt{ref}^{\,\phi} \mid \tau \xrightarrow{\phi} \tau' \mid \tau \times^{\phi} \tau'$$

where α is a metavariable ranging over an infinite set of *type variables* [1] distinct from the set of locality variables. The locality label of a type τ, if it has one, is given by $\mathbf{lab}\,(\tau)$.

[1]Following the 1997 Definition of Standard ML [20], we adopt "value polymorphism" [30]. and omit the imperative variables that appeared in the original Definition.

Remark 3.1 *Type variables are required to be* consistently labelled *with locality labels. That is, distinct occurrences of the same type variable have the same locality label.*

Our concern is with the locality properties of references, so clearly the locality labels belong on reference types. Intuitively, a reference that may escape from one processor to another will have a type with ϵ as its locality label; a reference that cannot escape can safely be assigned a type with locality label ℓ.

As an example, suppose that the expression below runs on processor ρ:

$$\begin{array}{l} \texttt{let}\ x = \texttt{ref unit} \\ \texttt{in}\ \ \texttt{rfork}_{\rho'}\ \texttt{fn}\, w.\ \texttt{deref}\ x \\ \texttt{end}\,. \end{array}$$

Using a primitive for remote thread creation ($\texttt{rfork}_{\rho'}$), a new thread is created on a processor ρ' to evaluate the body of the function $\texttt{fn}\, w.\ \texttt{deref}\ x$. The reference x, created on ρ, *escapes* to processor ρ', so its reference type should be tagged with ϵ.

In contrast, in the expression

$$\begin{array}{l} \texttt{let}\ x = \texttt{ref unit} \\ \texttt{in}\ \ (\texttt{fn}\, w.\ \texttt{deref}\ x)\ \texttt{unit} \\ \texttt{end} \end{array}$$

the reference x is a *local reference* as it does not escape to a processor different from the one where it was created. Its reference type can therefore be tagged with ℓ.

Function types, pair types and type variables also carry a locality label. A value that escapes from one processor to another may contain reference subterms, causing those references to escape with the value. In our language, the only values that may contain references, other than references themselves, are functions and pairs. So the function in the first example above will have a type tagged with ϵ and the function in the second example can have a type tagged with ℓ.

If we had a language with other kinds of value possibly containing references, such as lists or user-definable data types, then we would want to add locality labels to their types. Type variables carry a locality label because they may be instantiated to labelled types.

3.2 Locality Polymorphism

Some terms are *locality polymorphic*, because they can be used in term contexts that expect any locality. Particularly, functions and pairs of values may be polymorphic in their locality. Consider for example the expression

$$\begin{array}{l}\texttt{let}\ f = \texttt{fn}\,x.P\\ \texttt{in}\ \ (\texttt{fn}\,y.\ \texttt{rfork}_\rho\,f)\ \ (\texttt{if}\ \ldots\texttt{then}\ f\ \texttt{else}\ g)\\ \texttt{end}\,.\end{array}$$

We must consider f escaping, because it is an argument to an $\texttt{rfork}_\rho$ constant. However, there is no reason that f cannot also be used in a context that requires a local function. For example, if g is tagged as local in the example above, locality polymorphism allows us to consider f as local also. Similarly, a pair of values can also be made polymorphic in its locality.

Analogous to the well known fact that references cannot have polymorphic types [28], references in our type system cannot be locality polymorphic. Suppose a thread p runs on processor ρ and has the following expression associated to it

$$\begin{array}{l}\texttt{let}\ x = \texttt{ref}\ 3\\ \texttt{in}\ \ (\texttt{fn}\,y.\,\texttt{deref}\ x)\ (\texttt{rfork}_{\rho'}\ \ \texttt{fn}\,w.\,x := 5)\\ \texttt{end}\,.\end{array}$$

According to the operational semantics the program above is non-deterministic: the result returned by thread p can be either 5 or 3. If we admit references polymorphic in their locality, the body of the let expression above could be typed in an environment where x has the type scheme $\forall\kappa.\texttt{int ref}^{\,\kappa}$. Then, the first occurrence of x can have the type $\texttt{int ref}^{\,\ell}$ and the occurrence in the *rforked* function has type $\texttt{int ref}^{\,\epsilon}$. Based on this type and locality information a compiler could translate the program to:

$$\begin{array}{l}\texttt{let}\ x = \texttt{ref}\ 3\\ \texttt{in}\ \ (\texttt{fn}\,y.\,\texttt{deref}^{\,\ell}\,x)\ (\texttt{rfork}_{\rho'}\ \ \texttt{fn}\,w.\,x := 5)\\ \texttt{end}\end{array}$$

where $\texttt{deref}^{\,\ell}$ is the specialised local version of $\texttt{deref}$ (which does not rely on a coherency protocol). Although we have not yet given a semantics for $\texttt{deref}^{\,\ell}$ we can argue informally that the value 5 cannot be returned by thread p, so the program above no longer has a non-deterministic behaviour. Even if the assignment is executed before the dereferencing, updating the copy of x in processor ρ' with 5, the operation $\texttt{deref}^{\,\ell}$ on ρ cannot retrieve this most recent value as it does not call a coherency protocol. This claim will be made precise later on Chapter 6 where we define a translation to a target language which includes specialised local versions of `deref` and the assignment operator :=, and give an operational semantics for it.

3.3 Constrained Types

Consider the term `ref` (`ref` 3). If the outer reference escapes, it may be dereferenced, causing the inner reference to escape. Therefore, possible types for such a term include $\mathtt{int\ ref}^{\epsilon}\,\mathtt{ref}^{\epsilon}$, $\mathtt{int\ ref}^{\epsilon}\,\mathtt{ref}^{\ell}$ and $\mathtt{int\ ref}^{\ell}\,\mathtt{ref}^{\ell}$. Explicitly we want to exclude the type $\mathtt{int\ ref}^{\ell}\,\mathtt{ref}^{\epsilon}$. Such a type would give *unsound* locality information.

We also want to exclude the type $\mathtt{int\ ref}^{\kappa}\,\mathtt{ref}^{\epsilon}$. If the outer reference is labelled with ϵ, then we will expect that the type of the value this reference points to should be also labelled as escaping. We say that the type $\mathtt{int\ ref}^{\kappa}\,\mathtt{ref}^{\epsilon}$ provides *unresolved* information about locality.

The same ideas also apply to pairs. Suppose we have a pair like $(\mathtt{ref}\ 3, \mathtt{ref}\ 0)$. We have to exclude a type like $\mathtt{int\,ref}^{\ell} \times^{\epsilon} \mathtt{int\,ref}^{\ell}$, for example. Were we allowed to assign such a type, then the pair might escape, carrying with it references that are labelled with ℓ.

How then to give a general type for terms in such a way that undesired types are excluded? The solution we adopt is to give types with *constraints* on their locality labels. A *constrained type* $\tau \backslash C$ is a pair consisting of a type τ and a set C of *constraints* on locality labels possibly appearing in τ, each of the form $\phi \leq \phi'$. We sometimes use the metavariable γ to range over constrained types.

Definition 3.2 *We say that a constraint $\phi \leq \phi'$ is*

- unsound *iff $\phi = \epsilon$ and $\phi' = \ell$,*
- unresolved *iff $\phi = \epsilon$ and $\phi' \in LocVar$.*

Definition 3.3 *A constraint set C is* valid *iff no constraint in its transitive closure is unsound or unresolved. A constraint set is* invalid *iff it is not valid.*

We will see that rules of the type system form, at their conclusions, the union of the constraint sets of their premises The following lemma says that the union of valid constraint sets is also valid.

Lemma 3.1 *If C and C' are valid constraint sets, then $C \cup C'$ is also valid.*

Proof. By contradiction. We assume C and C' valid and we try to prove that $C \cup C'$ is invalid. According to Definition 3.3, for $C \cup C'$ to be invalid means that

1. $\epsilon \leq \ell \in (C \cup C')^+$ or

2. $\epsilon \leq \kappa \in (C \cup C')^+$ for some locality variable κ.

If (1) holds, then either $\epsilon \leq \ell \in (C \cup C')$ or there exists κ' such that $\epsilon \leq \kappa' \in (C \cup C')$ and $\kappa' \leq \ell \in (C \cup C')^+$. If $\epsilon \leq \ell \in (C \cup C')$, then this contradicts the hypothesis that C and C' are valid. On the other hand if $\epsilon \leq \kappa' \in (C \cup C')$, then this also contradicts the hypothesis that C and C' are valid.

Similarly for (2). ∎

Remark 3.2 *In order to avoid* unsound *and* unresolved *locality information we require that in a type derivation for a term, the type assigned to each of its subterms has an associated valid constraint set. It will be assumed that type derivations producing invalid constraints sets will be rejected.*

References are created with the `ref` constant. According to the typing rules presented in the next section, if we apply `ref` to a term of type $\texttt{int ref}^{\,\ell} \backslash \emptyset$ for example, the resulting constrained type will be of the form $\texttt{int ref}^{\,\ell}\, \texttt{ref}^{\,\phi} \backslash \phi \leq \ell$. By Remark 3.2 the label ϕ must be different than ϵ.

We shall show later on Chapter 5 that this local validity property is crucial in establishing the desired global property of the system, that references tagged as local indeed do not escape.

3.4 Type Judgements

ML type schemes have the form $\forall \vec{\alpha}.\, \tau$, where τ is an ML type and $\vec{\alpha}$ is a sequence of type variables. Our constrained types can have locality variables as well as type variables, hence a *constrained type scheme* has the form $\forall \vec{\alpha}\vec{\kappa}.\, \tau \backslash C$, where τ is a type, C is any constraint set not necessarily valid , and $\vec{\alpha}$ and $\vec{\kappa}$ are sequences of type and locality variables, respectively. The order of the binders in a constrained type scheme is not significant; we segregate type variables from locality variables for clarity. We use σ as a metavariable ranging over constrained type schemes. For a sequence of type variables $\vec{\alpha}$, let $\lceil \vec{\alpha} \rceil$ be the set consisting of the type variables in the sequence; similarly for locality variables.

Our type judgements use two notions of context:

- A *term type context* is a finite map from variables to constrained type schemes. The term type context $\Gamma[x \mapsto \sigma]$ is the same as Γ, but maps x to σ.
- A *store type context* is a finite map from store locations to constrained types of the form $\tau\,\mathtt{ref}^{\,\phi}\backslash C$. The store type context $\Delta[r \mapsto \tau\,\mathtt{ref}^{\,\phi}\backslash C]$ is the same as Δ, but maps r to $\tau\,\mathtt{ref}^{\,\phi}\backslash C$.

Because type variables do not appear in constraints, the set $FTV(\tau\backslash C)$ of free type variables of a constrained type $\tau\backslash C$ is given by

$$FTV(\tau\backslash C) = FTV(\tau).$$

The set $FTV(\tau)$ of free type variables of a type τ is given by

$$\begin{array}{l} FTV(\tau) \ = \{\} \quad \textit{if}\ \tau \in \{\mathtt{int}, \mathtt{bool}, \mathtt{unit}\} \\ FTV(\alpha^{\phi}) = \{\,\alpha\,\} \\ FTV(\tau_1 \times^{\phi} \tau_2) = FTV(\tau_1) \ \cup \ FTV(\tau_2) \\ FTV(\tau_1 \xrightarrow{\phi} \tau_2) = FTV(\tau_1) \ \cup \ FTV(\tau_2) \\ FTV(\tau\,\mathtt{ref}^{\,\phi}) = FTV(\tau). \end{array}$$

We define $FLV(\phi) = \{\phi\}$ if ϕ is a locality variable. $FLV(\phi) = \emptyset$ otherwise. The set $FLV(\tau\backslash C)$ of free locality variables of a constrained type $\tau\backslash C$ is defined in the following way

$$FLV(\tau\backslash C) = FLV(\tau) \ \cup \ FLV(C).$$

The set $FLV(\tau)$ of free locality variables of τ is given by

$$\begin{array}{l} FLV(\tau) = \{\} \quad \textit{if}\ \tau \in \{\mathtt{int}, \mathtt{bool}, \mathtt{unit}\} \\ FLV(\alpha^{\phi}) = FLV(\phi) \\ FLV(\tau_1 \times^{\phi} \tau_2) = FLV(\tau_1) \ \cup \ FLV(\tau_2) \ \cup \ FLV(\phi) \\ FTV(\tau_1 \xrightarrow{\phi} \tau_2) = FTV(\tau_1) \ \cup \ FLV(\tau_2) \ \cup \ FLV(\phi) \\ FLV(\tau\,\mathtt{ref}^{\,\phi}) = FTV(\tau) \ \cup \ FLV(\phi), \end{array}$$

and for a constraint set $C = \{\phi_1 \leq \phi'_1 \ldots, \phi_n \leq \phi'_n\}$, we have

$$FLV(C) = FLV(\phi_1) \cup FLV(\phi'_1) \ldots \cup FLV(\phi_n) \cup FLV(\phi'_n).$$

For a constrained type scheme $\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C$ define

$$\begin{array}{l} FTV(\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C) = FTV(\tau\backslash C) \setminus \lceil\vec{\alpha}\rceil, \text{ and} \\ FLV(\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C) \ = FLV(\tau\backslash C) \setminus \lceil\vec{\kappa}\rceil. \end{array}$$

For term type contexts and store type contexts, we define their sets of free type variables and free locality variables by pointwise union. We write $FLV(\Gamma,\Delta)$ for $FLV(\Gamma) \cup FLV(\Delta)$, and $FTV(\Gamma,\Delta)$ for $FTV(\Gamma) \cup FTV(\Delta)$.

Below, we define the notion of a *polymorphic-consistent* type scheme. The intuition behind this definition is that we only get locality polymorphism on type variables when we have type polymorphism.

Definition 3.4 *We say that $\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C$ is* polymorphic-consistent *iff for all $\alpha \in FTV(\tau)$ if $\mathbf{lab}\,(\alpha) \in \lceil\vec{\kappa}\rceil$ then $\alpha \in \lceil\vec{\alpha}\rceil$. We also say that a term type context Γ is polymorphic-consistent if all the constrained type schemes in its range are polymorphic-consistent.*

Similarly to Standard ML, we obtain polymorphism by generalising over the constrained types of **let**-bound variables. Our notion of generalisation includes generalisation over locality variables, as well as over type variables:

Definition 3.5 (Generalisation) $Gen(\tau\backslash C,\Gamma,\Delta) = \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C$ *where*

$$\lceil\vec{\kappa}\rceil = FLV(\tau) \setminus FLV(\Gamma,\Delta) \text{ and}$$
$$\lceil\vec{\alpha}\rceil = FTV(\tau) \setminus FTV(\Gamma,\Delta).$$

Generalising a constrained type under a polymorphic-consistent typing context produces a polymorphic-consistent constrained type scheme.

Lemma 3.2 *If Γ is polymorphic-consistent then $Gen(\tau\backslash C,\Gamma,\Delta)$ is polymorphic-consistent.*

Proof. Let $Gen(\tau\backslash C,\Gamma,\Delta) = \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C$ so that $\lceil\vec{\alpha}\rceil = FTV(\tau) \setminus FTV(\Gamma,\Delta)$ and $\lceil\vec{\kappa}\rceil = FLV(\tau) \setminus FLV(\Gamma,\Delta)$. Suppose that there are α and κ such that $\alpha \in FTV(\tau)$ and $\mathbf{lab}\,(\alpha)$ in τ is κ. Suppose also that $\kappa \in \lceil\vec{\kappa}\rceil$. We prove by contradiction that $\alpha \in \lceil\vec{\alpha}\rceil$. So we assume that $\alpha \notin \lceil\vec{\alpha}\rceil$. Because $\alpha \in FTV(\tau)$ but $\alpha \notin \lceil\vec{\alpha}\rceil$, it must be the case that α occurs free either in Γ or in Δ. By the consistent labelling assumption on the locality labels of type variables (Remark 3.1), all these occurrences of α in Γ and Δ must be labelled with κ. By the assumption that Γ is polymorphic-consistent, these occurrences of κ labelling α in Γ must also be free. But this contradicts the assumption that $\kappa \in \lceil\vec{\kappa}\rceil$. ∎

Since we have two kinds of variables appearing in types, we need to define substitutions involving these variables carefully. We split the definition into three parts: *locality substitution*, *type substitution* and what we call *extended type substitution* which combines both locality and type substitutions.

Definition 3.6 (Locality substitution) *A* locality substitution Z^{loc} *is a mapping from locality variables to locality labels. Extending the domain appropriately we define the application of* Z^{loc} *to locality labels, constraint sets, types and constrained types in the following way*

$$
\begin{aligned}
&Z^{loc}\,\phi = \phi \quad \textit{if} \;\; \phi \in \{\epsilon, \ell\}\\
&Z^{loc}\,C = \{Z^{loc}\phi \leq Z^{loc}\phi' \mid \phi \leq \phi' \in C\}\\
&Z^{loc}\,\tau = \tau \quad \textit{if}\; \tau \in \{\texttt{int}, \texttt{unit}, \texttt{bool}\}\\
&Z^{loc}\,(\alpha^{\phi}) = \alpha^{Z^{loc}\phi}\\
&Z^{loc}\,(\tau \xrightarrow{\phi} \tau') = (Z^{loc}\,\tau) \xrightarrow{Z^{loc}\phi} (Z^{loc}\,\tau')\\
&Z^{loc}\,(\tau \times^{\phi} \tau') = (Z^{loc}\,\tau) \times^{Z^{loc}\phi} (Z^{loc}\,\tau')\\
&Z^{loc}\,(\tau\,\texttt{ref}^{\,\phi}) = (Z^{loc}\,\tau)\ \texttt{ref}^{\,Z^{loc}\phi}\\
&Z^{loc}\,(\tau \backslash C) = Z^{loc}\,\tau \backslash Z^{loc}\,C.
\end{aligned}
$$

We assume that locality substitutions behave like the identity function when applied to locality variables outside their domains.

Definition 3.7 (Type substitution) *A* type substitution Z^{type} *is a mapping from type variables to types. Extending the domain of a type substitution appropriately we define the application of* Z^{type} *to types and constrained types in the following way*

$$
\begin{aligned}
&Z^{type}\,(\alpha^{\phi}) = \tau \;\textit{ if } Z^{type}(\alpha) = \tau \textit{ and either } \mathbf{lab}\,(\tau) = \phi \textit{ or } \mathbf{lab}\,(\tau) \textit{ is undefined}\\
&Z^{type}\,\tau = \tau \quad \textit{if}\; \tau \in \{\texttt{unit}, \texttt{int}, \texttt{bool}\}\\
&Z^{type}\,(\tau \xrightarrow{\phi} \tau') = (Z^{type}\,\tau) \xrightarrow{\phi} (Z^{type}\,\tau')\\
&Z^{type}\,(\tau \times^{\phi} \tau') = (Z^{type}\,\tau) \times^{\phi} (Z^{type}\,\tau')\\
&Z^{type}\,(\tau\,\texttt{ref}^{\,\phi}) = (Z^{type}\,\tau)\ \texttt{ref}^{\,\phi}\\
&Z^{type}\,(\tau \backslash C) = (Z^{type}\,\tau) \backslash C.
\end{aligned}
$$

In order to illustrate the previous definition suppose that Z^{type} is the map $\{\alpha \mapsto \texttt{int}\,\texttt{ref}^{\,\epsilon}, \beta \mapsto \texttt{int} \times^{\ell} \texttt{int}, \eta \mapsto \texttt{unit}\}$. Then the following holds:

$$
\begin{aligned}
&Z^{type}\,(\alpha^{\kappa}) \text{ is undefined}\\
&Z^{type}\,(\beta^{\ell}) = \texttt{int} \times^{\ell} \texttt{int}\\
&Z^{type}\,(\eta^{\epsilon}) = \texttt{unit}.
\end{aligned}
$$

In the first example above note that the type variable α is in the domain of Z^{type} but the result of applying Z^{type} to α^{ϕ} is undefined! We say that applications like that are not *well behaved*. We define a well behaved application of a type substitution in the following way:

Definition 3.8 *We say that $Z^{type}\,\alpha^{\phi}$ is* well behaved *iff $Z^{type}\,\alpha^{\phi}$ is undefined only when $\alpha \notin Dom(Z^{type})$. We say that $Z^{type}\,\tau$ is* well behaved *iff $Z^{type}\,\alpha^{\phi}$ is well behaved for all $\alpha \in FTV(\tau)$ with ϕ the label of α in τ.*

Next we define a substitution which is a combination of type and locality substitutions.

Definition 3.9 (Extended type substitution) *An* extended type substitution *Z is a pair (Z^{type}, Z^{loc}) of type and locality substitutions. We define the application of Z to constraint sets, types, and constrained types in the following way*

$$\begin{array}{l} Z\,C = Z^{loc}\,C \\ Z\,\tau = Z^{type}\,(Z^{loc}\,\tau) \\ Z\,(\tau \backslash C) = Z\,\tau \backslash Z\,C. \end{array}$$

Note that by the definition above, when applying an extended type substitution to a type τ the locality component is applied first. Similarly as we did for type substitution we define when the application of an extended type substitution is well behaved.

Definition 3.10 *Let $Z = (Z^{type}, Z^{loc})$ be an extended type substitution. We say that $Z\,\tau$ is well behaved iff $Z^{type}\,(Z^{loc}\tau)$ is well behaved; and we say that $Z\,(\tau\backslash C)$ is well behaved iff $Z\,\tau$ is well behaved.*

We also define the application of type and locality substitution to constrained type schemes separately. The added complexity is due to the need of renaming both type and locality variables to avoid capture. Later on, to keep some proofs simpler, we will assume that all bound type and locality variables are different from free variables in a given context, so we shall not need to worry about this cumbersome renaming of variables.

Definition 3.11 *We extend the application of locality substitution to constrained type schemes by first renaming its bound locality variables as follows*[2]

$$Z^{loc}\,(\forall\vec{\alpha}\ \kappa_1 \ldots \kappa_n.\ \gamma) \;=\; \forall\vec{\alpha}\ \kappa'_1 \ldots \kappa'_n.\ Z^{loc}\,(Z'^{\,loc}\,\gamma),$$

where $Z'^{\,loc} = \{\kappa_1 \mapsto \kappa'_1, \ldots \kappa_n \mapsto \kappa'_n\}$ and each κ'_i is a distinct and fresh locality variable.

[2]In this definition and in the next one, *all* bound type and locality variables are renamed to fresh ones. One could change these definitions so that renaming will occur only when needed. The first alternative results in clearer definitions.

Example 3.1 *Consider the constrained type scheme* $\forall\alpha\,\kappa_1\,\kappa_2.\ \beta^j \xrightarrow{\kappa_1} \beta^j\backslash\emptyset$ *and the locality substitution* $Z^{loc} = \{j \mapsto \kappa_1\}$. *We then have*

$$Z^{loc}\,(\forall\alpha\,\kappa_1\,\kappa_2.\ \beta^j \xrightarrow{\kappa_1} \beta^j\backslash\emptyset) \;=\; \forall\alpha\,\kappa_1'\,\kappa_2'.\ \beta^{\kappa_1} \xrightarrow{\kappa_1'} \beta^{\kappa_1}\backslash\emptyset \tag{3.1}$$

since

$$\begin{array}{lll} & Z^{loc}\,(\forall\alpha\,\kappa_1\,\kappa_2.\ \beta^j \xrightarrow{\kappa_1} \beta^j\backslash\emptyset) & \\ = & \forall\alpha\,\kappa_1'\,\kappa_2'.\ Z^{loc}\,(\{\kappa_1 \mapsto \kappa_1', \kappa_2 \mapsto \kappa_2'\}\,(\beta^j \xrightarrow{\kappa_1} \beta^j\backslash\emptyset)) & \text{def. 3.11} \\ = & \forall\alpha\,\kappa_1'\,\kappa_2'.\ Z^{loc}\,(\beta^j \xrightarrow{\kappa_1'} \beta^j\backslash\emptyset) & \text{def. 3.6} \\ = & \forall\alpha\,\kappa_1'\,\kappa_2'.\ \{j \mapsto \kappa_1\}\,(\beta^j \xrightarrow{\kappa_1'} \beta^j\backslash\emptyset) & \text{def. of } Z^{loc} \\ = & \forall\alpha\,\kappa_1'\,\kappa_2'.\ \beta^{\kappa_1} \xrightarrow{\kappa_1'} \beta^{\kappa_1}\backslash\emptyset & \text{def. 3.6.} \end{array}$$

Definition 3.12 *The application of a type substitution to a constrained type scheme requires the renaming of bound locality variables and bound type variables.*

$$Z^{type}\,(\forall\alpha_1 \ldots \alpha_m\,\kappa_1 \ldots \kappa_n.\ \gamma) \;=\; \forall\beta_1 \ldots \beta_m\,\kappa_1' \ldots \kappa_n'.\ Z^{type}\,(Z'^{\,type}\,Z'^{\,loc}\,\gamma)$$

where $Z'^{\,loc} = \{\kappa_1 \mapsto \kappa_1', \ldots \kappa_n \mapsto \kappa_n'\}$ *and each* κ_i' *is a distinct and fresh locality variable, and* $Z'^{\,type} = \{\alpha_1 \mapsto \beta_1^{\phi_1} \ldots \alpha_m \mapsto \beta_m^{\phi_m}\}$, *with each* β_i *a distinct and fresh type variable, and* ϕ_i *is the locality label of* α_i *in* $Z'^{loc}\,\gamma$, *if* $\alpha_i \in FTV(Z'^{loc}\,\gamma)$, *otherwise* ϕ_i *is any label.*

Example 3.2 *In order to illustrate this definition let* $Z^{type} = \{\beta \mapsto \alpha^{\kappa_2'}\,\texttt{ref}^{\,\kappa_1}\}$. *We then have*

$$Z^{type}\,(\forall\alpha\,\kappa_1'\,\kappa_2'.\ \beta^{\kappa_1} \xrightarrow{\kappa_1'} \beta^{\kappa_1}\backslash\emptyset) \;=\; \forall\alpha'\,\kappa_1''\,\kappa_2''.\ \alpha^{\kappa_2'}\,\texttt{ref}^{\,\kappa_1} \xrightarrow{\kappa_1''} \alpha^{\kappa_2'}\,\texttt{ref}^{\,\kappa_1}\backslash\emptyset \tag{3.2}$$

since

$$\begin{array}{lll} & Z^{type}\,(\forall\alpha\,\kappa_1'\,\kappa_2'.\ \beta^{\kappa_1} \xrightarrow{\kappa_1'} \beta^{\kappa_1}\backslash\emptyset) & \\ = & \forall\alpha'\,\kappa_1''\,\kappa_2''.\ Z^{type}\,(\{\alpha \mapsto \alpha'^{\phi}\}(\{\kappa_1' \mapsto \kappa_1'', \kappa_2' \mapsto \kappa_2''\}\,(\beta^{\kappa_1} \xrightarrow{\kappa_1'} \beta^{\kappa_1}\backslash\emptyset))) & \text{def. 3.12} \\ = & \forall\alpha'\,\kappa_1''\,\kappa_2''.\ Z^{type}\,(\{\alpha \mapsto \alpha'^{\phi}\}(\beta^{\kappa_1} \xrightarrow{\kappa_1''} \beta^{\kappa_1}\backslash\emptyset)) & \text{def. 3.6} \\ = & \forall\alpha'\,\kappa_1''\,\kappa_2''.\ Z^{type}\,(\beta^{\kappa_1} \xrightarrow{\kappa_1''} \beta^{\kappa_1}\backslash\emptyset) & \text{def. 3.7} \\ = & \forall\alpha'\,\kappa_1''\,\kappa_2''.\ \{\beta \mapsto \alpha^{\kappa_2'}\,\texttt{ref}^{\,\kappa_1}\}\,(\beta^{\kappa_1} \xrightarrow{\kappa_1''} \beta^{\kappa_1}\backslash\emptyset) & \text{def. of } Z^{type} \\ = & \forall\alpha'\,\kappa_1''\,\kappa_2''.\ \alpha^{\kappa_2'}\,\texttt{ref}^{\,\kappa_1} \xrightarrow{\kappa_1''} \alpha^{\kappa_2}\,\texttt{ref}^{\,\kappa_1}\backslash\emptyset & \text{def. 3.7.} \end{array}$$

We now define the result of applying an extended type substitution to a constrained type scheme. As for constrained types, the locality component is applied first. So for a constrained type scheme σ we have

$$(Z^{type}, Z^{loc})\,\sigma \;=\; Z^{type}\,(Z^{loc}\,\sigma).$$

So for example, let Z^{loc} be as in Example 3.1 and let Z^{type} be as in Example 3.2. Then

$$(Z^{type}, Z^{loc})\,(\forall\alpha\,\kappa_1\,\kappa_2.\ \beta^j \xrightarrow{\kappa_1} \beta^j\backslash\emptyset) \ = \ \forall\alpha'\,\kappa_1''\,\kappa_2''.\ \alpha^{\kappa_2'}\,\mathtt{ref}^{\,\kappa_1} \xrightarrow{\kappa_1''} \alpha^{\kappa_2}\,\mathtt{ref}^{\,\kappa_1}\backslash\emptyset$$

since

$$\begin{array}{lll} & (Z^{type}, Z^{loc})\,(\forall\alpha\,\kappa_1\,\kappa_2.\ \beta^j \xrightarrow{\kappa_1} \beta^j\backslash\emptyset) & \\ = & Z^{type}\,(Z^{loc}\,(\forall\alpha\,\kappa_1\,\kappa_2.\ \beta^j \xrightarrow{\kappa_1} \beta^j\backslash\emptyset)) & \\ = & Z^{type}\,(\forall\alpha\,\kappa_1'\,\kappa_2'.\ \beta^{\kappa_1} \xrightarrow{\kappa_1'} \beta^{\kappa_1}\backslash\emptyset) & \text{ex. 3.1} \\ = & \forall\alpha'\,\kappa_1''\,\kappa_2''.\ \alpha^{\kappa_2'}\,\mathtt{ref}^{\,\kappa_1} \xrightarrow{\kappa_1''} \alpha^{\kappa_2}\,\mathtt{ref}^{\,\kappa_1}\backslash\emptyset & \text{ex. 3.2.} \end{array}$$

Type, locality and extended type substitutions may be applied to term and store type contexts, by applying them pointwise.

The composition of type substitutions and the composition of locality substitutions is ordinary function composition. Next we define composition of extended type substitutions.

We use the same symbol $\circ$ for both composition of functions and composition of extended type substitutions. Any ambiguity is resolved by the context. Note that a locality substitution may also be applied to type substitution by pointwise application. So if $Z^{type} = \{\alpha_1 \mapsto \tau_1 \ldots, \alpha_n \mapsto \tau_n\}$ we have

$$Z^{loc}\,Z^{type} = \{\alpha_1 \mapsto Z^{loc}\tau_1 \ldots, \alpha_n \mapsto Z^{loc}\tau_n\}.$$

Definition 3.13 *Let $Z = (Z^{type}, Z^{loc})$ and $Z' = (Z'^{\,type}, Z'^{\,loc})$ be two extended type substitutions. We define the composition of Z and Z', written $Z \circ Z'$, as the following extended type substitution*

$$(Z^{type} \circ (Z^{loc}\,Z'^{type}),\ Z^{loc} \circ Z'^{\,loc}).$$

The following lemma states some properties of the composition of extended type substitutions.

Lemma 3.3 *Let Z_1, Z_2 and Z_3 be extended type substitutions. Then*

1. *if $Z_2\,\tau$ is well behaved then $(Z_1 \circ Z_2)\,(\tau\backslash C) = Z_1\,(Z_2\,(\tau\backslash C))$, and*
2. *$Z_1 \circ (Z_2 \circ Z_3) = (Z_1 \circ Z_2) \circ Z_3$.*

Proof. Straightforward from Definition 3.13, the way type and locality substitutions are defined and properties of function composition. ∎

As an example of why $Z_2\tau$ in the first case of the lemma above should be well behaved, suppose that $Z = (Id^{type},\ \{\kappa \mapsto \epsilon\})$ and $Z' = (\{\alpha \mapsto \texttt{int ref}^{\,\epsilon}\},\ Id^{loc})$. One can readily check that $(Z \circ Z')\ \alpha^\kappa \neq Z\ (Z'\alpha^\kappa)$.

Definition 3.14 (Instantiation) *We say that $\tau'\backslash C'$ is an* instance *of $\forall\vec{\alpha}\vec{\kappa}.\tau\backslash C$, written $\forall\vec{\alpha}\vec{\kappa}.\tau\backslash C \succ \tau'\backslash C'$ iff for some extended type substitution $Z = (Z^{type}, Z^{loc})$ such that $Dom(Z^{loc}) = \lceil\vec{\kappa}\rceil$ and $Dom(Z^{type}) = \lceil\vec{\alpha}\rceil$ we have $Z(\tau\backslash C) = \tau'\backslash C'$. Extending this notion to constrained type schemes, say that $\sigma \succ \sigma'$ iff for all τ, C, we have $\sigma' \succ \tau\backslash C$ implies $\sigma \succ \tau\backslash C$.*

Here are some examples of instantiation:

1. $\forall\alpha.\, \alpha^\epsilon \xrightarrow{\epsilon} \alpha^\epsilon\backslash\,\{\} \ \succ\ \texttt{int} \xrightarrow{\epsilon} \texttt{int}\backslash\{\}$
2. $\forall\alpha.\, \alpha^\epsilon \xrightarrow{\epsilon} \alpha^\epsilon\backslash\,\{\} \ \succ\ \texttt{int ref}^{\,\epsilon} \xrightarrow{\epsilon} \texttt{int ref}^{\,\epsilon}\,\texttt{ref}^{\,\epsilon}\backslash\{\}$
3. $\forall\alpha\kappa\kappa'\kappa''.\, \alpha^\kappa \xrightarrow{\kappa''} \alpha^\kappa\,\texttt{ref}^{\,\kappa'}\backslash\{\kappa' \leq \kappa\} \ \succ\ \texttt{int ref}^{\,\epsilon} \xrightarrow{\epsilon} \texttt{int ref}^{\,\epsilon}\,\texttt{ref}^{\,\epsilon}\backslash\{\epsilon \leq \epsilon\}$
4. $\forall\alpha\kappa\kappa'\kappa''.\, \alpha^\kappa \xrightarrow{\kappa''} \alpha^\kappa\,\texttt{ref}^{\,\kappa'}\backslash\{\kappa' \leq \kappa\} \ \succ\ \texttt{int ref}^{\,j} \xrightarrow{\epsilon} \texttt{int ref}^{\,j}\,\texttt{ref}^{\,j'}\backslash\{j' \leq j\}$
5. $\forall\alpha\kappa\kappa'\kappa''.\, \alpha^\kappa \xrightarrow{\kappa''} \alpha^\kappa\,\texttt{ref}^{\,\kappa'}\backslash\{\kappa' \leq \kappa\} \ \succ\ \texttt{int} \xrightarrow{\epsilon} \texttt{int ref}^{\,j'}\backslash\{j' \leq j\}$.

Examples 1 and 2 above show the instantiation of constrained type schemes with no bound locality variables and with empty sets of constraints. Note that, as illustrated by examples 3 and 5 above, instantiation may produce constrained types with some obsolete constraints. In example 3 the set $\{\epsilon \leq \epsilon\}$ is obsolete as it is not constraining any locality variable. In example 5 the locality variable j in the set $\{j' \leq j\}$ does not appear in the type, so this constraint also has no effect.

We assume the usual notion of α-conversion, so we say that $\sigma_1 =_\alpha \sigma_2$ if σ_1 and σ_2 differ only by the names of their bound type and locality variables. As a consequence of the next lemma from now on we will identify α-equivalent constrained type schemes.

Lemma 3.4 *If $\sigma =_\alpha \sigma'$ then $\sigma \succ \sigma'$ and $\sigma' \succ \sigma$.*

PROOF. By the definition of α-equivalence and by Definition 3.14 above. ∎

Next we give a series of definitions of *escaping* predicates. The intuition behind these predicates will be given in the next section where we explain the mechanisms used by the type system to detect references that can escape.

Definition 3.15 *The predicate* **esc?**(τ) *holds iff* **lab** $(\tau) = \epsilon$ *or* **lab** (τ) *is undefined.*

Definition 3.16 *The predicate* **esc?**(σ) *holds , iff there exists a constrained type $\tau\backslash C$ with C valid, such that $\sigma \succ \tau\backslash C$ and* **esc?**(τ).

Constrained types can be seen as constrained type schemes with no bound variables. Hence the following lemma is an immediate consequence of the previous definition.

Lemma 3.5 $\textbf{esc?}(\tau\backslash C)$ *iff* C *is valid and* $\textbf{esc?}(\tau)$.

PROOF. From Definition 3.16 and the fact that $\tau\backslash C \succ \tau\backslash C$. ∎

Definition 3.16 is based on the *existence* of an appropriate extended type substitution. We can view the result of the next auxiliary lemma as providing an alternative way to decide if $\textbf{esc?}(\sigma)$ holds based solely on the form of the type scheme σ itself. This lemma will be used to simplify some subsequent proofs. In the following, we write C^+ for the transitive closure of C.

Lemma 3.6 $\textbf{esc?}(\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C)$ *iff*

1. $\epsilon \leq \ell \notin C^+$ *and* $\epsilon \leq \kappa \in C^+$ *only if* $\kappa \in \lceil\vec{\kappa}\rceil$, *and*
2. *exactly one of the following holds:*
   (a) $\tau = \alpha^\ell$ *for some* $\alpha \in \lceil\vec{\alpha}\rceil$,
   (b) $\textbf{esc?}(\tau)$, *or*
   (c) $\textbf{lab}\,(\tau) \in \lceil\vec{\kappa}\rceil$, $\textbf{lab}\,(\tau) \leq \ell \notin C^+$ *and* $\textbf{lab}\,(\tau) \leq \kappa' \in C^+$ *only if* $\kappa' \in \lceil\vec{\kappa}\rceil$.

PROOF. See Appendix A.1. ∎

Definition 3.17 *The predicate* $\textbf{esc?}(\Gamma, \Delta, M)$ *holds iff*

- $FV(M) \subseteq Dom(\Gamma)$ *and for all* x *in* $FV(M)$, $\textbf{esc?}(\Gamma(x))$, *and*
- $Refs(M) \subseteq Dom(\Delta)$ *and for all* r *in* $Refs(M)$, $\textbf{esc?}(\Delta(r))$.

So for instance, the predicate

$$\textbf{esc?}([x \mapsto \texttt{int ref}^{\,\epsilon}\backslash\emptyset], [r \mapsto \texttt{int ref}^{\,\epsilon}\backslash\emptyset], \texttt{fn}\, y.(x, r))$$

holds since the constrained types of the free variable x and the reference constant r satisfy the predicate $\textbf{esc?}$. But the following predicate

$$\textbf{esc?}([x \mapsto \forall\kappa.\,\texttt{int ref}^{\,\ell} \times^\kappa \texttt{int}\backslash\kappa \leq \ell], \{\}, \texttt{fn}\, y.x)$$

$$
\begin{array}{lcl}
n & : & \mathtt{int} \backslash \emptyset \\
\mathtt{unit} & : & \mathtt{unit} \backslash \emptyset \\
\mathtt{true} & : & \mathtt{bool} \backslash \emptyset \\
\mathtt{false} & : & \mathtt{bool} \backslash \emptyset \\
\mathtt{succ} & : & \mathtt{int} \xrightarrow{\kappa} \mathtt{int} \backslash \emptyset \\
\mathtt{pred} & : & \mathtt{int} \xrightarrow{\kappa} \mathtt{int} \backslash \emptyset \\
\mathtt{zero?} & : & \mathtt{int} \xrightarrow{\kappa} \mathtt{bool} \backslash \emptyset \\
\mathtt{fst} & : & (\alpha^{\kappa} \times^{\kappa''} \beta^{\kappa'}) \xrightarrow{\kappa'''} \alpha^{\kappa} \backslash \emptyset \\
\mathtt{snd} & : & (\alpha^{\kappa} \times^{\kappa''} \beta^{\kappa'}) \xrightarrow{\kappa'''} \beta^{\kappa'} \backslash \emptyset \\
\mathtt{ref} & : & \alpha^{\kappa} \xrightarrow{\kappa'} \alpha^{\kappa}\, \mathtt{ref}^{\,\kappa''} \backslash \{\kappa'' \leq \kappa\} \\
\mathtt{deref} & : & \alpha^{\kappa}\, \mathtt{ref}^{\,\kappa'} \xrightarrow{\kappa''} \alpha^{\kappa} \backslash \emptyset \\
:= & : & (\alpha^{\kappa}\, \mathtt{ref}^{\,\kappa'} \times^{\kappa''} \alpha^{\kappa}) \xrightarrow{\kappa'''} \mathtt{unit} \backslash \emptyset \\
\mathtt{fork} & : & (\mathtt{unit} \xrightarrow{\kappa} \mathtt{unit}) \xrightarrow{\kappa'} \mathtt{unit} \backslash \emptyset \\
\mathtt{rfork}_{\rho} & : & (\mathtt{unit} \xrightarrow{\epsilon} \mathtt{unit}) \xrightarrow{\kappa} \mathtt{unit} \backslash \emptyset
\end{array}
$$

Figure 3.1: Constrained type schemes for constants.

does not hold since the constrained type scheme of the free variable x does not satisfy the predicate **esc?**. If the bound locality variable κ that labels the product type is instantiated to ϵ the invalid constraint set $\{\epsilon \leq \ell\}$ is produced.

Type judgements have the form $\Gamma; \Delta \vdash M : \tau \backslash C$ where M is a term of the language and Γ and Δ are term and store type contexts respectively and C must be a valid constraint set. We sometimes omit Γ and/or Δ when they are empty. The rules of the type system appear in Figures 3.1 and 3.2. Figure 3.1 has the constrained type schemes of all constants other than references. Each constant c has a constrained type scheme given by $\mathbf{typeof}(c)$. Quantifiers are omitted to simplify the presentation. We say that an expression M is *well-typed* if there are Γ, Δ and a constrained labelled type $\tau \backslash C$ such that $\Gamma; \Delta \vdash M : \tau \backslash C$.

Observe that in the conclusion of the type rules we often form the union of the constraints sets in the premises. Some of these constraints propagated towards the root of a type derivation may become obsolete. In an implementation, we might omit displaying obsolete constraints.

In the next section we explain the rules in Figure 3.2, giving the intuitions for the ways they assign locality labels to types.

VAR $$\frac{\mathbf{typeof}(c) \succ \tau \backslash C}{\Gamma; \Delta \vdash c : \tau \backslash C}$$

CTE $$\frac{\Gamma(x) \succ \tau \backslash C}{\Gamma; \Delta \vdash x : \tau \backslash C}$$

REF $$\Gamma; \Delta \vdash r : \Delta(r)$$

ABS$_\ell$ $$\frac{\Gamma[x \mapsto \tau_1 \backslash C']; \Delta \vdash M : \tau_2 \backslash C}{\Gamma; \Delta \vdash \mathtt{fn}\, x.M : \tau_1 \xrightarrow{\ell} \tau_2 \backslash C \cup C'}$$

ABS$_\epsilon$ $$\frac{\Gamma[x \mapsto \tau_1 \backslash C']; \Delta \vdash M : \tau_2 \backslash C \quad \mathbf{esc?(}\Gamma, \Delta, \mathtt{fn}\, x.M\mathbf{)}}{\Gamma; \Delta \vdash \mathtt{fn}\, x.M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \cup C'}$$

REC$_\ell$ $$\frac{\Gamma[f \mapsto \tau_1 \xrightarrow{\ell} \tau_2 \backslash C][x \mapsto \tau_1 \backslash C']; \Delta \vdash M : \tau_2 \backslash C''}{\Gamma; \Delta \vdash \mathtt{rec}\, f\, x.M : \tau_1 \xrightarrow{\ell} \tau_2 \backslash C \cup C' \cup C''}$$

REC$_\epsilon$ $$\frac{\Gamma[f \mapsto \tau_1 \xrightarrow{\phi} \tau_2 \backslash C][x \mapsto \tau_1 \backslash C']; \Delta \vdash M : \tau_2 \backslash C'' \quad \mathbf{esc?(}\Gamma, \Delta, \mathtt{rec}\, f\, x.M\mathbf{)}}{\Gamma; \Delta \vdash \mathtt{rec}\, f\, x.M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \cup C' \cup C''}$$

PAIR $$\frac{\Gamma; \Delta \vdash M_1 : \tau_1 \backslash C \quad \Gamma; \Delta \vdash M_2 : \tau_2 \backslash C'}{\Gamma; \Delta \vdash \mathtt{(}M_1, M_2\mathtt{)} : \tau_1 \times^{\phi} \tau_2 \backslash C \cup C' \cup C'' \cup C'''}$$

where :
$C'' = \{\phi \leq \mathbf{lab}\ (\tau_1)\}$, *if* $\mathbf{lab}\ (\tau_1)$ *is defined*, $C'' = \emptyset$ *otherwise*
$C''' = \{\phi \leq \mathbf{lab}\ (\tau_2)\}$, *if* $\mathbf{lab}\ (\tau_2)$ *is defined*, $C''' = \emptyset$ *otherwise*

APP $$\frac{\Gamma; \Delta \vdash M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \quad \Gamma; \Delta \vdash N : \tau_1 \backslash C'}{\Gamma; \Delta \vdash M N : \tau_2 \backslash C \cup C'}$$

COND $$\frac{\Gamma; \Delta \vdash M : \mathtt{bool} \backslash C \quad \Gamma; \Delta \vdash N : \tau \backslash C' \quad \Gamma; \Delta \vdash P : \tau \backslash C''}{\Gamma; \Delta \vdash \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ P : \tau \backslash C \cup C' \cup C''}$$

LET$_V$ $$\frac{\Gamma; \Delta \vdash V : \tau' \backslash C' \quad \Gamma[x \mapsto Gen(\tau' \backslash C', \Gamma, \Delta)]; \Delta \vdash N : \tau \backslash C}{\Gamma; \Delta \vdash \mathtt{let}\ x = V\ \mathtt{in}\ N : \tau \backslash C \cup C'}$$

LET $$\frac{\Gamma; \Delta \vdash M : \tau' \backslash C' \quad \Gamma[x \mapsto \tau' \backslash C']; \Delta \vdash N : \tau \backslash C}{\Gamma; \Delta \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N\ \mathtt{end} : \tau \backslash C \cup C'}$$

Figure 3.2: Typing rules.

3.5 Types and Locality

Our type system detects references that may escape from one processor to another – the others, in a result that we prove in Chapter 5, are sure to be local. There are four mechanisms in the type system to find out which types should be labelled with ϵ:

1. the first mechanism is the type scheme for $\texttt{rfork}_\rho$ constants in Figure 3.1:

   $$\texttt{rfork}_\rho : \forall\kappa.\,(\texttt{unit} \xrightarrow{\epsilon} \texttt{unit}) \xrightarrow{\kappa} \texttt{unit}\backslash\emptyset.$$

   This type scheme requires the argument of $\texttt{rfork}_\rho$ to be of a function type labelled with ϵ. This is sensible, since applying a remote fork constant in a parent thread spawns a child thread on a possibly different processor;

2. the second mechanism is the use of the predicate **esc?** in the rule in Figure 3.2 used to type functions with label ϵ or with a locality variable:

   $$\frac{\Gamma[x \mapsto \tau_1\backslash C'];\Delta \vdash P\colon \tau_2\backslash C \quad \textbf{esc?}(\Gamma;\Delta,\texttt{fn}\,x.P)}{\Gamma;\Delta \vdash \texttt{fn}\,x.P\colon \tau_1 \xrightarrow{\phi} \tau_2\backslash C \cup C'}$$

   As an example of how the predicate **esc?** is used consider the following expression

   $$\begin{array}{l}\texttt{let}\ x = \texttt{ref unit in}\\ \quad \texttt{let}\ p = (x, 0)\ \texttt{in}\\ \quad\quad \texttt{rfork}_{\rho'}\ \texttt{fn}\,u.(\texttt{deref}\,(\texttt{fst}\ \ p))\\ \quad \texttt{end}\\ \texttt{end}\,.\end{array}$$

   Clearly the reference x can escape to a processor different than the one where it is created. Its reference type should then be labelled with ϵ. The type system finds this out by requiring the predicate **esc?** to hold when typing the function $\texttt{fn}\,u.(\texttt{deref}\,(\texttt{fst}\ \ p))$. So suppose the *rforked* function is typed in the following type environment

   $$\Gamma = [x \mapsto \texttt{unit ref}^{\,\phi},\ \ p \mapsto \forall\kappa.\,\texttt{unit ref}^{\,\phi} \times^{\kappa} \texttt{int}\backslash\kappa \leq \phi].$$

   Then, according to Definition 3.17,

   $$\textbf{esc?}(\Gamma, \{\}, \texttt{fn}\,u.(\texttt{deref}\,(\texttt{fst}\ \ p))$$

   holds only if

   $$\textbf{esc?}(\forall\kappa.\,\texttt{unit ref}^{\,\phi} \times^{\kappa} \texttt{int}\backslash\kappa \leq \phi)$$

holds, which in its turn, according to Definition 3.16 (or Lemma 3.6) holds only if $\phi = \epsilon$.

If a function is to be polymorphic in its locality it may have an escaping instance. For this reason the type rule used for deriving $\Gamma; \Delta \vdash \texttt{fn}\, x.M : \tau_1 \xrightarrow{\kappa} \tau_2 \backslash C$ should also use the predicate $\mathbf{esc?}(\Gamma; \Delta, \texttt{fn}\, x.M)$. All we just said about function abstractions also applies to recursive functions;

3. the third mechanism is a combination of the type scheme we assign to the constant `ref` in Figure 3.1

$$\texttt{ref} : \forall \alpha\kappa\kappa'\kappa''.\alpha^{\kappa} \xrightarrow{\kappa'} \alpha^{\kappa}\, \texttt{ref}^{\,\kappa''} \backslash \{\kappa'' \leq \kappa\}$$

and the assumption that type derivations with invalid constraint sets are rejected. When `ref` is applied, if the resulting reference type is labelled with ϵ then the "inner" type cannot be labelled with ℓ or with a locality variable as that would produce an invalid constraint set.

4. the fourth mechanism is analogous to the previous one. It is a combination of the rule for typing pairs:

$$\frac{\Gamma; \Delta \vdash M_1 : \tau_1 \backslash C \quad \Gamma; \Delta \vdash M_2 : \tau_2 \backslash C'}{\Gamma; \Delta \vdash (M_1, M_2) : \tau_1 \times^{\phi} \tau_2 \backslash C \cup C' \cup C'' \cup C'''}$$

where :
$C'' = \{\phi \leq \mathbf{lab}\ (\tau_1)\}$, *if* $\mathbf{lab}\ (\tau_1)$ *is defined*, $C'' = \emptyset$ *otherwise*
$C''' = \{\phi \leq \mathbf{lab}\ (\tau_2)\}$, *if* $\mathbf{lab}\ (\tau_2)$ *is defined*, $C''' = \emptyset$ *otherwise*

and the assumption that type derivations producing invalid constraint sets are rejected. So if the type of a pair (M_1, M_2) is labelled with ϵ then τ_1 and τ_2, the type of its components must be such that $\mathbf{esc?}(\tau_1)$ and $\mathbf{esc?}(\tau_2)$, that is, they cannot be labelled with ℓ or with a locality variable.

An immediate consequence of the next lemma is that if a value has its type labelled with the escape label the types of its references subterms should also be labelled with ϵ.

Lemma 3.7 *If* $\Gamma; \Delta \vdash V : \tau \backslash C$ *and* $\mathbf{esc?}(\tau)$ *then* $\mathbf{esc?}(\Gamma, \Delta, V)$.

Proof. By induction on the height of type derivation for V.

case x

We have that $\Gamma; \Delta \vdash x : \tau \backslash C$. Then $\Gamma(x) \succ \tau \backslash C$ and as $\mathbf{esc?}(\tau)$ and C is valid, by Definition 3.17, we have $\mathbf{esc?}(\Gamma, \Delta, x)$.

case r

We have that $\Delta(r) = \tau' \, \texttt{ref}^{\,\epsilon} \backslash C$ with C valid. Then by Definition 3.17, we have **esc?**(Γ, Δ, r).

case c other than a reference

We have $FV(c) = \emptyset$ and $\mathit{Refs}\,(c) = \emptyset$. Then **esc?**$(\Gamma, \Delta, c)$ holds trivially.

case $\texttt{fn}\, x'.P$

We have a type derivation of the form

$$\frac{\Gamma[x' \mapsto \tau_1 \backslash C_1]; \Delta \vdash P \colon \tau_2 \backslash C_2 \quad \mathbf{esc?}(\Gamma, \Delta, \texttt{fn}\, x'.P)}{\Gamma; \Delta \vdash \texttt{fn}\, x'.P \colon \tau_1 \xrightarrow{\epsilon} \tau_2 \backslash C_1 \cup C_2}$$

where the predicate **esc?**$(\Gamma, \Delta, \texttt{fn}\, x'.P)$ holds as a premise.

case $\texttt{rec}\, f\, x.P$

Same as the proof for the previous case.

case (V_1, V_2)

The proof of $\Gamma; \Delta \vdash (V_1, V_2) : \tau_1 \times^{\epsilon} \tau_2 \backslash C$ must have been of this form:

$$\frac{\Gamma; \Delta \vdash V_1 \colon \tau_1 \backslash C_1 \quad \Gamma; \Delta \vdash V_2 \colon \tau_2 \backslash C_2}{\Gamma; \Delta \vdash (V_1, V_2) \colon \tau_1 \times^{\epsilon} \tau_2 \backslash C_1 \cup C_2 \cup C_3 \cup C_4}$$

where

$C_3 = \{\epsilon \leq \mathbf{lab}\ (\tau_1)\}$ *or* $C_3 = \emptyset$ *if* $\mathbf{lab}\ (\tau_1)$ *is undefined*

$C_4 = \{\epsilon \leq \mathbf{lab}\ (\tau_2)\}$ *or* $C_4 = \emptyset$ *if* $\mathbf{lab}\ (\tau_2)$ *is undefined*

As C_3 and C_4 are valid, $\mathbf{lab}\,(\tau_1) = \epsilon$ or it is undefined and the same for $\mathbf{lab}\,(\tau_2)$, which means that **esc?**(τ_1) and **esc?**(τ_2). Hence by the induction hypothesis and the first and second premises of the proof above we have that **esc?**(Γ, Δ, V_1) and **esc?**(Γ, Δ, V_2). We then conclude that **esc?**$(\Gamma, \Delta, (V_1, V_2))$. ∎

Observe that the lemma above does not hold for non-values terms. Consider for example the following type judgement

$$\Gamma, \Delta \vdash \texttt{snd}\ (x, 3) \colon \texttt{int} \backslash \{\}$$

where Γ maps x to the type $\texttt{int}\,\texttt{ref}^{\,\ell} \backslash \{\}$. One can easily check that this type judgement is derivable using the rules of the type system and that **esc?**$(\texttt{int})$ holds. But **esc?**$(\Gamma, \Delta, \texttt{snd}\ (x, 3))$ is false since the type of x in Γ is labelled with ℓ.

Later on Chapter 5 we prove that the locality analysis specified by the type system is safe meaning that no reference is considered local if it can escape.

Equally important, we note that the analysis is not trivial, that is, it allows references to be tagged as local.

Lemma 3.8 *The locality analysis given by the type system in Figures 3.1 and 3.2 is not trivial.*

PROOF. The proof consists of exhibiting type derivations where references can be labelled with ℓ. In any non-distributed program, for example, references that do not occur free in a locally polymorphic function can be labelled with ℓ. ∎

Next we give an example of a type derivation using the rules of Figure 3.2.

Example 3.3 *The following is a type derivation for the expression*

$$\texttt{let}\ x = \texttt{ref unit in}\ M\ \texttt{end}$$

where

$$M \equiv \texttt{fork fn}\, u.\, \texttt{deref}\ x.$$

The letters X and U stand for the following term type contexts

$$X \equiv [x \mapsto \texttt{unit ref}^{\,\ell} \backslash \ell \leq \kappa] \quad \textit{and} \quad Y \equiv [y \mapsto \texttt{unit} \backslash \emptyset].$$

$$\dfrac{\dfrac{\dfrac{\mathbf{typeof}(\texttt{ref}) \succ \texttt{unit} \xrightarrow{\phi} \texttt{unit ref}^{\,\ell} \backslash \ell \leq \kappa}{\vdash \texttt{ref} : \texttt{unit} \xrightarrow{\phi} \texttt{unit ref}^{\,\ell} \backslash \ell \leq \kappa} \qquad \vdash \texttt{unit} : \texttt{unit} \backslash \emptyset}{\vdash \texttt{ref unit} : \texttt{unit ref}^{\,\ell} \backslash \ell \leq \kappa} \qquad \dfrac{\textit{see below}}{X \vdash M : \texttt{unit} \backslash \ell \leq \kappa}}{\vdash \texttt{let}\ x = \texttt{ref unit in}\ M\ \texttt{end} : \texttt{unit} \backslash \ell \leq \kappa}$$

$$\dfrac{\dfrac{\mathbf{typeof}(\texttt{fork}) \succ (\texttt{unit} \xrightarrow{\phi} \texttt{unit}) \xrightarrow{\phi'} \texttt{unit} \backslash \emptyset}{X \vdash \texttt{fork} : (\texttt{unit} \xrightarrow{\phi} \texttt{unit}) \xrightarrow{\phi'} \texttt{unit} \backslash \emptyset} \qquad \dfrac{\textit{see below}}{X \vdash \texttt{fn}\, u.\, \texttt{deref}\ x : \texttt{unit} \xrightarrow{\phi} \texttt{unit} \backslash \ell \leq \kappa}}{X \vdash \texttt{fork fn}\, u.\, \texttt{deref}\ x : \texttt{unit} \backslash \ell \leq \kappa}$$

$$\dfrac{\dfrac{\dfrac{\mathbf{typeof}(\texttt{deref}) \succ \texttt{unit ref}^{\,\ell} \xrightarrow{\phi} \texttt{unit} \backslash \emptyset}{XU \vdash \texttt{deref} : \texttt{unit ref}^{\,\ell} \xrightarrow{\phi} \texttt{unit} \backslash \emptyset} \qquad \dfrac{\mathbf{typeof}(x) \succ \texttt{unit ref}^{\,\ell} \backslash \ell \leq \kappa}{XU \vdash x : \texttt{unit ref}^{\,\ell} \backslash \ell \leq \kappa}}{XU \vdash \texttt{deref}\ x : \texttt{unit} \backslash \ell \leq \kappa}}{X \vdash \texttt{fn}\, u.\, \texttt{deref}\ x : \texttt{unit} \xrightarrow{\phi} \texttt{unit} \backslash \ell \leq \kappa}$$

In the derivation above observe that constraints are introduced when instantiating the type scheme of the `ref` constant. Instantiation may produce *junk* locality variables, as is the case of the locality variable κ above. In an implementation, sets of constraints should be simplified to eliminate *junk* constraints.

3.6 Conservative Extension

Before considering in the following chapters other fundamental properties of our type system, we prove that the expressions accepted by it can also be typed under the Hindley-Milner polymorphic discipline. Because that type discipline forms the base of the type system for languages like SML, this result gives us an indication that our locality analysis can be extended to the full SML language.

The type system which corresponds to the Hindley-Milner type discipline is given by the first set of rules in Figure 3.3. We will refer to that type system as the ML type system. We write ι to represent ML types and all the other objects used in the ML type system are annotated with the subscript $_{\text{ML}}$.

For the purpose of the conservative extension proof it is enough to consider a simplified version of our type system. This simplified version, presented in the second set of rules in Figure 3.3, is the same as the type system in Figure 3.2 with the following modifications:

- type judgements do not have store typing contexts, so there is no rule for typing reference constants, and
- there are no typing rules for
  - constants,
  - conditional expression,
  - pairs,
  - let-expressions binding variables to non-value expressions, and
  - recursive functions.

We will prove the conservative extension result for source expressions which do not contain reference constants, for this reason we can omit store typing contexts from type judgements and the rule for typing references

The proof cases for the conditional expression and pairs follow the same lines as the proof case for application. The let expression binding a non-value to a variable can be seen as a sugared version of a function applied to a term. The conservative extension result for this case follows by composing the result for functions and applications. We also eliminate the rule for typing constants assuming that for each constant c in Figure 3.1 there is a variable named after it in the domain of term type contexts and that the type scheme associated to this variable is the same as **typeof** (c).

$$\frac{\Gamma(x) \succ_{\text{ML}} \iota}{\Gamma \vdash_{\text{ML}} x : \iota}$$

$$\frac{\Gamma[x \mapsto \iota_1] \vdash_{\text{ML}} M : \iota_2}{\Gamma \vdash_{\text{ML}} \texttt{fn}\, x.M : \iota_1 \to \iota_2}$$

$$\frac{\Gamma \vdash_{\text{ML}} M : \iota_1 \to \iota_2 \quad \Gamma \vdash_{\text{ML}} N : \iota_1}{\Gamma \vdash_{\text{ML}} M\ N : \iota_2}$$

$$\frac{\Gamma \vdash_{\text{ML}} V : \iota' \quad \Gamma[x \mapsto Gen_{\text{ML}}(\iota', \Gamma)] \vdash_{\text{ML}} N : \iota}{\Gamma \vdash_{\text{ML}} \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : \iota}$$

$$\frac{\Gamma(x) \succ \tau \backslash C}{\Gamma \vdash x : \tau \backslash C}$$

$$\frac{\Gamma[x \mapsto \tau_1 \backslash C'] \vdash M : \tau_2 \backslash C}{\Gamma \vdash \texttt{fn}\, x.M : \tau_1 \xrightarrow{\ell} \tau_2 \backslash C \cup C'}$$

$$\frac{\Gamma[x \mapsto \tau_1 \backslash C'] \vdash M : \tau_2 \backslash C \quad \textbf{esc?}(\Gamma, \texttt{fn}\, x.M)}{\Gamma \vdash \texttt{fn}\, x.M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \cup C'}$$

$$\frac{\Gamma \vdash M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \quad \Gamma \vdash N : \tau_1 \backslash C'}{\Gamma \vdash M N : \tau_2 \backslash C \cup C'}$$

$$\frac{\Gamma \vdash V : \tau' \backslash C' \quad \Gamma[x \mapsto Gen(\tau' \backslash C', \Gamma)] \vdash N : \tau \backslash C}{\Gamma \vdash \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : \tau \backslash C \cup C'}$$

Figure 3.3: Type systems for the conservative extension result.

The rules for recursive functions were also eliminated as the proof cases involving these rules are analogous to the proof cases for functions.

We define a map ψ which simply erases all locality information and constraint information from objects from our type system:

$$\psi(\tau) \;=\; \begin{cases} \alpha & \textit{if } \tau = \alpha^{\phi} \\ \tau & \textit{if } \tau \in \{\texttt{int}, \texttt{bool}, \texttt{unit}\} \\ \psi(\tau_1) \rightarrow \psi(\tau_2) & \textit{if } \tau = \tau_1 \xrightarrow{\phi} \tau_2 \\ \psi(\tau_1)\ \texttt{ref} & \textit{if } \tau = \tau_1\ \texttt{ref}^{\,\phi} \\ \psi(\tau_1) \times \psi(\tau_2) & \textit{if } \tau = \tau_1 \times^{\phi} \tau_2 \end{cases}$$

$$\psi(\tau \backslash C) \;=\; \psi(\tau)$$

$$\psi\left(\forall \vec{\alpha}\vec{\kappa}.\, \tau \backslash C\right) \;=\; \forall \vec{\alpha}.\, \psi(\tau \backslash C)$$

$$\begin{aligned} \psi(\Gamma) \;=\; & \Gamma_{\mathrm{ML}} \textit{ where } \mathit{Dom}(\Gamma_{\mathrm{ML}}) = \mathit{Dom}(\Gamma) \textit{ and} \\ & \quad \forall x \in \mathit{Dom}(\Gamma_{\mathrm{ML}}),\, \Gamma_{\mathrm{ML}}(x) = \psi(\Gamma(x)). \end{aligned}$$

Theorem 3.1 *Let M be a term of the language, then*

1. *if $\Gamma \vdash M : \gamma$ then $\psi(\Gamma) \vdash_{ML} M : \psi(\gamma)$, and*
2. *if $\Gamma_{ML} \vdash_{ML} M : \iota$ then there exist Γ and γ such that $\psi(\Gamma) = \Gamma_{ML}$, $\psi(\gamma) = \iota$, and $\Gamma \vdash M : \gamma$.*

PROOF. Part *1* is proved by induction on the type derivation of M using our type system. Part *2* is proved by induction on the ML type derivation of M (see Appendix A.1 for auxiliary lemmas and a complete proof). ∎

Chapter 4

Type Soundness

For clarity of presentation we split the soundness result of the type system with respect to the operational semantics in two parts. In this chapter we prove *type* soundness, not taking into account locality properties. *Locality* soundness is proved in the next chapter.

In order to prove type soundness we follow the syntactic approach of Wright and Felleisen [31] and Harper [12]: we first prove type preservation in Section 4.1, then we prove in Section 4.2 that expressions that may lead to execution errors are untypable, and finally we prove type soundness in Section 4.3.

4.1 Type Preservation

In order to prove type preservation we need a few preliminary results. The following two lemmas say that the type of a term and if the term satisfies or not the three-place predicate esc? depend only on its free variables and reference constants which are its subterms.

Lemma 4.1 *(1) If* $\mathbf{esc?}(\Gamma, \Delta, M)$ *and* $\Delta'(r) = \Delta(r)$ *for all* $r \in \mathit{Refs}(M)$ *then* $\mathbf{esc?}(\Gamma, \Delta', M)$*, and (2) if* $\Gamma; \Delta \vdash M : \tau \backslash C$ *and* $\Delta'(r) = \Delta(r)$ *for all* $r \in \mathit{Refs}(M)$ *then* $\Gamma; \Delta' \vdash M : \tau \backslash C$*.*

Proof. Part *1* follows from Definition 3.17. Part *2* is proved by induction on the height of type derivations using result *1* when necessary. ∎

Lemma 4.2 *(1) If* $\mathbf{esc?}(\Gamma, \Delta, M)$ *and* $\Gamma'(x) = \Gamma(x)$ *for all* $x \in FV(M)$ *then* $\mathbf{esc?}(\Gamma', \Delta, M)$*, and (2) if* $\Gamma; \Delta \vdash M : \tau \backslash C$ *and* $\Gamma'(x) = \Gamma(x)$ *for all* $x \in FV(M)$ *then* $\Gamma'; \Delta \vdash M : \tau \backslash C$*.*

Proof. Similar to the proof for the previous lemma. ∎

The following lemma says that types of variables in the term type context can be generalised without affecting the type of a term.

Lemma 4.3 *If* $\Gamma[x \mapsto \sigma]; \Delta \vdash M : \tau\backslash C$ *and* $\sigma' \succ \sigma$, *then* $\Gamma[x \mapsto \sigma']; \Delta \vdash M : \tau\backslash C$.

PROOF. By induction on the height of the type derivation for M. See Wright and Felleisen's Lemma 4.6 [31]. ∎

In proofs for the Hindley-Milner polymorphic type discipline, a result saying that

$$\textit{if } \Gamma \vdash M : \tau \textit{ and } S \textit{ is a substitution then } S\Gamma \vdash M : S\tau$$

is essential to prove a *substitution lemma* which, in its turn, is used to prove type preservation ([28], [31]). The proof of such a lemma also requires a lemma saying that substitution commutes with instantiation:

$$\textit{if } \sigma \succ \tau \textit{ and } S \textit{ is a substitution then } S\sigma \succ S\tau.$$

We shall prove similar results for our type system in Lemmas 4.8 and 4.7 below. We start by proving that the truth of our escaping predicates is preserved by extended type substitutions.

Lemma 4.4 *If* $\mathbf{esc?}(\sigma)$ *and there is* $\tau\backslash C$ *such that* $\sigma \succ \tau\backslash C$ *and* ZC *is valid then* $\mathbf{esc?}(Z\,\sigma)$.

PROOF. See Appendix A.2. ∎

Lemma 4.5 *If* $\mathbf{esc?}(\tau\backslash C)$ *and* ZC *is valid then* $\mathbf{esc?}(Z\,(\tau\backslash C))$.

PROOF. Follows immediately from the previous lemma, considering $\tau\backslash C$ as a constrained type scheme σ with no bound type and locality variables. ∎

Lemma 4.6 *Suppose that* $\mathbf{esc?}(\Gamma, \Delta, M)$ *and let* Z *be an extended type substitution. If for all* $r \in \mathit{Refs}(M)$ *with* $\Delta(r) = \tau\,\mathtt{ref}^{\,\epsilon}\backslash C$ *we have* ZC *is valid, and if for all* $x \in FV(M)$ *there is* $\tau\backslash C$ *such that* $\Gamma(x) \succ \tau\backslash C$ *and* ZC *is valid, then* $\mathbf{esc?}(Z\Gamma, Z\Delta, M)$.

PROOF. From the definition of the 3-place predicate **esc?** (Definition 3.17) and Lemmas 4.4 and 4.5. ∎

Extended type substitution commutes with instantiation of polymorphic-consistent constrained type schemes:

Lemma 4.7 *If* σ *is polymorphic-consistent and* $\sigma \succ \tau\backslash C$, *then* $Z\sigma \succ Z(\tau\backslash C)$.

PROOF. Similar to the proof of Lemma 2.16 in [28]. ∎

In order to illustrate why such condition is necessary consider the following example. Consider the constrained type scheme $\forall\kappa.\, \alpha^\kappa \xrightarrow{\epsilon} \alpha^\kappa \backslash \emptyset$. One readily checks that

$$\forall\kappa.\, \alpha^\kappa \xrightarrow{\epsilon} \alpha^\kappa \backslash \emptyset \;\; \succ \;\; \alpha^\epsilon \xrightarrow{\epsilon} \alpha^\epsilon \backslash \emptyset.$$

Now let $Z = (\{\alpha \mapsto \texttt{int ref}^{\,\epsilon}\}, \{\})$. We have that

$$Z\ (\forall\kappa.\, \alpha^\kappa \xrightarrow{\epsilon} \alpha^\kappa \backslash \emptyset) \;=\; \forall\kappa'.\, \alpha^{\kappa'} \xrightarrow{\epsilon} \alpha^{\kappa'} \backslash \emptyset$$

and

$$Z\ (\alpha^\epsilon \xrightarrow{\epsilon} \alpha^\epsilon \backslash \emptyset) \;=\; \texttt{int ref}^{\,\epsilon} \xrightarrow{\epsilon} \texttt{int ref}^{\,\epsilon} \backslash \emptyset,$$

but

$$Z\,(\forall\kappa. \alpha^\kappa \xrightarrow{\epsilon} \alpha^\kappa \backslash \emptyset) \;\; \not\succ \;\; Z\,(\alpha^\epsilon \xrightarrow{\epsilon} \alpha^\epsilon \backslash \emptyset).$$

The reason why substitution fails to commute with the instantiation above is that the type scheme $\forall\kappa.\, \alpha^\kappa \xrightarrow{\epsilon} \alpha^\kappa \backslash \emptyset$ is not *polymorphic consistent*, that is, a bound locality variable (κ) labels a free type variable (α).

Having proved that the truth of the escaping predicates and that the instantiation relation are preserved by extended type substitution, our next step is to prove that type derivability is preserved by extended type substitutions. Before proceeding, we state another general assumption, this time about bound type and locality variables:

Remark 4.1 *We assume that, in a given context (e.g. type derivation, proof, substitution) bound type variables are different from any other type variables appearing in the same context. Similarly for bound locality variables.*

This is not a restrictive assumption since, as we have stated, we are equating α-equivalent constrained type schemes and we assume the existence of a countable infinite set of type and locality variables.

Lemma 4.8 (Type Substitution) *If*

1. $\Gamma, \Delta \vdash M : \tau \backslash C$,

2. Γ *is polymorphic-consistent, and*

3. ZC *is valid*

then $Z\Gamma, Z\Delta \vdash M : Z\,(\tau \backslash C)$.

Proof. By induction on the height of the type derivation of M. We present proof details only for the interesting cases.

case x

We have that $\Gamma;\Delta \vdash x : \tau\backslash C$. By the rule for typing variables we have $\Gamma(x) \succ \tau\backslash C$. By the assumption that Γ is polymorphic-consistent, $\Gamma(x)$ is polymorphic-consistent. By Lemma 4.7 we then get $Z\,(\Gamma(x)) \succ Z\,(\tau\backslash C)$, which, because substitutions are applied pointwise to typing contexts, is equivalent to

$$(Z\,\Gamma)(x) \succ Z\,(\tau\backslash C). \tag{4.1}$$

By assumption that ZC is valid we can use (4.1) as premise for the rule for typing variables and get

$$Z\Gamma; Z\Delta \vdash x\colon Z\,(\tau\backslash C).$$

case $\mathtt{fn}\,x.P$

The type derivation for this case must have been of this form

$$\frac{\Gamma[x \mapsto \tau_1\backslash C_1]; \Delta \vdash P\colon \tau_2\backslash C_2 \quad (*)}{\Gamma; \Delta \vdash \mathtt{fn}\,x.P\colon \tau_1 \xrightarrow{\phi} \tau_2\backslash C_1 \cup C_2}$$

where $(*)$ represents the possible use of the predicate $\mathbf{esc?}(\Gamma;\Delta, \mathtt{fn}\,x.P)$.

Since Γ is polymorphic-consistent, we have

$$\Gamma[x \mapsto \tau_1\backslash C_1] \textit{ is polymorphic-consistent.} \tag{4.2}$$

By the assumption that $Z\,(C_1 \cup C_2)$ is valid it follows that

$$ZC_2 \textit{ is valid.} \tag{4.3}$$

By the induction hypothesis with (4.2), (4.3) and the premise of the proof above we get

$$Z\,(\Gamma[x \mapsto \tau_1\backslash C_1]); Z\,\Delta \vdash P\colon Z\,(\tau_2\backslash C_2)$$

which is equivalent to

$$Z\Gamma[x \mapsto Z\tau_1\backslash ZC_1]; Z\,\Delta \vdash P\colon Z\tau_2\backslash ZC_2. \tag{4.4}$$

If ϕ in the type derivation above is ℓ, or ϕ is a locality variable κ such that $Z^{loc}(\kappa) = \ell$ then we use (4.4) as premise for the rule for typing local functions and we get

$$Z\Gamma; Z\Delta \vdash \mathtt{fn}\,x.P\colon Z\tau_1 \xrightarrow{\ell} Z\tau_2\backslash ZC_1 \cup ZC_2,$$

which is equivalent to

$$Z\Gamma; Z\Delta \vdash \mathtt{fn}\, x.P \colon Z\,(\tau_1 \xrightarrow{\phi} \tau_2 \backslash C_1 \cup C_2).$$

If $\phi = \epsilon$, or ϕ is a locality variable κ such that either $Z^{loc}(\kappa) = \epsilon$ or $Z^{loc}(\kappa)$ is another locality variable, we still have to prove that $\mathbf{esc?(}Z\Gamma,\ Z\Delta\mathbf{,}\mathtt{fn}\, x.P\mathbf{)}$. For these cases

$$\mathbf{esc?(}\Gamma, \Delta\mathbf{,}\mathtt{fn}\, x.P\mathbf{)} \tag{4.5}$$

must have held in the type derivation above.

If $y \in FV(\mathtt{fn}\, x.P)$ then $\Gamma'(y) \succ \tau \backslash C$, for some $\Gamma' \supseteq \Gamma$ and $\tau \backslash C$, must be at the leaves of any derivation tree for $\Gamma, \Delta \vdash \mathtt{fn}\, x.P \colon \tau_1 \xrightarrow{\phi} \tau_2 \backslash C_1 \cup C_2$. Constraints are accumulated in the root so we know that $C \subseteq C_1 \cup C_2$. By the assumption that $Z\,(C_1 \cup C_2)$ is valid we have that ZC is also valid. We also have that $\Gamma(y) = \Gamma'(y)$. Hence, the following holds

$$\textit{for any } y \in FV(\mathtt{fn}x.P)\ \exists\, \tau\backslash C \textit{ such that } \Gamma(y) \succ \tau\backslash C \textit{ and } ZC \textit{ is valid,} \tag{4.6}$$

and by a similar reasoning we also have that

$$\textit{for any } r \in \textit{Refs}\,(\mathtt{fn}\, x.P) \textit{ if } \Delta(r) = \tau\, \mathtt{ref}^{\,\epsilon}\backslash C \textit{ then } ZC \textit{ is valid.} \tag{4.7}$$

By Lemma 4.6 with (4.5), (4.6) and (4.7) we then get

$$\mathbf{esc?(}Z\Gamma, Z\Delta\mathbf{,}\mathtt{fn}\, x.P\mathbf{)}. \tag{4.8}$$

case $\mathtt{let}\ x = V\ \mathtt{in}\ P\ \mathtt{end}$

The proof derivation for this case must have been of this form

$$\frac{\Gamma; \Delta \vdash V \colon \tau' \backslash C' \qquad \Gamma[x \mapsto \textit{Gen}(\tau' \backslash C', \Gamma, \Delta)]; \Delta \vdash N \colon \tau \backslash C}{\Gamma; \Delta \vdash \mathtt{let}\ x = V\ \mathtt{in}\ N\ \mathtt{end} \colon \tau \backslash C \cup C'}$$

By the assumption that $Z\,(C \cup C')$ is valid we have that

$$ZC' \textit{ is valid,} \tag{4.9}$$

and

$$ZC \textit{ is valid} \tag{4.10}$$

By the induction hypothesis with (4.9) and the first premise of the proof above we get

$$Z\Gamma; Z\Delta \vdash V \colon Z(\tau' \backslash C') \tag{4.11}$$

By Lemma 3.2 and the assumption that Γ is polymorphic-consistent we have that $Gen(\tau'\backslash C', \Gamma, \Delta)$ is polymorphic-consistent. Hence,

$$\Gamma[x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)] \textit{ is polymorphic-consistent.} \tag{4.12}$$

By the induction hypothesis with (4.12), (4.10) and the second premise of the proof above we get

$$Z\Gamma[x \mapsto Z\, Gen(\tau'\backslash C', \Gamma, \Delta)]; Z\Delta \vdash N : Z\,(\tau\backslash C). \tag{4.13}$$

We have that

$$Z\, Gen(\tau'\backslash C', \Gamma, \Delta) \;=\; Gen(Z\,(\tau'\backslash C'), Z\Gamma, Z\Delta) \tag{4.14}$$

since

$$\begin{array}{cll} & Z\, Gen(\tau'\backslash C', \Gamma, \Delta) & \\ = & Z\,(\forall\vec{\alpha}\vec{\kappa}.\, \tau'\backslash C') & \\ = & \forall\vec{\alpha}\vec{\kappa}.\, Z\,(\tau'\backslash C') & \text{by Remark 4.1} \\ = & Gen(Z(\tau'\backslash C'), Z\Gamma, Z\Delta) & \end{array}$$

By (4.13) and (4.14) above we get

$$Z\Gamma[x \mapsto Gen(Z(\tau'\backslash C'), Z\Gamma, Z\Delta)]; Z\Delta \vdash N : Z\,(\tau\backslash C \cup C'). \tag{4.15}$$

We then use (4.11) and (4.15) as premises for the rule for typing let expressions to get

$$Z\Gamma; Z\Delta \vdash \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : Z\,(\tau\backslash C \cup C')$$

as desired. ∎

A substitution lemma is used when proving type preservation for the evaluation rules which involve the substitution of a value for a variable in a term. In their proofs of Subject Reduction (Type Preservation), Wright and Felleisen are able to do with a single Substitution Lemma (Lemma 4.4 in [31]). In their β transition the type of the variable is the same as the type of the value. In the rule for let expressions the type of the variable is a generalisation of the type of the value. For this reason only one lemma, assuming a quantified type for the variable, is necessary. When the lemma is used in the proof of type preservation for the β transition the type of the variable is treated as a quantified type with no bound type variables.

Our introduction of constrained types though requires two such lemmas. The crucial difference here is that the constraint sets in the constrained type of the variable and in the constrained type of the value are not known to be the same. A single lemma is then not possible: a quantified version of the type of the variable does not correspond to a generalisation of the value's type.

We now present the first of our substitution lemmas. Before we give an auxiliary lemma which can be seen as a *substitution* lemma for the predicate **esc?**.

Lemma 4.9 *If* $\mathbf{esc?}(\Gamma;\Delta,M)$ *and* $\mathbf{esc?}(\Gamma;\Delta,V)$ *then* $\mathbf{esc?}(\Gamma;\Delta,M[V/x])$.

PROOF. By Definition 3.17 and the facts that $FV(M[V/x]) \subseteq FV(M) \cup FV(V)$ and $\mathit{Refs}(M[V/x]) \subseteq \mathit{Refs}(M) \cup \mathit{Refs}(V)$. ∎

Lemma 4.10 (Polymorphic Substitution) *If*

- Γ *is polymorphic-consistent,*
- $\Gamma[x \mapsto \mathit{Gen}(\tau'\backslash C', \Gamma, \Delta)]; \Delta \vdash M : \tau\backslash C$
- $x \notin \mathit{Dom}(\Gamma)$,
- $\Gamma; \Delta \vdash V : \tau'\backslash C'$.

then $\Gamma; \Delta \vdash M[V/x] : \tau\backslash C$.

PROOF. By induction on the height of the typing derivation for M.

case r

We have $\Gamma[x \mapsto \mathit{Gen}(\tau'\backslash C', \Gamma, \Delta)]; \Delta \vdash r : \tau\backslash C$. Then, as the type of r does not depend on the type of x, we have $\Gamma; \Delta \vdash r : \tau\backslash C$, and as $r = r[V/x]$ we have $\Gamma; \Delta \vdash r[V/x] : \tau\backslash C$.

cases c and y, where $y \neq x$

Similar to the previous case.

case x

Let $\forall\vec{\alpha}\vec{\kappa}.\tau'\backslash C' = \mathit{Gen}(\tau'\backslash C', \Gamma, \Delta)$ with $\lceil\vec{\alpha}\rceil = FTV(\tau') \setminus FTV(\Gamma, \Delta)$ and $\lceil\vec{\kappa}\rceil = FLV(\tau') \setminus FLV(\Gamma, \Delta)$. By the typing rule for variables, $\forall\vec{\alpha}\vec{\kappa}.\tau'\backslash C' \succ \tau\backslash C$. By definition of instantiation (Definition 3.14), there exists $Z = (Z^{type}, Z^{loc})$ such that $\mathit{Dom}(Z^{type}) = \lceil\vec{\alpha}\rceil$, $\mathit{Dom}(Z^{loc}) = \lceil\vec{\kappa}\rceil$ and $Z(\tau'\backslash C') = \tau\backslash C$. As C is valid by assumption it must be that

$$ZC' \text{ is valid.} \tag{4.16}$$

By Lemma 4.8 with the assumption that Γ is polymorphic-consistent and (4.16) we get

$$Z\Gamma; Z\Delta \vdash V : Z(\tau' \backslash C').$$

By the definition of the domains of Z^{loc} and Z^{type} we have that $Z\Gamma = \Gamma$ and $Z\Delta = \Delta$. Then the type judgement above is equivalent to

$$\Gamma; \Delta \vdash V : \tau \backslash C,$$

and since $V = x[V/x]$, we have

$$\Gamma; \Delta \vdash x[V/x] : \tau \backslash C.$$

case $\texttt{fn}\, x'.P$

For some τ_1, τ_2, and some ϕ, we have a type derivation of the form

$$\frac{\Gamma[x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma, \Delta)][x' \mapsto \tau_1 \backslash C_1]; \Delta \vdash P : \tau_2 \backslash C_2 \quad (*)}{\Gamma[x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma, \Delta)]; \Delta \vdash \texttt{fn}\, x'.P : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C_1 \cup C_2}$$

where $(*)$ represents the possible use of the predicate

$$\textbf{esc?}(\Gamma[x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma, \Delta)]; \Delta, \texttt{fn}\, x'.P). \tag{4.17}$$

Changing the order of the type assumptions we rewrite the first premise of the proof above as

$$\Gamma[x' \mapsto \tau_1 \backslash C_1][x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma, \Delta)]; \Delta \vdash P : \tau_2 \backslash C_2. \tag{4.18}$$

By Remark 4.1, we have $\mathit{FLV}(\tau_1 \backslash C_1) \cap (\mathit{FLV}(\tau') \setminus \mathit{FLV}(\Gamma, \Delta)) = \emptyset$ and $\mathit{FTV}(\tau_1 \backslash C_1) \cap (\mathit{FTV}(\tau') \setminus \mathit{FTV}(\Gamma, \Delta)) = \emptyset$. Hence

$$\mathit{Gen}(\tau' \backslash C', \Gamma, \Delta) = \mathit{Gen}(\tau' \backslash C', \Gamma[x' \mapsto \tau_1 \backslash C_1], \Delta),$$

and we can then write (4.18) as

$$\Gamma[x' \mapsto \tau_1 \backslash C_1][x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma[x' \mapsto \tau_1 \backslash C_1], \Delta)]; \Delta \vdash P : \tau_2 \backslash C_2. \tag{4.19}$$

Since Γ is polymorphic-consistent

$$\Gamma[x' \mapsto \tau_1 \backslash C_1] \textit{ is polymorphic consistent.} \tag{4.20}$$

By the variable convention, because x' is bound in $\texttt{fn}\, x'.P$ we have that

$$x' \notin FV(V), \tag{4.21}$$

and also

$$x \neq x'. \tag{4.22}$$

By the assumption that $x \notin Dom(\Gamma)$ and (4.22) we have that

$$x \notin Dom(\Gamma[x' \mapsto \tau_1 \backslash C_1]). \tag{4.23}$$

By Lemma 4.2 and (4.21) we may weaken Γ in the assumption $\Gamma; \Delta \vdash V : \tau' \backslash C'$ to get

$$\Gamma[x' \mapsto \tau_1 \backslash C_1]; \Delta \vdash V : \tau' \backslash C'. \tag{4.24}$$

By the induction hypothesis with (4.19), (4.20), (4.23) and (4.24) above, we get

$$\Gamma[x' \mapsto \tau_1 \backslash C_1]; \Delta \vdash P[V/x] : \tau_2 \backslash C_2.$$

We now have to consider the locality label ϕ associated with the function type. Suppose $\phi = \ell$. Then by the first type rule for abstractions using the type judgement above as premise we get

$$\Gamma; \Delta \vdash \mathtt{fn}\, x'.P[V/x] : \tau_1 \xrightarrow{\ell} \tau_2 \backslash C_1 \cup C_2.$$

By (4.21) and (4.22) the type judgement above is equivalent to

$$\Gamma; \Delta \vdash (\mathtt{fn}\, x'.P)[V/x] : \tau_1 \xrightarrow{\ell} \tau_2 \backslash C_1 \cup C_2.$$

Suppose instead that $\phi = \epsilon$ or $\phi = \kappa$. We have to prove that the predicate **esc?**$(\Gamma, \Delta, (\mathtt{fn}\, x'.P)[V/x])$ holds. For this subcase 4.17 must have held in the type derivation.

If $x \notin FV(\mathtt{fn}\, x'.P)$ then by Lemma 4.2 and (4.17) above we also have that **esc?**$(\Gamma; \Delta, \mathtt{fn}\, x'.P)$, and because in this case $\mathtt{fn}\, x'.P = (\mathtt{fn}\, x'.P)[V/x]$ we also have **esc?**$(\Gamma; \Delta, (\mathtt{fn}\, x'.P)[V/x])$.

On the other hand, if $x \in FV(\mathtt{fn}\, x'.P)$ then by Definition 3.17 and (4.17) above there is $\tau'' \backslash C''$, such that $Gen(\tau' \backslash C', \Gamma, \Delta) \succ \tau'' \backslash C''$ with C'' valid and **esc?**(τ''). By Definition 3.14 there is a suitable Z such that $Z(\tau' \backslash C') = \tau'' \backslash C''$.

By Lemma 4.8 and the assumption $\Gamma; \Delta \vdash V : \tau' \backslash C'$ we get

$$Z\Gamma; Z\Delta \vdash V : Z(\tau' \backslash C'),$$

By the way Z is defined, we have $Z\Gamma = \Gamma$ and $Z\Delta = \Delta$. The type judgement above is then the same as

$$\Gamma; \Delta \vdash V : \tau'' \backslash C''.$$

Because $\mathbf{esc?}(\tau'')$, by Lemma 3.7 we get that $\mathbf{esc?}(\Gamma;\Delta, V)$. By the assumptions that $x \notin Dom(\Gamma)$ and $\Gamma;\Delta \vdash V : \tau'\backslash C'$, we have that $x \notin FV(V)$, by Lemma 4.2 we then get

$$\mathbf{esc?}(\Gamma[x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)];\Delta, V). \tag{4.25}$$

By (4.17) and (4.25) above and by Lemma 4.9 we have

$$\mathbf{esc?}(\Gamma[x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)];\Delta, (\texttt{fn}\, x'.P)[V/x]).$$

As $x \notin FV((\texttt{fn}\, x'.P)[V/x])$, by Lemma 4.2 we finally conclude

$$\mathbf{esc?}(\Gamma;\Delta, (\texttt{fn}\, x'.P)[V/x]).$$

case $\texttt{rec}\, f\, x'.P$

Similar to the case for function abstractions.

case (P, Q)

Suppose $\Gamma' = \Gamma[x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)]$. We then have a proof of the form

$$\frac{\Gamma';\Delta \vdash P : \tau_1\backslash C_1 \qquad \Gamma';\Delta \vdash Q : \tau_2\backslash C_2}{\Gamma';\Delta \vdash (P, Q) : \tau_1 \times^{\phi} \tau_2\backslash C_1 \cup C_2 \cup C_3 \cup C_4}$$

where :
$C_3 = \{\phi \leq \mathbf{lab}\ (\tau_1)\},\ \textit{if } \mathbf{lab}\ (\tau_1)\ \textit{is defined},\ C_3 = \emptyset\ \textit{otherwise}$
$C_4 = \{\phi \leq \mathbf{lab}\ (\tau_2)\},\ \textit{if } \mathbf{lab}\ (\tau_2)\ \textit{is defined},\ C_4 = \emptyset\ \textit{otherwise}$

Applying the induction hypothesis to the premises in the proof above, we have $\Gamma;\Delta \vdash P[V/x] : \tau_1\backslash C_1$, and $\Gamma;\Delta \vdash Q[V/x] : \tau_2\backslash C_2$. By the rule for pairs with these two type judgements as premises we conclude

$$\Gamma;\Delta \vdash (P[V/x], Q[V/x]) : \tau_1 \times^{\phi} \tau_2\backslash C_1 \cup C_2 \cup C_3 \cup C_4,$$

which is the same as $\Gamma;\Delta \vdash (P, Q)[V/x] : \tau_1 \times^{\phi} \tau_2\backslash C_1 \cup C_2 \cup C_3 \cup C_4$.

cases $P\ Q$ and $\texttt{if}\ N\ \texttt{then}\ P\ \texttt{else}\ Q$

Similar to the previous case.

case $\texttt{let}\ x' = V'\ \texttt{in}\ Q\ \texttt{end}$

The derivation of the type judgement is of the form

$$\frac{\Gamma';\Delta \vdash V' : \tau''\backslash C'' \quad \Gamma'[x' \mapsto \sigma];\Delta \vdash Q : \tau\backslash C'''}{\Gamma';\Delta \vdash \texttt{let}\ x' = V'\ \texttt{in}\ Q\ \texttt{end} : \tau\backslash C'' \cup C'''}$$

where $\Gamma' = \Gamma[x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)]$ and $\sigma = Gen(\tau''\backslash C'', \Gamma', \Delta)$.

By the induction hypothesis and the first premise above

$$\Gamma; \Delta \vdash V'[V/x] : \tau''\backslash C''. \tag{4.26}$$

We may write the second premise in the proof above as

$$\Gamma[x' \mapsto \sigma][x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)]; \Delta \vdash Q : \tau\backslash C'''. \tag{4.27}$$

Now we have

$$FLV(\sigma) \subseteq FLV(\Gamma, \Delta) \tag{4.28}$$

since

$$\begin{array}{ll} & FLV(\sigma) \\ = & FLV(\tau'') \cap FLV(\Gamma[x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)], \Delta) \\ = & FLV(\tau'') \cap (FLV(\Gamma) \cup FLV(Gen(\tau'\backslash C', \Gamma, \Delta)) \cup FLV(\Delta)) \\ \subseteq & FLV(\Gamma) \cup FLV(Gen(\tau'\backslash C', \Gamma, \Delta)) \cup FLV(\Delta) \\ = & FLV(\Gamma) \cup (FLV(\tau') \cap FLV(\Gamma, \Delta)) \cup FLV(\Delta) \\ \subseteq & FLV(\Gamma, \Delta). \end{array}$$

With a similar derivation we can also prove that

$$FTV(\sigma) \subseteq FTV(\Gamma, \Delta). \tag{4.29}$$

Hence, with (4.28) and (4.29) we get

$$Gen(\tau'\backslash C', \Gamma[x' \mapsto \sigma], \Delta) \succ Gen(\tau'\backslash C', \Gamma, \Delta). \tag{4.30}$$

By Lemma 4.3 with (4.27) and (4.30) we get

$$\Gamma[x' \mapsto \sigma][x \mapsto Gen(\tau'\backslash C', \Gamma[x' \mapsto \sigma], \Delta)]; \Delta \vdash Q : \tau\backslash C'''. \tag{4.31}$$

Since x' is `let`-bound, by the variable convention we have that

$$x' \notin FV(V), \tag{4.32}$$

and also

$$x' \neq x. \tag{4.33}$$

By Lemma 4.2 and (4.32), we may weaken Γ in the assumption $\Gamma; \Delta \vdash V : \tau'\backslash C'$ to get:

$$\Gamma[x' \mapsto \sigma]; \Delta \vdash V : \tau'\backslash C'. \tag{4.34}$$

By (4.33) and the assumption that $x \notin Dom(\Gamma)$ we have that

$$x \notin Dom(\Gamma[x' \mapsto \sigma]). \tag{4.35}$$

By assumption that Γ is polymorphic consistent and Lemma 3.2 we have that $Gen(\tau'\backslash C', \Gamma, \Delta)$ is polymorphic consistent. Hence, because $\Gamma[x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)]$ is polymorphic consistent and Lemma 3.2 we have that σ is polymorphic consistent, consequently

$$\Gamma[x \mapsto \sigma] \textit{ is polymorphic consistent.} \tag{4.36}$$

By (4.31), (4.36), (4.34), (4.35) and by the induction hypothesis we get

$$\Gamma[x' \mapsto \sigma]; \Delta \vdash Q[V/x] : \tau\backslash C'''. \tag{4.37}$$

By assumption, $x \notin Dom(\Gamma)$, so

$$Gen(\tau''\backslash C'', \Gamma, \Delta) \succ Gen(\tau''\backslash C'', \Gamma[x \mapsto Gen(\tau'\backslash C', \Gamma, \Delta)], \Delta). \tag{4.38}$$

Hence by Lemma 4.3, with (4.37) and (4.38)

$$\Gamma[x' \mapsto Gen(\tau''\backslash C'', \Gamma, \Delta)]; \Delta \vdash Q[V/x] : \tau\backslash C'''. \tag{4.39}$$

By the first typing rule for `let` using (4.26) and (4.39) above as premises we have

$$\Gamma; \Delta \vdash \texttt{let } x' = V'[V/x] \texttt{ in } Q[V/x] \texttt{ end} : \tau\backslash C'' \cup C'''.$$

By (4.32) and (4.33) the judgement above is equivalent to

$$\Gamma; \Delta \vdash (\texttt{let } x' = V' \texttt{ in } Q)[V/x] \texttt{ end} : \tau\backslash C'' \cup C'''.$$

case `let` $x' = Q$ `in` P `end`, where Q not a value
Similar to the case for functions. ∎

Unlike the Polymorphic Substitution lemma, our other substitution lemma assumes an unquantified constrained type scheme for the variable x to be substituted.

Before proving the ordinary substitution lemma below we need the following two auxiliary lemmas.

Lemma 4.11 *If* $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\tau\backslash C]; \Delta \vdash M : \tau'\backslash C'$, $C \cup C''$ *is valid and* $FLV(C'') \cap \lceil\vec{\kappa}\rceil = \emptyset$ *then* $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\tau\backslash C \cup C'']; \Delta \vdash M : \tau'\backslash C'''$, *for some* $C''' \subseteq C' \cup C''$.

Proof. By induction on the height of the type derivation (see Appendix A.2). ∎

Lemma 4.12 *If* $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\tau \setminus C]; \Delta \vdash M : \tau' \setminus C'$ *and* $C'' \subseteq C$ *then* $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\tau \setminus C'']; \Delta \vdash M : \tau' \setminus C'''$, *for some* $C''' \subseteq C'$.

Proof. By induction on the height of the type derivation (see Appendix A.2). ∎

Lemma 4.13 (Ordinary Substitution) *If*

- $\Gamma[x \mapsto \tau' \setminus C']; \Delta \vdash M : \tau \setminus C$ *with* C' *valid,*
- $x \notin Dom(\Gamma)$ *and*
- $\Gamma; \Delta \vdash V : \tau' \setminus C''$

then $\Gamma; \Delta \vdash M[V/x] : \tau \setminus C'''$, *for some* $C''' \subseteq C \cup C''$.

Proof. By induction on the height of the typing derivation for M.

case r

We have $\Gamma[x \mapsto \tau' \setminus C']; \Delta \vdash r : \tau \setminus C$. The type of r does not depend on the type of x. By Lemma 4.2 we then conclude $\Gamma; \Delta \vdash r : \tau \setminus C$ which, as $r = r[V/x]$, is the same as $\Gamma; \Delta \vdash r[V/x] : \tau \setminus C$. Note that $C \subseteq C \cup C''$.

cases c and y, where $y \neq x$

Similar to the previous case.

case x

By the assumption that C' is valid we have $\Gamma[x \mapsto \tau' \setminus C']; \Delta \vdash x : \tau' \setminus C'$. By the assumption $\Gamma; \Delta \vdash V : \tau' \setminus C''$ and as $V = x[V/x]$, we get $\Gamma; \Delta \vdash x[V/x] : \tau' \setminus C''$. Note that $C'' \subseteq C' \cup C''$.

case $\mathtt{fn}\, x'.P$

We have a proof of the form

$$\frac{\Gamma[x \mapsto \tau' \setminus C'][x' \mapsto \tau_1 \setminus C_1]; \Delta \vdash P : \tau_2 \setminus C_2 \qquad (*)}{\Gamma[x \mapsto \tau' \setminus C']; \Delta \vdash \mathtt{fn}\, x'.P : \tau_1 \xrightarrow{\phi} \tau_2 \setminus C_1 \cup C_2}$$

where the possible use of the predicate **esc?**$(\Gamma[x \mapsto \tau' \setminus C']; \Delta, \mathtt{fn}\, x'.P)$ is marked by $(*)$.

We may write the premise of the proof above as

$$\Gamma[x' \mapsto \tau_1 \setminus C_1][x \mapsto \tau' \setminus C']; \Delta \vdash P : \tau_2 \setminus C_2. \tag{4.40}$$

By the variable convention, as x' is bound in $\mathtt{fn}\, x'.P$ we have that

$$x' \notin FV(V), \tag{4.41}$$

and

$$x' \neq x. \tag{4.42}$$

By (4.41) can weaken Γ in the assumption $\Gamma;\Delta \vdash V : \tau'\backslash C''$ and get

$$\Gamma[x' \mapsto \tau_1\backslash C_1];\Delta \vdash V : \tau'\backslash C''. \tag{4.43}$$

By the assumption that $x \notin Dom(\Gamma)$ and by (4.42) we have that

$$x \notin Dom(\Gamma[x' \mapsto \tau_1\backslash C_1]). \tag{4.44}$$

By the induction hypothesis with (4.40), (4.44) and (4.43) we get

$$\Gamma[x' \mapsto \tau_1\backslash C_1];\Delta \vdash P[V/x] : \tau_2\backslash C_3 \;\text{ where } C_3 \subseteq C_2 \cup C''. \tag{4.45}$$

Constraint sets C_1 and C_3 are valid, so by Lemma 3.1 we have that $C_1 \cup C_3$ is also valid. Suppose the locality label ϕ from the proof above is ℓ. Then by the first typing rule for function abstractions using (4.45) above we get

$$\Gamma;\Delta \vdash \mathtt{fn}\, x'.P[V/x] : \tau_1 \xrightarrow{\ell} \tau_2\backslash C_1 \cup C_3.$$

By (4.41) and (4.42) the type judgement above is equivalent to

$$\Gamma;\Delta \vdash (\mathtt{fn}\, x'.P)[V/x] : \tau_1 \xrightarrow{\ell} \tau_2\backslash C_1 \cup C_3.$$

Note that

$$\begin{array}{rcll} C_1 \cup C_3 & \subseteq & C_1 \cup (C_2 \cup C'') & \text{by (4.45)} \\ & = & (C_1 \cup C_2) \cup C''. & \end{array}$$

Suppose instead that ϕ is ϵ or a locality variable. Then we must also show that $\mathbf{esc?}(\Gamma,\Delta,(\mathtt{fn}\, x'.P)[V/x])$.

For this subcase

$$\mathbf{esc?}(\Gamma[x \mapsto \tau'\backslash C'];\Delta,\mathtt{fn}\, x'.P) \tag{4.46}$$

must have held in the type derivation.

If $x \notin FV(\mathtt{fn}\, x'.P)$ by Lemma 4.2 and (4.46) we have $\mathbf{esc?}(\Gamma;\Delta,\mathtt{fn}\, x'.P)$ and because $\mathtt{fn}\, x'.P = (\mathtt{fn}\, x'.P)[V/x]$ we get $\mathbf{esc?}(\Gamma;\Delta,(\mathtt{fn}\, x'.P)[V/x])$.

Suppose $x \in FV(\mathtt{fn}\, x'.P)$. By (4.46) and Definition 3.17 we have that $\mathbf{esc?}(\tau')$. By the assumption $\Gamma;\Delta \vdash V : \tau'\backslash C''$ and Lemma 3.7 we get that $\mathbf{esc?}(\Gamma;\Delta,V)$. By the assumptions that $x \notin Dom(\Gamma)$ and $\Gamma;\Delta \vdash V : \tau'\backslash C''$ we know that $x \notin FV(V)$. Then, by Lemma 4.2 we get

$$\mathbf{esc?}(\Gamma[x \mapsto \tau'\backslash C'];\Delta,V). \tag{4.47}$$

By (4.46) and (4.47) above and by Lemma 4.9 we get that

$$\textbf{esc?(}\Gamma[x \mapsto \tau'\backslash C']; \Delta\textbf{,}(\texttt{fn}\, x'.P)[V/x]\textbf{)}$$

and, as $x \notin FV((\texttt{fn}\, x'.P)[V/x])$, by Lemma 4.2 we finally conclude

$$\textbf{esc?(}\Gamma; \Delta\textbf{,}(\texttt{fn}\, x'.P)[V/x]\textbf{)}.$$

case $\texttt{rec}\, f\, x.P$

Similar to the previous case.

case $\texttt{(}P, Q\texttt{)}$

The proof of the type judgement must have been of the form

$$\frac{\Gamma[x \mapsto \tau'\backslash C']; \Delta \vdash P\colon \tau_1\backslash C_1 \quad \Gamma[x \mapsto \tau'\backslash C']; \Delta \vdash Q\colon \tau_2\backslash C_2}{\Gamma[x \mapsto \tau'\backslash C']; \Delta \vdash \texttt{(}P, Q\texttt{)}\colon \tau_1 \times^{\phi} \tau_2\backslash C_1 \cup C_2 \cup C_3 \cup C_4}$$

where :

$C_3 = \{\phi \leq \textbf{lab}\ (\tau_1)\}$, *if* $\textbf{lab}\ (\tau_1)$ *is defined*, $C_3 = \emptyset$ *otherwise*

$C_4 = \{\phi \leq \textbf{lab}\ (\tau_2)\}$, *if* $\textbf{lab}\ (\tau_2)$ *is defined*, $C_4 = \emptyset$ *otherwise*

By the induction hypothesis and the first and second premises above respectively we get

$$\Gamma; \Delta \vdash P[V/x]\colon \tau_1\backslash C_1''' \text{ where } C_1''' \subseteq C_1 \cup C'' \tag{4.48}$$

and

$$\Gamma; \Delta \vdash Q[V/x]\colon \tau_2\backslash C_2''' \text{ where } C_2''' \subseteq C_2 \cup C''. \tag{4.49}$$

By Lemma 3.1 we have that $C_2''' \cup C_1''' \cup C_3 \cup C_4$ is valid. Then by the typing rule for pairs using (4.48) and (4.49) as premises we have that $\Gamma; \Delta \vdash \texttt{(}P[V/x], Q[V/x]\texttt{)}\colon \tau_1 \times^{\phi} \tau_2\backslash C_1''' \cup C_2''' \cup C_3 \cup C_4$ which is equivalent to

$$\Gamma; \Delta \vdash \texttt{(}P, Q\texttt{)}[V/x]\colon \tau_1 \times^{\phi} \tau_2\backslash C_1''' \cup C_2''' \cup C_3 \cup C_4.$$

Note that

$$\begin{array}{ll} & C_1''' \cup C_2''' \cup C_3 \cup C_4 \\ \subseteq & (C_1 \cup C'') \cup (C_2 \cup C'') \cup C_3 \cup C_4 \quad \text{by (4.48) and (4.49)} \\ = & (C_1 \cup C_2 \cup C_3 \cup C_4) \cup C''. \end{array}$$

cases $P\ Q$ and $\texttt{if}\ N\, \texttt{then}\ P\, \texttt{else}\ Q$

Similar to the preceding case.

case $\texttt{let}\ x' = V'\ \texttt{in}\ Q\, \texttt{end}$

We have a proof of the form

$$\frac{\Gamma[x \mapsto \tau'\backslash C']; \Delta \vdash V'\colon \tau_4\backslash C_4 \quad \Gamma[x \mapsto \tau'\backslash C'][x' \mapsto \sigma]; \Delta \vdash Q\colon \tau\backslash C_5}{\Gamma[x \mapsto \tau'\backslash C']; \Delta \vdash \texttt{let}\ x' = V'\ \texttt{in}\ Q\, \texttt{end}\colon \tau\backslash C_4 \cup C_5}$$

where $\sigma = Gen(\tau_4 \backslash C_4, \Gamma[x \mapsto \tau' \backslash C'], \Delta)$.

By the induction hypothesis and the first premise above

$$\Gamma; \Delta \vdash V'[V/x] \colon \tau_4 \backslash C_4', \tag{4.50}$$

where

$$C_4' \subseteq C_4 \cup C''. \tag{4.51}$$

Changing the order of the type assumptions we may write the second premise of the proof above as

$$\Gamma[x' \mapsto Gen(\tau_4 \backslash C_4, \Gamma[x \mapsto \tau' \backslash C'], \Delta)][x \mapsto \tau' \backslash C']; \Delta \vdash Q \colon \tau \backslash C_5. \tag{4.52}$$

By assumption, $x \notin Dom(\Gamma)$, so

$$Gen(\tau_4 \backslash C_4, \Gamma, \Delta) \succ Gen(\tau_4 \backslash C_4, \Gamma[x \mapsto \tau' \backslash C'], \Delta). \tag{4.53}$$

By Lemma 4.3 with (4.52) and (4.53) above we get

$$\Gamma[x' \mapsto Gen(\tau_4 \backslash C_4, \Gamma, \Delta)][x \mapsto \tau' \backslash C']; \Delta \vdash Q \colon \tau \backslash C_5. \tag{4.54}$$

Hence by Lemma 4.11, the general assumption about bound locality variables (Remark 4.1) and the type judgement above we get

$$\Gamma[x' \mapsto Gen(\tau_4 \backslash C_4 \cup C'', \Gamma, \Delta)][x \mapsto \tau' \backslash C']; \Delta \vdash Q \colon \tau \backslash C_6, \tag{4.55}$$

where

$$C_6 \subseteq C_5 \cup C''. \tag{4.56}$$

By Lemma 4.12 with (4.51) and (4.55) we have

$$\Gamma[x' \mapsto Gen(\tau_4 \backslash C_4', \Gamma, \Delta)][x \mapsto \tau' \backslash C']; \Delta \vdash Q \colon \tau \backslash C_7, \tag{4.57}$$

where

$$C_7 \subseteq C_6. \tag{4.58}$$

Since x' is `let`-bound, by the variable convention, it does not occur in V. By Lemma 4.2, we may weaken Γ in the assumption $\Gamma; \Delta \vdash V : \tau' \backslash C''$ to get:

$$\Gamma[x' \mapsto Gen(\tau_4 \backslash C_4', \Gamma, \Delta)]; \Delta \vdash V \colon \tau' \backslash C''. \tag{4.59}$$

By the induction hypothesis with (4.57) and (4.59) above we get

$$\Gamma[x' \mapsto Gen(\tau_4 \backslash C_4', \Gamma, \Delta)]; \Delta \vdash Q[V/x] \colon \tau \backslash C_8, \tag{4.60}$$

where

$$C_8 \subseteq C_7 \cup C''. \tag{4.61}$$

By the first typing rule for `let` using (4.50) and (4.60) above as premises we get

$$\Gamma; \Delta \vdash \texttt{let}\ x' = V'[V/x]\ \texttt{in}\ Q[V/x] : \tau \backslash C'_4 \cup C_8.$$

By the variable convention, $x' \neq x$ and $x' \notin FV(V)$. The type judgement above is then equivalent to

$$\Gamma; \Delta \vdash (\texttt{let}\ x' = V'\ \texttt{in}\ Q)[V/x] : \tau \backslash C'_4 \cup C_8.$$

Note that

$$\begin{array}{rcll} C'_4 \cup C_8 & \subseteq & (C_4 \cup C'') \cup (C_7 \cup C'') & (4.51) \text{ and } (4.61) \\ & \subseteq & (C_4 \cup C'') \cup (C_6 \cup C'') & (4.58) \\ & \subseteq & (C_4 \cup C'') \cup ((C_5 \cup C'') \cup C'') & (4.56) \\ & = & (C_4 \cup C_5) \cup C''. & \end{array}$$

case `let` $x' = P$ `in` Q `end`

Similar to the previous case. ∎

Lemma 4.14 (Context-filler Type) *If $\Gamma; \Delta \vdash E[M] : \tau \backslash C$, then there exist τ' and C' such that $\Gamma; \Delta \vdash M : \tau' \backslash C'$.*

Proof. By the construction of derivations of type judgements.

Since M is a subterm of $E[M]$, the proof that $\Gamma; \Delta \vdash E[M] : \tau \backslash C$ must contain a subproof assigning a constrained type to the occurrence of M in the context hole. We want to show that in that subproof, the term type context is Γ and the store type context is Δ. By an easy induction on the height of type judgement proofs, the store type context at any subproof is the same as that at the root. Term type contexts may be extended at subproofs, though. The only instances in the typing rules where term type contexts are extended are when typing the bodies of functions and the bodies of `let`'s. By the grammar for contexts, the hole in $E[\,]$ can occur neither in a function body nor in the body of a `let`. So the term type context for typing the occurrence of M in the context hole must also be Γ. ∎

Lemma 4.15 (Derivation Pasting) *Suppose* $\Gamma; \Delta \vdash E[M] : \tau \backslash C$, *and the proof of this type judgement contains the subproof* $\Gamma; \Delta \vdash M : \tau' \backslash C'$ *for the occurrence of* M *at the location of the context hole. Then if* $\Gamma; \Delta \vdash N : \tau' \backslash C''$, *there exists* C''' *such that* $\Gamma; \Delta \vdash E[N] : \tau \backslash C'''$.

PROOF. By replacement of the subproof giving the constrained type of M at the context hole with the proof giving the constrained type of N, and the facts that constraints accumulate towards the root of typing derivations and that the validity of constraint sets is closed under union according to Lemma 3.1. ∎

A similar lemma appears in [31], based on the lemma in [16, p. 181]. We extend the result to handle constrained types.

There is a small subtlety here not found in the other versions of the lemma mentioned above. Suppose the replaced subproof involving M contains a use of the 3-place version of **esc?**. How do we know that the predicate holds when typing N? By inspection of the typing rules, the 3-place version of **esc?** is used only when typing function bodies. Context holes do not occur in function bodies; hence the predicate is not used when typing M.

The next definition states the requirements for a store typing context to be a typing for the references in the domain of a store.

Definition 4.1 *A store type context* Δ respects *a store* S *given a term context* Γ *iff*

- $Dom(S) \subseteq Dom(\Delta)$,
- *for all* $r \in Dom(S)$, *there exists* τ *such that*
  1. $\Gamma; \Delta \vdash S(r) : \tau \backslash C$ *for some constraint set* C, *and*
  2. $\Delta(r) = \tau\, \mathtt{ref}^{\,\phi} \backslash C'$ *where* C' *is valid and, if* **lab** (τ) *is defined, then* $\{\phi \leq \mathbf{lab}\ (\tau)\}$ *is in the transitive closure of* C'.

The next theorem says that types are preserved by sequential evaluation. We then use this result to prove type preservation for concurrent evaluation. Observe that the theorem below says that only the *type* part of constrained types are preserved by evaluation. The sets of constraints remain valid but they can be different. There is no specific relation between the sets of constraints because the rule Deref introduces a value in the right-hand side of the transition, and the constraint set of its constrained type has no specific relation with the constraint set of the reference in the left-hand side of the transition. We hope this point will become clearer when looking at the proof of the theorem below, in particular to the proof case for the Deref transition.

We say that Δ' *extends* Δ iff $Dom(\Delta') \supseteq Dom(\Delta)$ and for all $r \in Dom(\Delta)$, $\Delta'(r) = \Delta(r)$; we write $\Delta' \supseteq \Delta$ to indicate this relation.

Theorem 4.1 (Sequential Type Preservation) *Let S be a store and Δ a store type context respecting S given Γ a polymorphic-consistent term type context. Suppose $\Gamma; \Delta \vdash M : \tau \backslash C$ and suppose also that $S, M \underset{\mathsf{seq}}{\longrightarrow}^* S', M'$. Then there exist Δ' and C' such that*

- $\Delta' \supseteq \Delta$,
- Δ' *respects* S' *given* Γ *and*
- $\Gamma; \Delta' \vdash M' : \tau \backslash C'$.

Proof. By induction on the length of sequential evaluation.

If $\underset{\mathsf{seq}}{\longrightarrow}^*$ represents zero evaluation steps, the results hold by assumption. For the induction, consider the last evaluation step. Suppose $S, M \underset{\mathsf{seq}}{\longrightarrow}^* S'', M''$ and let the last step be $S'', M'' \underset{\mathsf{seq}}{\longrightarrow} S', M'$. The term M'' is not a value so, by Lemma 2.1, it can be uniquely represented as $E[N'']$ where N'' is a redex. We then consider transitions of the form $S'', E[N''] \underset{\mathsf{seq}}{\longrightarrow} S', E[N']$, where $E[N''] = M''$ and $E[N'] = M'$. By the induction hypothesis, there are Δ'' and C'' such that $\Delta'' \supseteq \Delta$, Δ'' respects S'' given Γ, and $\Gamma; \Delta'' \vdash E[N''] : \tau \backslash C''$.

For each transition, we have to produce a Δ', and show that it has the stated properties. Using that Δ' we need to find a type derivation for M'.

In all transitions where $S' = S''$ we choose Δ' to be Δ''. For these cases the facts about Δ' follow from the induction hypothesis. For the **assign** case the store can be modified but we also choose Δ' to be Δ'' so $\Delta' \supseteq \Delta$ holds by the induction hypothesis, and we shall prove that Δ'' respects the modified store. For the **ref** case we will construct a suitable Δ'.

For any of the above choices of Δ', we have $\Delta' \supseteq \Delta''$, so by Lemma 4.1 $\Gamma; \Delta' \vdash E[N''] : \tau \backslash C''$. By Lemma 4.14, there are τ' and C_3 such that $\Gamma; \Delta' \vdash N'' : \tau' \backslash C_3$. If we can also show that $\Gamma; \Delta' \vdash N' : \tau' \backslash C_4$ where C_4 is some constraint set, then by Lemma 4.15, there exists C' such that $\Gamma; \Delta' \vdash E[N'] : \tau \backslash C'$. So for each case below, we derive a suitable constrained type for the term in the context hole on the right hand side of the transition.

case Succ

We have the judgement $\Gamma; \Delta' \vdash \texttt{succ}\ \ n_i : \texttt{int} \backslash \emptyset$. The judgement $\Gamma; \Delta' \vdash n_{i+1} : \texttt{int} \backslash \emptyset$ follows from the typing rules.

cases Pred, Zero-True, Zero-False

Similar to the case for Succ.

case Fst

We have a proof of the form:

$$\dfrac{\dfrac{\textbf{typeof}\,(\texttt{fst}\,) \succ \ldots}{\Gamma; \Delta' \vdash \texttt{fst} : \ldots} \quad \dfrac{\Gamma; \Delta' \vdash V : \tau' \backslash C_4 \quad \Gamma; \Delta' \vdash V' : \tau'' \backslash C_5}{\Gamma; \Delta' \vdash (V, V') : \tau' \times^{\phi'} \tau'' \backslash C_4 \cup C_5 \cup C'' \cup C'''}}{\Gamma; \Delta' \vdash \texttt{fst} \;\; (V, V') : \tau' \backslash C_4 \cup C_5 \cup C'' \cup C'''}$$

where :
1. $C'' = \{\phi' \leq \textbf{lab}\,(\tau')\}$ *or* $C'' = \emptyset$ *if* $\textbf{lab}\,(\tau')$ *is undefined*
2. $C''' = \{\phi' \leq \textbf{lab}\,(\tau'')\}$ *or* $C''' = \emptyset$ *if* $\textbf{lab}\,(\tau'')$ *is undefined*

By this proof we have that $\Gamma; \Delta' \vdash V : \tau' \backslash C_4$.

case Snd

Similar to the previous case.

case β

We have a proof of the form:

$$\dfrac{\dfrac{\Gamma[x \mapsto \tau'' \backslash C_5]; \Delta' \vdash P : \tau' \backslash C_6 \quad (1) \quad (*)}{\Gamma; \Delta' \vdash \texttt{fn}\, x.P : \tau'' \xrightarrow{\phi} \tau' \backslash C_5 \cup C_6} \qquad \Gamma; \Delta' \vdash V : \tau'' \backslash C_7 \quad (2)}{\Gamma; \Delta' \vdash (\texttt{fn}\, x.P)\; V : \tau' \backslash C_5 \cup C_6 \cup C_7}$$

where $(*)$ indicates a possible use of the three-place version of **esc?**. By the bound variable convention, as x is bound in $\texttt{fn}\, x.P$, we have that $x \notin Dom(\Gamma)$. Then by the judgements marked (1) and (2) and by the Ordinary Substitution lemma we get $\Gamma; \Delta' \vdash P[V/x] : \tau' \backslash C_4$ for some C_4.

case Rec-Unroll

We have a proof of this form:

$$\dfrac{\dfrac{\Gamma[f \mapsto \tau'' \xrightarrow{\phi} \tau' \backslash C_7][x \mapsto \tau'' \backslash C_6]; \Delta' \vdash P : \tau' \backslash C_5 \quad (1) \quad (*)}{\Gamma; \Delta' \vdash \texttt{rec}\, f\, x.P : \tau'' \xrightarrow{\phi} \tau' \backslash C_7 \cup C_6 \cup C_5 \quad (2)} \qquad \Gamma; \Delta' \vdash V : \tau'' \backslash C_9 \quad (3)}{\Gamma; \Delta' \vdash (\texttt{rec}\, f\, x.P)\; V : \tau' \backslash C_7 \cup C_6 \cup C_5 \cup C_9}$$

where $(*)$ indicates a possible use of **esc?**.

We need more complicated reasoning here than we used in the β case. Let us rearrange the order of the term type context in the premise marked with (1) in the proof above to get:

$$\Gamma[x \mapsto \tau'' \backslash C_6][f \mapsto \tau'' \xrightarrow{\phi} \tau' \backslash C_7]; \Delta' \vdash P : \tau' \backslash C_5. \qquad (4.62)$$

Since x is bound in $\mathtt{rec}\, f\, x.P$, by Lemma 4.2 and the premise marked (2) in the proof above we get

$$\Gamma[x \mapsto \tau''\backslash C_6]; \Delta' \vdash \mathtt{rec}\, f\, x.P : \tau'' \xrightarrow{\phi} \tau'\backslash C_7 \cup C_6 \cup C_5. \tag{4.63}$$

By the variable convention, as f is bound in $\mathtt{rec}\, f\, x.P$ we have that $f \notin Dom(\Gamma[x \mapsto \tau''\backslash C_6])$. This fact together with (4.62), (4.63) and the Ordinary Substitution lemma give us that for some C_8

$$\Gamma[x \mapsto \tau''\backslash C_6]; \Delta' \vdash P[\mathtt{rec}\, f\, x.P/f] : \tau'\backslash C_8.$$

By Lemma 3.1 we have that $C_6 \cup C_8$ is valid. With the judgement above as premise for any of the typing rules for functions in Figure 3.2 we get

$$\Gamma; \Delta' \vdash \mathtt{fn}\, x.P[(\mathtt{rec}\, f\, x.P)/f] : \tau'' \xrightarrow{\phi} \tau'\backslash C_6 \cup C_8.$$

Since bound variables are distinct, $f \neq x$, so we may rewrite the judgement above as

$$\Gamma; \Delta' \vdash (\mathtt{fn}\, x.P)[(\mathtt{rec}\, f\, x.P)/f] : \tau'' \xrightarrow{\phi} \tau'\backslash C_6 \cup C_8.$$

By Lemma 3.1 we have that $C_6 \cup C_8 \cup C_9$ is valid. Hence we can use the judgement above and the premise (3) from the proof above as premises for the rule for typing applications to conclude

$$\Gamma; \Delta' \vdash (\mathtt{fn}\, x.P)[(\mathtt{rec}\, f\, x.P)/f] \;\; V : \tau'\backslash C_6 \cup C_8 \cup C_9.$$

cases Cond-True, Cond-False

Similar to the case for Fst.

case Ref

For any $\Delta' \supseteq \Delta''$ we have a proof of the form

$$\frac{\Gamma; \Delta' \vdash \mathtt{ref} : \tau'' \xrightarrow{\phi'} \tau''\, \mathtt{ref}^{\,\phi}\backslash C_6 \quad \Gamma; \Delta' \vdash V : \tau''\backslash C_5}{\Gamma; \Delta' \vdash \mathtt{ref}\ V : \tau''\, \mathtt{ref}^{\,\phi}\backslash C_6 \cup C_5}$$

where $C_6 = \{\phi \leq \mathbf{lab}\,(\tau'')\}$, if $\mathbf{lab}\,(\tau'')$ is defined.

On the right-hand side, the context-filler is a fresh reference r. We define Δ' to be $\Delta''[r \mapsto \tau''\, \mathtt{ref}^{\,\phi}\backslash C_6]$. Hence by the rule for typing references we get $\Gamma; \Delta' \vdash r : \tau''\, \mathtt{ref}^{\,\phi}\backslash C_6$.

We still have to show that Δ' respects S' given Γ. By the induction hypothesis, Δ'' respects S'' given Γ, so we only have to handle the fresh element

r in $Dom(S')$. By the above proof, $\Gamma; \Delta' \vdash V : \tau'' \backslash C_5$ and as $S'(r) = V$ we also have $\Gamma; \Delta' \vdash S'(r) : \tau'' \backslash C_5$. From the proof above we know that $\{\phi \leq \textbf{lab}\,(\tau'')\} \subseteq C_6$ if $\textbf{lab}\,(\tau'')$ is defined. Since $\Delta'(r) = \tau''\,\texttt{ref}^{\,\phi} \backslash C_6$, by Definition 4.1, Δ' respects S' given Γ.

case Deref

The left-hand side term in the context is $\texttt{deref}\ \ r$. We have a type derivation of the form:

$$\dfrac{\dfrac{\textbf{typeof}\,(\texttt{deref}\,) \succ \tau'\,\texttt{ref}^{\,\phi} \xrightarrow{\phi'} \tau' \backslash \emptyset}{\Gamma; \Delta' \vdash \texttt{deref} : \tau'\,\texttt{ref}^{\,\phi} \xrightarrow{\phi'} \tau' \backslash \emptyset} \qquad \Gamma; \Delta' \vdash r : \tau'\,\texttt{ref}^{\,\phi} \backslash C_3}{\Gamma; \Delta' \vdash \texttt{deref}\ r : \tau' \backslash C_3}$$

On the right-hand side, the evaluation context becomes filled by $S'(r)$ which is the same as $S''(r)$. By the induction hypothesis Δ' respects S' given Γ. From the derivation just given, $\Delta'(r) = \tau'\,\texttt{ref}^{\,\phi} \backslash C_3$ hence, by Definition 4.1 we have that $\Gamma; \Delta' \vdash S'(r) : \tau' \backslash C_4$ for some C_4.

case Assign

For this case we choose Δ' to be the same as Δ''. We have $\Gamma; \Delta' \vdash r := V : \texttt{unit} \backslash C_3$ for the left-hand side term filling the context, and $\Gamma; \Delta' \vdash \texttt{unit} : \texttt{unit} \backslash \emptyset$ for the right-hand context-filler. We still need to show that Δ' respects S' given Γ.

For the term filling the context in the left-hand side, we have a derivation of the form

$$\dfrac{\dfrac{\textbf{typeof}\,(:=) \succ \ldots}{\Gamma; \Delta' \vdash := : \ldots} \qquad \dfrac{\Gamma; \Delta' \vdash r : \tau_1\,\texttt{ref}^{\,\phi} \backslash C_1 \quad (1) \qquad \Gamma; \Delta' \vdash V : \tau_1 \backslash C_2 \quad (2)}{\Gamma; \Delta' \vdash (r', V) : \tau_1\,\texttt{ref}^{\,\phi} \times^{\phi'} \tau_1 \backslash C_1 \cup C_2 \cup \{\phi' \leq \phi\} \cup C_3}}{\Gamma; \Delta' \vdash r' := V : \texttt{unit} \backslash C_1 \cup C_2 \cup \{\phi' \leq \phi\} \cup C_3}$$

where:
$C_3 = \{\phi \leq \textbf{lab}\ (\tau_1)\}$, *if* $\textbf{lab}\ (\tau_1)$ *is defined*, $C_3 = \emptyset$ *otherwise*

From (2) in the proof above, and as $S'(r) = V$ we have $\Gamma; \Delta' \vdash S'(r) : \tau_1 \backslash C_2$. By (1) in the proof above $\Delta'(r) = \tau_1\,\texttt{ref}^{\,\phi} \backslash C_1$. In order to satisfy the requirements of Definition 4.1, we must also show that if $\textbf{lab}\,(\tau_1)$ is defined, then $\phi \leq \textbf{lab}\,(\tau_1) \in C_1$. By the induction hypothesis, Δ'' respects S'' given Γ. Since $\Delta'(r) = \Delta''(r) = \tau_1\,\texttt{ref}^{\,\phi} \backslash C_1$ the required condition on C_1 holds.

case Let

The typing derivation for the left-hand side term is of the form:

$$\dfrac{\Gamma; \Delta' \vdash V : \tau'' \backslash C_5 \qquad \Gamma[x \mapsto \sigma]; \Delta' \vdash N : \tau' \backslash C_6}{\Gamma; \Delta' \vdash \texttt{let}\ x = V\ \texttt{in}\ N\,\texttt{end} : \tau' \backslash C_5 \cup C_6}$$

where $\sigma = Gen(\tau''\backslash C_5, \Gamma, \Delta')$.

By the bound variable convention, as x is bound in $\texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end}$ we have that $x \notin Dom(\Gamma)$. Hence, by the Polymorphic Substitution lemma, the two premises of the proof above and the assumption that Γ is polymorphic consistent we have that $\Gamma; \Delta' \vdash N[V/x] : \tau'\backslash C_5 \cup C_6$. ∎

The following theorem says that the type of threads is preserved by concurrent evaluation. It also says that threads continue across evaluation, remaining on the same processor where they are created, and that new threads have type `unit`.

Theorem 4.2 (Concurrent Type Preservation) *Let S, Π be a configuration and Δ a store type context respecting S given a polymorphic consistent term type context Γ. Let p be any thread identifier in $Dom(\Pi)$ and $\Pi(p) = (\rho, M)$ for some processor ρ and term M. Suppose that $\Gamma; \Delta \vdash M : \tau \backslash C$. Suppose also that $S, \Pi \underset{\mathsf{con}}{\longrightarrow}^* S', \Pi'$. Then $\Pi'(p) = (\rho, M')$ for some term M' and there are Δ' and C' such that:*

- $\Delta' \supseteq \Delta$,
- Δ' *respects* S' *given* Γ,
- $\Gamma; \Delta' \vdash M' : \tau\backslash C'$ *and*
- *for any thread* $q \in Dom(\Pi') - Dom(\Pi)$, *there is* C'' *such that* $\Gamma; \Delta' \vdash N' : \texttt{unit}\backslash C''$ *where* N' *is the term associated to* q *in* Π'.

PROOF. By induction on the length of the reduction sequence.

By inspecting the evaluation rules in Figure 2.2 we see that once a thread is created, it continues across each evaluation step. Moreover, threads remain on the processor where they are created. Those facts gives us the result that $\Pi'(p) = (\rho, M')$ for some M'.

If $\underset{\mathsf{con}}{\longrightarrow}^*$ represents zero evaluation steps, the results about Δ', C' and the type of M' hold by assumption. The result about new threads holds trivially.

For the induction, consider the last evaluation step. Suppose $S, \Pi \underset{\mathsf{con}}{\longrightarrow}^* S'', \Pi''$, and let the last step be $S'', \Pi'' \underset{\mathsf{con}}{\longrightarrow} S', \Pi'$. By the induction hypothesis, for any $p \in Dom(\Pi'')$ with $\Pi''(p) = (\rho, M'')$, for some M'', there are Δ'' and C''' such that $\Delta'' \supseteq \Delta$, Δ'' respects S'' given Γ, and $\Gamma; \Delta'' \vdash M'' : \tau\backslash C'''$.

We have to prove the existence of Δ' with the required properties. Whatever this suitable Δ' is, it should be such that $\Delta' \supseteq \Delta''$. So, if p does not change from the left to the right side, $M' = M''$ and by Lemma 4.2 we get $\Gamma; \Delta' \vdash M' : \tau\backslash C'''$.

We now proceed by case analysis on the last step, considering only threads that change.

case Seq

This transition has the sequential transition $S'', M'' \underset{\mathsf{seq}}{\longrightarrow} S', M'$ as premise. By the Sequential Type Preservation theorem there is C' and $\Delta' \supseteq \Delta''$, with Δ' respecting S' given Γ, such that $\Gamma; \Delta' \vdash M' : \tau \backslash C'$.

The result about the type of new threads holds trivially for this case as $Dom(\Pi') = Dom(\Pi'')$.

case Fork

In this transition $S' = S''$ so we choose Δ' to be Δ''. By the induction hypothesis we have $\Delta'' \supseteq \Delta$ and Δ'' respects S'' given Γ. Since $\Delta'' = \Delta'$ and $S'' = S'$ we have $\Delta' \supseteq \Delta$ and Δ' respects S' given Γ. We now examine the threads that change.

For the parent thread: by the induction hypothesis we have the following type judgement: $\Gamma; \Delta' \vdash E[\texttt{fork fn}\, x.P] : \tau \backslash C'''$. By Lemma 4.14 we have that $\Gamma; \Delta' \vdash \texttt{fork fn}\, x.P : \texttt{unit} \backslash C_3$ for some C_3. As $\Gamma; \Delta' \vdash \texttt{unit} : \texttt{unit} \backslash \emptyset$, by Lemma 4.15 there exists C' such that $\Gamma; \Delta' \vdash E[\texttt{unit}] : \tau \backslash C'$.

For the child thread $P[\texttt{unit}/x]$: the typing derivation for the judgement $\Gamma; \Delta' \vdash \texttt{fork fn}\, x.P : \texttt{unit} \backslash C_3$ must have been of this form:

$$\dfrac{\dfrac{\textbf{typeof}(\texttt{fork}) \succ \ldots}{\Gamma; \Delta' \vdash \texttt{fork} : \ldots} \qquad \dfrac{\Gamma[x \mapsto \texttt{unit} \backslash C_1]; \Delta' \vdash P : \texttt{unit} \backslash C_2 \quad (*) \quad (1)}{\Gamma; \Delta' \vdash \texttt{fn}\, x.P : \texttt{unit} \xrightarrow{\phi} \texttt{unit} \backslash C_1 \cup C_2}}{\Gamma; \Delta' \vdash \texttt{fork fn}\, x.P : \texttt{unit} \backslash C_1 \cup C_2}$$

where $(*)$ marks the possible use of the predicate $\textbf{esc?}(\Gamma, \Delta', \texttt{fn}\, x.P)$. By the bound variable convention, as x is bound in $\texttt{fn}\, x.P$ we have that $x \notin Dom(\Gamma)$. Then by the premise marked with (1) in the typing derivation above and as $\Gamma; \Delta' \vdash \texttt{unit} : \texttt{unit} \backslash \emptyset$, by Lemma 4.13 (Ordinary Substitution) we conclude that $\Gamma; \Delta' \vdash P[\texttt{unit}/x] : \texttt{unit} \backslash C''$, for some C''.

case Rfork

Similar to the Fork case. ∎

4.2 Faulty Threads are Untypable

In the previous section we proved that if the term associated to a thread has a type, this type is preserved by the reduction rules. This is not enough to prove type soundness of the type system with respect to the operational semantics. We still have to prove that if evaluating the term associated to a thread in a given configuration leads to a configuration in which the thread is faulty (see Definition 2.1) then the term is untypable.

As an example consider the following reduction sequence

$$S, [p : \rho,\ (\mathtt{fn}\,x.\,x\ \mathtt{true})\ \ \mathtt{succ}\,] \ \underset{\mathsf{con}}{\longrightarrow}\ S, [p : \rho,\ \mathtt{succ}\ \ \mathtt{true}].$$

The evaluation of the thread p in configuration $S, [p : \rho, (\mathtt{fn}\,x.x\ \mathtt{true})\ \ \mathtt{succ}\,]$ leads to the configuration $S, [p : \rho, \mathtt{succ}\ \ \mathtt{true}]$ where p is a faulty thread. One can easily show that the term $\mathtt{succ}\ \ \mathtt{true}$ is untypable. Because we proved last section that the relation $\underset{\mathsf{con}}{\longrightarrow}$ preserves types, the term $(\mathtt{fn}\,x.x\ \mathtt{true})\ \ \mathtt{succ}$ is also untypable.

If arbitrary configurations are allowed, then some threads may be faulty, even if typable. Consider this example

$$S, [p : \rho,\ (\mathtt{fn}\,x.\,x\ y)\ \ \mathtt{succ}\,] \ \underset{\mathsf{con}}{\longrightarrow}\ S, [p : \rho,\ \mathtt{succ}\ \ y].$$

As before, the evaluation of p in configuration $S, [p : \rho,\ (\mathtt{fn}\,x.x\ y)\ \ \mathtt{succ}\,]$ leads to the configuration $S, [p : \rho,\ \mathtt{succ}\ \ y]$ where p is faulty. But one can easily show that $\Gamma; \Delta \vdash \mathtt{succ}\ \ y : \mathtt{int} \setminus C$ for any Δ and any Γ such that $\Gamma(y) = \mathtt{int} \setminus C$. Another example is

$$[r' \mapsto 3], [p : \rho,\ (\mathtt{fn}\,x.\ \mathtt{succ}\ \ (\mathtt{deref}\ x))\ r] \ \underset{\mathsf{con}}{\longrightarrow}\ [r' \mapsto 3], [p : \rho,\ \mathtt{succ}\ \ (\mathtt{deref}\ r)].$$

The evaluation of thread p in configuration $[r' \mapsto 3], [p : \rho,\ (\mathtt{fn}x.\ \mathtt{succ}\ \ (\mathtt{deref}\ x))\ r]$ leads to a configuration where p is faulty (in this case the thread is faulty because $r \notin \mathit{Dom}([r' \mapsto 3])$). But again $\Gamma; \Delta \vdash \mathtt{succ}\ \ (\mathtt{deref}\ r) : \mathtt{int}\setminus C$ for any Γ and Δ such that $\Delta(r) = \mathtt{int}\,\mathtt{ref}^{\,\phi}\setminus C$.

Next we define what we call *well-formed* configurations and then we prove that if a thread is faulty in a well-formed configuration then its associated term is untypable. This is not a restrictive assumption since, as we also prove later, well-formedness is preserved by evaluation and, according to Definition 2.5, all the computations of a program start in a well-formed configuration.

We first define what we mean by a store to be well-formed:

Definition 4.2 *S is a* well-formed store *if for all terms $M \in \mathit{Range}(S)$, if $r \in \mathit{Refs}(M)$ then $r \in \mathit{Dom}(S)$.*

The set of store locations appearing in a thread map Π is given by

$$\mathit{Refs}(\Pi) = \{r \mid r \in \mathit{Refs}(M)\ \mathit{and}\ \exists\, p \text{ such that } \Pi(p) = (\rho, M)\}.$$

The set of free variables appearing in Π is given by

$$FV(\Pi) = \{x \mid x \in FV(M) \ \textit{and} \ \exists p \text{ such that } \Pi(p) = (\rho, M)\}.$$

A configuration S, Π is well-formed if S is well-formed and all terms in Π are closed and have their reference subterms mapped in store.

Definition 4.3 S, Π *is a* well-formed configuration *if S is a well-formed store, $FV(\Pi) = \emptyset$ and $Refs(\Pi) \subseteq Dom(S)$.*

Well-formedness of configurations is preserved by evaluation:

Lemma 4.16 *If S, Π is a well-formed configuration and $S, \Pi \xrightarrow[\mathsf{con}]{}^{*} S', \Pi'$ then S', Π' is well-formed.*

Proof. By induction on the number of evaluation steps. ∎

Finally, combining the type preservation theorems with the following lemma we achieve type soundness for our type system.

Lemma 4.17 (Faulty Threads are Untypable) *If p is a faulty thread in a well-formed configuration S, Π and $\Pi(p) = (\rho, M)$ then M is untypable.*

Proof. By Definition 2.1 the term M is not a value then, by Lemma 2.1 it can be uniquely decomposed as $E[R]$ where R is a redex. We have to prove that there are no Δ and no constrained type $\tau \backslash C$ such that $\Delta \vdash E[R] \colon \tau \backslash C$. We assume that there is such typing for $E[R]$ and we try to get a contradiction from this. By Lemma 4.14 we have that $\Delta \vdash R \colon \tau' \backslash C'$ for some type τ' and constraint set C'.

Since S, Π is well-formed, p is faulty iff R is an improper redex, that is, R is a redex of one of these forms:

1. `pred` V, `succ` V, `zero?` V where V is not a numeral,
2. `fst` V, `snd` V where V is not a pair of values,
3. `deref` V, where V is not a reference constant,
4. $V_1 := V_2$, where V_1 is not a reference constant,
5. `fork` V, $\texttt{rfork}_\rho\, V$ where V is not a function abstraction,
6. `if` V `then` M `else` N where V is neither `true` nor `false`,
7. $V_1\, V_2$ where V_1 is neither a function constant nor a function abstraction.

It is easy to show that if R has one of the forms above it is untypable. As a consequence $E[R]$ is also untypable. ∎

4.3 Type Soundness

Here we finally prove that the type system is sound with respect to the operational semantics. In fact, the main result of this section is only part of the soundness result as it does not take into account if the locality information labelling types is sound with respect to the operational semantics. Locality soundness is proved in the next chapter.

The following type soundness theorem encompasses all possible computations of a program P, so it applies to the initial thread and to all possible new threads created along evaluation. In the following we say that a program P is *well-typed* if $\vdash P : \tau \backslash C$ for some constrained type $\tau \backslash C$.

Theorem 4.3 (Syntactic Type Soundness) *Let P be a well-typed program, T be any trace in Comp(P), and p be any thread in Threads(T). Suppose S_i, Π_i is the configuration of T where p occurs for the first time and let M be the term associated with p in Π_i. Then there exist Δ respecting S_i, τ and C such that $\Delta \vdash M : \tau \backslash C$ and either:*

- *$p \Uparrow_T$ or*
- *$p \Downarrow_T V$ and there are $\Delta' \supseteq \Delta$ and C' such that $\Delta' \vdash V : \tau \backslash C'$.*

Proof. All traces in *Comp*(P) start in a well-formed configuration $\emptyset, \Pi_0$ where $\Pi_0 = [p_0 : (\rho_0, P)]$. The existence of Δ respecting S_i, τ and C such that $\Delta \vdash M : \tau \backslash C$ follows from the Concurrent Type Preservation theorem (Theorem 4.2). By the Uniform Evaluation lemma (Lemma 2.2), we know that either $p \Uparrow_T$, $p \Downarrow_T V$ or p is a faulty thread in a configuration of T.

First we prove that p is never faulty. We suppose that p is faulty in a configuration S_j, Π_j of T, for some $j \geq i$, and we try to get a contradiction from that. By Lemma 4.16 the configuration S_j, Π_j is well-formed. Suppose that $\Pi_j(p) = (\rho, M')$. By the Concurrent Type Preservation theorem there are Δ' respecting S_j and C' such that $\Delta' \vdash M' : \tau \backslash C'$. But this contradicts Lemma 4.17 so p cannot be a faulty thread in any configuration of T.

If $p \Uparrow_T$ we are done.

If $p \Downarrow_T V$: let S_n, Π_n, $n \geq i$, be a configuration in T such that $\Pi_n(p) = (\rho, V)$. By the Concurrent Type Preservation theorem there are Δ' respecting S_n and C' such that $\Delta' \vdash V : \tau \backslash C'$. ∎

In order to state a more traditional soundness result we need the following definition

Definition 4.4 (eval) *For a computation T, define the evaluation of a thread p in T as:*

$$eval_T(p) = \begin{cases} \textbf{wrong} & \textit{if } p \textit{ is a faulty thread in a configuration of } T \\ V & \textit{if } p \Downarrow_T V. \end{cases}$$

With this definition we can state weak and strong soundness results.

Theorem 4.4 (Weak Type Soundness) *Let P be a well-typed program, T be any trace in Comp(P) and p be any thread in Threads(T). Then $eval_T(p) \neq$* **wrong**.

PROOF. By Theorem 4.3 and Definition 4.4. ∎

Theorem 4.5 (Strong Type Soundness) *Let P be a well-typed program, T be any trace in Comp(P), p be any thread in Threads(T), S_i, Π_i be the configuration of T where p occurs for the first time and let M be the term associated with p in Π_i such that $\Delta \vdash M : \tau \backslash C$ for Δ respecting S_i. Then if $eval_T(p) = V$, there is $\Delta' \supseteq \Delta$ and C' such that $\Delta' \vdash V : \tau \backslash C'$.*

PROOF. By Theorem 4.3 and Definition 4.4. ∎

Chapter 5

Locality Soundness

We call *local references* those references whose types are labelled with ℓ or with a locality variable. In this chapter we prove the claim made in Chapter 3 that local references are certain not to escape according to the operational semantics. This property, which we call *Locality Soundness*, together with type soundness proved last chapter, completes the result that our type system is sound with respect to the operational semantics given in Chapter 2.

5.1 Notions of Reachability

We define three notions of *reachability* of local references that will be used to state locality soundness and we prove some of their properties.

The first notion is the set $\mathit{Reach}^{\Delta}(M)$ of reference constants of M which are local according to store type context Δ. We define this set as follows:

$$\mathit{Reach}^{\Delta}(M) \;=\; \{\, r \in \mathit{Refs}(M) \cap \mathit{Dom}(\Delta) \mid \mathbf{lab}\,(\Delta(r)) \neq \epsilon \,\}.$$

The following lemma says that if the type of a value is escaping then it has no local references as subterms.

Lemma 5.1 *If* $\Gamma; \Delta \vdash V : \tau \backslash C$ *and* $\mathbf{esc?}(\tau)$ *then* $\mathit{Reach}^{\Delta}(V) = \emptyset$.

Proof. By the assumptions $\Gamma; \Delta \vdash V : \tau \backslash C$ and $\mathbf{esc?}(\tau)$ we have, by Lemma 3.7 that $\mathbf{esc?}(\Gamma; \Delta, V)$, then by Definition 3.17, for all $r \in \mathit{Refs}(V)$, $\mathbf{lab}\,(\Delta(r)) = \epsilon$. Hence $\mathit{Reach}^{\Delta}(V) = \emptyset$. ∎

The set of reachable local references from a term M with respect to S and Δ is given by the following recursive definition:

$$\mathit{Reach}^{\Delta,S}(M) \;=\; \mathit{Reach}^{\Delta}(M) \;\cup \bigcup_{r \,\in\, \mathit{Refs}(M) \cap \mathit{Dom}(S)} \mathit{Reach}^{\Delta,S}(S(r)).$$

More formally, $Reach^{\Delta,S}$ is the least fixed point of the functional $F^{\Delta,S}\colon D \to D$ defined by

$$F^{\Delta,S}(f^{\Delta,S})(N) \;=\; Reach^{\Delta}(N) \;\;\cup \bigcup_{r \,\in\, Refs(N) \cap Dom(S)} f^{\Delta,S}(S(r))$$

where D is the set of functions from terms of the language to sets of references, ordered by $f^{\Delta,S} \sqsubseteq_D g^{\Delta,S}$ iff $f^{\Delta,S}(M) \subseteq g^{\Delta,S}(M)$ for all terms M. The least fixed point exists because D is clearly a directed complete partial order with a least element $\perp \stackrel{\text{def}}{=} \lambda M.\, \emptyset$, and one readily checks that $F^{\Delta,S}$ is continuous.

The following lemma says that, under certain conditions, if the type of a value is escaping then no local references can be reached from it.

Lemma 5.2 *Let Δ be a store typing context respecting S given a term type context Γ, and let V be a value. If $\Gamma; \Delta \vdash V : \tau \backslash C$ and $\mathbf{esc?(}\tau\mathbf{)}$ then $Reach^{\Delta,S}(V) = \emptyset$.*

Proof. We prove by fixed point induction [29] that $Reach^{\Delta,S}$ satisfies the predicate

$$P(f^{\Delta,S}) \stackrel{\text{def}}{=} \textit{for any value } V\textit{, if } \Gamma; \Delta \vdash V : \tau \backslash C \textit{ and } \mathbf{esc?(}\tau\mathbf{)} \textit{ then } f^{\Delta,S}(V) = \emptyset.$$

For clarity, we omit the superscripts Δ and S as they are clear from the context. First we prove that P is admissible: one has to show that if each member of an ascending chain $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \ldots$ satisfies the property, so does its *supremum* $\bigsqcup_{n \geq 0} f_n$. But this is immediate because $(\bigsqcup_{n \geq 0} f_n)(M) = \bigcup_{n \geq 0} f_n(M)$.

For the base case: we have to show that $P(\perp)$ holds which is also immediate.

Now the induction step: assume that $P(f)$ holds. We have to conclude that so does $P(F(f))$. In order to conclude this, we assume that both the conditions

$$\Gamma; \Delta \vdash V : \tau \backslash C \quad \textit{and} \quad \mathbf{esc?(}\tau\mathbf{)} \tag{5.1}$$

hold, and from this we deduce that $F(f)(V) = \emptyset$.

By Lemma 5.1 and (5.1) above we have

$$Reach^{\Delta}(V) = \emptyset. \tag{5.2}$$

Hence, by (5.2) above and the definition of $Reach^{\Delta}$, if $r \in Refs(V)$ we have $\Delta(r) = \tau' \, \mathtt{ref}^{\,\epsilon} \backslash C'$ for some type τ' and some constraint set C'. Because Δ respects S, if the same r is such that $r \in Dom(S)$, then

$$\Gamma; \Delta \vdash S(r) : \tau' \backslash C'', \tag{5.3}$$

for some constraint set C''. Also by the definition of respects (Definition 4.1) τ' is either ground or is labelled with ϵ, so we have

$$\mathbf{esc?}(\tau'). \tag{5.4}$$

Because $P(f)$ holds by hypothesis, by (5.3) and (5.4) we have

$$f(S(r)) = \emptyset, \ \mathit{for} \ \ r \in \mathit{Refs}(V) \cap \mathit{Dom}(S). \tag{5.5}$$

Therefore, $F(f)(V) = \emptyset$ since

$$\begin{array}{rcllll} F(f)(V) & = & \mathit{Reach}^{\Delta}(V) & \cup & \bigcup_{r \in \mathit{Refs}(V) \cap \mathit{Dom}(S)} f(S(r)) & \\ & = & \emptyset & \cup & \bigcup_{r \in \mathit{Refs}(V) \cap \mathit{Dom}(S)} f(S(r)) & \text{by (5.2)} \\ & = & \emptyset & \cup & \emptyset & \text{by (5.5)} \end{array}$$

which concludes the proof. ∎

We finally give our third notion of reachability. The set of local references reachable from processor ρ in thread map Π, given a store type context Δ and a store S is defined as follows

$$\mathit{Reach}^{\Pi}_{\Delta,S}(\rho) \ = \bigcup_{q \in \mathit{Dom}(\Pi) \wedge \Pi(q) = (\rho, M)} \mathit{Reach}^{\Delta,S}(M).$$

The lemmas 5.3 to 5.7 below state some useful properties about *Reach* to be used in other proofs. We omit their proofs as they follow immediately from the definitions of reachability presented.

Lemma 5.3 *If* $\mathit{Reach}^{\Delta,S}(M) = \mathit{Reach}^{\Delta,S}(M')$ *then*

1. $\mathit{Reach}^{\Pi[p:\rho,E[M]]}_{\Delta,S}(\rho) = \mathit{Reach}^{\Pi[p:\rho,E[\mathtt{unit}]][q:\rho,M']}_{\Delta,S}(\rho)$, *if* $q \notin \mathit{Dom}(\Pi)$, *and*
2. $\mathit{Reach}^{\Pi[p:\rho,E[M]]}_{\Delta,S}(\rho) = \mathit{Reach}^{\Pi[p:\rho,E[M']]}_{\Delta,S}(\rho)$.

Proof. Immediate from the definition of *Reach*. ∎

Lemma 5.4 *If* $\mathit{Reach}^{\Delta,S}(M) \supseteq \mathit{Reach}^{\Delta,S}(M')$ *then*

$$\mathit{Reach}^{\Pi[p:\rho,E[M]]}_{\Delta,S}(\rho) \supseteq \mathit{Reach}^{\Pi[p:\rho,E[M']]}_{\Delta,S}(\rho).$$

Proof. Immediate from the definition of *Reach*. ∎

Lemma 5.5 *If* $Reach^{\Delta,S}(M)=\emptyset$ *and* $p\notin Dom(\Pi)$ *then*

$$Reach^{\Pi}_{\Delta,S}(\rho)=Reach^{\Pi[p:\rho,M]}_{\Delta,S}(\rho).$$

PROOF. Immediate from the definition of *Reach*. ∎

Lemma 5.6 $Reach^{\Pi'[p:\rho,E[M]]}_{\Delta,S}(\rho)=Reach^{\Pi'[p:\rho,E[c]]}_{\Delta,S}(\rho)\cup Reach^{\Delta,S}(M)$.

PROOF. Immediate from the definition of *Reach*. ∎

Lemma 5.7 *If* Δ *respects* S *given a term context* Γ*,* $Refs(\Pi)\subseteq Dom(\Delta)$ *and* $\Delta'\supseteq\Delta$ *then*

$$Reach^{\Pi}_{\Delta,S}(\rho)=Reach^{\Pi}_{\Delta',S}(\rho).$$

PROOF. Immediate from the definition of *Reach*. ∎

Lemma 5.8 *If* r *is new or* **lab** $(\Delta(r))=\epsilon$ *then*

$$Reach^{\Pi}_{\Delta,S}(\rho)=Reach^{\Pi}_{\Delta,S[r\mapsto V]}(\rho).$$

PROOF. Immediate from the definition of *Reach*. ∎

5.2 Locality Preservation

The following lemma says that, under certain conditions, if *the sets of local references reachable from different processors are disjoint* they remain so along evaluation. Before, we define the set of processors associated with a thread map Π as follows

$$ProcSet(\Pi)=\{\rho\mid\exists\, q \textit{ such that } \Pi(q)=(\rho,M)\}.$$

Lemma 5.9 *Let* S,Π *be a configuration, and assume that all threads in* Π *are well-typed in a store type context* Δ *that respects* S *given* Γ*, a term type context. Suppose that* $Reach^{\Pi}_{\Delta,S}(\rho)\cap Reach^{\Pi}_{\Delta,S}(\rho')=\emptyset$ *for all distinct* $\rho,\rho'\in ProcSet(\Pi)$*. Suppose also that* $S,\Pi\ \underset{\mathsf{con}}{\longrightarrow}^{*}\ S',\Pi'$*. Then there is* Δ' *that respect* S' *given* Γ*,* $\Delta'\supseteq\Delta$*, and, for all such* Δ'

$$Reach^{\Pi'}_{\Delta',S'}(\rho)\cap Reach^{\Pi'}_{\Delta',S'}(\rho')=\emptyset \textit{ for all distinct } \rho,\rho'\in ProcSet(\Pi').$$

PROOF. By induction on the length of the reduction sequence. For a sequence of length zero the result holds by assumption. For the induction consider the last evaluation step. Suppose $S, \Pi \underset{\mathsf{con}}{\longrightarrow}^* S'', \Pi''$ and let the last step be $S'', \Pi'' \underset{\mathsf{con}}{\longrightarrow} S', \Pi'$.

By the induction hypothesis $\mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho) \cap \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho') = \emptyset$. The existence of $\Delta' \supseteq \Delta''$ respecting S' follows by the Concurrent Type Preservation theorem. Now we consider each possible transition proceeding as follows:

- for the transition Fork we have that $\mathit{ProcSet}(\Pi') = \mathit{ProcSet}(\Pi'')$. We prove that $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$ for any $\rho \in \mathit{ProcSet}(\Pi')$. The result then follows from the induction hypothesis,

- for the transition Rfork we prove that for any processor $\rho' \in \mathit{ProcSet}(\Pi')$, $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$. If the new thread is created on ρ' already in $\mathit{ProcSet}(\Pi'')$ we have $\mathit{ProcSet}(\Pi') = \mathit{ProcSet}(\Pi'')$ and the result also holds by the induction hypothesis. If $\rho' \notin \mathit{ProcSet}(\Pi'')$ we prove $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho') = \emptyset$.

- For the transition Seq we have that $\mathit{ProcSet}(\Pi') = \mathit{ProcSet}(\Pi'')$. For Seq using sequential transitions Succ, Pred, Zero-True and Zero-False we prove that $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$ for any $\rho \in \mathit{ProcSet}(\Pi')$. The result then follows from the induction hypothesis,

- for Seq using the sequential transitions Fst, Snd, Cond-True, Cond-False, β, Rec-Unroll, Let, Deref and Assign as premises, we prove $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) \subseteq \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$ for ρ involved in the transition. For any other processor ρ' not involved, we prove $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho') = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho')$. We then get the desired result by the induction hypothesis and by elementary set theory,

- if the Ref transition is used in the premise of Seq we give a proof that for ρ' not involved in the transition $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho') = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho')$. If ρ is the processor where the selected thread is we prove either $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$, so the result holds by the induction hypothesis, or we have that $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho) \cup \{r'\}$ where r' is the new reference created, in which case by induction hypothesis and by set theory we obtain the desired conclusion.

case Fork

The last evaluation step has this form:

$$S'', \Pi'''[p : \rho, E[\texttt{fork fn}\,x.M]] \underset{\mathsf{con}}{\longrightarrow} S'', \Pi'''[p : \rho, E[\texttt{unit}]][q : \rho, M[\texttt{unit}/x]].$$

For processor $\rho' \neq \rho$ we have

$$\begin{array}{lll} & \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho') & \\ = & \mathit{Reach}^{\Pi''}_{\Delta',S''}(\rho') & \text{Lemma 5.7} \\ = & \mathit{Reach}^{\Pi''}_{\Delta',S'}(\rho') & S' = S'' \\ = & \mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho') & \rho \neq \rho'. \end{array}$$

One can readily check that

$$\mathit{Reach}^{\Delta'',S''}(\texttt{fork}\,\texttt{fn}\,x.M) = \mathit{Reach}^{\Delta'',S''}(M[\texttt{unit}/x]). \tag{5.6}$$

So for the processor ρ we can prove

$$\begin{array}{lll} & \mathit{Reach}^{\Pi'''[p:\rho,E[\texttt{fork fn}\,x.M]]}_{\Delta'',S''}(\rho) & \\ = & \mathit{Reach}^{\Pi'''[p:\rho,E[\texttt{unit}]]\,[q:\rho,M[\texttt{unit}/x]]}_{\Delta'',S''}(\rho) & \text{Lemma 5.3.(1) and (5.6)} \\ = & \mathit{Reach}^{\Pi'''[p:\rho,E[\texttt{unit}]]\,[q:\rho,M[\texttt{unit}/x]]}_{\Delta',S''}(\rho) & \text{Lemma 5.7} \\ = & \mathit{Reach}^{\Pi'''[p:\rho,E[\texttt{unit}]]\,[q:\rho,M[\texttt{unit}/x]]}_{\Delta',S'}(\rho) & S' = S'. \end{array}$$

case Rfork

The last evaluation step has this form:

$$S'', \Pi'''[p\colon \rho, E[\texttt{rfork}_{\rho'}\,\texttt{fn}\,x.M]] \underset{\mathsf{con}}{\longrightarrow} S'', \Pi'''[p\colon \rho, E[\texttt{unit}]][q\colon \rho', M[\texttt{unit}/x]].$$

The proof that $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho'') = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho'')$, for ρ'' not involved in the transition is similar to the case for Fork for processors not involved in that transition.

By the Concurrent Type Preservation theorem, for some constrained type $\tau \backslash C$, we have that $\Gamma; \Delta'' \vdash E[\texttt{rfork}_{\rho'}\ \texttt{fn}\,x.M] : \tau \backslash C$.

By the Context Filler lemma , $\Gamma; \Delta'' \vdash \texttt{rfork}_{\rho'}\ \texttt{fn}\,x.M : \texttt{unit} \backslash C'''$ and the proof of this judgement must have been of this form:

$$\dfrac{\dfrac{\textbf{typeof}(\texttt{rfork}_{\rho'}) \succ \ldots}{\Gamma; \Delta'' \vdash \texttt{rfork}_{\rho'} : \ldots} \qquad \Gamma; \Delta'' \vdash \texttt{fn}\,x.M : \texttt{unit} \xrightarrow{\epsilon} \texttt{unit} \backslash C''' \quad (1)}{\Gamma; \Delta'' \vdash \texttt{rfork}_{\rho'}\,\texttt{fn}\,x.M : \texttt{unit} \backslash C'''}$$

By Lemma 5.2 and the premises marked with (1) in the type derivation above, it follows that $\mathit{Reach}^{\Delta'',S''}(\texttt{fn}\,x.M) = \emptyset$. Therefore we can easily prove the following two equations

$$\mathit{Reach}^{\Delta'',S''}(\texttt{rfork}_{\rho'}\ \texttt{fn}\,x.M) = \mathit{Reach}^{\Delta'',S''}(\texttt{unit}) \tag{5.7}$$

and

$$\mathit{Reach}^{\Delta'',S''}(M[\texttt{unit}/x]) = \emptyset. \tag{5.8}$$

For processor ρ we can then prove:

$$
\begin{array}{lll}
 & Reach^{\Pi'''[p:\rho,E[\mathtt{rfork}_{\rho'}\ \mathtt{fn}\ x.M]]}_{\Delta'',S''}(\rho) & \\
= & Reach^{\Pi'''[p:\rho,E[\mathtt{unit}]]}_{\Delta'',S''}(\rho) & \text{Lemma 5.3(2) and (5.7)} \\
= & Reach^{\Pi'''[p:\rho,E[\mathtt{unit}]]\,[q:\rho',M[\mathtt{unit}/x]]}_{\Delta'',S''}(\rho) & \rho' \neq \rho \\
= & Reach^{\Pi'''[p:\rho,E[\mathtt{unit}]]\,[q:\rho',M[\mathtt{unit}/x]]}_{\Delta',S''}(\rho) & \text{Lemma 5.7} \\
= & Reach^{\Pi'''[p:\rho,E[\mathtt{unit}]]\,[q:\rho',M[\mathtt{unit}/x]]}_{\Delta',S'}(\rho) & S' = S''.
\end{array}
$$

For the processor ρ', where the new thread is created:

$$
\begin{array}{lll}
 & Reach^{\Pi'''[p:\rho,E[\mathtt{rfork}_{\rho'}\ \mathtt{fn}\ x.M]]}_{\Delta'',S''}(\rho') & \\
= & Reach^{\Pi'''[p:\rho,E[\mathtt{unit}]]}_{\Delta'',S''}(\rho') & \rho \neq \rho' \\
= & Reach^{\Pi'''[p:\rho,E[\mathtt{unit}]][q:\rho',M[\mathtt{unit}/x]]}_{\Delta'',S''}(\rho') & \text{Lemma 5.5 and (5.8)} \\
= & Reach^{\Pi'''[p:\rho,E[\mathtt{unit}]][q:\rho',M[\mathtt{unit}/x]]}_{\Delta',S''}(\rho') & \text{Lemma 5.7} \\
= & Reach^{\Pi'''[p:\rho,E[\mathtt{unit}]][q:\rho',M[\mathtt{unit}/x]]}_{\Delta',S'}(\rho') & S' = S''.
\end{array}
$$

If $\rho' \in ProcSet(\Pi'')$ then $ProcSet(\Pi') = ProcSet(\Pi'')$ and we get the desired result for this case by the induction hypothesis. On the other hand, if $\rho' \notin ProcSet(\Pi'')$, we still have to prove that $Reach^{\Pi'}_{\Delta',S'}(\rho') \cap Reach^{\Pi'}_{\Delta',S'}(\rho) = \emptyset$ for all $\rho \in ProcSet(\Pi')$ different than ρ'. But that clearly holds since, by (5.8) above, we have

$$Reach^{[p:\rho' M[\mathtt{unit}/x]]}_{\Delta',S'}(\rho') = \emptyset.$$

For the concurrent transition Seq we consider each possible sequential rule it uses as premise.

case Succ

The last evaluation step has this form:

$$S'', \Pi'''[p:\rho, E[\mathsf{succ}\ n_i]] \underset{\mathsf{con}}{\longrightarrow} S'', \Pi'''[p:\rho, E[n_{i+1}]].$$

For $\rho' \neq \rho$ the proof that $Reach^{\Pi'}_{\Delta',S'}(\rho') = Reach^{\Pi''}_{\Delta'',S''}(\rho')$ is the same as in the case for Fork. For the processor ρ we have:

$$
\begin{array}{lll}
 & Reach^{\Pi'''[p:\rho,E[\mathsf{succ}\, n_i]]}_{\Delta'',S''}(\rho) & \\
= & Reach^{\Pi'''[p:\rho,E[n_{i+1}]]}_{\Delta'',S''}(\rho) & \text{Lemma 5.3 (2)} \\
= & Reach^{\Pi'''[p:\rho,E[n_{i+1}]]}_{\Delta',S''}(\rho) & \text{Lemma 5.7} \\
= & Reach^{\Pi'''[p:\rho,E[n_{i+1}]]}_{\Delta',S'}(\rho) & S' = S''.
\end{array}
$$

cases Pred, $\mathsf{Zero\text{-}True}$, $\mathsf{Zero\text{-}False}$

Proof as in the previous case.

case Fst

The last evaluation step is:

$$S'', \Pi''[p : \rho, E[\texttt{fst}\ (V, V')]] \underset{\mathsf{con}}{\longrightarrow} S'', \Pi''[p : \rho, E[V]].$$

For $\rho' \neq \rho$ the proof that $\mathit{Reach}^{\Pi'}_{\Delta', S'}(\rho') = \mathit{Reach}^{\Pi''}_{\Delta'', S''}(\rho')$ is the same as in the Fork case.

We can easily prove that

$$\mathit{Reach}^{\Delta'', S''}(\texttt{fst}\ (V, V')) \supseteq \mathit{Reach}^{\Delta'', S''}(V). \tag{5.9}$$

Then for processor ρ we have:

$$\begin{array}{lll} & \mathit{Reach}^{\Pi'''[p:\rho, E[\texttt{fst}\,(V,V')]]}_{\Delta'', S''}(\rho) & \\ \supseteq & \mathit{Reach}^{\Pi'''[p:\rho, E[V]]}_{\Delta'', S''}(\rho) & \text{Lemma 5.4 and (5.9)} \\ = & \mathit{Reach}^{\Pi'''[p:\rho, E[V]]}_{\Delta', S''}(\rho) & \text{Lemma 5.7} \\ = & \mathit{Reach}^{\Pi'''[p:\rho, E[V]]}_{\Delta', S'}(\rho) & S' = S''. \end{array}$$

cases Snd, Cond-True, Cond-False, β, Rec-Unroll, Let

Similar to the previous case.

case Deref

The last evaluation step has this form:

$$S'', \Pi'''[p : \rho, E[\texttt{deref}\ r']] \underset{\mathsf{con}}{\longrightarrow} S'', \Pi'''[p : \rho, E[V]] \qquad S''(r') = V.$$

For $\rho' \neq \rho$ the proof that $\mathit{Reach}^{\Pi'}_{\Delta', S'}(\rho) = \mathit{Reach}^{\Pi''}_{\Delta'', S''}(\rho)$ is the same as in the Fork case.

By the Concurrent Type Preservation theorem, for some τ' and C', we know that $\Gamma; \Delta'' \vdash E[\texttt{deref}\ r'] : \tau' \backslash C'$. By Lemma 4.14, for some τ'' and C'', we have $\Gamma; \Delta'' \vdash \texttt{deref}\ r' : \tau'' \backslash C''$ and its proof must have been of this form:

$$\dfrac{\dfrac{\textbf{typeof}\ (\texttt{deref}\ ) \succ \tau''\ \texttt{ref}\ ^{\phi} \xrightarrow{\phi'} \tau'' \backslash \emptyset}{\Gamma; \Delta'' \vdash \texttt{deref}\ : \tau''\ \texttt{ref}\ ^{\phi} \xrightarrow{\phi'} \tau'' \backslash \emptyset} \qquad \Gamma; \Delta'' \vdash r' : \tau''\ \texttt{ref}\ ^{\phi} \backslash C'' \quad (1)}{\Gamma; \Delta'' \vdash \texttt{deref}\ r' : \tau'' \backslash C''}$$

Suppose ϕ in type judgement (1) in the derivation above is ϵ. By Lemma 5.2 it follows that $\mathit{Reach}^{\Delta'', S''}(r') = \emptyset$. As Δ'' respects S'', by definition of respects, we have that $\Gamma; \Delta'' \vdash S''(r') : \tau'' \backslash C'''$ and $\textbf{esc?}(\tau'')$. By Lemma 5.2 we then have that $\mathit{Reach}^{\Delta'', S''}(S''(r')) = \emptyset$ which, as $S''(r') = V$, is the same as

$$\mathit{Reach}^{\Delta'', S''}(V) = \emptyset. \tag{5.10}$$

Suppose now that $\phi = \ell$ or $\phi = \kappa$. By the way we define *Reach* we have that

$$Reach^{\Delta'',S''}(r') = \{r'\} \cup Reach^{\Delta'',S''}(V). \tag{5.11}$$

So in either case, by 5.10 and 5.11 we have

$$Reach^{\Delta'',S''}(r') \supseteq Reach^{\Delta'',S''}(V). \tag{5.12}$$

We can then prove that:

$$\begin{array}{lll} & Reach_{\Delta'',S''}^{\Pi'''[p:\rho,E[\mathtt{deref}\,r']]}(\rho) & \\ \supseteq & Reach_{\Delta'',S''}^{\Pi'''[p:\rho,E[V]]}(\rho) & \text{Lemma 5.4 and (5.12)} \\ = & Reach_{\Delta',S''}^{\Pi'''[p:\rho,E[V]]}(\rho) & \text{Lemma 5.7} \\ = & Reach_{\Delta',S'}^{\Pi'''[p:\rho,E[V]]}(\rho) & S' = S''. \end{array}$$

case Assign

The last evaluation step is:

$$S'', \Pi'''[p\colon \rho, E[r' := V]] \underset{\mathsf{con}}{\longrightarrow} S''[r' \mapsto V], \Pi'''[p\colon \rho, E[\mathtt{unit}]] \quad r' \in Dom(S'').$$

By the Concurrent Type Preservation theorem, $\Gamma; \Delta'' \vdash E[r' := V] : \tau \backslash C$ for some $\tau \backslash C$. By the Context Filler lemma there is C' such that $\Gamma; \Delta'' \vdash r' := V : \mathtt{unit} \backslash C'$ and its proof must have been of this form:

$$\dfrac{\dfrac{\mathbf{typeof}(:=) \succ \ldots}{\Gamma; \Delta'' \vdash := : \ldots} \qquad \dfrac{\Gamma; \Delta'' \vdash r' : \tau_1\,\mathtt{ref}^{\,\phi} \backslash C_1 \;\; (1) \qquad \Gamma; \Delta'' \vdash V : \tau_1 \backslash C_2 \;\; (2)}{\Gamma; \Delta'' \vdash (r', V) : \tau_1\,\mathtt{ref}^{\,\phi} \times^{\phi'} \tau_1 \backslash C_1 \cup C_2 \cup \{\phi' \leq \phi\} \cup C_3}}{\Gamma; \Delta'' \vdash r' := V : \mathtt{unit} \backslash C_1 \cup C_2 \cup \{\phi' \leq \phi\} \cup C_3}$$

where $C_3 = \{\phi' \leq \mathbf{lab}\ (\tau_1)\}$ or $C_3 = \emptyset$ if $\mathbf{lab}\ (\tau_1)$ is undefined.

We now proceed by case analysis on ϕ of the type judgement (1) from the derivation above.

case $\phi = \epsilon$

By Lemma 5.2 and premise marked with (1) it follows that

$$Reach^{\Delta'',S''}(r') = \emptyset. \tag{5.13}$$

As Δ'' respects S'', by definition of respects we have that

$$\Gamma; \Delta'' \vdash S''(r') : \tau_1 \backslash C''' \textit{ and } \mathbf{esc?}(\tau_1). \tag{5.14}$$

By Lemma 5.2 and (5.14) we have that

$$Reach^{\Delta'',S''}(S''(r')) = \emptyset. \tag{5.15}$$

Also by Lemma 5.2 and type judgement (2) in the proof above we have that

$$Reach^{\Delta'',S''}(V) = \emptyset. \tag{5.16}$$

By (5.13) and (5.16) one can easily check that

$$Reach^{\Delta'',S''}(r' := V) = Reach^{\Delta'',S''}(\texttt{unit}). \tag{5.17}$$

We can then have:

$$\begin{array}{lll} & Reach^{\Pi'''[p:\rho,E[r':=V]]}_{\Delta'',S''}(\rho) & \\ = & Reach^{\Pi'''[p:\rho,E[\texttt{unit}]]}_{\Delta'',S''}(\rho) & \text{Lemma 5.3(2) and (5.17)} \\ = & Reach^{\Pi'''[p:\rho,E[\texttt{unit}]]}_{\Delta',S''}(\rho) & \text{Lemma 5.7} \\ = & Reach^{\Pi'''[p:\rho,E[\texttt{unit}]]}_{\Delta',S''[r'\mapsto V]}(\rho) & \text{Lemma 5.8} \end{array}$$

and for $\rho' \neq \rho$:

$$\begin{array}{lll} & Reach^{\Pi'''[p:\rho,E[r':=V]]}_{\Delta'',S''}(\rho') & \\ = & Reach^{\Pi'''[p:\rho,E[\texttt{unit}]]}_{\Delta'',S''}(\rho') & \rho \neq \rho' \\ = & Reach^{\Pi'''[p:\rho,E[\texttt{unit}]]}_{\Delta',S''}(\rho') & \text{Lemma. 5.7} \\ = & Reach^{\Pi'''[p:\rho,E[\texttt{unit}]]}_{\Delta',S''[r'\mapsto V]}(\rho') & \text{Lemma 5.8} \end{array}$$

case $\phi = \ell$ or $\phi = \kappa$

We first consider the case for the processor ρ involved in the transition. Suppose that $V' = S''(r')$. We then have

$$\begin{array}{lll} & Reach^{\Pi'''[p:\rho,E[r':=V]]}_{\Delta'',S''[r'\mapsto V']}(\rho) & \\ = & Reach^{\Pi'''[p:\rho,E[r':=V]]}_{\Delta',S''[r'\mapsto V']}(\rho) & \text{Lemma 5.7} \\ \supseteq & Reach^{\Pi'''[p:\rho,E[r':=V]]}_{\Delta',S''[r'\mapsto V]}(\rho) & \text{see (*) below} \\ \supseteq & Reach^{\Pi'''[p:\rho,E[\texttt{unit}]]}_{\Delta',S''[r'\mapsto V]}(\rho) & \text{Lemma 5.4.} \end{array}$$

(*) The second step in the derivation above is justified by the fact that r' is not pointing to V' anymore. The resulting set can then be smaller because local references of V' might not be reachable from ρ anymore.

For ρ' not involved in the transition: because in Π'' the reference r' appears on processor ρ, by the induction hypotheses we have that

$$r' \notin Reach^{\Pi''}_{\Delta'',S''}(\rho'). \tag{5.18}$$

In other words, r' is not in any thread running on ρ'. We then have

$$\begin{array}{lll} & \mathit{Reach}_{\Delta'',S''}^{\Pi'''[p:\rho,E[r':=V]]}(\rho') & \\ = & \mathit{Reach}_{\Delta'',S''}^{\Pi'''[p:\rho,E[\mathtt{unit}]]}(\rho') & \rho' \neq \rho \\ = & \mathit{Reach}_{\Delta',S''}^{\Pi'''[p:\rho,E[\mathtt{unit}]]}(\rho') & \text{Lemma 5.7} \\ = & \mathit{Reach}_{\Delta',S''[r'\mapsto V]}^{\Pi'''[p:\rho,E[\mathtt{unit}]]}(\rho') & (5.18) \end{array}$$

case Ref

The last evaluation step has this form:

$$S'', \Pi'''[p:\rho, E[\mathtt{ref}\ V]] \underset{\mathsf{con}}{\longrightarrow} S''[r' \mapsto V], \Pi'''[p:\rho, E[r']] \quad r'\ \mathit{new}$$

For $\rho' \neq \rho$:

$$\begin{array}{lll} & \mathit{Reach}_{\Delta'',S''}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho') & \\ = & \mathit{Reach}_{\Delta',S''}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho') & \text{Lemma 5.7} \\ = & \mathit{Reach}_{\Delta',S''[r'\mapsto V]}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho') & \text{Lemma 5.8} \\ = & \mathit{Reach}_{\Delta',S''}^{\Pi'''[p:\rho,E[r']]}(\rho') & \rho \neq \rho' \end{array}$$

For the processor ρ involved in the transition we first consider the case when $\phi = \epsilon$. One can easily proof that

$$\mathit{Reach}^{\Delta',S'}(r') = \mathit{Reach}^{\Delta',S'}(\mathtt{ref}\ V) = \emptyset. \tag{5.19}$$

We then have

$$\begin{array}{lll} & \mathit{Reach}_{\Delta'',S''}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho) & \\ = & \mathit{Reach}_{\Delta',S''}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho) & \text{Lemma 5.7} \\ = & \mathit{Reach}_{\Delta',S''[r'\mapsto V]}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho) & \text{Lemma 5.8} \\ = & \mathit{Reach}_{\Delta',S''[r'\mapsto V]}^{\Pi'''[p:\rho,E[r']]}(\rho) & \text{Lemma 5.3(2) and 5.19} \end{array}$$

For the case when $\phi = \ell$ or ϕ is a locality variable we have

$$\begin{array}{lll} & \mathit{Reach}_{\Delta'',S''}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho)\ \cup\ \{r'\} & \\ = & \mathit{Reach}_{\Delta',S''}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho)\ \cup\ \{r'\} & \text{Lemma 5.7} \\ = & \mathit{Reach}_{\Delta',S'}^{\Pi'''[p:\rho,E[\mathtt{ref}\ V]]}(\rho)\ \cup\ \{r'\} & \text{Lemma 5.8} \\ = & \mathit{Reach}_{\Delta',S'}^{\Pi'''[p:\rho,E[V]]}(\rho)\ \cup\ \{r'\} & \text{Lemma 5.3(2)} \\ = & \mathit{Reach}_{\Delta',S'}^{\Pi'''[p:\rho,E[\mathtt{unit}]]}(\rho)\ \cup\ \mathit{Reach}^{\Delta',S'}(V)\ \cup\ \{r'\} & \text{Lemma 5.6} \\ = & \mathit{Reach}_{\Delta',S'}^{\Pi'''[p:\rho,E[\mathtt{unit}]]}(\rho)\ \cup\ \mathit{Reach}^{\Delta',S'}(r') & \text{def. of } \mathit{Reach} \\ = & \mathit{Reach}_{\Delta',S'}^{\Pi'''[p:\rho,E[r']]}(\rho) & \text{Lemma 5.6} \end{array}$$

∎

An immediate consequence of the previous lemma is that if a local reference is reachable from just one processor in a configuration it remains so along evaluation, but that lemma does not say anything about whether the processor where the local reference is reachable from is the same.

Lemma 5.10 *Let S, Π be a configuration such that all threads in Π are well-typed under a store typing context Δ that respects S given Γ, a term type context. Suppose that $\mathit{Reach}^{\Pi}_{\Delta,S}(\rho) \cap \mathit{Reach}^{\Pi}_{\Delta,S}(\rho') = \emptyset$ for all distinct $\rho, \rho' \in \mathit{ProcSet}(\Pi)$. Suppose also that $S, \Pi \underset{\mathsf{con}}{\longrightarrow}^{*} S', \Pi'$. Then for any $\rho \in \mathit{ProcSet}(\Pi)$, $\rho' \in \mathit{ProcSet}(\Pi')$, $r \in \mathit{Reach}^{\Pi}_{\Delta,S}(\rho)$ and $r \in \mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho')$, where $\Delta' \supseteq \Delta$ respects S', implies $\rho = \rho'$.*

Proof. By induction on the length of the reduction sequence. For a sequence of length zero the result holds by assumption. For the induction consider just the last evaluation step. Suppose $S, \Pi \underset{\mathsf{con}}{\longrightarrow}^{*} S'', \Pi''$ and let the last step be $S'', \Pi'' \underset{\mathsf{con}}{\longrightarrow} S', \Pi'$.

By Lemma 5.9 for any $\Delta'' \supseteq \Delta$ respecting S'' the following holds

$$\textit{for all distinct } \rho, \rho' \in \mathit{ProcSet}(\Pi''), \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho) \cap \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho') = \emptyset. \quad (5.20)$$

Suppose $r \in \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$ for $\rho \in \mathit{ProcSet}(\Pi'')$. By 5.20 we know that $r \notin \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho')$ for any $\rho' \neq \rho$. So $r \in \mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho')$ in the right side of a transition for $\rho' \neq \rho$ only if there is a transition that modifies the set of local reachable references of processor ρ' so to include r.

We now analyse each transition showing that none satisfies the above requirement, allowing us to conclude that if $r \in \mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho')$ in the right side of a transition then $\rho' = \rho$

Transitions Seq with sequential transitions Succ, Pred, Zero-True and Zero-False are excluded. That is because $\mathit{ProcSet}(\Pi') = \mathit{ProcSet}(\Pi'')$ and, as we saw in the proof of the previous lemma (Lemma5.9), no local reachable set is modified by this transitions, that is, $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$ for all $\rho \in \mathit{ProcSet}(\Pi')$.

The transitions Seq with premises Fst, Snd, Cond-True, Cond-False, β, Rec-Unroll, Let, Deref and Assign are also excluded. For these transitions $\mathit{ProcSet}(\Pi') = \mathit{ProcSet}(\Pi'')$ and, in the proof of the previous lemma, we demonstrated that the sets of local reachable references of processors not involved are *not* modified, that is, $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) = \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$ for all $\rho \in \mathit{ProcSet}(\Pi')$ not involved in the transition. The set of local reachable references of a processor ρ involved can be modified though. But for these transitions we proved in the previous lemma that $\mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho) \subseteq \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho)$. That is, if modified after the transition, the set can only become smaller so $r \notin \mathit{Reach}^{\Pi'}_{\Delta',S'}(\rho)$.

If the Seq transition has Ref as premise. We also know from the previous lemma that for the processors ρ not involved in the transition $Reach^{\Pi'}_{\Delta',S'}(\rho) = Reach^{\Pi''}_{\Delta'',S''}(\rho)$. The processor involved has its set of local reachable references modified but only with the inclusion of a *new* reference.

The transition Fork is also excluded. For this case $ProcSet(\Pi') = ProcSet(\Pi'')$ and we prove in the previous lemma that $Reach^{\Pi'}_{\Delta',S'}(\rho) = Reach^{\Pi''}_{\Delta'',S''}(\rho')$ for all $\rho, \rho' \in ProcSet(\Pi')$, that is, no local reachable set is modified.

For the transition Rfork we also demonstrated in the previous lemma that the sets of local reachable references of processors not involved and of the processor where the *rfork* is executed are not modified. For the processor where the new thread is created we proved that its set of local reachable references is empty. ∎

5.3 Locality Soundness

Lemma 5.11 *Let P be a well-typed program, T be any trace in $Comp(P)$, S_i, Π_i be a configuration of T, and r a reference constant. Suppose that $r \in Reach^{\Pi_i}_{\Delta_i,S_i}(\rho)$ for some $\rho \in ProcSet(\Pi_i)$ and Δ_i respecting S_i. Suppose also S_j, Π_j is a configuration of T with $j \geq i$. Then, for any $\Delta_j \supseteq \Delta_i$, respecting S_j, $r \in Reach^{\Pi_j}_{\Delta_j,S_j}(\rho')$ implies $\rho' = \rho$.*

Proof. By Definition 2.5, all traces in $Comp(P)$ start with a configuration S_0, Π_0 where S_0 is the empty store and $\Pi_0 = [p_0 : (\rho_0, P)]$. So the fact

$$\textit{all threads in } \Pi_0 \textit{ are well-typed} \tag{5.21}$$

holds by assumption, and the fact

$$\textit{for all distinct } \rho, \rho' \in Dom(\Pi_0), Reach^{\Pi_0}_{S_0,\Delta_0}(\rho) \cap Reach^{\Pi_0}_{S_0,\Delta_0}(\rho') = \emptyset \tag{5.22}$$

holds trivially for any type context Δ_0.

By the Concurrent Type Preservation theorem, there exists Δ_i respecting S_i such that $\Delta_i \supseteq \Delta_0$. Hence, by Lemma 5.9 with (5.21) and (5.22) above we get that

$$\textit{for all distinct } \rho, \rho' \in Dom(\Pi_i), Reach^{\Pi_i}_{\Delta_i,S_i}(\rho) \cap Reach^{\Pi_i}_{\Delta_i,S_i}(\rho') = \emptyset. \tag{5.23}$$

By the Concurrent Type Preservation theorem there is Δ_j respecting S_j, $\Delta_j \supseteq \Delta_i$. Hence by (5.23) above and Lemma 5.10 we have that if $r \in Reach^{\Pi_i}_{\Delta_i,S_i}(\rho)$ and $r \in Reach^{\Pi_j}_{\Delta_j,S_j}(\rho')$ then $\rho = \rho'$. ∎

Finally we can state and prove the main result of this chapter. In the following, we write *subterm of a thread* meaning *subterm of the term associated to a thread.*

Theorem 5.1 (Locality Soundness) *Let P be a well-typed program, T be any trace in Comp(P). Suppose that the first occurrence of a constant r in T is in configuration S_i, Π_i as a subterm of a thread running on processor ρ. Suppose also that* $\mathbf{lab}\,(\Delta_i(r)) \neq \epsilon$ *where Δ_i is a store typing constant respecting S_i. If later on the same trace the reference r appears as subterm of a thread running on ρ', then $\rho' = \rho$.*

Proof. By the previous lemma and by the definition of *Reach*. ∎

Chapter 6

Translation

In this chapter we illustrate how the locality information provided by the types can be used to translate a program into another where store operations with local references run faster.

We start by presenting, in Section 6.1, the target language for the translation. We then give an operational semantics for it at a level of abstraction that distinguishes local and non-local operations on the store.

In Section 6.2 we define a translation from well typed source expressions to expressions in the target language and we prove, in Section 6.3, that the translation preserves the behaviour of source programs.

6.1 Target Language and its Semantics

The terms of the target language are given by the following grammar

$$\begin{array}{rcl} m & ::= & x \\ & \mid & \mathtt{c} \\ & \mid & \mathtt{fn}\, x.\, m \\ & \mid & \mathtt{rec}\, f\, x.\, m \\ & \mid & m\ m' \\ & \mid & (m, m') \\ & \mid & \mathtt{if}\ m\, \mathtt{then}\ m'\, \mathtt{else}\ m'' \\ & \mid & \mathtt{let}\ x = m\, \mathtt{in}\ m'\, \mathtt{end}\ . \end{array}$$

In the grammar above the metavariable $\mathtt{c}$ ranges over the constants given by the grammar

$$\mathtt{c} \ ::= \ c \mid \mathtt{deref}^{\,\ell} \mid :=^{\ell} \mid \mathtt{ref}^{\,\ell}$$

where c is a metavariable ranging over the set of constants of the *source* language. The target language is then the same as the source language given in Chapter 2 with $\mathtt{deref}^{\,\ell}$ and $:=^{\ell}$, the local versions of dereference and assignment operations respectively, and $\texttt{ref}^{\,\ell}$, a constant used to create local references, added to the set of constants.

As in the source language, *values* are the following irreducible terms: free variables, constants, functions, recursive functions and pairs of values. We use v as a metavariable ranging over values in the target language. We also write $m :=^{\ell} m'$ for the application of the local assignment operator $:=^{\ell}$ to the pair (m, m').

We use the notation $e[]$ for evaluation contexts, which are defined in the same way as evaluation contexts for the source language. Non-value expressions can also be uniquely decomposed as an evaluation context with a redex filling its hole.

The operational semantics is divided in two parts. The sequential part, shown in Figure 6.1, is much the same as the sequential semantics for the source language, the difference is that the rules that modify the store are in the concurrent part shown in Figure 6.2.

As before, a thread component map is a finite, non-empty map from thread identifiers to thread components. We write π for a thread map, instead of Π, to remind us that a thread component is a pair (ρ, m) where ρ identifies the processor where the thread component runs and m is a term from the *target* language.

As expected the novelty is in the way the store and the rules that modify it are defined. A *store* is a finite, possibly empty map from references to *triples*.

The first component of these triples is what we call a locality indicator taken from a set $\{\mathsf{E}, L_\rho, L_{\rho'}, \ldots\}$ of locality indicators constants. If a reference is created by the default $\texttt{ref}$ operation the first position of its triple has the locality indicator E. If a reference is created by the $\texttt{ref}^{\,\ell}$ operation the first element in the triple is the locality indicator L_ρ, where ρ is the idenfication of the processor where the $\texttt{ref}^{\,\ell}$ operation is executed. We assume the existence of a supply of locality indicators $L_\rho, L_{\rho'}, \ldots$ one for each possible processor identifier $\rho, \rho', \ldots$. Observe that the first element in the triples is determined at the moment a reference is created and is not modified by any transition.

The second component of a triple is a set of processor identifiers that informs which processors have the most recent version of the data pointed by a reference.

Immediately after creation (ref and ref^{ℓ} transitions) a reference is available only in the processor where the creation occurred.

The ways in which the set of processors is updated by operations with store represent the differences between local and global dereference operations and between local and global assignment operations. The updating of the set of processor identifiers in the rules deref and assign for global dereference and global succ:

$e[\,\texttt{succ}\ n_i] \underset{seq}{\longrightarrow} e[n_{i+1}]$

pred:

$e[\,\texttt{pred}\ n_i] \underset{seq}{\longrightarrow} e[n_{i-1}]$

zero-true:

$e[\,\texttt{zero?}\ 0] \underset{seq}{\longrightarrow} e[\texttt{true}]$

zero-false:

$e[\,\texttt{zero?}\ n] \underset{seq}{\longrightarrow} e[\texttt{false}] \quad n \neq 0$

fst:

$e[\,\texttt{fst}\ \texttt{(}v, v'\texttt{)}] \underset{seq}{\longrightarrow} e[v]$

snd:

$e[\,\texttt{snd}\ \texttt{(}v, v'\texttt{)}] \underset{seq}{\longrightarrow} e[v']$

β:

$e[(\texttt{fn}\,x.m)\ v] \underset{seq}{\longrightarrow} e[m[v/x]]$

rec-unroll:

$e[(\texttt{rec}\,f\,x.m)\ v] \underset{seq}{\longrightarrow} e[(\texttt{fn}\,x.m)[\texttt{rec}\,f\,x.m/f]\ v]$

cond-true:

$e[\,\texttt{if true then}\ m\,\texttt{else}\ m'] \underset{seq}{\longrightarrow} e[m]$

cond-false:

$e[\,\texttt{if false then}\ m\,\texttt{else}\ m'] \underset{seq}{\longrightarrow} e[m']$

let:

$e[\,\texttt{let}\ x = v\ \texttt{in}\ m\,\texttt{end}\ ] \underset{seq}{\longrightarrow} e[m[v/x]]$

Figure 6.1: Sequential evaluation for the target language

seq:

$$\frac{e[m] \underset{seq}{\longrightarrow} e[m']}{s, \pi[\,p : \rho, e[m]] \underset{con}{\longrightarrow} s, \pi[\,p : \rho, e[m']]}$$

ref:

$$\frac{r \notin Dom(s) \quad Refs\,(v) \cap Local\,(\rho, s) = \emptyset}{s, \pi[\,p : \rho, e[\,\texttt{ref}\ \ v]] \underset{con}{\longrightarrow} s[r \mapsto E, \{\rho\}, v], \pi[\,p : \rho, e[r]]}$$

deref:

$$\frac{\forall \rho' \neq \rho.\ Refs\,(v) \cap Local\,(\rho', s) = \emptyset}{s[r \mapsto t, \mathcal{P}, v], \pi[\,p : \rho, e[\,\texttt{deref}\ \ r]] \underset{con}{\longrightarrow} s[r \mapsto t, \mathcal{P} \cup \{\rho\}, v], \pi[\,p : \rho, e[v]]}$$

assign:

$$\frac{t = L_\rho \quad \vee \quad Refs\,(v) \cap Local\,(\rho, s) = \emptyset}{s[r \mapsto t, \mathcal{P}, v'], \pi[\,p : \rho, e[r := v]] \underset{con}{\longrightarrow} s[r \mapsto t, \{\rho\}, v], \pi[\,p : \rho, e[\texttt{unit}]]}$$

ref$^\ell$:

$$\frac{r \notin Dom(s)}{s, \pi[\,p : \rho, e[\,\texttt{ref}\ \ v]] \underset{con}{\longrightarrow} s[r \mapsto L_\rho, \{\rho\}, v], \pi[\,p : \rho, e[r]]}$$

deref$^\ell$:

$$\frac{\forall\, \rho' \neq \rho.\ Refs\,(v) \cap Local\,(\rho', s) = \emptyset}{s[r \mapsto L_\rho, \{\rho\}, v], \pi[\,p : \rho, e[\texttt{deref}^{\,\ell}\ \ r]] \underset{con}{\longrightarrow} s[r \mapsto L_\rho, \{\rho\}, v], \pi[\,p : \rho, e[v]]}$$

assign$^\ell$:

$$\frac{}{s[r \mapsto L_\rho, \{\rho\}, v'], \pi[\,p : \rho, e[r :=^\ell v]] \underset{con}{\longrightarrow} s[r \mapsto L_\rho, \{\rho\}, v], \pi[\,p : \rho, e[\texttt{unit}]]}$$

fork:

$$\frac{q \notin Dom(\Pi) \cup \{p\}}{s, \pi[\,p : \rho, e[\,\texttt{fork}\ \texttt{fn}\, x.m]] \underset{con}{\longrightarrow} s, \pi[\,p : \rho, e[\texttt{unit}]][\,q : \rho, m[\texttt{unit}/x]]}$$

rfork:

$$\frac{q \notin Dom(\Pi) \cup \{p\} \quad \forall\, \rho'' \neq \rho'.\ Refs\,(m[\texttt{unit}/x]) \cap Local\,(\rho'', s)}{s, \pi[\,p : \rho, e[\,\texttt{rfork}_{\rho'}\ \ \texttt{fn}\, x.m]] \underset{con}{\longrightarrow} s, \pi[\,p : \rho, e[\texttt{unit}]][\,q : \rho', m[\texttt{unit}/x]]}$$

Figure 6.2: Thread map evaluation for the target language

assignment represents the cost involved in keeping the coherency of the distributed copies of mutable data (the *coherency problem* explained in Chapter 1.)

In the deref transition the identifier of the processor where the dereferencing operation is executed is added to the set of processor identifiers. This represents the *validation* of the copy associated to that processor.

After a global assignment (assign transition), the set of processor identifiers associated with the reference is updated to contain only the identifier of the processor where the assignment operation is executed. We can see this as modelling the fact that all the copies of the data pointed by the reference are *invalid* after the assignment, except the copy associated with the processor where the assignment operation is executed.

Finally the third component of the triple is a value from the target language. We use s, t and $\mathcal{P}$ as metavariables for stores, locality indicators and set of processor identifiers, respectively. Sometimes we call s a *target store*, π a *target thread map*, and a pair s, π a *target configuration*.

In the rules deref$^{\ell}$ and assign$^{\ell}$, for the local versions of dereference and assignment operations, the set of processor identifiers associated to the reference is the same before and after the transition, and its single element identifies the processor where the operation is executed. This represents the fact that local versions of dereference and assignment operations are less costly than the global ones because there is no need for a coherency protocol.

Note also that the set of processor identifiers in the range of the store should not be confused with the processors where references have escaped to. It is possible that a reference might have escaped to several processors but only a few of them have the most updated version of the reference's value.

Some rules also make use of the function *Local* defined as follows:

$$Local(\rho, s) = \{r \in Dom(s) \mid s(r) = (L_\rho, \mathcal{P}, v)\}$$

where $\mathcal{P}$ and v are some processor set and target value, respectively. A set $Local(\rho, s)$ is then the set of references in the domain of store s which were created by a `ref` $^{\ell}$ operation executed on processor ρ. We say that a reference created by `ref` $^{\ell}$ on processor ρ is *local to* ρ.

The specialised `deref` $^{\ell}$ and $:=^{\ell}$ operations do not work properly if their reference operand is not local. That is represented in their rules by the locality indicator in store. Similar restriction is not needed in the rules for `deref` and $:=$ because the default operations have *full power* to access and modify a distributed shared memory in a consistent way.

Now lets consider the rules assign and ref. After a reference is created or an assignment is executed the effect is that a reference will be pointing to a value in the store. The ref operation creates references that can escape so the value pointed by the reference cannot have references local to the processor where `ref` is executed. This fact is represented by that rule's proviso.

In the rule for := the proviso has the same role. If r is a reference that was created by a `ref` operation ($t \neq L_\rho$) then r can escape. The assignment can then only be executed if the references of v are not local to ρ.

The rule seq, fork and rfork in Figure 6.2 are very similar to their counterparts in the semantics for the source language. The rfork rule also requires that no local references be in the *rforked* term.

6.2 Translation

In Chapter 2 we defined a language and its operational semantics, and in Chapter 3 we gave a type system for it where types are labelled with locality information. In this section we illustrate how we can use a type derivation for a term from the source language to guide a translation for the target language defined in the previous section.

It is convenient to have a *linear* representations of type derivation trees in the form of *explicitly typed expressions*. In Figure 6.3 we present a function T from type derivations to explicitly typed expressions. We use ∇, ∇', ... as metavariables for type derivations. The $*$ in clauses for functions and recursive function represents the possible occurrence of the 3-place predicate **esc?**.

We use the notation $\Gamma; \Delta \vdash \overline{M}_\gamma$ for the linear representation of a type derivation for the type judgement $\Gamma; \Delta \vdash M : \gamma$, or simply $\overline{M}$, if we do not need to make explicitly the type contexts Γ and Δ, and the outermost type γ. In order to simplify the definition of the translation we assume that in all type derivations the set of free locality variables is distinct from the set of the bound locality variables. This is not a restrictive assumption because we can always change the names of bound locality variables.

The translation of a term M based on a type derivation $\overline{M}$ consists simply of:

- replacing occurrences of $\mathtt{deref} : \tau\,\mathtt{ref}^{\,\phi} \xrightarrow{\phi'} \tau \backslash C$, with $\mathtt{deref}^{\,\ell}$, whenever $\phi = \ell$ or ϕ is a locality variable that does not occur bound in $\overline{M}$,
- replacing occurrences of $:=: \tau\,\mathtt{ref}^{\,\phi} \times^{\phi'} \tau \xrightarrow{\phi''} \mathtt{unit} \backslash C$, with $:=^{\ell}$, whenever $\phi = \ell$ or ϕ is a locality variable that does not occur bound in $\overline{M}$, and
- replacing occurrences of $\mathtt{ref} : \tau \xrightarrow{\phi'} \tau\,\mathtt{ref}^{\,\phi} \backslash C$, with $\mathtt{ref}^{\,\ell}$, whenever $\phi = \ell$ or ϕ is a locality variable that does not occur bound in $\overline{M}$.

We define a translation Φ in the following way:

$$\Phi(\overline{M}) \;=\; \Phi'(\overline{M}, \{\})$$

$$
\begin{array}{rcl}
\mathsf{T}\left(\dfrac{\textbf{typeof}(c) \succ \gamma}{\Gamma;\Delta \vdash c\colon \gamma}\right) & = & c : \gamma \\[2ex]
\mathsf{T}\left(\dfrac{\Gamma(x) \succ \gamma}{\Gamma;\Delta \vdash x\colon \gamma}\right) & = & (x : \Gamma(x)) : \gamma \\[2ex]
\mathsf{T}\,(\,\Gamma;\Delta \vdash r : \gamma\,) & = & r : \gamma \\[2ex]
\mathsf{T}\left(\dfrac{\nabla \quad *}{\Gamma;\Delta \vdash \mathtt{fn}\, x.M : \gamma}\right) & = & (\mathtt{fn}\, x : \gamma' .\, \mathsf{T}\,(\nabla)) : \gamma \\
 & & \quad \textit{where } \nabla = \dfrac{\nabla'}{\Gamma[x \mapsto \gamma'];\Delta \vdash M\colon \gamma''} \\[2ex]
\mathsf{T}\left(\dfrac{\nabla \quad *}{\Gamma;\Delta \vdash \mathtt{rec}\, f\, x.M : \gamma}\right) & = & (\mathtt{rec}\, f\colon \gamma'\, x\colon \gamma'' .\, \mathsf{T}\,(\nabla)) : \gamma \\
 & & \quad \textit{where } \nabla = \dfrac{\nabla'}{\Gamma[f \mapsto \gamma'][x \mapsto \gamma''];\Delta \vdash M : \gamma'''} \\[2ex]
\mathsf{T}\left(\dfrac{\nabla \quad \nabla'}{\Gamma;\Delta \vdash \mathtt{(}M_1, M_2\mathtt{)} : \gamma}\right) & = & \mathtt{(}\mathsf{T}\,(\nabla), \mathsf{T}\,(\nabla')\mathtt{)} : \gamma \\[2ex]
\mathsf{T}\left(\dfrac{\nabla \quad \nabla'}{\Gamma;\Delta \vdash M_1 \;\; M_2\colon \gamma}\right) & = & (\mathsf{T}\,(\nabla)\;\; \mathsf{T}\,(\nabla')) : \gamma \\[2ex]
\mathsf{T}\left(\dfrac{\nabla \quad \nabla' \quad \nabla''}{\Gamma;\Delta \vdash \mathtt{if}\; M\, \mathtt{then}\; N\, \mathtt{else}\; P\colon \gamma}\right) & = & (\mathtt{if}\;\, \mathsf{T}\,(\nabla)\, \mathtt{then}\;\, \mathsf{T}\,(\nabla')\, \mathtt{else}\;\, \mathsf{T}\,(\nabla'')) : \gamma \\[2ex]
\mathsf{T}\left(\dfrac{\nabla \qquad \nabla'}{\Gamma;\Delta \vdash \mathtt{let}\;\; x = M\, \mathtt{in}\; N\, \mathtt{end} : \gamma}\right) & = & (\mathtt{let}\;\; x : \sigma = \mathsf{T}\,(\nabla)\, \mathtt{in}\;\, \mathsf{T}\,(\nabla')\, \mathtt{end}\,) : \gamma \\
 & & \quad \textit{where } \nabla' = \dfrac{\nabla''}{\Gamma[x \mapsto \sigma];\Delta \vdash M\colon \gamma'}
\end{array}
$$

Figure 6.3: Linear represention of type derivations

$$
\begin{array}{rcl}
\Phi'(r:\gamma,B) & = & r \\
\Phi'((x:\sigma):\gamma,B) & = & x \\
\Phi'(c:\gamma,B) & = & c \;\; \textit{for } c \notin \{\texttt{deref}\,, := \,\texttt{ref}\,\} \\
\Phi'(\texttt{deref}\,:\tau\,\texttt{ref}^{\,\phi} \xrightarrow{\phi'} \tau\backslash C, B) & = & \left\{ \begin{array}{ll} \texttt{deref} & \textit{if } \phi = \epsilon \textit{ or } \phi \in B \\ \texttt{deref}^{\,\ell} & \textit{otherwise} \end{array} \right. \\
\Phi'(:=:\tau\,\texttt{ref}^{\,\phi} \xrightarrow{\phi'} \tau\backslash C, B) & = & \left\{ \begin{array}{ll} := & \textit{if } \phi = \epsilon \textit{ or } \phi \in B \\ :=^{\ell} & \textit{otherwise} \end{array} \right. \\
\Phi'(\texttt{ref}\,:\tau \xrightarrow{\phi'} \tau\,\texttt{ref}^{\,\phi}\backslash C, B) & = & \left\{ \begin{array}{ll} \texttt{ref} & \textit{if } \phi = \epsilon \textit{ or } \phi \in B \\ \texttt{ref}^{\,\ell} & \textit{otherwise} \end{array} \right. \\
\Phi'((\texttt{fn}\,x:\sigma.\overline{N}):\gamma,B) & = & \texttt{fn}\,x.\Phi'(\overline{N},B) \\
\Phi'((\texttt{rec}\,f:\sigma\,x:\sigma'.\overline{N}):\gamma,B) & = & \texttt{rec}\,f\,x.\Phi'(\overline{N},B) \\
\Phi'((\overline{N}\;\overline{P}):\gamma,B) & = & \Phi'(\overline{N},B)\;\Phi'(\overline{P},B) \\
\Phi'(((\overline{N},\overline{P})):\gamma,B) & = & (\Phi'(\overline{N},B),\Phi'(\overline{P},B)) \\
\Phi'((\texttt{if}\;\overline{N}\,\texttt{then}\;\overline{P}\,\texttt{else}\;\overline{Q}):\gamma,B) & = & \texttt{if}\;\Phi'(\overline{N},B)\,\texttt{then}\;\Phi'(\overline{P},B)\,\texttt{else}\;\Phi'(\overline{Q},B) \\
\Phi'((\texttt{let}\;x:\gamma=\overline{N}\,\texttt{in}\;\overline{P}\,\texttt{end}\,):\gamma',B) & = & \texttt{let}\;x=\Phi'(\overline{N},B)\,\texttt{in}\;\Phi'(\overline{P},B)\,\texttt{end} \\
\Phi'((\texttt{let}\;x:\forall\vec{\alpha}\vec{\kappa}.\gamma=\overline{V}\,\texttt{in}\;\overline{P}\,\texttt{end}\,):\gamma',B) & = & \texttt{let}\;x=\Phi'(\overline{V},\lceil\vec{\kappa}\rceil\cup B)\,\texttt{in}\;\Phi'(\overline{P},B)\,\texttt{end}
\end{array}
$$

Figure 6.4: Translation

where Φ', given in Figure 6.4, takes a linear representation of a type derivation tree for a term M, a set B of locality variables and returns a term in the target language. The interesting clauses of Φ' are those for the constants `deref` , := and `ref` . Observe that the translation Φ of a term starts with an empty set of locality variables. Variables are introduced in this set by the clause for `let` expressions binding a variable to a value.

As we said before, one of the purposes of this chapter is to illustrate how the locality information labelling types can be used in a program translation. We assume the translation operates on original source expressions but that does not need to be the case. If types are kept through other program transformations, further translations based on locality information can be revealed.

Consider for example an expression like

$$\begin{array}{ll} \texttt{let} & d = \texttt{deref} \\ \texttt{in} & \ldots(d\,r)\ldots(d\,r')\ldots \\ \texttt{end}, & \end{array}$$

and suppose that r is a local integer reference and that r' is an escaping integer reference. The first occurrence of d has the type $\texttt{int ref}^{\,\ell} \xrightarrow{\phi} \texttt{int}$ and the second one has the type $\texttt{int ref}^{\,\epsilon} \xrightarrow{\phi'} \texttt{int}$, but no transformation is done by Φ. But suppose that we keep type and locality information and the expression above is transformed, using some kind of *inline expansion*, to

$$\ldots(\texttt{deref}\ \ r)\ldots(\texttt{deref}\ \ r')\ldots$$

The locality information could then be used by Φ to produce

$$\ldots(\texttt{deref}^{\,\ell}\ r)\ldots(\texttt{deref}\ \ r')\ldots$$

6.3 Correctness of Translation

We have to prove that the translation given in the previous section preserves the behaviour of original programs. Here, thread maps are evaluated non deterministically and evaluation may fail to terminate. Therefore our proof of correction of the translation Φ will be based on the notion of a *bisimulation relation* [19].

We define the set $\Phi(\Gamma;\Delta,\,M)$ of possible translations of a term M given Γ and Δ as follows

$$\Phi(\Gamma;\Delta,\,M) = \{m \ \mid \exists\ \Gamma;\Delta \vdash \overline{M}\ \mathit{and}\ m = \Phi(\Gamma;\Delta \vdash \overline{M})\}.$$

We say that the target term m is a translation of M if $m \in \Phi(\Gamma; \Delta, M)$. We extend this notion of translation to source stores as follows:

$$\begin{array}{rcl} \Phi(\Gamma, \Delta, S) & = & s \ \ \textit{such that} \\ & & \textit{Dom}(s) = \textit{Dom}(S) \ \textit{and for all} \ r \in \textit{Dom}(S) \\ & & \textit{if} \ S(r) = V \ \textit{then} \ s(r) = (t, \mathcal{P}, v), \ \textit{where} \ v \in \Phi(\Gamma, \Delta, V), \\ & & \quad t \ = \ \left\{ \begin{array}{ll} \mathsf{E} & \textit{if} \ \mathbf{esc?}(\Delta(r)) \\ L_\rho \ \textit{for some} \ \rho & \textit{otherwise} \end{array} \right. \\ & & \\ & & \quad \mathcal{P} \ = \ \left\{ \begin{array}{ll} \textit{any processor set} & \textit{if} \ \mathbf{esc?}(\Delta(r)) \\ \{\rho\}, \ \textit{for some} \ \rho & \textit{otherwise.} \end{array} \right. \end{array}$$

So

We also define the translation of a thread map in the following way:

$$\begin{array}{rcl} \Phi(\Gamma, \Delta, \ \Pi) & = & \pi \ \ \textit{such that} \\ & & \textit{Dom}(\pi) = \textit{Dom}(\Pi) \ \textit{and for all} \ p \in \textit{Dom}(\pi) \\ & & \textit{if} \ \Pi(p) = (\rho, M) \ \textit{then} \ \pi(p) = (\rho, m) \\ & & \textit{where} \ m \in \Phi(\Gamma, \Delta, M). \end{array}$$

Finally we combine the two previous definitions to define the set of translations of a source configuration as follows:

$$\Phi(\Gamma, \Delta, \ S, \Pi) = \{(s, \pi) \mid s \in \Phi(\Gamma, \Delta, \ S) \ \textit{and} \ \pi \in \Phi(\Gamma, \Delta, \ \Pi)\}.$$

At this point it might be useful to see the main result that we want to achieve in this chapter. In the following, we abbreviate the initial source configuration $\emptyset$, $[p_0 : \rho_0, P]$ and the initial target configuration $\emptyset$, $[p_0 : \rho_0, \mathsf{p}]$ by writing only P and p respectively. We also write $\Phi(P)$ for $\Phi(\Gamma, \Delta, \ P)$ when Γ and Δ are empty, both when P is a source program or when P stands for the initial configuration $\emptyset, [p_0 : \rho_0, P]$. The result we want to proof is then the following:

Let P be a well-typed source program and p be such that $\mathsf{p} \in \Phi(P)$. Then the following hold:

1. *if $P \xrightarrow[\mathsf{con}]{}^n \ S', \Pi'$ then there is a configuration s', π' such that $\mathsf{p} \xrightarrow[con]{}^n \ s', \pi'$ and $(s', \pi') \ \in \ \Phi(\Delta', \ S', \Pi')$ for Δ' respecting S', and*

2. *if $\mathsf{p} \xrightarrow[con]{}^n \ s', \pi'$ then there is a configuration S', Π' such that $P \xrightarrow[\mathsf{con}]{}^n \ S', \Pi'$ and $(s', \pi') \ \in \ \Phi(\Delta', \ S', \Pi')$ for Δ' respecting S'.*

In order to prove the result above, we need an auxiliary lemma. When we proved type preservation lemmas in Chapter 4 we prove that the *outermost* type of a term can be preserved. When proving the Sequential Type Preservation theorem we also proved that the outermost type of the term *filling* the evaluation context in the left hand side of the transition is preserved in the term filling the same evaluation context on the right side. But none of these two results say anything about preservation of other *internal types*. The proof of the lemma below involves showing that *internal types* can also be preserved by evaluation.

Lemma 6.1 *Let S, Π be a source configuration and Δ be a store type context respecting S given a term type context Γ. Suppose that all terms associated to thread components of Π are well-typed under Γ and Δ. Suppose also that $\mathit{Reach}^{\Pi}_{\Delta,S}(\rho) \cap \mathit{Reach}^{\Pi}_{\Delta,S}(\rho')$ for distinct ρ, ρ' in $\mathit{ProcSet}(\Pi)$.*

Now let s, π be a target configuration such that $(s, \pi) \in \Phi(\Gamma; \Delta, S, \Pi)$. Then the following hold:

1. *if $S, \Pi \xrightarrow[\mathsf{con}]{}^{n} S', \Pi'$, then there exist s', π' such that $s, \pi \xrightarrow[\mathit{con}]{}^{n} s', \pi'$ and $(s', \pi') \in \Phi(\Gamma; \Delta', S', \Pi')$ for some $\Delta' \supseteq \Delta$ respecting S' given Γ, and*
2. *if $s, \pi \xrightarrow[\mathit{con}]{}^{n} s', \pi'$ then there exist S', Π' such that $S, \Pi \xrightarrow[\mathsf{con}]{}^{n} S', \Pi'$ and $(s', \pi') \in \Phi(\Gamma; \Delta', S', \Pi')$ for some $\Delta' \supseteq \Delta$ respecting S' given Γ*

Proof. By induction on the length of the reduction sequence. We prove part *1* first. For a sequence of length zero the result holds by assumption. We assume a sequence $S, \Pi \xrightarrow[\mathsf{con}]{}^{n-1} S'', \Pi''$ and we consider the last step $S'', \Pi'' \xrightarrow[\mathsf{con}]{} S', \Pi'$. We proceed by case analysis on each possible transition $\xrightarrow[\mathsf{con}]{}$ from Figure 2.2.

For the Seq transition we consider each possible sequential transition it has as premise. The interesting cases are for transitions that modify the store, so for the transitions that *do not* modify the store we give the proof only for Seq with Succ as premise. The proofs for the other transitions follow the same lines.

case Seq with Succ

We have the following transition

$$S'', \Pi'''[p : \rho, E[\mathtt{succ}\ n_i]] \xrightarrow[\mathsf{con}]{} S'', \Pi'''[p : \rho, E[n_{i+1}]].$$

By the induction hypothesis and by the definition of Φ we have the following target configuration

$$(s'', \pi'''[p : \rho, e[\mathtt{succ}\ n_i]]) \in \Phi(\Gamma; \Delta'', S'', \Pi'''[p : \rho, E[\mathtt{succ}\ n_i]]) \quad (6.1)$$

where

$$s'' \in \Phi(\Gamma; \Delta'', S''), \quad (6.2)$$

$$\pi''' \in \Phi(\Gamma; \Delta'', \Pi'''), \quad (6.3)$$

and

$$e[\texttt{succ}\ n_i] \ \in\ \Phi(\Gamma; \Delta'',\ E[\texttt{succ}\ n_i]). \tag{6.4}$$

With p the selected thread in the target configuration of (6.1) we have the following transition

$$s'', \pi'''[p : \rho, e[\texttt{succ}\ n_i]] \underset{con}{\longrightarrow} s'', \pi'''[p : \rho, e[n_{i+1}]].$$

For this case $\Delta' = \Delta''$. By (6.2) and (6.3) we then get

$$s'' \ \in\ \Phi(\Gamma; \Delta',\ S'') \tag{6.5}$$

and

$$\pi''' \in\ \Phi(\Gamma; \Delta',\ \Pi'''). \tag{6.6}$$

We still have to prove that $e[n_{i+1}] \in \Phi(\Gamma, \Delta',\ E[n_{i+1}])$.

By (6.4) above, the fact that $\Delta'' = \Delta'$, and the definition of Φ we know that

$$\exists\ \Gamma; \Delta' \vdash \overline{E[\texttt{succ}\ n_i]}\ \textit{such that}\ e[\texttt{succ}\ n_i] = \Phi(\Gamma; \Delta' \vdash \overline{E[\texttt{succ}\ n_i]}) \tag{6.7}$$

which implies that

$$\exists\ \Gamma; \Delta' \vdash \overline{\texttt{succ}\ n_i}_{\texttt{int}\backslash\emptyset}. \tag{6.8}$$

For the value n_{i+1} we can have the following type derivation

$$\Gamma; \Delta' \vdash n_{i+1} : \texttt{int} \backslash \emptyset. \tag{6.9}$$

By removing, from the type derivation $\Gamma; \Delta' \vdash \overline{E[\texttt{succ}\ n_i]}$ of (6.7) above, the subderivation for $\texttt{succ}\ n_i$ and replacing it with the derivation of (6.9) we get a derivation $\Gamma; \Delta' \vdash \overline{E[n_{i+1}]}$ where the types for subterms of $E[]$ are the same as the types for the occurrences of these subterms in $\Gamma; \Delta' \vdash \overline{E[\texttt{succ}\ n_i]}$. We then have $e[n_{i+1}] = \Phi(\Gamma; \Delta' \vdash \overline{E[n_{i+1}]})$, which, by the definition of Φ implies

$$e[n_{i+1}] \ \in\ \Phi(\Gamma; \Delta',\ E[n_{i+1}]). \tag{6.10}$$

Finally, with (6.5), (6.6) and (6.10) we get

$$s'', \pi''''[p : \rho, e[n_{i+1}]] \quad \in \quad \Phi(\Gamma, \Delta',\ S'', \Pi'''[p : \rho, E[n_{i+1}]],$$

as desired.

case Seq with Ref

We have the following transition:

$$S'', \Pi'''[p : \rho, E[\texttt{ref}\ V]] \underset{\mathsf{con}}{\longrightarrow} S''[r \mapsto V], \Pi'''[p : \rho, E[r]].$$

By induction hypothesis and the definition of Φ, we have a target configuration

$$(s'', \pi'''[p : \rho, e[c\ v]]) \ \in\ \Phi(\Gamma; \Delta'', S'', \Pi'''[p : \rho, E[\texttt{ref}\ V]]), \tag{6.11}$$

where c is either $\texttt{ref}$ or $\texttt{ref}^{\,\ell}$, and

$$s'' \ \in\ \Phi(\Gamma; \Delta'', S''), \tag{6.12}$$

$$\pi''' \ \in\ \Phi(\Gamma; \Delta'', \Pi'''), \tag{6.13}$$

$$e[c\ v] \ \in\ \Phi(\Gamma; \Delta'', E[\texttt{ref}\ V]), \tag{6.14}$$

and

$$v \ \in\ \Phi(\Gamma; \Delta'', V). \tag{6.15}$$

We build a Δ' respecting $S''[r \mapsto V]$ as an extension of Δ'' with a typing for the new reference r. Because r is new, from (6.12), (6.13), (6.14), and (6.15) we get

$$s'' \ \in\ \Phi(\Gamma; \Delta', S''), \tag{6.16}$$

$$\pi''' \ \in\ \Phi(\Gamma; \Delta', \Pi'''), \tag{6.17}$$

$$e[c\ v] \ \in\ \Phi(\Gamma; \Delta', E[\texttt{ref}\ V]), \tag{6.18}$$

and

$$v \ \in\ \Phi(\Gamma; \Delta', V). \tag{6.19}$$

By (6.18) above and the definition of Φ we know that

$$\exists\ \Gamma; \Delta' \vdash \overline{E[\texttt{ref}\ V]}\ \textit{such that}\ e[c\ v] = \Phi(\Gamma; \Delta' \vdash \overline{E[\texttt{ref}\ V]}) \tag{6.20}$$

which implies that

$$\exists\ \Gamma; \Delta' \vdash \overline{\texttt{ref}\ V}_{\tau\texttt{ref}^{\phi}\backslash C} \tag{6.21}$$

and one can easily check that in this type derivation

$$\Gamma; \Delta' \vdash \texttt{ref} : \tau \xrightarrow{\phi'} \tau\,\texttt{ref}^{\,\phi}\backslash C', \tag{6.22}$$

and

$$\Gamma; \Delta' \vdash V : \tau \backslash C'', \tag{6.23}$$

for some constraint sets C', C'', and labels ϕ and ϕ'.

We choose

$$\Delta' = \Delta''[r \mapsto \tau\,\texttt{ref}^{\,\phi}\backslash C']. \tag{6.24}$$

which implies that

$$\Gamma ; \Delta' \vdash r : \tau\, \texttt{ref}^{\,\phi} \backslash C'. \tag{6.25}$$

By removing, from the type derivation $\Gamma ; \Delta' \vdash \overline{E[\texttt{ref}\ V]}$ of (6.20) above, the subderivation for $\texttt{ref}\ V$ and replacing it with the derivation of (6.25) we get a derivation $\Gamma ; \Delta' \vdash \overline{E[r]}$ where the types for subterms of $E[]$ are the same as the types for the occurrences of these subterms in $\Gamma ; \Delta' \vdash \overline{E[\texttt{ref}\ V]}$. We then have $e[r] = \Phi(\Gamma ; \Delta' \vdash \overline{E[r]})$, which, by the definition of Φ, implies

$$e[r] \ \in \ \Phi(\Gamma ; \Delta',\, E[r]). \tag{6.26}$$

We now proceed by case analysis on the label ϕ of (6.22).

case $\phi = \epsilon$

We can rewrite (6.11) as

$$s'', \pi'''[p : \rho, e[\texttt{ref}\ v]] \ \in \ \Phi(\Gamma ; \Delta'',\, S'', \Pi'''[p : \rho, E[\texttt{ref}\ V]]). \tag{6.27}$$

By (6.16), (6.17), (6.19), (6.26), and the definition of Φ we get

$$s''[r \mapsto \mathsf{E}, \mathcal{P}, v], \pi'''[p : \rho, e[r]] \ \in \Phi(\Gamma ; \Delta',\, S''[r \mapsto V], \Pi'''[p : \rho, E[r]])$$

where $\mathcal{P}$ is any processor set.

By the fact that Δ' respects the store S' and by the typing of r and V in (6.22) and (6.23) we have that **esc?**(τ). By Lemma 3.7 we then have that **esc?**$(\Gamma ; \Delta', V)$ which implies that for all $r \in \mathit{Refs}(V)$, **esc?**$(\Delta'(r))$. We then have that $\mathit{Refs}(v) \cap \mathit{Local}(\rho, s'') = \emptyset$. We can then conclude that

$$s'', \pi'''[p : \rho, e[\texttt{ref}\ v]] \underset{\mathsf{con}}{\longrightarrow} s''[r \mapsto \mathsf{E}, \mathcal{P}, v], \pi'''[p : \rho, e[r]].$$

case $\phi \neq \epsilon$

We can rewrite (6.11) as

$$s'', \pi'''[p : \rho, e[\texttt{ref}^{\,\ell}\, v]] \ \in \ \Phi(\Gamma ; \Delta'',\, S'', \Pi'''[p : \rho, E[\texttt{ref}^{\,\ell}\, V]]). \tag{6.28}$$

By (6.16), (6.17), (6.19), (6.26), and the definition of Φ we get

$$s''[r \mapsto L_\rho, \{\rho\}, v], \pi'''[p : \rho, e[r]] \ \in \Phi(\Gamma ; \Delta',\, S''[r \mapsto V], \Pi'''[p : \rho, E[r]].$$

And we have the following transition

$$s'', \pi'''[p : \rho, e[\texttt{ref}^{\,\ell}\, v]] \underset{\mathsf{con}}{\longrightarrow} s''[r \mapsto L_\rho, \{\rho\}, v], \pi'''[p : \rho, e[r]].$$

case Deref

We have the following transition:

$$S'''[r \mapsto V], \Pi'''[p : \rho, E[\texttt{deref}\ r]] \xrightarrow[\mathsf{con}]{} S'''[r \mapsto V], \Pi'''[p : \rho, E[V]].$$

By the induction hypothesis and the definition of Φ we have that

$$s''' \in \Phi(\Gamma; \Delta'', S'''), \tag{6.29}$$

$$v \in \Phi(\Gamma; \Delta'', V), \tag{6.30}$$

and

$$e[c\ r] \in \Phi(\Gamma; \Delta'', E[\texttt{deref}\ r]). \tag{6.31}$$

where c is either `deref` or $\texttt{deref}\ \ell$.

We have that $\Delta' = \Delta''$, so by (6.29), (6.30), and (6.31),

$$s''' \in \Phi(\Gamma; \Delta', S'''), \tag{6.32}$$

$$v \in \Phi(\Gamma; \Delta', V), \tag{6.33}$$

and

$$e[c\ r] \in \Phi(\Gamma; \Delta', E[\texttt{deref}\ r]). \tag{6.34}$$

By a reasoning similar to the one used in the previous cases, with (6.33) and (6.34) we have

$$e[v] \in \Phi(\Gamma; \Delta', E[V]). \tag{6.35}$$

Also from (6.34) an the definition of Φ we know that

$$\exists \Gamma; \Delta' \vdash \overline{\texttt{deref}\ r}$$

which implies that

$$\Gamma; \Delta' \vdash \texttt{deref} : \tau\, \texttt{ref}^{\,\phi} \xrightarrow{\phi'} \tau \backslash \emptyset, \tag{6.36}$$

and

$$\Gamma; \Delta' \vdash r : \tau\, \texttt{ref}^{\,\phi} \backslash C, \tag{6.37}$$

for some type τ, constraint set C and labels ϕ and ϕ'. We now proceed by case analysis on ϕ

case $\phi = \epsilon$

By (6.29), (6.30), (6.31), and the definition of Φ we have that

$$\begin{gathered} s'''[r \mapsto \mathsf{E}, \mathcal{P}, v],\ \pi'''[p : \rho, e[\texttt{deref}\ r]] \\ \in \\ \Phi(\Gamma; \Delta'', S'''[r \mapsto V], \Pi'''[p : \rho, E[\texttt{deref}\ r]]. \end{gathered}$$

where $\mathcal{P}$ is any processor set. From (6.32), (6.33), (6.35) and the definition of Φ we have that

$$s'''[r \mapsto \mathsf{E}, \mathcal{P} \cup \{\rho\}, v],\ \pi'''[p : \rho, e[v]] \\ \in \\ \Phi(\Gamma; \Delta',\ S'''[r \mapsto V],\ \Pi'''[p : \rho, E[V]]).$$

In order to show that

$$s'''[r \mapsto \mathsf{E}, \mathcal{P}, v],\ \pi'''[p : \rho, e[\texttt{deref}\ r]] \\ \xrightarrow[con]{} \\ s'''[r \mapsto \mathsf{E}, \mathcal{P} \cup \{\rho\}, v],\ \pi'''[p : \rho, e[v]]$$

we still have to prove that the proviso for this rule holds. We have to show that for all $r' \in \mathit{Refs}(v)$ their locality indicator should be either E or $L_{\rho'}$ with $\rho' = \rho$.

By (6.37) and the fact that Δ'' respects S'' we know that $\Gamma; \Delta'' \vdash V : \tau \backslash C'''$ for some C'''. Also by the definition of respects we know that $\mathbf{esc?(}\tau\mathbf{)}$. By Lemma 3.7 we then have that $\mathbf{esc?(}\Gamma; \Delta''\mathbf{,}V\mathbf{)}$, which implies that $\mathbf{esc?(}\Delta''(r')\mathbf{)}$ holds for all $r' \in \mathit{Refs}(V)$. By the definition of Φ the locality indicator of all the references of v are then E.

case $\phi \neq \epsilon$

By (6.29), (6.30), (6.31), and the definition of Φ we have that

$$s'''[r \mapsto L_\rho, \{\rho\}, v)],\ \pi'''[p : \rho, e[\texttt{deref}^{\,\ell}\, r]] \\ \in \\ \Phi(\Gamma; \Delta'',\ S'''[r \mapsto V],\ \Pi'''[p : \rho, E[\texttt{deref}\ r]]).$$

From (6.32), (6.33), (6.35) and the definition of Φ we have that

$$s'''[r \mapsto L_\rho, \{\rho\}, v],\ \pi'''[p : \rho, e[v]] \\ \in \\ \Phi(\Gamma; \Delta',\ S'''[r \mapsto V],\ \Pi'''[p : \rho, E[V]]).$$

In order to show that

$$s'''[r \mapsto L_\rho, \{\rho\}, v],\ \pi'''[p : \rho, e[\texttt{deref}^{\,\ell}\, r]] \\ \xrightarrow[con]{} \\ s'''[r \mapsto L_\rho, \{\rho\}, v],\ \pi'''[p : \rho, e[v]]$$

We still have to prove that the proviso of this rule holds. Note that

$$\mathit{Reach}^{\Delta'',S''}(V) \subseteq\ \mathit{Reach}^{\Delta'',S''}(V) \subseteq \mathit{Reach}^{\Pi''}_{\Delta'',S''}(\rho).$$

By Lemma 5.9 and the assumption that $\mathit{Reach}^{\Pi}_{\Delta,S}(\rho) \cap \mathit{Reach}^{\Pi}_{\Delta,S}(\rho')$ for all $\rho' \neq \rho$ we have that $\mathit{Reach}^{\Pi}_{\Delta,S}(\rho) \cap \mathit{Reach}^{\Pi}_{\Delta,S}(\rho')$ for all $\rho' \neq \rho$.

We then know that the references of V in $Reach^{\Delta'',S''}(V)$ cannot be reachable from any other processor $\rho' \neq \rho$. The `ref` operations that created this local references of V, translated to $\texttt{ref}^{\,\ell}$ operations, were executed at processor ρ only. The proviso of the rule, then follows immediately.

case Seq with Assign

We have the following transition:

$$S'''[r \mapsto V'], \Pi'''[p : \rho, E[r := V]] \underset{\mathsf{con}}{\longrightarrow} S'''[r \mapsto V], \Pi'''[p : \rho, E[\texttt{unit}]].$$

By the induction hypothesis and the definition of Φ we have that

$$s''' \in \Phi(\Gamma; \Delta'', S''') \tag{6.38}$$

$$v' \in \Phi(\Gamma; \Delta'', V'), \tag{6.39}$$

$$e[r\ c\ v] \in \Phi(\Gamma; \Delta'', E[r := V]) \tag{6.40}$$

and

$$v \in \Phi(\Gamma; \Delta'', V) \tag{6.41}$$

where c is either $:=$ or $:=^{\ell}$.

We have that $\Delta' = \Delta''$, so by (6.38), (6.39), (6.40) and (6.41)

$$s''' \in \Phi(\Gamma; \Delta', S'''), \tag{6.42}$$

$$v' \in \Phi(\Gamma; \Delta', V'), \tag{6.43}$$

$$e[r\ c\ v] \in \Phi(\Gamma; \Delta', E[r := V]), \tag{6.44}$$

and

$$v \in \Phi(\Gamma; \Delta', V). \tag{6.45}$$

By a reasoning similar to previous cases using (6.44) we have

$$e[\texttt{unit}] \in \Phi(\Gamma; \Delta', E[\texttt{unit}]). \tag{6.46}$$

Also by (6.44) we know that

$$\exists \Gamma; \Delta' \vdash \overline{r := V}$$

which implies that

$$\Gamma; \Delta' \vdash := : \tau\, \texttt{ref}^{\,\phi} \times^{\phi''} \tau \xrightarrow{\phi'} \texttt{unit} \backslash C, \tag{6.47}$$

$$\Gamma; \Delta' \vdash r : \tau\, \mathtt{ref}^{\phi} \backslash C', \tag{6.48}$$

and

$$\Gamma; \Delta' \vdash V : \tau \backslash C'', \tag{6.49}$$

for some τ, ϕ, ϕ' and constraint sets C, C' and C''.

We now proceed by case analysis on the label ϕ of the reference r above.

case $\phi = \epsilon$

By (6.38), (6.39), (6.40), and the definition of Φ we have that

$$\begin{array}{c} s'''[r \mapsto \mathsf{E}, \mathcal{P}, v')],\ \pi'''[p : \rho, e[r := v]] \\ \in \\ \Phi(\Gamma; \Delta'',\ S'''[r \mapsto V'],\ \Pi'''[p : \rho, E[r := V]]) \end{array}$$

where $\mathcal{P}$ is any processor set.

From (6.42), (6.45), (6.46) and the definition of Φ we have that

$$\begin{array}{c} s'''[r \mapsto \mathsf{E}, \{\rho\}, v],\ \pi'''[p : \rho, e[\mathtt{unit}]] \\ \in \\ \Phi(\Gamma; \Delta',\ S'''[r \mapsto V],\ \Pi'''[p : \rho, E[\mathtt{unit}]]). \end{array}$$

In order to show that

$$\begin{array}{c} s'''[r \mapsto \mathsf{E}, \mathcal{P}, v'],\ \pi'''[p : \rho, e[r := v]] \\ \xrightarrow[con]{} \\ s'''[r \mapsto \mathsf{E}, \{\rho\}, v)],\ \pi'''[p : \rho, e[\mathtt{unit}]]. \end{array}$$

we still have to prove that the proviso for this transition is satisfied. Because $\Delta'' = \Delta'$ respects S'' and by (6.49) we know that **esc?(**τ'**)**. By Lemma 3.7 we then have that **esc?(**$\Gamma; \Delta''$**,**V**)**, which implies that **esc?(**$\Delta''(r')$**)** holds for all $r' \in \mathit{Refs}(V)$. By the definition of Φ the locality indicator t of all the references of v are then E.

case $\phi \neq \epsilon$

By (6.38), (6.39), (6.44), and the definition of Φ we have that

$$\begin{array}{c} s'''[r \mapsto L_{\rho}, \{\rho\}, v'],\ \pi'''[p : \rho, e[r :=^{\ell} v]] \\ \in \\ \Phi(\Gamma; \Delta'',\ S'''[r \mapsto V],\ \Pi'''[p : \rho, E[r := V]]). \end{array}$$

From (6.42), (6.45), (6.46) and the definition of Φ we have that

$$\begin{array}{c} s'''[r \mapsto L_{\rho}, \{\rho\}, v],\ \pi'''[p : \rho, e[\mathtt{unit}]] \\ \in \\ \Phi(\Gamma; \Delta',\ S'''[r \mapsto V],\ \Pi'''[p : \rho, E[\mathtt{unit}]]). \end{array}$$

And we can also have that

$$s'''[r \mapsto L_\rho, \{\rho\}, v'],\ \pi'''[p : \rho, e[r :=^{\ell} v]] \underset{con}{\longrightarrow} s'''[r \mapsto L_\rho, \{\rho\}, v],\ \pi'''[p : \rho, e[\mathtt{unit}]].$$

as desired.

case Rfork

We have the following transition:

$$S'', \Pi'''[p : \rho, E[\mathtt{rfork}_{\rho'}\ \mathtt{fn} x.M]] \underset{\mathsf{con}}{\longrightarrow} S'', \Pi'''[p : \rho, E[\mathtt{unit}]][q : \rho', M[\mathtt{unit}/x]].$$

By the induction hypothesis and the definition of Φ we have

$$s'' \ \in\ \Phi(\Gamma; \Delta'', S''), \tag{6.50}$$

$$\pi''' \ \in\ \Phi(\Gamma; \Delta'', \Pi'''), \tag{6.51}$$

and

$$e[\mathtt{rfork}_{\rho'}\ \ \mathtt{fn}\, x.m] \ \in\ \Phi(\Gamma; \Delta'', E[\mathtt{rfork}_{\rho'}\ \ \mathtt{fn}\, x.M]). \tag{6.52}$$

We have that $\Delta' = \Delta''$ hence by (6.50), (6.51) and (6.52)

$$s'' \ \in\ \Phi(\Gamma; \Delta', S''), \tag{6.53}$$

$$\pi''' \ \in\ \Phi(\Gamma; \Delta', \Pi'''), \tag{6.54}$$

and

$$e[\mathtt{rfork}_{\rho'}\ \ \mathtt{fn}\, x.m] \ \in\ \Phi(\Gamma; \Delta', E[\mathtt{rfork}_{\rho'}\ \ \mathtt{fn}\, x.M]). \tag{6.55}$$

By a similar reasoning used in the previous cases we can prove that

$$e[\mathtt{unit}] \ \in\ \Phi(\Gamma; \Delta', E[\mathtt{unit}]) \tag{6.56}$$

$$m[\mathtt{unit}/x] \ \in\ \Phi(\Gamma; \Delta', M[\mathtt{unit}/x]) \tag{6.57}$$

Then, by the definition of Φ with (6.53), (6.54), (6.56) and (6.57) we conclude

$$s'', \pi'''[p : \rho, e[\mathtt{unit}]][q : \rho, e[m[\mathtt{unit}/x]]] \in \Phi(\Gamma; \Delta', S'', \Pi'''[p : \rho, E[\mathtt{unit}]][q : \rho, E[M[\mathtt{unit}/x]]]).$$

In order to conclude the transition

$$s'', \pi'''[p : \rho, e[\mathtt{rfork}_{\rho'}\, \mathtt{fn}\, x.m]] \underset{con}{\longrightarrow} s'', \pi'''[p : \rho, e[\mathtt{unit}]][q : \rho', m[\mathtt{unit}/x]].$$

We still have to show that the proviso for this rule holds, that is, $\forall \rho'' \neq \rho'.\, \mathit{Refs}\,(m[\mathtt{unit}/x]) \cap \mathit{Local}(\rho'', s)$. For some constrained type $\tau \backslash C$, we have that $\Gamma; \Delta'' \vdash E[\mathtt{rfork}_{\rho'}\ \mathtt{fn}\, x.M] : \tau \backslash C$. By the typing rules, the type of any *rforked* function must be labelled with ϵ. By Lemma 3.7 we then have that for all $r \in \mathit{Refs}\,(\mathtt{fn}\, x.M)$, $\mathbf{esc?}(\Delta''(r))$. One can easily check that $\mathit{Refs}\,(\mathtt{fn}\, x.M) = \mathit{Refs}\,(m[\mathtt{unit}/x])$, so according to the definition of Φ, all the locality indicators of references of the term $m[\mathtt{unit}/x]$ are E.

case Fork

Similar to the previous case.

The proof of part *2* is analogous to the proof of part *1*. ∎

We finally prove the main result of this chapter: translation of well typed programs induced by their type derivations preserves the behaviour of original programs.

In the following, we abbreviate the initial source configuration $\emptyset$, $[p_0 : \rho_0, P]$ and the initial target configuration $\emptyset$, $[p_0 : \rho_0, \mathsf{p}]$ by writing only P and p respectively. We also write $\Phi(P)$ for $\Phi(\Gamma, \Delta,\, P)$ when Γ and Δ are empty, both when P is a source program or when P stands for the initial configuration $\emptyset, [p_0 : \rho_0, P]$.

Theorem 6.1 *Let P be a well-typed source program and p be such that $\mathsf{p} \in \Phi(P)$. Then the following hold:*

1. *if $P \xrightarrow[\mathsf{con}]{}^{n} S', \Pi'$ then there is a configuration s', π' such that $\mathsf{p} \xrightarrow[con]{}^{n} s', \pi'$ and $(s', \pi') \in \Phi(\Delta',\, S', \Pi')$ for Δ' respecting S', and*

2. *if $\mathsf{p} \xrightarrow[con]{}^{n} s', \pi'$ then there is a configuration S', Π' such that $P \xrightarrow[\mathsf{con}]{}^{n} S', \Pi'$ and $(s', \pi') \in \Phi(\Delta',\, S', \Pi')$ for Δ' respecting S'.*

Proof. By Lemma 6.1 and the facts that both the source configuration $\emptyset, [p_0 : \rho_0, P]$ and the target configuration $\emptyset, [p_0 : \rho_0, \mathsf{p}]$ satisfy the conditions required by that lemma. ∎

Chapter 7

Type Inference

In this chapter we give a type reconstruction algorithm for our type system.

We begin by presenting, in Section 7.1, a type system which is essentially the same as the type system presented in Chapter 3, but with some simplifications which help us to concentrate on the key ideas of the algorithm

The algorithm, which we call $\mathcal{W}_{loc}$, returns an explicitly typed expression which corresponds to its input expression with type and locality information reconstructed. We present these explicitly typed expressions in Section 7.2. $\mathcal{W}_{loc}$ is presented in Section 7.3. It is divided in two sub-algorithms: $\mathcal{W}_{\epsilon}$ and $\mathcal{L}$. $\mathcal{W}_{\epsilon}$ and its proof of soundness are presented in Section 7.3.1 and Section 7.4.1 respectively. The algorithm $\mathcal{L}$ and its proof of soundness are given in Section 7.3.2 and Section 7.4.2. In Section 7.4 these two results are put together to produce a proof of soundness for $\mathcal{W}_{loc}$. Soundness of $\mathcal{W}_{loc}$ with respect to the type system combined with the result of type and locality soundness of the type system with respect to the operational semantics, means that the locality analysis performed by $\mathcal{W}_{loc}$ is safe. Section 7.5 concludes this chapter with a discussion of the restricted form of *best locality* property that we conjecture holds for $\mathcal{W}_{loc}$.

7.1 Type System for Type Reconstruction

We will prove the algorithm $\mathcal{W}_{loc}$ sound with respect to the type system in Figure 7.1. This type system is in essence the same as the one given in Chapter 3 (Figure 3.2), whose type and locality soundness were proved with respect to the operational semantics for the language. The only differences are:

- type judgements do not have store typing contexts, so there is no rule for typing reference constants, and
- there are no typing rules for

VAR $\dfrac{\Gamma(x) \succ \tau \backslash C}{\Gamma \vdash x : \tau \backslash C}$

ABS$_\ell$ $\dfrac{\Gamma[x \mapsto \tau_1 \backslash C'] \vdash M : \tau_2 \backslash C}{\Gamma \vdash \texttt{fn}\, x.M : \tau_1 \xrightarrow{\ell} \tau_2 \backslash C \cup C'}$

ABS$_\epsilon$ $\dfrac{\Gamma[x \mapsto \tau_1 \backslash C'] \vdash M : \tau_2 \backslash C \quad \textbf{esc?}(\Gamma, \texttt{fn}\, x.M)}{\Gamma \vdash \texttt{fn}\, x.M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \cup C'}$

PAIR $\dfrac{\Gamma \vdash M_1 : \tau_1 \backslash C \quad \Gamma \vdash M_2 : \tau_2 \backslash C'}{\Gamma \vdash \texttt{(}M_1, M_2\texttt{)} : \tau_1 \times^\phi \tau_2 \backslash C \cup C' \cup C'' \cup C'''}$

where:
$C'' = \{\phi \le \textbf{lab}\,(\tau_1)\}$, *if* $\textbf{lab}\,(\tau_1)$ *is defined*, $C'' = \emptyset$ *otherwise*
$C''' = \{\phi \le \textbf{lab}\,(\tau_2)\}$, *if* $\textbf{lab}\,(\tau_2)$ *is defined*, $C''' = \emptyset$ *otherwise*

APP $\dfrac{\Gamma \vdash M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \quad \Gamma \vdash N : \tau_1 \backslash C'}{\Gamma \vdash MN : \tau_2 \backslash C \cup C'}$

LET $\dfrac{\Gamma \vdash V : \tau' \backslash C' \quad \Gamma[x \mapsto Gen(\tau' \backslash C', \Gamma)] \vdash N : \tau \backslash C}{\Gamma \vdash \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : \tau \backslash C \cup C'}$

Figure 7.1: Typing rules for type inference.

– constants,

– conditional expression,

– let-expressions binding variables to non-value expressions, and

– recursive functions.

We had store typing contexts and a rule for typing reference constants in the type system of Figure 3.2 because the proof method used to prove type soundness in Chapter 4 and locality soundness in Chapter 5 uses the type system itself to type intermediate states of evaluation, which might contain reference constants.

We also do not consider conditional expressions and let-expressions binding variables to non-value expressions as their type reconstruction are very similar to the type reconstruction of applications. We also eliminate the rule for typing constants assuming that for each constant c in Figure 3.1 there is a variable named after it in the domain of term type contexts and that the type scheme associated to this variable is the same as **typeof** (c).

We also omit recursive functions. Although the type reconstruction for recursive functions has its own peculiarities, its locality information reconstruction is similar to the reconstruction of locality for non-recursive functions.

As expected the type system in Figure 7.1 shares several properties with the type system in Figure 3.2. When necessary we will refer to chapters 3 and 4 for definitions and proofs.

7.2 Explicitly Typed Expressions

The type reconstruction algorithm returns what we call *extended explicitly typed* expressions. The constrained types and constrained type schemes annotating these extended explicitly typed expressions are of the form $\tau\backslash C, C'$ and $\forall\vec{\alpha}\vec{\kappa}.\tau\backslash C, C'$ respectively, where τ is a labelled type and C and C' are constraint sets as defined in Chapter 3. Note that they have an extra set of constraints, for this reason we call them *extended constrained types* and *extended constrained type schemes*.

In $\tau\backslash C, C'$ we say that C is a *regular* constraint set and C' is an *extra* constraint set. Constraints are added to the extra set of constraints only in the clause of the algorithm for abstractions. Extra sets of constraints have information relating the locality of functions with the locality of their free term variables. This information helps the algorithm to infer locality labels for the types of term variables occurring free in escaping and polymorphic functions.

The metavariables δ and ς will be used to range over extended constrained types and extended constrained type schemes, respectively.

Extended explicitly typed expressions are given by the following grammar

$$\begin{array}{rcl}
\overline{\mathsf{M}} & ::= & (x : \varsigma) : \delta \\
& | & (\mathtt{fn}\, x : \varsigma.\ \overline{\mathsf{M}}) : \delta \\
& | & (\overline{\mathsf{M}}\ \overline{\mathsf{N}}) : \delta \\
& | & (\overline{\mathsf{M}}, \overline{\mathsf{N}}) : \delta \\
& | & \mathtt{let}\ x : \varsigma = \overline{\mathsf{V}}\,\mathtt{in}\ \overline{\mathsf{N}}\ \mathtt{end}\ : \delta.
\end{array}$$

We write $\overline{\mathsf{M}}$, $\overline{\mathsf{M}}'$, $\overline{\mathsf{M}}''$, ... for different extended explicitly typed versions of expression M, and $\overline{\mathsf{V}}$, $\overline{\mathsf{V}}'$, $\overline{\mathsf{V}}''$, ... for different extended explicitly typed versions of value V For simplicity we omit constraint sets when they are empty. We sometimes write $\overline{\mathsf{M}}_{\tau\backslash C,C'}$ to indicate that $\tau\backslash C, C'$ is the outermost extended constrained type of $\overline{\mathsf{M}}$.

Example 7.1 *Consider the expression* $\mathtt{let}\ id = \mathtt{fn}\, x.\, x\ \ \mathtt{in}\ \ id\ id\ \mathtt{end}$. *A possible explicitly typed version of this expression is*

$$\begin{array}{l}
\mathtt{let}\ id : \varsigma = (\mathtt{fn}\, x : \alpha^{\kappa}.(x : \alpha^{\kappa}) : \alpha^{\kappa}) :\ \alpha^{\kappa} \xrightarrow{\kappa'} \alpha^{\kappa} \\
\ \mathtt{in}\quad ((id : \varsigma) : (\beta^{j} \xrightarrow{j'} \beta^{j}) \xrightarrow{j''} (\beta^{j} \xrightarrow{j'} \beta^{j}) \qquad (id : \varsigma) : \beta^{j} \xrightarrow{j'} \beta^{j}) :\ \beta^{j} \xrightarrow{j'} \beta^{j} \\
\mathtt{end}\ : \beta^{j} \xrightarrow{j'} \beta^{j}
\end{array}$$

where $\varsigma = \forall \alpha\kappa\kappa'.\alpha^\kappa \xrightarrow{\kappa'} \alpha^\kappa$.

The extra sets of constraints have their role in the algorithm but ultimately we will be interested only in the *regular* sets. If $\overline{\mathsf{M}}$ is an extended explicitly typed expression, $Reg(\overline{\mathsf{M}})$, defined below, erases extra constraint sets from all extended constrained types and extended constrained type schemes of $\overline{\mathsf{M}}$. We call the resulting expression a *regular explicitly typed expression*, or simply an *explicitly typed expression*. Observe that linear representations for type derivation trees as given in the previous chapter are regular explicitly typed expressions.

$$\begin{array}{rcl}
Reg(\tau\backslash C, C') & = & \tau\backslash C \\
Reg(\forall\vec{\alpha}\vec{\kappa}\,\tau\backslash C, C') & = & \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C \\
Reg((x:\varsigma):\delta) & = & (x : Reg(\varsigma)) : Reg(\delta) \\
Reg((\mathtt{fn}\,x:\varsigma.\overline{\mathsf{P}}):\delta) & = & (\mathtt{fn}\,x : Reg(\varsigma).\,Reg(\overline{\mathsf{P}})) : Reg(\delta) \\
Reg(\mathtt{let}\ x\!:\varsigma = \overline{\mathsf{V}}\ \mathtt{in}\ \overline{\mathsf{P}}\,\mathtt{end}:\delta) & = & \mathtt{let}\ x\!: Reg(\varsigma) = Reg(\overline{\mathsf{V}})\ \mathtt{in}\ Reg(\overline{\mathsf{P}})\,\mathtt{end} : Reg(\delta).
\end{array}$$

In the remaining of this section we give other auxiliary functions over extended explicitly type expressions. These functions will helps us state and prove properties about $\mathcal{W}_{loc}$ and its components.

The set $BLV(\overline{\mathsf{M}})$ of bound locality variables of $\overline{\mathsf{M}}$ is given by

$$\begin{array}{rcl}
BLV((x:\varsigma):\delta) & = & \{\} \\
BLV((\mathtt{fn}\,x:\varsigma.\overline{\mathsf{P}}):\delta) & = & BLV(\overline{\mathsf{P}}) \\
BLV((\overline{\mathsf{M}}\ \overline{\mathsf{N}}):\delta) & = & BLV(\overline{\mathsf{M}}) \ \cup\ BLV(\overline{\mathsf{N}}) \\
BLV(\mathtt{(}\overline{\mathsf{M}},\overline{\mathsf{N}}\mathtt{)}:\delta) & = & BLV(\overline{\mathsf{M}}) \ \cup\ BLV(\overline{\mathsf{N}}) \\
BLV(\mathtt{let}\ x:\forall\vec{\alpha}\vec{\kappa}.\delta' = \overline{\mathsf{V}}\ \mathtt{in}\ \overline{\mathsf{P}}\,\mathtt{end}:\delta) & = & BLV(\overline{\mathsf{V}}) \ \cup\ BLV(\overline{\mathsf{P}}) \ \cup\ \lceil\vec{\kappa}\rceil.
\end{array}$$

The set $FV(\overline{\mathsf{M}})$ of free term variables of $\overline{\mathsf{M}}$, paired with their type schemes is given by

$$\begin{array}{rcl}
FV((x:\varsigma):\delta) & = & \{x\mapsto\varsigma\} \\
FV((\mathtt{fn}\,x:\varsigma.\overline{\mathsf{P}}):\delta) & = & FV(\overline{\mathsf{P}}) \ \backslash\ \{x\mapsto\varsigma\} \\
FV((\overline{\mathsf{M}}\ \overline{\mathsf{N}}):\delta) & = & FV(\overline{\mathsf{M}}) \ \cup\ FV(\overline{\mathsf{N}}) \\
FV(\mathtt{(}\overline{\mathsf{M}},\overline{\mathsf{N}}\mathtt{)}:\delta) & = & FV(\overline{\mathsf{M}}) \ \cup\ FV(\overline{\mathsf{N}}) \\
FV(\mathtt{let}\ x:\varsigma = \overline{\mathsf{V}}\ \mathtt{in}\ \overline{\mathsf{P}}\,\mathtt{end}:\delta) & = & FV(\overline{\mathsf{V}}) \ \cup\ (FV(\overline{\mathsf{P}}) \ \backslash\ \{x\mapsto\varsigma\}).
\end{array}$$

The set $FVoccur(\overline{\mathsf{M}})$ of *occurrences* of free term variables of $\overline{\mathsf{M}}$ is given by

$$\begin{array}{rcl}
FVoccur((x:\varsigma):\delta) & = & \{(x:\varsigma):\delta\} \\
FVoccur((\mathtt{fn}\,x:\varsigma.\overline{\mathsf{P}}):\delta) & = & FVoccur(\overline{\mathsf{P}}) \backslash\ \{(x'\!:\varsigma')\!:\delta' \mid x'=x, \varsigma'=\varsigma\} \\
FVoccur((\overline{\mathsf{M}}\ \overline{\mathsf{N}}):\delta) & = & FVoccur(\overline{\mathsf{M}}) \ \cup\ FVoccur(\overline{\mathsf{N}}) \\
FVoccur(\mathtt{(}\overline{\mathsf{M}},\overline{\mathsf{N}}\mathtt{)}:\delta) & = & FVoccur(\overline{\mathsf{M}}) \ \cup\ FVoccur(\overline{\mathsf{N}}) \\
FVoccur(\mathtt{let}\ x\!:\varsigma = \overline{\mathsf{V}}\ \mathtt{in}\ \overline{\mathsf{P}}\,\mathtt{end}:\delta) & = & FVoccur(\overline{\mathsf{V}}) \ \cup \\
 & & (FVoccur(\overline{\mathsf{P}}) \backslash\ \{(x'\!:\varsigma')\!:\delta' \mid x'=x, \varsigma'=\varsigma\}).
\end{array}$$

Observe the difference between $FVoccur(\overline{\mathsf{M}})$ and $FV(\overline{\mathsf{M}})$: if $\overline{\mathsf{M}}$ is the pair $\mathtt{(}(x\!:\varsigma)\!:\delta,\ (x\!:\varsigma)\!:\delta'\mathtt{)}:\delta''$ for example, with $\delta\neq\delta'$ we have that $FV(\overline{\mathsf{M}}) = \{\, x\!:\varsigma \,\}$ and $FVoccur(\overline{\mathsf{M}}) = \{\, (x\!:\varsigma)\!:\delta,\ (x\!:\varsigma)\!:\delta' \,\}$.

The set the set $Sub(\overline{\mathsf{M}})$ of subexpressions of $\overline{\mathsf{M}}$ is given by

$$\begin{array}{rcl}
Sub((x : \varsigma) : \delta) & = & \{(x : \varsigma) : \delta\} \\
Sub((\mathtt{fn}\, x : \varsigma.\overline{\mathsf{P}}) : \delta) & = & Sub(\overline{\mathsf{P}}) \ \cup \ \{(\mathtt{fn}\, x : \varsigma.\overline{\mathsf{P}}) : \delta\} \\
Sub((\overline{\mathsf{M}}\ \overline{\mathsf{N}}) : \delta) & = & Sub(\overline{\mathsf{M}}) \ \cup \ Sub(\overline{\mathsf{N}}) \ \cup \ \{(\overline{\mathsf{M}}\ \overline{\mathsf{N}}) : \delta\} \\
Sub(\mathtt{(}\overline{\mathsf{M}}, \overline{\mathsf{N}}\mathtt{)} : \delta) & = & Sub(\overline{\mathsf{M}}) \ \cup \ Sub(\overline{\mathsf{N}}) \ \cup \ \{\mathtt{(}\overline{\mathsf{M}}, \overline{\mathsf{N}}\mathtt{)} : \delta\} \\
Sub(\mathtt{let}\ x : \varsigma\ = \overline{\mathsf{V}}\ \mathtt{in}\ \overline{\mathsf{P}}\,\mathtt{end} : \delta) & = & Sub(\overline{\mathsf{V}}) \ \cup \ Sub(\overline{\mathsf{P}}) \ \cup \\
& & \{(\mathtt{let}\ x : \varsigma\ = \overline{\mathsf{V}}\ \mathtt{in}\ \overline{\mathsf{P}}\,\mathtt{end}) : \delta\}.
\end{array}$$

Type, locality and extended type substitutions are applied to extended explicitly typed expressions inductively on their structure, as expected.

7.3 The Algorithm $\mathcal{W}_{loc}$

The type reconstruction algorithm $\mathcal{W}_{loc}$, is presented in Figure 7.2. The first argument of $\mathcal{W}_{loc}$ is a map from term variables to *extended constrained type schemes*. We use the metavariable Θ to range over these maps. When given a map Θ and an expression M, $\mathcal{W}_{loc}(\Theta, M)$ returns an extended explicitly typed version of M.

The algorithm $\mathcal{W}_{loc}$ is divided into two sub-algorithms called $\mathcal{W}_\epsilon$ and $\mathcal{L}$.

$$\begin{array}{rcll}
\mathcal{W}_{loc}(\Theta,\ M) & = & \mathtt{let}\ \ (Z,\ \overline{\mathsf{M}},\ X)\ =\ \mathcal{W}_\epsilon(\Theta,\ M) & (1) \\
& & \qquad\quad Z^{loc}\ =\ \mathcal{L}(\overline{\mathsf{M}},\ \overline{\mathsf{M}}) & (2) \\
& & \mathtt{in} & \\
& & \qquad\quad Z^{loc}\,\overline{\mathsf{M}} & (3) \\
& & \mathtt{end}\ . &
\end{array}$$

Figure 7.2: Algorithm $\mathcal{W}_{loc}$.

The sub-algorithm $\mathcal{W}_\epsilon(\Theta,\ M)$ returns a pair $Z = (Z^{type}, Z^{loc})$ of type and locality substitution, an extended explicitly typed expression $\overline{\mathsf{M}}$, and an auxiliary set X (whose role in $\mathcal{W}_\epsilon$ we explain later).

$\mathcal{W}_\epsilon$ is the component of $\mathcal{W}_{loc}$ responsible for inferring types and for inferring all escaping labels. The only labels appearing in types inferred by $\mathcal{W}_\epsilon$ are the escape label ϵ and locality variables. $\mathcal{W}_\epsilon$ provides all the type and locality information needed for an optimising transformation as defined in the previous chapter. $\mathcal{W}_{loc}$ fails only if $\mathcal{W}_\epsilon$ fails

The reason for having the sub-algorithm $\mathcal{L}$ is that the output of $\mathcal{W}_\epsilon$ *cannot* be proved sound with respect to the type system in Figure 7.1. That type system requires the predicate **esc?** to hold (rule ABS_ϵ) when labelling a function abstraction with a locality variable or with ϵ. But, as we will prove later, that is

$$\text{VAR}_{\text{aux}} \quad \frac{\Gamma(x) \succ \tau \backslash C}{\Gamma \vdash_{\text{aux}} x : \tau \backslash C}$$

$$\text{ABS}_{\text{aux}} \quad \frac{\Gamma[x \mapsto \tau_1 \backslash C'] \vdash_{\text{aux}} M : \tau_2 \backslash C}{\Gamma \vdash_{\text{aux}} \mathtt{fn}\, x.M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \cup C'}$$

$$\text{PAIR}_{\text{aux}} \quad \frac{\Gamma \vdash_{\text{aux}} M_1 : \tau_1 \backslash C \quad \Gamma \vdash_{\text{aux}} M_2 : \tau_2 \backslash C'}{\Gamma \vdash_{\text{aux}} (M_1, M_2) : \tau_1 \times^{\phi} \tau_2 \backslash C \cup C' \cup C'' \cup C'''}$$

where:
$C'' = \{\phi \leq \mathbf{lab}\ (\tau_1)\}$, *if* $\mathbf{lab}\ (\tau_1)$ *is defined*, $C'' = \emptyset$ *otherwise*
$C''' = \{\phi \leq \mathbf{lab}\ (\tau_2)\}$, *if* $\mathbf{lab}\ (\tau_2)$ *is defined*, $C''' = \emptyset$ *otherwise*

$$\text{APP}_{\text{aux}} \quad \frac{\Gamma \vdash_{\text{aux}} M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C \quad \Gamma \vdash_{\text{aux}} N : \tau_1 \backslash C'}{\Gamma \vdash_{\text{aux}} MN : \tau_2 \backslash C \cup C'}$$

$$\text{LET}_{\text{aux}} \quad \frac{\Gamma \vdash_{\text{aux}} V : \tau' \backslash C' \quad \Gamma[x \mapsto Gen(\tau' \backslash C', \Gamma)] \vdash_{\text{aux}} N : \tau \backslash C}{\Gamma \vdash_{\text{aux}} \mathtt{let}\ x = V\ \mathtt{in}\ N\ \mathtt{end} : \tau \backslash C \cup C'}$$

Figure 7.3: Auxiliary type system for $\mathcal{W}_\epsilon$

the case for $\mathcal{W}_\epsilon$ only when the locality label of a function type is ϵ or is a *bound* locality variable. The only role $\mathcal{L}$ plays in $\mathcal{W}_{loc}$ is to go through the explicitly typed expression produced by $\mathcal{W}_\epsilon$ and, whenever it finds an abstraction labelled with a *free locality variable* κ, it replaces κ with ℓ.

7.3.1 The Algorithm $\mathcal{W}_\epsilon$

The algorithm $\mathcal{W}_\epsilon$ is given in Figure 7.6. Before explaining $\mathcal{W}_\epsilon$ and its components we present a formal system against which we prove some properties of $\mathcal{W}_\epsilon$.

7.3.1.1 Type System for $\mathcal{W}_\epsilon$

As we mentioned before we cannot prove $\mathcal{W}_\epsilon$ sound with respect to the type system in Figure 7.1. In this section we then present a formal system, in the form of a type system, against which we prove $\mathcal{W}_\epsilon$ sound. This type system is given in Figure 7.3. The differences between this *auxiliary* type system and the one in Figure 7.1 are the following:

- there is only one rule for typing functions and it does not require the predicate **esc?**,
- the language of labelled types has only ϵ and locality variables as locality labels, which implies that

- a constraint set is invalid iff it has a constraint like $\epsilon \leq \kappa$ in its transitive closure.

As expected the type system in Figure 7.3 shares several definitions and properties with the type system in Figure 7.1 (which in its turn inherits its properties from the type system in Figure 3.2).

In the next subsections we present some of the components of $\mathcal{W}_\epsilon$.

7.3.1.2 Unification of Labelled Types

The following notation is adopted to give a description of the algorithm for unification of labelled types as general as possible: the metavariable t_n is used to range over type constructors of arity $n \geq 0$. In our case the type constructors of arity $n = 0$ are the basic types `int`, `bool` and `unit`. The type constructor `ref` has arity $n = 1$, and $\times$ and $\rightarrow$ are type constructors of arity $n = 2$. We write t_n^ϕ to indicate that ϕ is the label of type constructor t_n, $n > 0$.

Figure 7.4 shows the unification algorithm $\mathcal{U}$ used by $\mathcal{W}_\epsilon$. Given two labelled types τ_1 and τ_2, $\mathcal{U}(\tau_1, \tau_2)$ returns a unifying substitution (Z^{type}, Z^{loc}) such that $(Z^{type}, Z^{loc})\,\tau_1 = (Z^{type}, Z^{loc})\,\tau_2$.

The algorithm $\mathcal{U}$ calls $\mathcal{U}_{tg}$, given below, to unify locality labels

$$\begin{array}{lllll}
\mathcal{U}_{tg}(\phi, \phi) & = & Id^{loc} & & \\
\mathcal{U}_{tg}(\epsilon, \kappa) & = & \mathcal{U}_{tg}(\kappa, \epsilon) & = & \{\kappa \mapsto \epsilon\} \\
\mathcal{U}_{tg}(\kappa, \kappa') & = & \{\kappa \mapsto \kappa'\}. & &
\end{array}$$

The following lemma establish the correctness of unification in the usual way.

Lemma 7.1 *$\mathcal{U}(\tau_1, \tau_2)$ terminates, and if it returns Z then*

1. *$Z\,\tau_1 = Z\,\tau_2$, and*

2. *for any Z' such that $Z'\tau_1 = Z'\tau_2$, there exists some Z'' such that $Z' = Z'' \circ Z$.*

PROOF. Similar to the proof of Theorem 15 given in page 36 of [11]. ∎

7.3.1.3 Resolving Constraints

When given a set C of constraints, the algorithm $\mathcal{R}$ for resolution of constraints, in Figure 7.5, returns a locality substitution Z^{loc} such that no constraint of the form $\epsilon \leq \kappa$ is in the transitive closure of $Z^{loc}\,C$. The following lemma establish the correctness of $\mathcal{R}$

Lemma 7.2 *$\mathcal{R}$ terminates and if $Z^{loc} = \mathcal{R}(C)$ then*

$$
\begin{array}{l}
\mathcal{U}(\tau,\tau) \;\;=\;\; (Id^{type}, Id^{loc}) \\[1em]
\mathcal{U}(\alpha^{\phi}, t_0) \;\;=\;\; \mathcal{U}(t_0, \alpha^{\phi}) \;\;=\;\; (\{\alpha \mapsto t_0\},\ Id^{loc}) \\[1em]
\mathcal{U}(\alpha^{\phi}, \beta^{\phi'}) \;\;=\;\; \texttt{let } Z^{loc} = \mathcal{U}_{tg}(\phi,\phi') \\
\qquad\qquad\qquad\;\; \texttt{in} \\
\qquad\qquad\qquad\qquad (\{\alpha \mapsto \beta^{Z^{loc}\,\phi'}\},\ Z^{loc}) \\
\qquad\qquad\qquad\;\; \texttt{end} \\[1em]
\mathcal{U}(\alpha^{\phi},\ t^{\phi'}(\tau_1,\ldots\tau_n)) \;\;=\;\; \mathcal{U}(t^{\phi'}(\tau_1,\ldots\tau_n),\ \alpha^{\phi}) = \\
\qquad\qquad\qquad\qquad\qquad \texttt{if } \alpha \in FTV(\tau_1) \cup \ldots \cup FTV(\tau_n) \texttt{ then } \mathit{fail} \\
\qquad\qquad\qquad\qquad\qquad \texttt{else let } Z^{loc} = \mathcal{U}_{tg}(\phi,\phi') \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\{\alpha \mapsto Z^{loc}(t^{\phi'}(\tau_1,\ldots\tau_n))\},\ Z^{loc}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{end} \\[1em]
\mathcal{U}(t^{\phi}(\tau_1,\ldots\tau_n),\ t^{\phi'}(\tau'_1,\ldots\tau'_n)) \;\;=\;\; \\
\qquad\qquad\qquad\qquad\qquad \texttt{let } Z_0^{loc} = \mathcal{U}_{tg}(\phi,\phi') \\
\qquad\qquad\qquad\qquad\qquad\quad Z_0 = (Id^{type}, Z_0^{loc}) \\
\qquad\qquad\qquad\qquad\qquad\quad Z_1 = \mathcal{U}(Z_0\,\tau_1, Z_0\,\tau'_1) \\
\qquad\qquad\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\qquad\qquad\quad Z_n = \mathcal{U}(Z_{n-1} \circ \ldots \circ Z_0\,\tau_n,\ Z_{n-1} \circ \ldots \circ Z_0\,\tau'_n) \\
\qquad\qquad\qquad\qquad\qquad \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\quad (Z_n \circ \ldots \circ Z_0) \\
\qquad\qquad\qquad\qquad\qquad \texttt{end} \\[1em]
\mathcal{U}(\_,\_) \;\;=\;\; \mathit{fail}
\end{array}
$$

Figure 7.4: Unification of labelled types.

$$
\begin{array}{l}
\mathcal{R}(C) \;\;=\;\; \texttt{let } \mathit{dom} = \{\kappa_i \mid (\epsilon \leq \kappa_i) \in C\} \\
\qquad\qquad\;\; \texttt{in if } \mathit{dom} = \emptyset \texttt{ then } Id^{loc} \\
\qquad\qquad\qquad \texttt{else let } Z^{loc} = \{\kappa \mapsto \epsilon \mid \kappa \in \mathit{dom}\} \\
\qquad\qquad\qquad\qquad\qquad\;\; C' = Z^{loc}\,C \\
\qquad\qquad\qquad\qquad \texttt{in } Z^{loc} \circ \mathcal{R}(C') \\
\qquad\qquad\qquad\qquad \texttt{end} \\
\qquad\qquad\;\; \texttt{end}
\end{array}
$$

Figure 7.5: Resolution of constraints

1. *there is no κ such that $\epsilon \leq \kappa \in (Z^{loc}\, C)^+$,*

2. *$Dom(Z^{loc}) \subseteq FLV(C)$,*

3. *if $\kappa \leq \epsilon \in C$ and $\epsilon \leq \kappa \notin C^+$ then $\kappa \leq \epsilon \in (Z^{loc} C)^+$.*

Proof. $\mathcal{R}$ clearly terminates since at each iteration the number of constraints of the form $\epsilon \leq \kappa$ gets smaller. The program stops when there are no constraints of the form $\epsilon \leq \kappa$ in C. The other results clearly hold since a locality variable κ is in the domain of the substitutions built at each call of $\mathcal{R}$ only if $\epsilon \leq \kappa$ is in the transitive closure of C^+. ∎

The significance of each of the above results about $\mathcal{R}$ will be made clear in subsequent proofs. Result *1* is obviously needed because we want the algorithm to eliminate invalid constraint sets (in doing so it infers escaping labels). This result is used in the proof of soundness.

Results *2* and *3* say that $\mathcal{R}$ substitutes ϵ for locality variables *only* when that is necessary for the validity of the set of constraints. In doing so it is trying to infer as few escaping labels as possible.

7.3.1.4 Instantiation and Generalisation

The function *Inst* used by $\mathcal{W}_\epsilon$ to produce an instance of an extended constrained type scheme, is defined as follows. We write $[\kappa_i \mapsto \kappa'_i]$ for the locality substitution $\{\kappa_1 \mapsto \kappa'_1 \ldots, \kappa_n \mapsto \kappa'_n\}$ and $\lceil \kappa_i \rceil$ for $\{\kappa_i \ldots, \kappa_n\}$. Similarly for type substitutions.

$Inst\,(\forall \vec{\alpha}\vec{\kappa}.\, \delta) =$

  `let` $\delta_0 = [\kappa_i \mapsto \kappa'_i]\, \delta$

  `in` $[\alpha_i \mapsto \beta_i^{\phi_i}]\, \delta_0$

  where:

   $\lceil \kappa_i \rceil = \lceil \vec{\kappa} \rceil$ *and κ'_i is distinct and fresh*

   $\lceil \alpha_i \rceil = \lceil \vec{\alpha} \rceil$, *$\beta_i$ is distinct and fresh, and* $\phi_i = \mathbf{lab}\,(\alpha_i)$ *in* δ_0

    if $\alpha_i \in FTV(\delta_0)$, *otherwise ϕ is any label*

  `end`.

We prove a result that relates *Inst* with the instantiation relation $\succ$.

Lemma 7.3 *If $Inst\,(\forall \vec{\alpha}\vec{\kappa}.\tau \backslash C, C') = \tau_0 \backslash C_0, C'_0$ then*

1. *there is Z such that $Dom(Z^{loc}) = \lceil \vec{\kappa} \rceil$, $Dom(Z^{type}) = \lceil \vec{\alpha} \rceil$, and $Z\tau = \tau_0$, $ZC = C_0$ and $ZC' = C'_0$, and also*

2. *$\forall \vec{\alpha}\vec{\kappa}.\tau \backslash C \succ \tau_0 \backslash C_0$.*

PROOF. Part *1*: immediate from the way the function *Inst* is defined above. Part *2*: by part *1*, there is Z such that $Dom(Z^{loc}) = \lceil\vec{\kappa}\rceil$, $Dom(Z^{type}) = \lceil\vec{\alpha}\rceil$, and $Z\tau = \tau_0$, $ZC = C_0$. By the Definition 3.5 of $\succ$ this corresponds exactly to $\forall\vec{\alpha}\vec{\kappa}.\tau\backslash C \succ \tau_0\backslash C_0$. ∎

The algorithm $\mathcal{W}_\epsilon$ generalises type and locality variables of extended constrained types using $Gen_{\mathcal{W}_\epsilon}$ defined in the following way

$$
\begin{aligned}
&Gen_{\mathcal{W}_\epsilon}(\tau\backslash C, C',\ \Theta) = \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C, C'\\
&\quad \textit{where:}\\
&\quad\quad \lceil\vec{\kappa}\rceil = FLV(\tau) \setminus FLV(\Theta)\\
&\quad\quad \lceil\vec{\alpha}\rceil = FTV(\tau) \setminus FTV(\Theta).
\end{aligned}
$$

The following lemma relates the generalisation procedure used by the algorithm and the operation of generalisation of the type system.

Lemma 7.4 *$Gen(Reg(\delta),\ Reg(\Theta))\ \succ\ Reg\,(Gen_{\mathcal{W}_\epsilon}(\delta,\ \Theta))$.*

PROOF. The type variables that are generalised are the same on both sides of the relation $\succ$. But more locality variables can be generalised in $Gen(Reg(\delta),\ Reg(\Theta))$, though, since $FLV(Reg(\Theta)) \subseteq FLV(\Theta)$. ∎

7.3.1.5 Explaining $\mathcal{W}_\epsilon$

Notation. We adopt the following conventions in the presentation of $\mathcal{W}_\epsilon$.

1. substitutions with a ϵ as a subscript are locality substitutions,
2. we write $Z_\epsilon \circ Z$ as an abbreviation for $(Id^{type}, Z_\epsilon)\ \circ\ Z$,
3. if X is the set $\{(x_1\colon \varsigma_1)\colon \delta_1 \ldots, (x_n\colon \varsigma_n)\colon \delta_n\}$ we write $X \setminus \{x\}$ instead of $X \setminus \{(x_i\colon \varsigma_i)\colon \delta_i \mid x_i = x\}$,
4. the definitions of type, locality and extended type substitutions given in Chapter 3 are extended to Θ in the expected way.

If we *erase* from $\mathcal{W}_\epsilon$ all the machinery used to infer locality information, $\mathcal{W}_\epsilon$ behaves exactly as Milner's algorithm $\mathcal{W}$ [18] when inferring types. We assume familiarity with $\mathcal{W}$ and we concentrate on explaining how $\mathcal{W}_\epsilon$ infers locality information.

The clauses for variables, application and pairs are analogous to the their counterparts in $\mathcal{W}$. Resolution of constraints ($\mathcal{R}$) is added at appropriated places in order to infer valid locality information. Substitutions, including those produced by $\mathcal{R}$ are propagated as expected.

$$
\begin{array}{lllr}
\mathcal{W}_\epsilon(\Theta,\ x) & = & x \notin Dom(\Theta) \ \rightarrow\ fail & \\
 & \mid & \mathtt{let}\ \tau\backslash C, C' = Inst(\Theta(x)) & (1) \\
 & & \qquad Z_\epsilon = \mathcal{R}\,(C \cup C') & (2) \\
 & & \mathtt{in}\ \ (Z_\epsilon,\ Z_\epsilon\,((x : \Theta(x)) : \tau\backslash C, C'),\ \{Z_\epsilon((x : \Theta(x)) : \tau\backslash C, C')\}) & \\
 & & \mathtt{end} &
\end{array}
$$

$$
\begin{array}{lllr}
\mathcal{W}_\epsilon(\Theta,\ \mathtt{fn}\,x.\,P) & = & \mathtt{let}\ \ (Z,\ \overline{\mathsf{P}}_{\tau\backslash C,C'},\ X') =\ \mathcal{W}_\epsilon(\Theta[x \mapsto \alpha^{\kappa}],\ P) \quad \alpha, \kappa'\ new & (3) \\
 & & \mathtt{in}\ \ (Z,\ (\mathtt{fn}\,x : Z(\alpha^{\kappa'}).\ \overline{\mathsf{P}}) : Z(\alpha^{\kappa'}) \xrightarrow{\kappa} \tau\backslash C, C' \cup C'',\ X) & \\
 & & \mathtt{end} & \\
 & & where & \\
 & & \kappa\ new,\ X = X' \setminus \{x\} & (4) \\
 & & C'' = \{\kappa \leq \mathbf{lab}\,(\delta) \mid \overline{x}_\delta \in X\ \ and\ \ \mathbf{lab}\,(\delta)\ \ is\ defined\} & (5)
\end{array}
$$

$$
\begin{array}{lllr}
\mathcal{W}_\epsilon(\Theta,\ N\ P) & = & \mathtt{let}\,(Z_1,\ \overline{\mathsf{N}}_{\tau_1\backslash C_1,C'_1},\ X) = \mathcal{W}_\epsilon(\Theta,\ N) & (6) \\
 & & \quad (Z_2,\ \overline{\mathsf{P}}_{\tau_2\backslash C_2,C'_2},\ X') = \mathcal{W}_\epsilon(Z_1\,\Theta,\ P) & (7) \\
 & & \quad Z_u = \mathcal{U}(Z_2\tau_1,\ \tau_2 \xrightarrow{\kappa} \alpha^{\kappa'}) \quad \alpha, \kappa, \kappa'\ new & (8) \\
 & & \quad Z_\epsilon = \mathcal{R}\,(Z_u Z_2(C_1 \cup C'_1) \cup Z_u(C_2 \cup C'_2)) & (9) \\
 & & \mathtt{in} & \\
 & & \quad (Z_\epsilon \circ Z_u \circ Z_2 \circ Z_1, & \\
 & & \quad\ Z_\epsilon\, Z_u\,((Z_2\,\overline{\mathsf{N}}\ \overline{\mathsf{P}}) : \alpha^{\kappa'} \backslash Z_2 C_1 \cup C_2, Z_2 C'_1 \cup C'_2), & \\
 & & \quad\ Z_\epsilon Z_u\,(Z_2\,X\ \ \cup\ \ X')) & \\
 & & \mathtt{end} &
\end{array}
$$

$$
\begin{array}{lllr}
\mathcal{W}_\epsilon(\Theta,\ (N, P)) & = & \mathtt{let}\,(Z_1,\ \overline{\mathsf{N}}_{\tau_1\backslash C_1,C'_1},\ X) = \mathcal{W}_\epsilon(\Theta,\ N) & (10) \\
 & & \quad (Z_2,\ \overline{\mathsf{P}}_{\tau_2\backslash C_2,C'_2},\ X') = \mathcal{W}_\epsilon(Z_1\,\Theta,\ P) & (11) \\
 & & \quad Z_\epsilon = \mathcal{R}\,(Z_2\,(C_1 \cup C'_1) \cup C_2 \cup C'_2) & (12) \\
 & & \quad \kappa\ new & \\
 & & \quad C_3 = \mathbf{lab}\,(Z_\epsilon Z_2 \tau_1)\ \ undef.\ \ \rightarrow\ \ \{\}\ \ \mid\ \ \{\kappa \leq \mathbf{lab}\,(Z_\epsilon Z_2 \tau_1)\} & (13) \\
 & & \quad C_4 = \mathbf{lab}\,(Z_\epsilon \tau_2)\ \ undef.\ \ \rightarrow\ \ \{\}\ \ \mid\ \ \{\kappa \leq \mathbf{lab}\,(Z_\epsilon \tau_2)\} & (14) \\
 & & \quad C = C_3 \cup C_4 & \\
 & & \mathtt{in} & \\
 & & \quad (Z_\epsilon \circ Z_2 \circ Z_2, & \\
 & & \quad\ Z_\epsilon\,(Z_2 \overline{\mathsf{N}}, \overline{\mathsf{P}})\ : & \\
 & & \quad\quad Z_\epsilon\,(Z_2 \tau_1 \times^{\kappa} \tau_2) \backslash Z_\epsilon\,(Z_2\,C_1 \cup C_2) \cup C, Z_\epsilon\,(Z_2\,C'_1 \cup C'_2), & \\
 & & \quad Z_\epsilon\,(Z_2 X \cup X')) & \\
 & & \mathtt{end} &
\end{array}
$$

$$
\begin{array}{lr}
\mathcal{W}_\epsilon(\Theta,\ \mathtt{let}\ x = V\ \ \mathtt{in}\ P) = & \\
\qquad \mathtt{let}\,(Z_1,\ \overline{\mathsf{V}}_{\tau_1\backslash C_1,C'_1},\ X) = \mathcal{W}_\epsilon(\Theta,\ V) & (15) \\
\qquad\quad \vec{\kappa_f} = (FLV(\tau_1) \setminus FLV(Z_1\Theta))\ \cap\ \{j \mid\ \ j \leq \phi \in C'_1\} & (16) \\
\qquad\quad Z_{\epsilon 1} = \{\kappa \mapsto \epsilon \mid\ \ j \leq \kappa \in C'_1\ \wedge\ j \in \lceil \vec{\kappa_f} \rceil\} & (17) \\
\qquad\quad Z_{\epsilon 2} = \mathcal{R}(Z_{\epsilon 1}\,C_1 \cup Z_{\epsilon 1}\,C'_1) & (18) \\
\qquad\quad \varsigma = \forall \vec{\alpha}\vec{\kappa}.\ Z_{\epsilon 2}\,Z_{\epsilon 1}\,(\tau_1 \backslash C_1, C'_1) & \\
\qquad\qquad = Gen_{\mathcal{W}_\epsilon}\,(Z_{\epsilon 2}\,Z_{\epsilon 1}\,(\tau_1 \backslash C_1, C_1),\ Z_{\epsilon 2}\,Z_{\epsilon 1}\,Z_1\,\Theta) & (19) \\
\qquad\quad (Z_2,\ \overline{\mathsf{P}}_{\tau_2\backslash C_2,C'_2},\ X',\ ) = \mathcal{W}_\epsilon((Z_{\epsilon 2}\,Z_{\epsilon 1}\,Z_1\,\Theta)[x \mapsto \varsigma],\ P) & (20) \\
\qquad\quad Z_{\epsilon 3} = \mathcal{R}\,(Z_2\,Z_{\epsilon 2}\,Z_{\epsilon 1}\,(C_1 \cup C'_1) \cup C_2 \cup C'_2) & (21) \\
\qquad \mathtt{in} & \\
\qquad\quad (Z_{\epsilon 3} \circ Z_2 \circ Z_{\epsilon 2} \circ Z_{\epsilon 1} \circ Z_1, & \\
\qquad\quad\ Z_{\epsilon 3}\,(\mathtt{let}\ x : Z_2 \varsigma = Z_2\,Z_{\epsilon 2}\,Z_{\epsilon 1}\,(\overline{\mathsf{V}})\ \ \mathtt{in}\ \overline{\mathsf{P}}) : & \\
\qquad\quad\quad Z_{\epsilon 3}\,(\tau_2 \backslash Z_2\,Z_{\epsilon 2}\,Z_{\epsilon 1}\,C_1 \cup C_2, Z_2\,Z_{\epsilon 2}\,Z_{\epsilon 1}\,C'_1 \cup C'_2), & \\
\qquad\quad\ Z_{\epsilon 3} Z_2\,Z_{\epsilon 2}\,Z_{\epsilon 1}\,X\ \ \cup\ \ Z_{\epsilon 3}\,(X' \setminus \{x\})) & \\
\qquad \mathtt{end} &
\end{array}
$$

Figure 7.6: Algorithm $\mathcal{W}_\epsilon$.

Later we show that if $\mathcal{W}_\epsilon(\Theta, M) = (Z, \overline{\mathsf{M}}, X)$ then $X = FVoccur(\overline{\mathsf{M}})$, that is X has all the occurrences of free variables of $\overline{\mathsf{M}}$.

The unusual aspects of clauses for functions and let expressions when compared to $\mathcal{W}$ are due to the need to ensure that functions labelled with a *bound* locality variable satisfy the 3-place predicate **esc?**.

Constraints are added to the extra set of constraints only in the clause for abstractions. This is also the only place where the set X is used. Extra sets of constraints have information relating the locality of functions with the locality of their free term variables.

The clause for `let` expressions is the most interesting: in $\mathcal{W}_\epsilon(16)$ locality variables labelling function types which are *candidates* for generalisation are collected. Among these locality variables are those labelling abstractions occurring in V.

In $\mathcal{W}_\epsilon(17)$ a substitution is built mapping to ϵ the locality variables labelling instances of free variables of functions abstractions in V. This will ensure that these functions, that will be made polymorphic in their locality, are escape-expectant.

Note that this is the only place in $\mathcal{W}_\epsilon$ where the information in the extra set of constraints is used.

7.3.2 The Algorithm $\mathcal{L}$

The goal of the algorithm $\mathcal{L}$ presented in Figure 7.7 is to return the explicitly typed expression produced by $\mathcal{W}_\epsilon$ with its *free* locality variables labelling functions replaced by ℓ. As we said before $\mathcal{L}$ is only included to establish soundness of $\mathcal{W}_{loc}$ with respect to the type system in Figure 7.1. Note also that it is formulated in a very inefficient way. We choose to do so in order to make it simpler to formulate an inductive hypothesis for its soundness proof.

The only interesting clause of $\mathcal{L}$ is the one for function abstractions: if the outermost type of the function is labelled with a locality variable κ which is not bound in $\overline{\mathsf{M}}$ then a locality substitution is produced mapping κ to ℓ

7.4 Soundness of $\mathcal{W}_{loc}$

In this section we present soundness of $\mathcal{W}_{loc}$ with respect to the type system in Figure 7.1. The proof is organised in the following way:

1. first we prove, in Section 7.4.1, that $\mathcal{W}_\epsilon$ is sound with respect to the type system in Figure 7.3,

$$
\begin{array}{l}
\mathcal{L}\,(\overline{\mathsf{M}},\ (x : \varsigma) : \delta) \quad = \quad Id \\[1em]
\mathcal{L}\,(\overline{\mathsf{M}},\ (\mathtt{fn}\, x : \varsigma.\overline{\mathsf{P}}) : \delta) = \\
\qquad\qquad \mathtt{let}\ \ Z_\ell = \mathtt{if}\ \mathbf{lab}\,(\delta) = \kappa\ \mathit{and}\ \kappa \notin \mathit{BLV}\,(\overline{\mathsf{M}}) \\
\qquad\qquad\qquad\qquad \mathtt{then}\ \ [\kappa \mapsto \ell] \\
\qquad\qquad\qquad\qquad \mathtt{else}\ Id \\
\qquad\qquad\qquad\ Z'_\ell\ =\ \mathcal{L}\,(\overline{\mathsf{M}},\ \overline{\mathsf{P}}) \\
\qquad\qquad \mathtt{in} \\
\qquad\qquad\quad Z'_\ell \circ Z_\ell \\
\qquad\qquad \mathtt{end} \\[1em]
\mathcal{L}\,(\overline{\mathsf{M}},\ (\overline{\mathsf{P}}\ \overline{\mathsf{Q}}) : \delta) \ \ = \\
\qquad\qquad \mathtt{let}\ Z_\ell\ =\ \mathcal{L}\,(\overline{\mathsf{M}},\ \overline{\mathsf{P}}) \\
\qquad\qquad\qquad\ Z'_\ell\ =\ \mathcal{L}\,(\overline{\mathsf{M}},\ \overline{\mathsf{Q}}) \\
\qquad\qquad \mathtt{in} \\
\qquad\qquad\quad Z'_\ell \circ Z_\ell \\
\qquad\qquad \mathtt{end} \\[1em]
\mathcal{L}\,(\overline{\mathsf{M}},\ (\overline{\mathsf{P}}, \overline{\mathsf{Q}}) : \delta) \ \ = \\
\qquad\qquad \mathtt{let}\ Z_\ell\ =\ \mathcal{L}\,(\overline{\mathsf{M}},\ \overline{\mathsf{P}}) \\
\qquad\qquad\qquad\ Z'_\ell\ =\ \mathcal{L}\,(\overline{\mathsf{M}},\ \overline{\mathsf{Q}}) \\
\qquad\qquad \mathtt{in} \\
\qquad\qquad\quad Z'_\ell \circ Z_\ell \\
\qquad\qquad \mathtt{end} \\[1em]
\mathcal{L}\,(\overline{\mathsf{M}},\ \mathtt{let}\ x : \varsigma = \overline{\mathsf{V}}\,\mathtt{in}\ \overline{\mathsf{P}}\,\mathtt{end}\ : \delta) \quad = \\
\qquad\qquad \mathtt{let}\ Z_\ell\ =\ \mathcal{L}\,(\overline{\mathsf{M}},\ \overline{\mathsf{V}}) \\
\qquad\qquad\qquad\ Z'_\ell\ =\ \mathcal{L}\,(\overline{\mathsf{M}},\ \overline{\mathsf{P}}) \\
\qquad\qquad \mathtt{in} \\
\qquad\qquad\quad Z'_\ell \circ Z_\ell \\
\qquad\qquad \mathtt{end}
\end{array}
$$

Figure 7.7: Algorithm $\mathcal{L}$.

2. in the same section we prove that if a function $\overline{\mathsf{F}}$ is a subexpression of the explicitly typed expression $\overline{\mathsf{M}}$ returned by $\mathcal{W}_\epsilon$ and it is labelled with ϵ or with a locality variable which is bound in $\overline{\mathsf{M}}$, then the function satisfies the 3-place predicate **esc?**,

3. we then prove soundness of $\mathcal{L}$ in Section 7.4.2,

4. finally the results about $\mathcal{W}_\epsilon$ and $\mathcal{L}$ are put together to proof that $\mathcal{W}_{loc}$ is sound.

We start defining some useful notation.

Notation. 1. We write Γ_Θ to represent the *regular* term type context that is obtained from map Θ in the following way: $Dom(\Gamma_\Theta) = Dom(\Theta)$ and

$$\textit{for all } x \in Dom(\Gamma_\Theta),\ \Gamma_\Theta(x) = Reg(\Theta(x)),$$

2. we write $\Gamma_{\overline{\mathsf{M}}}$ for $Reg(FV(\overline{\mathsf{M}}))$,

3. we write $\overline{\mathsf{F}}$ for the extended typed expression of a function abstraction.

7.4.1 Soundness of $\mathcal{W}_\epsilon$

Several auxiliary lemmas are used in order to prove the two main results about $\mathcal{W}_\epsilon$. In order to keep the main line of reasoning as clear as possible we state and prove these auxiliary lemmas in the Appendix A.3.

The first main result states the soundness of $\mathcal{W}_\epsilon$ with respect to the type system in Figure 7.3. Before we have to define what we mean by Θ be polymorphic consistent.

Definition 7.1 Θ *is* polymorphic consistent *iff* $Reg(\Theta)$ *is polymorphic consistent.*

Note that $Reg(\Theta)$ is a regular term type context and polymorphic consistency of regular term type contexts was defined in Chapter 4 (Definition 3.4).

Lemma 7.5 *If* Θ *is polymorphic consistent and* $\mathcal{W}_\epsilon(\Theta, M) = (Z,\ \overline{\mathsf{M}}_{\tau\backslash C,C'},\ X)$ *then* $Z\Gamma_\Theta \vdash_{aux} M : \tau\backslash C$.

PROOF. By induction on the structure of M (See Appendix A.3). ∎

Our second main result about $\mathcal{W}_\epsilon$ says that function abstractions labelled either with ϵ or with a bound locality variable by the algorithm $\mathcal{W}_\epsilon$ are escape-expectant.

Lemma 7.6 *If*

1. $\mathcal{W}_\epsilon(\Theta, M) = (Z,\ \overline{\mathsf{M}},\ X)$,
2. $\overline{\mathsf{F}}_\delta \in Sub(\overline{\mathsf{M}})$, *and*
3. $\mathbf{lab}\,(\delta) = \epsilon$ *or* $\mathbf{lab}\,(\delta) \in BLV(\overline{\mathsf{M}})$,

then $\mathbf{esc?}(\Gamma_{\overline{\mathsf{F}}}, F)$

Proof. See Appendix A.3. ∎

7.4.2 Soundness of $\mathcal{L}$

The next lemma says that $\mathcal{L}$ is sound with respect to the main type system in Figure 7.1 in the following sense

Lemma 7.7 *Suppose that* $\overline{\mathsf{M}}_{\tau\backslash C,C'}$ *and* $\overline{\mathsf{N}}_{\tau_2\backslash C_2,C_2'}$ *are such that*

1. $\Gamma_{\overline{\mathsf{M}}} \vdash_{aux} M : \tau\backslash C$ *and*
2. *if* $\overline{\mathsf{F}}_\delta \in Sub(\overline{\mathsf{M}})$ *and* $\mathbf{lab}\,(\delta) = \epsilon$ *or* $\mathbf{lab}\,(\delta) \in BLV(\overline{\mathsf{M}})$ *then* $\mathbf{esc?}(\Gamma_{\overline{\mathsf{F}}}, F)$ *and*
3. $\overline{\mathsf{N}}_{\tau_2\backslash C_2,C_2'} \ \in\ Sub(\overline{\mathsf{M}}_{\tau\backslash C,C'})$.

Then $\mathcal{L}(\overline{\mathsf{M}}, \overline{\mathsf{N}}) = Z_\ell$ *implies* $Z_\ell\,\Gamma_{\overline{\mathsf{N}}} \vdash N : Z_\ell\,(\tau_2\backslash C_2)$.

Proof. By induction on the structure of $\overline{\mathsf{N}}$. See Appendix A.3. ∎

7.4.3 Soundness of $\mathcal{W}_{loc}$

We compose the results for $\mathcal{W}_\epsilon$ and $\mathcal{L}$ stated in the previous sections to prove $\mathcal{W}_{loc}$ sound with respect to the type system in Figure 7.1.

The proof makes use of the following auxiliary lemma:

Lemma 7.8 *If* $\mathcal{W}_\epsilon(\Theta, M) = (Z, \overline{\mathsf{M}}, X)$ *then* $(Z\Gamma_\Theta)\,(x) = \Gamma_{\overline{\mathsf{M}}}(x)$ *for all* $x \in FV(M)$.

Proof. See Appendix A.3. ∎

Theorem 7.1 *If Θ is polymorphic consistent and $\mathcal{W}_{loc}(\Theta, M) = \overline{\mathsf{M}}_{\tau\backslash C,C'}$ then $\Gamma_{\overline{\mathsf{M}}} \vdash M : \tau\backslash C$.*

Proof. If $\mathcal{W}_{loc}(\Theta, M)$ returns $\overline{\mathsf{M}}_{\tau\backslash C,C'}$ the sub-algorithm $\mathcal{W}_\epsilon$ is successful and by $\mathcal{W}_{loc}(1)$ in Figure 7.2 we have

$$\mathcal{W}_\epsilon(\Theta, M) \;=\; (Z,\ \overline{\mathsf{M}}'_{\tau_1\backslash C_1,C_1'},\ X) \tag{7.1}$$

for some Z, $\overline{\mathsf{M}}'_{\tau_1\backslash C_1,C_1'}$ and X.

By Lemma 7.5 (Soundness of $\mathcal{W}_\epsilon$) and (7.1) we have

$$Z\Gamma_\Theta \vdash_{\text{aux}} M : \tau_1\backslash C_1. \tag{7.2}$$

By Lemma 7.8 and (7.1) we have

$$(Z\Gamma_\Theta)\,(x) = (\Gamma_{\overline{\mathsf{M}}'})\,(x)\ \textit{for all}\ x \in FV(M). \tag{7.3}$$

The type of a term depends only on the type of its free variables, so by (7.2) and (7.3) we have

$$\Gamma_{\overline{\mathsf{M}}'} \vdash_{\text{aux}} M : \tau_1\backslash C_1. \tag{7.4}$$

By Lemma 7.6 with (7.1) the following holds

$$\textit{if}\ \overline{\mathsf{F}}_\delta \in \textit{Sub}(\overline{\mathsf{M}}')\ \textit{and}\ \mathbf{lab}\,(\delta) = \epsilon\ \textit{or}\ \mathbf{lab}\,(\delta) \in \textit{BLV}(\overline{\mathsf{M}}')\ \textit{then}\ \mathbf{esc?}(\Gamma_{\overline{\mathsf{F}}}, F). \tag{7.5}$$

From $\mathcal{W}_{loc}(2)$ in Figure 7.2 we also have

$$\mathcal{L}(\overline{\mathsf{M}}'_{\tau_1\backslash C_1,C_1'},\ \overline{\mathsf{M}}'_{\tau_1\backslash C_1,C_1'}) \;=\; Z_\ell \tag{7.6}$$

for some Z_ℓ. Then, by the assumption that $\mathcal{W}_{loc}(\Theta,\ M) = \overline{\mathsf{M}}_{\tau\backslash C,C'}$ and by $\mathcal{W}_{loc}(3)$ we have

$$Z_\ell\,(\overline{\mathsf{M}}'_{\tau_1\backslash C_1,C_1'}) \;=\; \overline{\mathsf{M}}_{\tau\backslash C,C'}. \tag{7.7}$$

By Lemma 7.7 (Soundness of $\mathcal{L}$) with (7.6), (7.4) and (7.5) we get

$$Z_\ell\,\Gamma_{\overline{\mathsf{M}}'} \vdash M : Z_\ell\,(\tau_1\backslash C_1),$$

which, by (7.7) is equivalent to

$$\Gamma_{\overline{\mathsf{M}}} \vdash M : \tau\backslash C,$$

as desired. ∎

7.5 Best Locality of $\mathcal{W}_{loc}$

Ignoring locality labels, $\mathcal{W}_{loc}$ behaves exactly as Milner's algorithm $\mathcal{W}$ [18] when inferring the most general type of an expression.

The proof of completeness of the type reconstruction algorithm $\mathcal{W}_{loc}$, not considering locality labels, follows the same lines of the standard completeness result for Milner's algorithm $\mathcal{W}$ [9].

Regarding locality information, the ideal type and locality reconstruction algorithm is one that satisfies what we call the *best locality* property. We have not defined precisely this property but the intuition is clear: an algorithm satisfies best locality if it infers as many local references, or, equivalently, as few escaping references as allowed by the type system.

An algorithm satisfying best locality allows the greatest number of dereference and assignment operations to be translated to their local versions.

We conjecture that the algorithm $\mathcal{W}_{loc}$ satisfies a restricted form of best locality

Conjecture 7.1 *For expressions with no* $\texttt{rfork}_\rho$ *constant and where references are not reachable from locality polymorphic functions we conjecture that* $\mathcal{W}_{loc}$ *satisfies the best locality property.*

For such expressions a reference will be labelled with ϵ only if it is reachable from a function labelled with ϵ. When inferring the type of a function, $\mathcal{W}_\epsilon$ adds to the extra set of constraints of the outermost type constraints $\kappa \leq \phi_1 \ldots, \kappa \leq \phi_n$ where κ is the label of the function and $\phi_1 \ldots, \phi_n$ are the labels of its free variables. Only if κ is unified with ϵ resolution of constraints $\mathcal{R}$, will built a locality substitution mapping locality variables in $\phi_1 \ldots, \phi_n$ to ϵ.

In order to illustrate the reason for the restriction consider the following expression:

$$
\begin{array}{l}
\texttt{let } r = \texttt{ref } 3 \\
\texttt{in let } f = \texttt{fn}\, x.\ \texttt{fst } (x, \texttt{deref } r) \\
\quad \texttt{in } f\ 3 \\
\quad \texttt{end} \\
\texttt{end} .
\end{array}
$$

Note that the expression above has no $\texttt{rfork}_\rho$ constant, so one would expect the reference r to be labelled with ℓ. In fact the rules of the type system allow ℓ to be inferred for the label of the function above and consequently allows ℓ to be inferred as the label for the reference type of r. But the component $\mathcal{W}_\epsilon$ though, generalises the locality variable labelling the function $\texttt{fn}\, x.\ \texttt{fst } (x, \texttt{deref } r)$. In doing so, it also ensures that r is labelled with ϵ even though the expression above is *non-distributed* (it has no $\mathtt{rfork}_\rho$ constant). But observe that there is no reason for f be polymorphic in its locality!

For the following expression for instance, the type inference algorithm infers best locality:

$$
\begin{array}{l}
\mathtt{let}\ r = \mathtt{ref}\ 3 \\
\mathtt{in}\ \mathtt{let}\ f = (\mathtt{fn}\, z.\, \mathtt{fn}\, x.\ \mathtt{fst}\ (x, \mathtt{deref}\ r))\ \ \mathtt{unit} \\
\quad \mathtt{in}\ f\ 3 \\
\quad \mathtt{end} \\
\mathtt{end}\ .
\end{array}
$$

Because the term bound to the variable f is not a value the type of the function where the reference r occurs free is not polymorphic. The algorithm then infers that the type of r can be labelled with ℓ.

In order to achieve best locality we have to find an algorithm which also infers *minimal polymorphism over locality labels*. To understand what we mean by that we can consider the analogous situation to "normal" type polymorphism:

$$
\begin{array}{l}
\mathtt{let}\ id\ = \mathtt{fn}\, x.x \\
\mathtt{in} \\
\quad id\ \ 3 \\
\mathtt{end}\ .
\end{array}
$$

In the expression above *id* is polymorphic but there is no need for that as it is only instantiated to type $\mathtt{int} \to \mathtt{int}$. In [7] there is proposal for a type inference algorithm that tries to reduce polymorphism to situations where it can be useful. A possible continuation for the work on this thesis is to adapt that proposal to obtain *minimal locality polymorphism*.

We implemented $\mathcal{W}_{loc}$ using Moscow/ML exactly as described in Chapter 7. We tested the implementation on several small programs. Our preliminary tests support the conjecture that $\mathcal{W}_{loc}$ satisfies the restricted form of best locality property.

Chapter 8

Conclusions and Future Work

We have devised a polymorphic type system that soundly describes the locality of dynamically-created references in a distributed language. The performance payoff for this analysis promises to be great. In the current implementation of Distributed Poly/ML, a program that does not use the distribution features and so uses only local references will execute more slowly than if run under (non-distributed) Poly/ML. A program that does use the distribution features but uses local references runs more slowly than it should. Our analysis provides a basis for redressing these problems.

We believe that our type-based approach can be useful for other distributed languages with mutable objects based on a shared memory system. Note that our type system requires that a program itself must contain information of which threads are to be executed remotely. Because of that our locality analysis cannot be used for parallel languages where the compiler decides which parts of a program will constitute concurrent threads.

Any non-specialised operation to access mutable data can be used wherever a specialised version, not calling a coherency protocol is expected. Because of that subtypes seems to be a natural choice for a type-based locality analysis to detect locality. When we started the work in this thesis one of our goals was the integration of our type-based analysis in an experimental version of Distributed Poly/ML. For this reason and because the type system of languages based on the SML type discipline does not support subtypes we decided to adopt variables ranging over locality labels. Our intention was to treat locality variables in a way analogous to the way the SML type discipline treats type variables. Later, when technical developments lead us to the adoption of constraints over locality variables, we realized that the additional effort to integrate our type system with that of an ML-like language could have justified the adoption of subtyping in the first place.

In order to prove soundness of locality we had to define notions of reachability, and we had to use fixed point induction to get our main result of Chapter 5. An alternative way to prove soundness of locality can be achieved by introducing a *wrong* value in the target language and by adding rules to the operational semantics given in Chapter 6 returning this *wrong* value every time locality is violated. The semantics given in Chapter 6 has this rule for example

$$\frac{\forall \rho' \neq \rho.\ \mathit{Refs}(v) \cap \mathit{Local}(\rho', s) = \emptyset}{s[r \mapsto t, \mathcal{P}, v], \pi[p : \rho, e[\,\texttt{deref}\ \ r]] \underset{\mathit{con}}{\longrightarrow} s[r \mapsto t, \mathcal{P} \cup \{\rho\}, v], \pi[p : \rho, e[v]]}$$

where the proviso ensures that local references do not escape as subterms of v. A rule returning a *wrong* value in case locality is violated would look like the following

$$\frac{\exists \rho' \neq \rho.\ \mathit{Refs}(v) \cap \mathit{Local}(\rho', s) \neq \emptyset}{s[r \mapsto t, \mathcal{P}, v], \pi[p : \rho, e[\,\texttt{deref}\ \ r]] \underset{\mathit{con}}{\longrightarrow} \mathit{wrong}}$$

The statement of soundness of locality would then be: *if P is well typed program and* $\mathsf{p} \in \Phi(\overline{P})$ *where* $\overline{P}$ *is a type derivation for* P *then*

$$\mathsf{p} \underset{\mathit{con}}{\not\longrightarrow} \mathit{wrong}.$$

The proof that our type system is a conservative extension of the Hindley-Milner type discipline provides an evidence that we can extend our locality analysis to the full SML language.

One of the advantages of type based analysis of programs is that it allows *modular analysis*. Non-modular or global analyses are not compatible with separate compilation. Sometimes the information necessary for optimising a module of a program depends on information that is collected on another module. If these modules are to be compiled separately, the compiler cannot have access to other modules' information. It has to assume a conservative approach which often disallows optimising transformations.

Extending our locality analysis to the language of modules of SML the locality information can be part of the signature of modules. In this way, if a module is compiled separately, the compiler can have access to locality information of other modules.

Further work should be done to find a correct type inference algorithm with minimal locality polymorphism. Only then, as we discussed in Chapter 7 we can achieve the best locality property.

Having a correct type inference algorithm inferring best locality information, we believe the next steps before scaling it up to SML is to study a better internal representation for its output. So another possible future work would be, for example, to integrate our locality labels with the work done in [14] which proposes a type structure of SML by giving an explicitly-typed, polymorphic function calculus to capture the essential aspects of both the core and the module language.

Bibliography

[1] S. Prasad A. Giacalone, P. Mishra. FACILE: A symmetric integration of concurrent and functional programming. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 2*, volume 352 of *LNCS*, pages 184–209, Berlin, March 1989. Springer.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[3] Álvaro Moreira and Paul Steckler. An ML-like type system for detecting local references in a distributed language. Talk given at the Fifth LOMAPS Workshop - Analysis and Verification of Multiple-Agent Languages. Swedish Institute of Computer Science, June 1996.

[4] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 2nd edition, 1984.

[5] David Berry, Robin Milner, and David N.Turner. A semantics for ML concurrency primitives. In *Conference Record of 19st ACM Symposium on Principles of Programming Languages*, pages 84–97, January 1992.

[6] Bernard Berthomieu and Thierry Le Sergent. Programming with behaviors in an ML framework—the syntax and semantics of LCS. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 89–104, Edinburgh, U.K., 11–13 April 1994. Springer.

[7] N. Bjorner. Minimal typing derivations. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 1994.

[8] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory

systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[9] Luis Damas. *Type Assignment in Programming Languages.* PhD thesis, LFCS, University of Edinburgh, April 1982.

[10] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–37, 1987.

[11] Paola Giannini. Type checking and type deduction techniques for polymorphic programming languages. Technical Report CMU-CS-85-187, Computer Science Department, Carnegie-Mellon University, December 1985.

[12] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [13].

[13] Robert Harper. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, January 1996. See [12].

[14] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnagie-Mellon University, 1997.

[15] Mark Hayden. Distributed communication in ML. Technical Report TR97-1652, Cornell University, Computer Science, November 11, 1997.

[16] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Number 1 in London Mathematical Society Student Texts. Cambridge University Press, 1986.

[17] David C.J. Matthews. A distributed concurrent implementation of Standard ML. LFCS Report Series ECS-LFCS-91-174, University of Edinburgh, August 1991.

[18] Robin Milner. A theory of type polymorphism in programming. *Jornal of Computer Systems Sciences*, 17:348–75, 1978.

[19] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[20] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, MA, 1997.

[21] Jelica Protić, Milo Tomašević, and Veljko Milutinović. Distributed shared memory: concepts and systems. *IEEE parallel and distributed technology: systems and applications*, 4(2):63–79, Summer 1996.

[22] J.H. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, 1992. Report 92-1285.

[23] Thierry Le Sargent and David C. J. Matthews. Adaptative selection of protocols for strict coherency in distributed shared memory. Technical Report ECS-LFCS-94-036, LFCS, Department of computer Science, University of Edinburgh, September 1994.

[24] Peter Sewell. Global/local subtyping and capability inference for a distributed pi-calculus. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *25th Colloquium on Automata, Languages and Programming (ICALP) (Aalborg, Denmark)*, volume 1443 of *LNCS*, pages 695–706. Springer, July 1998. Full version as Technical Report 435, Computer Laboratory, University of Cambridge.

[25] Paul Steckler. Detecting local channels in Distributed Poly/ML. Technical Report ECS-LFCS-96-340, Laboratory for the Foundations of Computer Science, Edinburgh University, January 1996.

[26] Paul Steckler and Mitchell Wand. Selective thunkification. In Baudouin Le Charlier, editor, *Proceedings of the 1st International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 162–78, Berlin, Heidelberg, and New York, 1994. Springer-Verlag.

[27] Paul Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–46, January 1997.

[28] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(2):1–34, 1990.

[29] Glynn Winskel. *The Formal Semantics of Programming Languages : an Introduction.* MIT Press, Cambridge, MA, 1993.

[30] Andrew K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8:343–355, 1995.

[31] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1992.

[32] Nobuko Yoshida and Matthew Henessy. Subtyping and locality in distributed higher order mobile processes. In *CONCUR'99*, LNCS. Springer, 1999.

Appendix A

Proofs of Auxiliary Lemmas

A.1 Lemmas from Chapter 3

Lemma 3.4 $\mathbf{esc?}(\forall\vec{\alpha}\vec{\kappa}.\tau\backslash C)$ *iff*

1. $\epsilon \leq \ell \notin C^+$ *and* $\epsilon \leq \kappa \in C^+$ *only if* $\kappa \in \lceil\vec{\kappa}\rceil$, *and*

2. *exactly one of the following holds:*

   (a) $\tau = \alpha^\ell$ *for some* $\alpha \in \lceil\vec{\alpha}\rceil$,

   (b) $\mathbf{esc?}(\tau)$, *or*

   (c) $\mathbf{lab}\,(\tau) = \kappa$, $\kappa \in \lceil\vec{\kappa}\rceil$, $\kappa \leq \ell \notin C^+$ *and* $\kappa \leq \kappa' \in C^+$ *only if* $\kappa' \in \lceil\vec{\kappa}\rceil$.

Proof. ($\Rightarrow$) By Definitions 3.14 and 3.16 we know that there exists $Z = (Z^{type}, Z^{loc})$ such that $Dom(Z^{loc}) = \lceil\vec{\kappa}\rceil$, $Dom(Z^{type}) = \lceil\vec{\alpha}\rceil$, ZC is valid and $\mathbf{esc?}(Z\tau)$.

Because ZC is valid we know that $\epsilon \leq \ell \notin C^+$, and, if $\epsilon \leq \kappa \in C^+$ then $Z^{loc}(\kappa) = \epsilon$. By the way Z^{loc} is defined we then have that $\kappa \in \lceil\vec{\kappa}\rceil$. These facts prove 1 above.

By Definition 3.15 and because $\mathbf{esc?}(Z\tau)$ we know that either

- $\tau = \alpha^\ell$, in which case Z should be such that $Z^{type}(\alpha^\ell) = \tau'$ and $\mathbf{lab}\,(\tau')$ is undefined. By the way Z^{type} is defined we then have that $\alpha \in \lceil\vec{\alpha}\rceil$,

- τ is already an escape-expectant type i.e. τ is such that $\mathbf{esc?}(\tau)$, or

- τ is labelled with a bound locality variable κ, in which case Z should be such that $Z^{loc}(\kappa) = \epsilon$. For ZC to be valid, there should be no $\kappa \leq \ell$ in the transitive closure of C, and, if $\kappa \leq \kappa'$ is in C^+ then $Z^{loc}(\kappa') = \epsilon$ implying that $\kappa' \in \lceil\vec{\kappa}\rceil$.

($\Leftarrow$) For each possible constrained type scheme $\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C$ satisfying the conditions on the right side of the implication we can build an extended type substitution $Z = (Z^{type}, Z^{loc})$ such that $Dom(Z^{type}) = \lceil\vec{\alpha}\rceil$, $Dom(Z^{loc}) = \lceil\vec{\kappa}\rceil$ and ZC is valid and **esc?**$(Z\tau)$. Then by definitions 3.14 and 3.16 we conclude that **esc?**$(\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C)$. ∎

Theorem 3.1 *Let M be a term of the language, then*

1. *if $\Gamma \vdash M : \gamma$ then $\psi(\Gamma) \vdash_{ML} M : \psi(\gamma)$, and*
2. *if $\Gamma_{ML} \vdash_{ML} M : \iota$ then there exist Γ and γ such that $\psi(\Gamma) = \Gamma_{ML}$, $\psi(\gamma) = \iota$, and $\Gamma \vdash M : \gamma$.*

PROOF. Part 1 is proved by induction on the type derivation of M using our type system. Part 2 is proved by induction on the ML type derivation of M. We start proving part 1.

case x

If $\Gamma \vdash x : \gamma$ then by the rule for typing variables $\Gamma(x) \succ \gamma$ must hold. By part 1 of Lemma A.2 we know that $\psi(\Gamma(x)) \succ_{\mathrm{ML}} \psi(\gamma)$, which is the same as $(\psi(\Gamma))(x) \succ_{\mathrm{ML}} \psi(\gamma)$. We can then conclude that $\psi(\Gamma) \vdash_{\mathrm{ML}} M : \psi(\gamma)$.

case $\mathtt{fn}\, x.M$

The typing derivation for $\Gamma \vdash \mathtt{fn}\, x.M : \tau_1 \xrightarrow{\phi} \tau_2\backslash C_1 \cup C_2$ must have been of this form

$$\frac{\Gamma[x \mapsto \tau_1\backslash C_1] \vdash M : \tau_2\backslash C_2 \quad (*)}{\Gamma \vdash \mathtt{fn}\, x.M : \tau_1 \xrightarrow{\phi} \tau_2\backslash C_1 \cup C_2}$$

where $(*)$ marks the possible use of the three-place predicate **esc?**.

From the first premise above and by the induction hypothesis we have that

$$\psi(\Gamma[x \mapsto \tau_1\backslash C_1]) \vdash_{\mathrm{ML}} M : \psi(\tau_2\backslash C_2)$$

which is equivalent to

$$\psi(\Gamma)[x \mapsto \psi(\tau_1)] \vdash_{\mathrm{ML}} M : \psi(\tau_2).$$

By the judgement above and the rule for functions we conclude

$$\psi(\Gamma) \vdash_{\mathrm{ML}} \mathtt{fn}\, x.M : \psi(\tau_1) \to \psi(\tau_2),$$

which one can readily check is the same as

$$\psi(\Gamma) \vdash_{\mathrm{ML}} \mathtt{fn}\, x.M : \psi(\tau_1 \xrightarrow{\phi} \tau_2\backslash C_1 \cup C_2).$$

case $N\ P$

The proof derivation of $\Gamma \vdash N\ P : \tau \backslash CUC'$ must have been of this form

$$\frac{\Gamma \vdash N : \tau' \xrightarrow{\phi} \tau \backslash C' \quad \Gamma \vdash P : \tau' \backslash C}{\Gamma \vdash N\ P : \tau \backslash CUC'}$$

By the induction hypothesis and the first premise above we have

$$\psi(\Gamma) \vdash_{\mathrm{ML}} N : \psi(\tau') \rightarrow \psi(\tau).$$

By the induction hypothesis and the second premise above we have

$$\psi(\Gamma) \vdash_{\mathrm{ML}} P : \psi(\tau').$$

By the application rule using these two type judgements as premises we get

$$\psi(\Gamma) \vdash_{\mathrm{ML}} N\ P : \psi(\tau)$$

which one readily checks is the same as

$$\psi(\Gamma) \vdash_{\mathrm{ML}} N\ P : \psi(\tau \backslash CUC').$$

case $\texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end}$

The proof derivation of $\Gamma \vdash \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : \tau \backslash C$ must have been of this form

$$\frac{\Gamma \vdash V : \tau' \backslash C' \quad \Gamma[x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma)] \vdash N : \tau \backslash C}{\Gamma \vdash \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : \tau \backslash C' \cup C}$$

By the induction hypothesis and the first premise above we have that

$$\psi(\Gamma) \vdash_{\mathrm{ML}} V : \psi(\tau' \backslash C'). \tag{A.1}$$

By the induction hypothesis and the second premises above we get

$$\psi(\Gamma)[x \mapsto \psi(\mathit{Gen}(\tau' \backslash C', \Gamma))] \vdash_{\mathrm{ML}} N : \psi(\tau). \tag{A.2}$$

By Lemma A.1 we have that $\psi(\mathit{Gen}(\tau' \backslash C', \Gamma)) = \mathit{Gen}_{\mathrm{ML}}(\psi(\tau' \backslash C'), \psi(\Gamma))$. We can then rewrite (A.2) above as

$$\psi(\Gamma)[x \mapsto \mathit{Gen}_{\mathrm{ML}}(\psi(\tau' \backslash C'), \psi(\Gamma))] \vdash_{\mathrm{ML}} N : \psi(\tau). \tag{A.3}$$

Using (A.1) and (A.3) as premises for the ML rule for polymorphic `let` we conclude

$$\psi(\Gamma) \vdash_{\mathrm{ML}} \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : \psi(\tau)$$

which one readily checks is equivalent to

$$\psi(\Gamma) \vdash_{\mathrm{ML}} \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : \psi(\tau \backslash C \cup C')$$

We now prove part (2).

case x

If $\Gamma_{\mathrm{ML}} \vdash_{\mathrm{ML}} x : \iota$ then by the rule for typing variables $\Gamma_{\mathrm{ML}}(x) \succ_{\mathrm{ML}} \iota$. By part 2 of Lemma A.2 we know that there are σ and valid γ such that $\psi(\sigma) = \Gamma_{\mathrm{ML}}(x)$, $\psi(\gamma) = \iota$, and $\sigma \succ \gamma$. We then build a term context Γ such that x is mapped to σ and any other y is mapped to a σ' such that $\psi(\sigma') = \Gamma_{\mathrm{ML}}(y)$. We then have that $\Gamma(x) \succ \gamma$ and we can use the rule for typing variables to conclude that $\Gamma \vdash x : \gamma$.

case $\mathtt{fn}\, x.M$

We have a type derivation of this form

$$\frac{\Gamma_{\mathrm{ML}}[x \mapsto \iota_1] \vdash_{\mathrm{ML}} M : \iota_2}{\Gamma_{\mathrm{ML}} \vdash_{\mathrm{ML}} \mathtt{fn}\, x.M : \iota_1 \to \iota_2}$$

By the inductive hypothesis and the premise of the proof above there exist Γ' with $\psi(\Gamma') = \Gamma_{\mathrm{ML}}[x \mapsto \iota_1]$, τ_2 and C_2 with $\psi(\tau_2 \backslash C_2) = \iota_2$ such that $\Gamma' \vdash M : \tau_2 \backslash C_2$.

As we have that $\psi(\Gamma') = \Gamma_{\mathrm{ML}}[x \mapsto \iota_1]$ we know that, for any C_1 and for any τ_1 such that $\psi(\tau_1) = \iota_1$, Γ' must be of the form $\Gamma[x \mapsto \tau_1 \backslash C_1]$ where $\psi(\Gamma) = \Gamma_{\mathrm{ML}}$. We chose some valid C_1. C_2 is also valid, so by Lemma 3.1 we have that $C_1 \cup C_2$ is valid. Hence we can use type judgement $\Gamma \vdash M : \tau_2 \backslash C_2$ as the premise for the rule for typing abstractions in order to get $\Gamma \vdash \mathtt{fn}\, x.M : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C_1 \cup C_2$ and one readily checks that $\psi(\tau_1 \xrightarrow{\ell} \tau_2 \backslash C_1 \cup C_2) = \iota_1 \to \iota_2$.

case $N\ P$

We have a type derivation of this form

$$\frac{\Gamma_{\mathrm{ML}} \vdash_{\mathrm{ML}} N : \iota' \to \iota \qquad \Gamma_{\mathrm{ML}} \vdash_{\mathrm{ML}} P : \iota'}{\Gamma_{\mathrm{ML}} \vdash_{\mathrm{ML}} N\ P : \iota}$$

By the inductive hypothesis and the first premise of the proof above we get

$$\Gamma \vdash N : \tau' \xrightarrow{\phi} \tau \backslash C.$$

By the inductive hypothesis and the premise of the proof above we can have

$$\Gamma \vdash P : \tau' \backslash C'.$$

By the rule for applications and the two type judgements above we conclude

$$\Gamma \vdash N\ P : \tau \backslash C \cup C'.$$

case $\texttt{let}\ x = V\ \texttt{in}\ P\ \texttt{end}$

The proof derivation for this case must have been of this form

$$\frac{\Gamma_{\mathrm{ML}} \vdash_{\mathrm{ML}} V : \iota' \quad \Gamma_{\mathrm{ML}}[x \mapsto \mathit{Gen}_{\mathrm{ML}}(\iota', \Gamma_{\mathrm{ML}})] \vdash_{\mathrm{ML}} N : \iota}{\Gamma_{\mathrm{ML}} \vdash_{\mathrm{ML}} \texttt{let}\ x = V\ \texttt{in}\ N : \iota}$$

By the induction hypothesis and the first premise above we know that there are Γ and $\tau' \backslash C'$ such that $\psi(\Gamma) = \Gamma_{\mathrm{ML}}$, $\psi(\tau' \backslash C') = \iota'$ and

$$\Gamma \vdash V : \tau' \backslash C' \tag{A.4}$$

By the induction hypothesis and the second premise above we know that there are Γ'' and $\tau \backslash C$ such that $\psi(\Gamma'') = \Gamma_{\mathrm{ML}}[x \mapsto \mathit{Gen}_{\mathrm{ML}}(\iota', \Gamma_{\mathrm{ML}})]$, $\psi(\tau \backslash C) = \iota$ and

$$\Gamma' \vdash N : \tau \backslash C \tag{A.5}$$

If $\psi(\Gamma'') = \Gamma_{\mathrm{ML}}[x \mapsto \mathit{Gen}_{\mathrm{ML}}(\iota', \Gamma_{\mathrm{ML}})]$ then Γ'' must be of the form $\Gamma[x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma)]$ since

$$\begin{array}{rl} & \psi(\Gamma[x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma)]) \\ = & \psi(\Gamma)[x \mapsto \psi(\mathit{Gen}(\tau' \backslash C', \Gamma))] \\ = & \Gamma_{\mathrm{ML}}[[x \mapsto \mathit{Gen}_{\mathrm{ML}}(\iota', \Gamma_{\mathrm{ML}})]. \end{array}$$

We can the rewrite (A.5) above as

$$\Gamma[x \mapsto \mathit{Gen}(\tau' \backslash C', \Gamma)] \vdash N : \tau \backslash C. \tag{A.6}$$

Using (A.4) and (A.6) as premises for the rule for typing let expressions we get

$$\Gamma \vdash \texttt{let}\ x = V\ \texttt{in}\ N\ \texttt{end} : \tau \backslash C' \cup C$$

as desired. ∎

Lemma A.1 *$\mathit{Gen}_{ML}(\psi(\tau \backslash C), \psi(\Gamma)) = \psi(\mathit{Gen}(\tau \backslash C, \Gamma))$.*

Proof. Immediate from the definitions of ψ, $\mathit{Gen}_{\mathrm{ML}}$ and Gen and by the facts that $\mathit{FTV}_{\mathrm{ML}}(\psi(\tau \backslash C)) = \mathit{FTV}(\tau)$ and $\mathit{FTV}_{\mathrm{ML}}(\psi(\Gamma)) = \mathit{FTV}(\Gamma)$. ∎

Lemma A.2 *(1) If $\forall \vec{\alpha}\vec{\kappa}.\, \tau' \backslash C' \succ \tau \backslash C$ then $\psi(\forall \vec{\alpha}\vec{\kappa}.\, \tau' \backslash C') \succ_{ML} \psi(\tau \backslash C)$, and (2) if $\forall \vec{\alpha}.\, \iota' \succ_{ML} \iota$ then for any $\vec{\kappa}$, τ' and C' such that $\forall \vec{\alpha}.\, \iota' = \psi(\forall \vec{\alpha}\vec{\kappa}.\, \tau' \backslash C')$ there is $\tau \backslash C$ such that $\forall \vec{\alpha}\vec{\kappa}.\, \tau' \backslash C' \succ \tau \backslash C$ and $\psi(\tau \backslash C) = \iota$.*

PROOF. For proving part *1* we build an "ML" type substitution S as follows: $Dom(S) = Dom(Z^{type}) = \lceil\vec{\kappa}\rceil$ and

$$S(\alpha) = \psi(Z(\alpha^{\phi})) \textit{ where } \phi = \mathbf{lab}\,(\alpha) \textit{ in } \tau' \textit{ if } \alpha \in FTV(\tau'), \textit{ otherwise } \phi \textit{ is any label.}$$

The proof then proceeds by induction on the structure of τ'. For part *2* we build an extended type substitution $Z = (Z^{type}, Z^{loc})$ as follows: Z^{loc} is such that $Dom(Z^{loc}) = \lceil\vec{\kappa}\rceil$ and $Z^{loc}C' = C$. The component Z^{type} is such that $Dom(Z^{type}) = Dom(S) = \lceil\vec{\alpha}\rceil$ and

$$Z^{type}(\alpha) = Z^{loc}\tau'' \textit{ where } \tau'' \textit{ is such that } \psi(\tau'') = S(\alpha).$$

Having devised Z, the proof proceeds by induction on the structure of ι'. ∎

A.2 Lemmas from Chapter 4

Lemma 4.4 *If* $\mathbf{esc?}(\sigma)$ *and there is* $\tau\backslash C$ *such that* $\sigma \succ \tau\backslash C$ *and* ZC *is valid then* $\mathbf{esc?}(Z\,\sigma)$.

PROOF. Let σ be $\forall\vec{\alpha}\vec{\kappa}.\tau'\backslash C'$. Remember from Chapter 3 that we are equating $\alpha-$it equivalent type schemes, so we can assume that σ is such that all its bound type variables and bound locality variables are different from any other type and locality variable in Z. So we can have

$$Z\,(\forall\vec{\alpha}\vec{\kappa}.\,\tau'\backslash C') = \forall\vec{\alpha}\vec{\kappa}.\,Z\tau'\backslash ZC'. \tag{A.7}$$

We prove by contradiction that

$$\epsilon \leq \ell \notin (Z^{loc}\,C'^{+}). \tag{A.8}$$

Suppose $\epsilon \leq \ell \in (Z^{loc}\,C'^{+})$. That can only happens if $\phi \leq \phi' \in C'^{+}$ and

1. $\phi \leq \phi' = \epsilon \leq \ell$, or

2. $\phi \leq \phi$ is either $\epsilon \leq \kappa$, $\kappa \leq \ell$ or $\kappa \leq \kappa'$, and $Z^{loc}\,(\phi \leq \phi') = \epsilon \leq \ell$.

By Lemma 3.6, 1 above contradicts the assumption that $\mathbf{esc?}(\forall\vec{\alpha}\vec{\kappa}.\tau'\backslash C')$. Case 2 above implies that κ and κ' are not in $\lceil\vec{\kappa}\rceil$. By the definition of instantiation we then know that $\phi \leq \phi'$ is in the constraint set of all possible instantiations of $\forall\vec{\alpha}\vec{\kappa}.\tau'\backslash C'$. But that violates the assumption that there is $\tau\backslash C$ such that $\forall\vec{\alpha}\vec{\kappa}.\tau'\backslash C' \succ \tau\backslash C$ and ZC is valid.

We prove by contradiction that

$$\epsilon \leq \kappa \notin (Z^{loc}\,C')^{+} \textit{ for } \kappa \notin \lceil\vec{\kappa}\rceil. \tag{A.9}$$

Suppose $\epsilon \leq \kappa \in (Z^{loc}C')^{+}$ for $\kappa \notin \lceil\vec{\kappa}\rceil$. That can only happens if $\phi \leq \phi' \in C'^{+}$ and either $\phi \leq \phi' = \epsilon \leq \kappa'$, or $\phi \leq \phi = \kappa' \leq \kappa''$ and $Z^{loc}(\phi \leq \phi') = \epsilon \leq \kappa$. But that implies that κ' and κ'' are not in $\lceil\vec{\kappa}\rceil$. By the definition of instantiation we then know that $\phi \leq \phi'$ is in the constraint set of all possible instantiations of $\forall\vec{\alpha}\vec{\kappa}.\tau'\backslash C'$. But that violates the assumption that there is $\tau\backslash C$ such that $\forall\vec{\alpha}\vec{\kappa}.\tau'\backslash C' \succ \tau\backslash C$ and ZC is valid.

We still have to prove that

$$\begin{array}{l} \textit{exactly one of the following holds :} \\ \quad - Z\tau' = \alpha^{\ell} \\ \quad - \textbf{esc?}(Z\tau') \\ \quad - \textbf{lab}\,(Z\tau') = \kappa \textit{ for } \kappa \in \lceil\vec{\kappa}\rceil, \kappa \leq \ell \notin ZC'^{+}, \kappa \leq \kappa' \notin ZC'^{+} \textit{ for } \kappa' \notin \lceil\vec{\kappa}\rceil. \end{array} \tag{A.10}$$

By Lemma 3.6 we know that $\forall\vec{\alpha}\vec{\kappa}.\,\tau'\backslash C'$ is such that exactly one of the following holds

1. $\tau' = \alpha^{\ell}$ for some $\alpha \in \lceil\vec{\alpha}\rceil$,

2. $\textbf{esc?}(\tau')$, or

3. $\textbf{lab}\,(\tau') = \kappa$ *for* $\kappa \in \lceil\vec{\kappa}\rceil, \kappa \leq \ell \notin C'^{+}, \kappa \leq \kappa' \notin C'^{+}$ *for* $\kappa' \notin \lceil\vec{\kappa}\rceil$

In case 1 above holds one readily checks that $Z\tau' = \alpha^{\ell}$ If $\textbf{esc?}(\tau')$ we also have that $\textbf{esc?}(Z\tau')$. Finally, if $\textbf{lab}\,(\tau') = \kappa$ for some $\kappa \in \lceil\vec{\kappa}\rceil$ then $\textbf{lab}\,(Z\tau') = \kappa$. Assume $\kappa \leq \ell$ is in the transitive closure of $Z^{loc}C'$. So either $\kappa \leq \ell$ or $\kappa \leq \kappa'$, with $Z^{loc}(\kappa') = \ell$ and $\kappa' \notin \lceil\vec{\kappa}\rceil$ are in the transitive closure of C'. But these contradicts *3* above so we have that

$$\kappa \leq \ell \notin Z^{loc}C'^{+}.$$

Assume $\kappa \leq \kappa'$ is in the transitive closure of $Z^{loc}C'$ for some $\kappa' \notin \lceil\vec{\kappa}\rceil$. That can only occur if $\kappa \leq \kappa''$ for some $\kappa' \notin \lceil\vec{K}\rceil$ is in the transitive closure of C' and $Z^{loc}(\kappa'') = \kappa'$. But this again contradicts 3 above, so we have that

$$\kappa \leq \kappa' \ \notin Z^{loc}C'^{+} \textit{ for } \kappa' \notin \lceil\vec{\kappa}\rceil.$$

Hence, by Lemma 3.6 with (A.8), (A.9) and (A.10) we have

$$\textbf{esc?}(\forall\vec{\alpha}\vec{\kappa}.\,Z\tau'\backslash ZC')$$

as desired. ∎

Lemma 4.12 *If* $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C]; \Delta \vdash M : \tau'\backslash C'$, $C \cup C''$ *is valid and* $FLV(C'') \cap \lceil\vec{\kappa}\rceil = \emptyset$ *then* $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C \cup C'']; \Delta \vdash M : \tau'\backslash C'''$, *where* $C''' \subseteq C' \cup C''$.

Proof. By induction on the type derivation of M. The interesting case is when $M = x$.

case x**.** If $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C]; \Delta \vdash x : \tau'\backslash C'$ then by the typing rule for variables $\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C \succ \tau'\backslash C'$. By Definition 3.14, we know that there exists $Z = (Z^{type}, Z^{loc})$ such that $Dom(Z^{loc}) = \lceil\vec{\kappa}\rceil$, $Dom(Z^{type}) = \lceil\vec{\alpha}\rceil$, $Z\,\tau = \tau'$ and $Z^{loc}\,C = C'$.

We have that $Z\,(C \cup C'') \subseteq C' \cup C''$ since

$$\begin{array}{rcl} Z^{loc}\,(C \cup C'') & = & Z^{loc}\,C \cup Z^{loc}\,C'' \\ & = & C' \cup Z^{loc}\,C'' \\ & = & C' \cup C''. \end{array}$$

Hence $\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C \cup C'' \succ \tau'\backslash C \cup C''$ and by the rule for typing variables we conclude

$$\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C \cup C'']; \Delta \vdash x : \tau'\backslash C \cup C''$$

as desired. ∎

Lemma 4.13 *If* $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C]; \Delta \vdash M : \tau'\backslash C'$ *and* $C'' \subseteq C$ *then* $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C'']; \Delta \vdash M : \tau'\backslash C'''$, *where* $C''' \subseteq C'$.

Proof. By induction on the type derivation for M. The interesting case is when $M = x$.

case x**.** If $\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C]; \Delta \vdash x : \tau'\backslash C'$ then by the typing rule for variables $\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C \succ \tau'\backslash C'$.

By Definition 3.14, we know that there exists $Z = (Z^{type}, Z^{loc})$ such that $Dom(Z^{loc}) = \lceil\vec{\kappa}\rceil$, $Dom(Z^{type}) = \lceil\vec{\alpha}\rceil$ and $Z\,\tau = \tau'$ and $Z^{loc}\,C = C'$.

Since, by assumption, $C'' \subseteq C$, we have that $Z^{loc}\,C'' \subseteq C'$. Let $Z^{loc}\,C'' = C'''$. Hence we have that $\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C'' \succ \tau'\backslash C'''$ and we conclude that

$$\Gamma[x \mapsto \forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C'']; \Delta \vdash x : \tau'\backslash C'''$$

as desired. ∎

A.3 Lemmas from Chapter 7

Lemma A.3 *If Θ is polymorphic consistent then*

1. *$Z\Theta$ is polymorphic consistent,*
2. *$Z\Gamma_\Theta$ is polymorphic consistent,*
3. *$Gen_{\mathcal{W}_\epsilon}(\delta, \Theta)$ is polymorphic consistent.*

PROOF. Part *1* follows by the definition of polymorphic consistency and by the definition of application of an extended type substitution to an extended constrained type scheme. From *1* we also have that $\Gamma_{Z\Theta}$ is polymorphic consistent. Part *2* then follows from the fact that $Z\Gamma_\Theta = \Gamma_{Z\Theta}$. The proof of *3* is analogous to the proof of Lemma 3.2. ∎

We define the set $InvLV(Z)$ of locality variables *involved* in $Z = (Z^{type}, Z^{loc})$ in the following way

$$InvLV(Z^{type}, Z^{loc}) = FLV(Range(Z^{type})) \cup Dom(Z^{loc}) \cup FLV(Range(Z^{loc})).$$

The set $InvTV(Z)$ of type variables *involved* in $Z = (Z^{type}, Z^{loc})$ is defined as

$$InvTV(Z^{type}, Z^{loc}) = Dom(Z^{type}) \cup FTV(Range(Z^{type})).$$

Lemma A.4 *If $\mathcal{W}_\epsilon(\Theta, M) = (Z, \overline{\mathsf{M}}, X)$ then*

1. *$BLV(\Theta) \cap InvLV(Z) = \emptyset$,*
2. *$BTV(\Theta) \cap InvTV(Z) = \emptyset$.*

PROOF. By induction on the structure of M. ∎

Lemma 7.13 *If Θ is polymorphic consistent and $\mathcal{W}_\epsilon(\Theta, M) = (Z,\ \overline{\mathsf{M}}_{\tau\backslash C,C'},\ X)$ then $Z\Gamma_\Theta \vdash_{aux} M : \tau\backslash C$.*

PROOF. By induction on the structure of M.

case x

By Lemma 7.3(*2*) and $\mathcal{W}_\epsilon(1)$ we have $\Gamma_\Theta(x) \succ \tau\backslash C$. By the assumption that Θ is polymorphic consistent Γ_Θ is polymorphic consistent, By Lemma 4.7 we then get $Z_\epsilon(\Gamma_\Theta(x)) \succ Z_\epsilon(\tau\backslash C)$, which is equivalent to

$$(Z_\epsilon \Gamma_\Theta)(x) \succ Z_\epsilon(\tau\backslash C). \tag{A.11}$$

By $\mathcal{W}_\epsilon(2)$ and Lemma 7.2(*1*)

$$\nexists\ \epsilon \leq \kappa \ \in \ (Z_\epsilon C)^+. \tag{A.12}$$

Because of (A.12) we can use (A.11) as premise for the $\mathsf{VAR}_{\text{aux}}$ rule to get

$$Z_\epsilon\,\Gamma_\Theta \vdash_{\text{aux}} x\colon Z_\epsilon\,(\tau \backslash C).$$

case $\mathtt{fn}\, x.\, P$

$\Theta[x \mapsto \alpha^{\kappa'}]$ is polymorphic consistent, so by $\mathcal{W}_\epsilon(3)$ and the induction hypothesis we have $Z\,\Gamma_{\Theta[x\mapsto\alpha^{\kappa'}]} \vdash_{\text{aux}} P\colon \tau\backslash C$ which is the same as

$$Z\,\Gamma_\Theta[x \mapsto Z(\alpha^{\kappa'})] \vdash_{\text{aux}} P\colon \tau\backslash C.$$

Using the type judgement above as premise for the $\mathsf{ABS}_{\text{aux}}$ rule, choosing κ as the label for the function type, gives us

$$Z\,\Gamma_\Theta \vdash_{\text{aux}} \mathtt{fn}\, x.\, P\colon Z(\alpha^{\kappa'}) \xrightarrow{\kappa} \tau\backslash C.$$

case $N\ P$

By $\mathcal{W}_\epsilon(6)$ and the induction hypothesis we have

$$Z_1\,\Gamma_\Theta \vdash_{\text{aux}} N\colon \tau_1\backslash C_1. \tag{A.13}$$

By assumption that Θ is polymorphic consistent and Lemma A.3.(*1*)

$$Z_1\Theta \ \textit{is polymorphic consistent.} \tag{A.14}$$

Hence, by $\mathcal{W}_\epsilon 7$ and the induction hypothesis we have $Z_2\,\Gamma_{Z_1\Theta} \vdash_{\text{aux}} P\colon \tau_2\backslash C_2$, which one can readily check is the same as

$$Z_2\,Z_1\,\Gamma_\Theta \vdash_{\text{aux}} P\colon \tau_2\backslash C_2. \tag{A.15}$$

By $\mathcal{W}_\epsilon 8$ and by Lemma 7.1(*1*) we have that

$$Z_u Z_2 \tau_1 \ \equiv \ Z_u \tau_2 \xrightarrow{Z_u\kappa} Z_u\,(\alpha^{\kappa'}). \tag{A.16}$$

By $\mathcal{W}_\epsilon 9$ and Lemma 7.2(*1*) one readily checks that

$$\nexists\ \epsilon \leq \kappa \in (Z_\epsilon Z_u\, Z_2\, C_1)^+ \tag{A.17}$$

and also

$$\nexists\ \epsilon \leq \kappa \in (Z_\epsilon Z_u\, C_2)^+. \tag{A.18}$$

By A.14 we have that $\Gamma_{Z\Theta}$ is polymorphic consistent. Because $\Gamma_{Z\Theta} = Z_1\Gamma_\Theta$ we have

$$Z_1\Gamma_\Theta \textit{ is polymorphic consistent.} \tag{A.19}$$

By Lemma 4.8 with (A.13), (A.19) and (A.17) we get

$$Z_\epsilon Z_u\, Z_2\, Z_1\, \Gamma_\Theta \vdash_{\text{aux}} N\colon Z_\epsilon Z_u\, Z_2\, (\tau_1 \backslash C_1). \tag{A.20}$$

With (A.16) we rewrite (A.20) as

$$Z_\epsilon Z_u\, Z_2\, Z_1\, \Gamma_\Theta \vdash_{\text{aux}} N\colon\ Z_\epsilon Z_u \tau_2 \overset{Z_\epsilon Z_u \kappa}{\to} Z_\epsilon Z_u\, (\alpha^{\kappa'}) \backslash Z_\epsilon Z_u\, Z_2 C_1. \tag{A.21}$$

By (A.14) above and Lemma A.3 we have that $Z_2(Z_1\Theta)$ is polymorphic consistent which implies that $Z_2 Z_1\, \Gamma_\Theta$ is polymorphic consistent. Hence, by Lemma 4.8 with (A.15) and (A.18) we get

$$Z_\epsilon Z_u\, Z_2\, Z_1\, \Gamma_\Theta \vdash_{\text{aux}} P\colon (Z_\epsilon Z_u \tau_2) \backslash Z_\epsilon Z_u C_2. \tag{A.22}$$

By the rule $\mathsf{APP}_{\text{aux}}$ using (A.21) and (A.22) as premises we get

$$Z_\epsilon Z_u\, Z_2\, Z_1\, \Gamma_\Theta \vdash_{\text{aux}} N\ P\colon Z_\epsilon Z_u\, (\alpha^{\kappa'}) \backslash Z_\epsilon Z_u\, (Z_2 C_1 \cup C_2).$$

case (N, P)

By induction hypothesis and $\mathcal{W}_\epsilon 10$ we have

$$Z_1\, \Gamma_\Theta \vdash_{\text{aux}} N\colon \tau_1 \backslash C_1. \tag{A.23}$$

By the same reasoning used in the previous case we have

$$Z_1\, \Theta \textit{ is polymorphic consistent,} \tag{A.24}$$

and

$$Z_1\, \Gamma_\Theta \textit{ is polymorphic consistent.} \tag{A.25}$$

By $\mathcal{W}_\epsilon(11)$, (A.24) above and the induction hypothesis we have

$$Z_2\, Z_1\, \Gamma_\Theta \vdash_{\text{aux}} P\colon \tau_2 \backslash C_2. \tag{A.26}$$

$\mathcal{W}_\epsilon 12$ and Lemma 7.2(*1*) imply that

$$\nexists\ \epsilon \leq \kappa \in (Z_\epsilon Z_2\, C_1)^+, \tag{A.27}$$

and

$$\nexists\ \epsilon \leq \kappa \in (Z_\epsilon C_2)^+. \tag{A.28}$$

By Lemma 4.8 with (A.23), (A.25) and (A.27) we get

$$Z_\epsilon Z_2\, Z_1\, \Gamma_\Theta \vdash_{\text{aux}} N \colon Z_\epsilon Z_2\, (\tau_1 \backslash C_1). \tag{A.29}$$

By Lemma A.3.(*2*) and (A.24), $Z_2\Gamma_{Z_1\Theta}$ is polymorphic consistent which implies that $Z_2 Z_1 \Gamma_\Theta$ is polymorphic consistent. Then, by Lemma 4.8 with (A.26) and (A.28) we get

$$Z_\epsilon\, Z_2\, Z_1\, \Gamma_\Theta \vdash_{\text{aux}} P \colon (Z_\epsilon\, \tau_2) \backslash Z_\epsilon\, C_2. \tag{A.30}$$

By the way C_3 and C_4 are constructed in $\mathcal{W}_\epsilon 13$ and $\mathcal{W}_\epsilon 14$ we know they are valid. Hence, by the Union Lemma with (A.27) and (A.28) we get

$$\nexists\, \epsilon \leq \kappa \in (Z_\epsilon(Z_2\, C_1 \cup C_2) \cup C_3 \cup C_4)^+. \tag{A.31}$$

Because (A.31) holds we can use the $\mathsf{PAIR}_{\text{aux}}$ rule with (A.29) and (A.30) as premises and get

$$Z_\epsilon\, Z_2\, Z_1\, \Gamma_\Theta \vdash_{\text{aux}} (N, P) \colon Z_\epsilon(Z_2 \tau_1 \times^\kappa \tau_2) \backslash Z_\epsilon(Z_2\, C_1 \cup C_2) \cup C_3 \cup C_4.$$

case `let` $x = V$ `in` P `end`

By the induction hypothesis and $\mathcal{W}_\epsilon 15$ we have

$$Z_1 \Gamma_\Theta \vdash_{\text{aux}} V \colon \tau_1 \backslash C_1. \tag{A.32}$$

By similar reasoning used in the previous cases we have that

$$Z_1\, \Theta \text{ \textit{is polymorphic consistent},} \tag{A.33}$$

and

$$Z_1\, \Gamma_\Theta \text{ \textit{is polymorphic consistent.}} \tag{A.34}$$

By $\mathcal{W}_\epsilon(21)$ and Lemma 7.2.(*1*) we have

$$\nexists \epsilon \leq \kappa \in Z_{\epsilon 3} Z_2 Z_{\epsilon 2} Z_{\epsilon 1}\, C_1^+. \tag{A.35}$$

By Lemma 4.8 with (A.32), (A.34) and (A.35) we get

$$Z_{\epsilon 3} Z_2 Z_{\epsilon 2} Z_{\epsilon 1} Z_1 \Gamma_\Theta \vdash_{\text{aux}} V \colon Z_{\epsilon 3} Z_2 Z_{\epsilon 2} Z_{\epsilon 1} (\tau_1 \backslash C_1). \tag{A.36}$$

By A.33 and Lemma A.3.(*1*) we have

$$Z_{\epsilon 2} Z_{\epsilon 1} Z_1 \Theta \text{ \textit{is polymorphic consistent.}} \tag{A.37}$$

By $\mathcal{W}_\epsilon(19)$ we have

$$\varsigma = Gen_{\mathcal{W}_\epsilon}\,(Z_{\epsilon 2}Z_{\epsilon 1}(\tau_1\backslash C_1, C_1'), Z_{\epsilon 2}Z_{\epsilon 1}Z_1\Theta) \tag{A.38}$$

By Lemma A.3.(*3*) with (A.37) and (A.38) above we have that

$$\varsigma \textit{ is polymorphic consistent.} \tag{A.39}$$

By (A.37) and (A.39) we get

$$Z_{\epsilon 2}Z_{\epsilon 1}Z_1\Theta\,[x \mapsto \varsigma] \textit{ is polymorphic consistent.} \tag{A.40}$$

By $\mathcal{W}_\epsilon(20)$ and (A.40) we can use the induction hypothesis to get

$$Z_2\,(\Gamma_{(Z_{\epsilon 2}Z_{\epsilon 1}Z_1\Theta)[x\mapsto\varsigma]}) \vdash_{\text{aux}} P\colon \tau_2\backslash C_2,$$

which one readily checks is equivalent to

$$Z_2\,Z_{\epsilon 2}Z_{\epsilon 1}Z_1\Gamma_\Theta[x \mapsto Z_2 Reg(\varsigma)] \vdash_{\text{aux}} P\colon \tau_2\backslash C_2. \tag{A.41}$$

By $\mathcal{W}_\epsilon(21)$ and Lemma 7.2.(*1*)

$$\nexists\,\epsilon \leq \kappa \in (Z_{\epsilon 3}\,C_2)^+. \tag{A.42}$$

Then by Lemma 4.8 with (A.42) and (A.41) we get

$$Z_{\epsilon 3}\,Z_2\,Z_{\epsilon 2}Z_{\epsilon 1}Z_1\Gamma_\Theta[x \mapsto Z_{\epsilon 3}\,Z_2 Reg(\varsigma)] \vdash_{\text{aux}} P\colon Z_{\epsilon 3}\,(\tau_2\backslash C_2). \tag{A.43}$$

We now consider two possibilities:

case $x \notin FV(P)$

In this case, by Lemma 4.2 and (A.43) above we get

$$Z_{\epsilon 3}\,Z_2\,Z_{\epsilon 2}Z_{\epsilon 1}Z_1\Gamma_\Theta[x \mapsto \sigma] \vdash_{\text{aux}} P\colon Z_{\epsilon 3}\,(\tau_2\backslash C_2), \tag{A.44}$$

where

$$\sigma = Gen(Z_{\epsilon 3}\,Z_2\,Z_{\epsilon 2}\,Z_{\epsilon 1}\,(\tau_1\backslash C_1), Z_{\epsilon 3}\,Z_2\,Z_{\epsilon 2}\,Z_{\epsilon 1}\,Z_1\Gamma_\Theta).$$

Then by the rule for typing let expressions with (A.44) and (A.36) as premises we get

$$Z_{\epsilon 3}Z_2Z_{\epsilon 2}Z_{\epsilon 1}Z_1\Gamma_\Theta \vdash_{\text{aux}} \texttt{let}\ x = V\ \texttt{in}\ P\,\texttt{end} : Z_{\epsilon 3}\,(\tau_2\backslash C_2 \cup Z_2Z_{\epsilon 2}Z_{\epsilon 1}C_1)$$

and we are done.

case $x \in FV(P)$

For this subcase we have that

$$Z_{\epsilon 3} \;=\; Id^{loc}. \tag{A.45}$$

since in this case $Z_2 Z_{\epsilon 2} Z_{\epsilon 1} C_1 \subseteq C_2$ and also $Z_2 Z_{\epsilon 2} Z_{\epsilon 1} C_1' \subseteq C_2'$. Because C_2 and C_2' are valid $Z_2 Z_{\epsilon 2} Z_{\epsilon 1} C_1$ and $Z_2 Z_{\epsilon 2} Z_{\epsilon 1} C_1'$ are valid.

We also have that

$$Gen_{\mathcal{W}_\epsilon}(Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, (\tau_1 \backslash C_1, C_1'),\; Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, Z_1\, \Theta) = Z_2\, \varsigma \tag{A.46}$$

since, by Lemma A.4, $BLV((Z_{\epsilon 2} Z^{\epsilon 1} Z_1 \Theta)[x \mapsto \varsigma]) \cap\; InvLV(Z_2) = \emptyset$ and $BTV((Z_{\epsilon 2} Z^{\epsilon 1} Z_1 \Theta)[x \mapsto \varsigma]) \cap\; InvTV(Z_2) = \emptyset$, which implies that $BLV(\varsigma) \cap\; InvLV(Z_2) = \emptyset$ and $BTV(\varsigma) \cap\; InvTV(Z_2) = \emptyset$.

With these two results we have

$$\sigma \succ Z_{\epsilon 3}\, Z_2\, Reg(\varsigma) \tag{A.47}$$

since,

$$\begin{array}{lll}
 & \sigma & \\
= & Gen(Z_{\epsilon 3}\, Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, (\tau_1 \backslash C_1),\; Z_{\epsilon 3}\, Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, Z_1 \Gamma_\Theta) & \\
= & Gen(Id\, Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, (\tau_1 \backslash C_1),\; Id\, Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, Z_1 \Gamma_\Theta) & (A.45) \\
= & Gen(Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, (\tau_1 \backslash C_1),\; Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, Z_1\, \Gamma_\Theta) & \\
= & Gen(Reg\,(Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, (\tau_1 \backslash C_1, C_1')),\; Reg\,(Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, Z_1\, Reg(\Theta)) & \\
\succ & Reg\;\; Gen_{\mathcal{W}_\epsilon}(Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, (\tau_1 \backslash C_1, C_1'),\; Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, Z_1 \Theta) & \text{Lemma 7.4} \\
= & Reg\;\, (Z_2\; Gen_{\mathcal{W}_\epsilon}(Z_{\epsilon 2}\, Z_{\epsilon 1}\, (\tau_1 \backslash C_1),\; Z_{\epsilon 2}\, Z_{\epsilon 1}\, Z_1 \Gamma_\Theta)) & (A.46) \\
= & Reg\;\, (Z_2\, \varsigma) & \\
= & Z_2 Reg(\varsigma) & \\
= & Z_{\epsilon 3} Z_2 Reg(\varsigma) &
\end{array}$$

By (A.47), (A.43) and Lemma 4.3 we get

$$Z_{\epsilon 3}\, Z_2\, Z_{\epsilon 2}\, Z_{\epsilon 1}\, Z_1\, \Gamma_\Theta[x \mapsto \sigma] \vdash_{\text{aux}} P\colon Z_{\epsilon 3}\, (\tau_2 \backslash C_2), \tag{A.48}$$

We can then use (A.36) and (A.48) as premises for the `Let` rule to conclude

$$Z_{\epsilon 3} Z_2 Z_{\epsilon 2} Z_{\epsilon 1} Z_1 \Gamma_\Theta \vdash_{\text{aux}} \texttt{let}\ x = V\ \texttt{in}\ P\ \texttt{end} : Z_{\epsilon 3}\, (\tau_2 \backslash C_2 \cup Z_2 Z_{\epsilon 2} Z_{\epsilon 1} C_1)$$

and we are done. ∎

The following lemma says that the set X returned by $\mathcal{W}_\epsilon$ has the occurrences of free variables in the type reconstruction for the input expression. This set is only used for adding elements to the extra set of constraints in line $\mathcal{W}_\epsilon(5)$.

Lemma A.5 *If* $\mathcal{W}_\epsilon(\Theta, M) = (Z, \overline{\mathsf{M}}, X)$ *then* $X = FVoccur(\overline{\mathsf{M}})$

Proof. By induction on the structure of M. ∎

The following lemma says about the role of the extra set on constraints. If a function abstraction is a subexpression of the explicitly typed expression $\overline{\mathsf{M}}_{\tau\backslash C,C'}$, returned by $\mathcal{W}_\epsilon$, then the constraints relating the label of the abstraction and the labels of its free variable occurrences are collected at the extra constraint set C' of $\overline{\mathsf{M}}$.

Lemma A.6 *If*

- $\mathcal{W}_\epsilon(\Theta, M) = (Z,\ \overline{\mathsf{M}}_{\tau\backslash C,C'},\ X)$,
- $\overline{\mathsf{F}}'_\delta\ \in\ Sub(\overline{\mathsf{M}}_{\tau\backslash C,C'})$,
- $\overline{\mathsf{y}}_\delta \in FVoccur(\overline{\mathsf{F}})$, *and*
- $\mathbf{lab}\,(\delta)$ *is defined,*

then $\mathbf{lab}\,(\delta') \leq \mathbf{lab}\,(\delta) \in C'$.

Proof. By induction on the structure of M. ∎

Lemma A.7 *If* $\mathcal{W}_\epsilon(\Theta, M) = (Z, \overline{\mathsf{M}}_{\tau\backslash C,C'}, X)$, *then*

1. $\epsilon \leq \phi \in C'^{+}$ *implies* $\phi = \epsilon$,
2. $\kappa \leq \phi \in C'^{+}$ *with* $\kappa \in BLV(\overline{\mathsf{M}})$, *implies* $\phi = \epsilon$.

Proof. By induction on the structure of M. ∎

The following lemma can be viewed as a version of the Context-Filler Type lemma for explicitly typed expressions.

Lemma A.8 *Let* $\overline{\mathsf{M}}_{\tau_1\backslash C_1,C_1'}$ *be such that* $\Gamma_{\overline{\mathsf{M}}} \vdash_{aux} M : \tau_1 \setminus C_1$. *If* $\overline{\mathsf{N}}_{\tau_2\backslash C_2,C_2'}$ *is a subexpression of* $\overline{\mathsf{M}}_{\tau_1\backslash C_1,C_1'}$ *then* $\Gamma_{\overline{\mathsf{N}}} \vdash_{aux} N : \tau_2 \setminus C_2$.

Proof. By induction on the structure of $\overline{\mathsf{M}}$. ∎

Lemma 7.14 *If*

1. $\mathcal{W}_\epsilon(\Theta, M) = (Z,\ \overline{\mathsf{M}},\ X)$*, for* Θ *polymorphic consistent,*

2. $\overline{\mathsf{F}}_\delta \in \mathit{Sub}(\overline{\mathsf{M}})$*, and*

3. $\mathbf{lab}\,(\delta) = \epsilon$ *or* $\mathbf{lab}\,(\delta) \in \mathit{BLV}(\overline{\mathsf{M}})$*,*

then $\mathbf{esc?}(\Gamma_{\overline{\mathsf{F}}}, F)$

Proof. According to the Definition 3.17 in order to prove $\mathbf{esc?}(\Gamma_{\overline{\mathsf{F}}}, F)$ we have to prove that for all $y \in \mathit{FV}(F)$ we have $\mathbf{esc?}(\sigma)$ where $\sigma = \Gamma_{\overline{\mathsf{F}}}(y)$.

Assume that $y \in \mathit{FV}(F)$. Hence $(y : \varsigma) : \tau_2 \backslash C_2, C_2' \in \mathit{FVoccur}(\overline{\mathsf{F}})$ for some ς, τ_2, C_2 and C_2'. Because $\overline{\mathsf{F}} \in \mathit{Sub}(\overline{\mathsf{M}})$ we also have $(y : \varsigma) : \tau_2 \backslash C_2, C_2' \in \mathit{Sub}(\overline{\mathsf{M}})$.

By Lemma 7.5 and A.8 we have that $[y \mapsto \sigma] \vdash_{\text{aux}} y \colon \tau_2 \backslash C_2$ which by the rule $\mathsf{VAR}_{\text{aux}}$ implies that

$$\sigma \succ_{\text{aux}} \tau_2 \backslash C_2,\ \textit{with } C_2 \textit{ valid.} \tag{A.49}$$

If $\mathbf{lab}\,(\tau_2)$ is undefined then, by the definition of escape-expectant constrained type scheme and (A.49) we have that $\mathbf{esc?}(\sigma)$.

Suppose that $\mathbf{lab}\,(\tau_2)$ is defined. Then by Lemma A.6 with assumptions 1 and 2 we have that

$$\mathbf{lab}\,(\gamma) \leq \mathbf{lab}\,(\tau_2) \in C'.$$

By the assumption 3 we have that $\mathbf{lab}\,(\gamma) = \epsilon$ or $\mathbf{lab}\,(\gamma) \in \mathit{BLV}(\overline{\mathsf{M}})$. By Lemma A.7, we know that $\mathbf{lab}\,(\tau_2) = \epsilon$. Then, by the definition of escape-expectant constrained type scheme and (A.49) we have that $\mathbf{esc?}(\sigma)$. ∎

The following is the proof that $\mathcal{L}$ is sound.

Lemma 7.15 *Suppose that* $\overline{\mathsf{M}}_{\tau\backslash C, C'}$ *and* $\overline{\mathsf{N}}_{\tau_2\backslash C_2, C_2'}$ *are such that*

1. $\Gamma_{\overline{\mathsf{M}}} \vdash_{aux} M \colon \tau \backslash C$ *and*

2. *if* $\overline{\mathsf{F}}_\delta \in \mathit{Sub}(\overline{\mathsf{M}})$ *and* $\mathbf{lab}\,(\delta) = \epsilon$ *or* $\mathbf{lab}\,(\delta) \in \mathit{BLV}(\overline{\mathsf{M}})$ *then* $\mathbf{esc?}(\Gamma_{\overline{\mathsf{F}}}, F)$ *and*

3. $\overline{\mathsf{N}}_{\tau_2\backslash C_2, C_2'} \in \mathit{Sub}(\overline{\mathsf{M}}_{\tau\backslash C, C'})$.

Then $\mathcal{L}(\overline{\mathsf{M}}, \overline{\mathsf{N}}) = Z_\ell$ *implies* $Z_\ell\, \Gamma_{\overline{\mathsf{N}}} \vdash N \colon Z_\ell\,(\tau_2 \backslash C_2)$.

Proof. By induction on the structure of $\overline{\mathsf{N}}$.

case $\overline{\mathsf{N}} \equiv (x\colon \varsigma) : \tau_2 \backslash C_2, C_2'$

We have $\mathcal{L}(\overline{\mathsf{M}}, (x : \varsigma) : \tau_2 \backslash C_2, C_2') = \mathit{Id}$. So we have to prove that $\mathit{Id}\,\Gamma_{(x:\varsigma):\tau_2\backslash C_2, C_2'} \vdash x\colon \mathit{Id}\,(\tau \backslash C)$.

By the assumptions that $\overline{\mathsf{M}}_{\tau\backslash C,C'}$ is such that $\Gamma_{\overline{\mathsf{M}}} \vdash_{\mathrm{aux}} M : \tau \backslash C$ and that $(x : \varsigma)\colon \tau_2 \backslash C_2, C_2' \in \mathit{Sub}(\overline{\mathsf{M}}_{\tau\backslash C,C'})$, by Lemma A.8 we get

$$\Gamma_{(x:\varsigma):\tau_2\backslash C_2, C_2'} \vdash_{\mathrm{aux}} x\colon \tau_2 \backslash C_2,$$

which is equivalent to

$$\mathit{Id}\,\Gamma_{(x:\varsigma):\tau_2\backslash C_2, C_2'} \vdash x\colon \mathit{Id}\,(\tau_2 \backslash C_2).$$

case $\overline{\mathsf{N}} \equiv (\mathtt{fn}\, x\colon \tau_1 \backslash C_1, C_1'.\ \overline{\mathsf{P}}_{\tau_2\backslash C_2, C_2'}) : \tau_1 \xrightarrow{\phi} \tau_2 \backslash C_1 \cup C_2, C_1' \cup C_2'$

By the assumption that $\overline{\mathsf{N}} \in \mathit{Sub}(\overline{\mathsf{M}}_{\tau\backslash C,C'})$ we have that

$$\overline{\mathsf{P}}_{\tau_2\backslash C_2, C_2'} \ \in \ \mathit{Sub}(\overline{\mathsf{M}}_{\tau\backslash C,C'}). \tag{A.50}$$

The assumption that $\Gamma_{\overline{\mathsf{N}}}$ is consistent implies that

$$\Gamma_{\overline{\mathsf{P}}} \textit{ is consistent.} \tag{A.51}$$

By $\mathcal{L}(3)$, (A.50), (A.51) and the induction hypothesis we get

$$Z_\ell'\,\Gamma_{\overline{\mathsf{P}}} \vdash P\colon Z_\ell'\,(\tau_2 \backslash C_2). \tag{A.52}$$

By Lemma A.3 with (A.51) we have that

$$Z_\ell\, Z_\ell'\,\Gamma_{\overline{\mathsf{P}}} \textit{ is consistent.} \tag{A.53}$$

By (A.52) we know that $Z_\ell'\,C_2$ is valid. Because ℓ is the only locality label in the range of Z_ℓ we have

$$Z_\ell\, Z_\ell'\, C_2 \textit{ is valid.} \tag{A.54}$$

By Lemma 4.8 with (A.52), (A.53) and (A.54) we get

$$Z_\ell Z_\ell'\,\Gamma_{\overline{\mathsf{P}}} \vdash P\colon Z_\ell Z_\ell'\,(\tau_2 \backslash C_2). \tag{A.55}$$

Note that if $x\colon \tau_1 \backslash C_1$ is not in $\Gamma_{\overline{\mathsf{P}}}$ we can *weakening* it to contain $x\colon \tau_1 \backslash C_1$. For simplicity, we then assume that $x\colon \tau_1 \backslash C_1$ is in $\Gamma_{\overline{\mathsf{P}}}$. One can then easily check that

$$\Gamma_{\overline{\mathsf{P}}} = \Gamma_{\overline{\mathsf{N}}}[x \mapsto \tau_1 \backslash C_1].$$

We can then write type judgement (A.55) above as

$$(Z_\ell Z'_\ell\,\Gamma_{\overline{\mathsf{N}}})\,[x \mapsto Z_\ell Z'_\ell\,(\tau_1\backslash C_1)] \vdash P\colon Z_\ell Z'_\ell\,(\tau_2\backslash C_2). \tag{A.56}$$

Observe also that the order in which the locality substitutions are composed in the algorithm is not important as they do not have locality variables in their range.

If ϕ is such that $Z_\ell Z'_\ell\,\phi = \ell$ we use the rule ABS with the type judgement above as premise to conclude

$$Z_\ell Z'_\ell\,\Gamma_{\overline{\mathsf{N}}} \vdash \mathtt{fn}\,x.P\colon Z_\ell Z'_\ell\,\tau_1 \overset{Z_\ell Z'_\ell\phi}{\to} Z_\ell Z'_\ell\,\tau_2\backslash Z_\ell Z'_\ell\,(C_1 \cup C_2).$$

If ϕ is such that $Z_\ell Z'_\ell\,\phi = \epsilon$ of $Z_\ell Z'_\ell\,\phi = \kappa$ we still have to prove that

$$\mathbf{esc?}(Z_\ell Z'_\ell\,\Gamma_{\overline{\mathsf{N}}}, \mathtt{fn}\,x.P)$$

In this case one readily checks that either $\phi = \epsilon$ or $\phi = \kappa$ and $\kappa \in BLV(\overline{\mathsf{M}})$. By assumption we then know that

$$\mathbf{esc?}(\Gamma_{\overline{\mathsf{N}}}, \mathtt{fn}\,x.P) \tag{A.57}$$

By Lemma 4.6 and (A.57), we get

$$\mathbf{esc?}(Z_\ell Z'_\ell\,\Gamma_{\overline{\mathsf{N}}},\mathtt{fn}\,x.P)$$

as desired.

case $\overline{\mathsf{N}} \equiv (\overline{\mathsf{P}}_{\tau_1\backslash C_1, C'_1}, \overline{\mathsf{Q}}_{\tau_2\backslash C_2, C'_2}) : \tau_1 \times^{\phi} \tau_2\backslash C_1 \cup C_2 \cup C3,$

By the assumption that $\overline{(\mathsf{P},\mathsf{Q})} \in Sub(\overline{\mathsf{M}})$ we have that

$$\overline{\mathsf{P}} \in Sub(\overline{\mathsf{M}}). \tag{A.58}$$

The assumption that $\Gamma_{\overline{(\mathsf{P},\mathsf{Q})}}$ is consistent implies that

$$\Gamma_{\overline{\mathsf{P}}} \textit{ is consistent.} \tag{A.59}$$

By $\mathcal{L}$.(4) and the induction hypothesis with (A.58) and (A.59) we get

$$Z_\ell\,\Gamma_{\overline{\mathsf{P}}} \vdash P\colon Z_\ell\,(\tau_1\backslash C_1,) \tag{A.60}$$

By (A.60) we have that $Z_\ell\,Z_1$ is valid. One then readily checks that

$$Z'_\ell Z_\ell\,Z_1 \textit{ is valid.} \tag{A.61}$$

By (A.59) and Lemma A.3 we get

$$Z'_\ell Z_\ell \Gamma_{\overline{\mathsf{P}}} \textit{ is consistent.} \tag{A.62}$$

By Lemma 4.8 with (A.60), (A.62) and (A.61) we get

$$Z'_\ell\, Z_\ell\, \Gamma_{\overline{\mathsf{P}}} \vdash P\colon Z'_\ell\, Z_\ell\, (\tau_1 \backslash C_1,) \tag{A.63}$$

One can easily check that

$$(Z'_\ell\, Z_\ell\, \Gamma_{\overline{\mathsf{(P,Q)}}})\,(x) = (Z'_\ell\, Z_\ell\, \Gamma_{\overline{\mathsf{P}}})\,(x) \textit{ for all } x \in FV(P) \tag{A.64}$$

By Lemma 4.2 with (A.63) and (A.64) we then get

$$Z'_\ell\, Z_\ell\, \Gamma_{\overline{\mathsf{(P,Q)}}} \vdash P\colon Z'_\ell\, Z_\ell\, (\tau_1 \backslash C_1,) \tag{A.65}$$

By a similar reasoning used above to prove (A.65) we can prove

$$Z'_\ell\, Z_\ell\, \Gamma_{\overline{\mathsf{(P,Q)}}} \vdash Q\colon Z'_\ell\, Z_\ell\, (\tau_2 \backslash C_2,). \tag{A.66}$$

With (A.65) and (A.66) as premises for the rule PAIR_{aux} we can conclude

$$Z_\ell Z'_\ell\, \Gamma_{\overline{\mathsf{(P,Q)}}} \vdash \texttt{(}P, Q\texttt{)}\colon Z_\ell Z'_\ell\, (\tau_1 \times^{\phi} \tau_2 \backslash C_1 \cup C_2 \cup C_3,)$$

case $\overline{\mathsf{N}} \equiv (\overline{\mathsf{P}}\ \overline{\mathsf{Q}}) : \delta$
Similar to the previous case

case $\overline{\mathsf{N}} \equiv \texttt{let}\ x\colon \varsigma = \overline{\mathsf{V}}\ \texttt{in}\ \overline{\mathsf{P}}\ \texttt{end} : \delta$
Similar to the previous case.

∎

Appendix B

Experiments

We explain some tests we did with two experimental versions of the Distributed Poly ML compiler. The goal of these tests was to evaluate the expected improvements on performance we hope to achieve with our analysis.

Distributed Poly/ML (DP/ML) is a variation on Standard ML (SML) that includes primitives for running threads on workstations distributed across a network . The language run-time system implements distributed shared memory, so that a program's address space is the same on all machines. From the programmer's viewpoint, the same reference on different machines refers to the same data object in the store. Stored data is in fact distributed among the machines running different threads. The DP/ML run time system checks for each dereference operation whether the associated data is available on the current machine, and if not, retrieve it over the network. A similar cost burden is also imposed for assignment operations.

The only differences between the two experimental versions of the compiler is that one of them has a *local dereference* operation (written `!!`) and a *local assignment* operation (written `::=`) in addition to the default dereference and assignment operations (`!` and `:=` respectively).

The tests were carried out in the following way: we wrote non-distributed imperative programs with no free occurrences of references in locality polymorphic functions. We executed them using the version of DP/ML with default dereference and assignment operations keeping record of their execution times. As all the references in non-distributed programs are local, we rewrote the same programs replacing `!` by `!!`, and `:=` by `::=` and we executed them, this time with the experimental version of DP/ML with local dereference and assignments, also keeping record of their execution times.

Our conclusion is that the programs using specialised dereference and assignment operations are both at least three times faster than those using the default

```
fun irev l =
  let val resultp = ref []
      and lp      = ref l
  in while not (null (!! lp)) do
         (resultp ::= hd (!! lp) :: !!resultp;
          lp      ::= tl (!! lp)
         );
     !!resultp
  end;
PolyML.timing true;
  irev buildlist(10000),
PolyML.timing false;
```

Figure B.1: Imperative *reverse* with local assignment and dereference

operations.

Figure B.2 and Figure B.1 give an example of how we measure the execution time in the experiments. Both figures show a non-distributed imperative program to reverse lists. The program in Figure B.2 uses `!` and `:=` and the program in Figure B.1 uses `!!` and *::=*. DP/ML provides the function `Poly.timing` to measure run-time efficiency. Calling `Poly.timing true` activates the time profiling. The time taken to evaluate the reversal of a list will be printed after the result. `Poly.timing false` deactivates the time facility.

```
fun irev l =
  let val resultp = ref []
      and lp      = ref l
  in while not (null (! lp)) do
         (resultp := hd (! lp) :: !resultp;
          lp      := tl (! lp)
         );
     !resultp
  end;
PolyML.timing true;
 irev buildlist(10000),
PolyML.timing false;
```

Figure B.2: Imperative *reverse* with default assignment and dereference