UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

COMPILING THE π-CALCULUS INTO A MULTITHREADED TYPED ASSEMBLY LANGUAGE

Tiago Soares Cogumbreiro Garcia

MESTRADO EM INFORMÁTICA

2009

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

COMPILING THE π-CALCULUS INTO A MULTITHREADED TYPED ASSEMBLY LANGUAGE

Tiago Soares Cogumbreiro Garcia

DISSERTAÇÃO

Projecto orientado pelo Prof. Doutor Francisco Martins

MESTRADO EM INFORMÁTICA

2009

Agradecimentos

Agradeço ao meu orientador, Francisco Martins, pelo seu apoio, paciência, rigor e minúcia. Ensinou-me a ter força para “não morrer na praia”, a ter perseverança para superar últimos 10% do trabalho sem descurar no primor. A qualidade científica que imprime no seu trabalho foi fundamental para o desenvolvimento desta tese.

Agradeço também ao Vasco Vasconcelos por me ensinar a olhar pela primeira vez algo que o sei de cor. Também por me relembrar da máxima: “premature optimisation is the root of all evil”. O seu contributo é também marcante no produto final que apresento.

Aproveito para agradecer a ambos pela oportunidade de participar numa visão que me permitiu trabalhar as fundações do meu conhecimento na licenciatura, desenvolver um trabalho mais teórico no mestrado e que me abriu portas para um doutoramento promissor. É uma honra trabalhar convosco.

Resumo

Restrições físicas e eléctricas limitam o aumento da velocidade do relógio dos processadores, pelo que não se espera que o poder computacional por unidade de processamento aumente muito num futuro próximo. Em vez disso, os fabricantes estão a aumentar o número de unidades de processamento (*cores*) por processador para continuarem a criar produtos com aumentos de performance. A indústria fez grandes investimentos em projectos como o RAMP e BEE2 que permitem a emulação de arquitecturas multi-core, mostrando interesse em suportar as fundações para a investigação de *software* que se destina a essas arquitecturas.

Para tirar vantagem de arquitecturas multi-core, tem de se dominar tanto a programação concorrente como a paralela. Com a grande disponibilidade de sistemas paralelos que vai desde sistemas embebidos até a super computadores, acreditamos que os programadores têm de fazer uma mudança de paradigma passando da programação sequencial para a programação paralela e produzir *software* adaptado, de raiz, a plataformas multi-core.

O *multithreading* é uma escolha bem conhecida e usada pela indústria para desenvolver sistemas explicitamente paralelos. Os *locks* são um mecanismo utilizado para sincronizar programas *multithreaded* de uma forma altamente eficiente. Porém, é comum surgirem problemas relacionados com concorrência como deter um *lock* tempo demais, não compreender quando se usam *locks* leitores/escritores ou escolher um mecanismo de sincronização desadequado. Estes problemas podem ser mitigados com recurso a abstracções de concorrência.

Os tipos e as regras de tipos são formas simples e eficazes de garantir segurança. Os tipos não são só usados por sistemas de tipos para garantir que programas não têm erros de execução, mas servem também como especificações (verificáveis de uma forma automática). A informação dada pelos tipos é expressiva o suficiente para representar propriedades operacionais importantes, como segurança de memória. Os compiladores que *preservam tipos* perduram a informação dada pelos tipos por todos os passos de compilação, permitindo uma compilação mais segura e que preserva a semântica (representada pelos tipos).

Propomos um compilador que endereça os problemas que levantámos até agora:

- uma linguagem fonte com abstracções para concorrência e para o paralelismo;

- um compilador que se destine a uma arquitectura multi-core;
- uma tradução que preserve a informação dada pelos tipos.

Em relação à linguagem fonte, os cálculos de processos evidenciam-se como um bom modelo de programação para a computação concorrente. O cálculo π, em particular, tem uma semântica bem compreendida, consiste num conjunto pequeno de operadores em que a comunicação é o passo fundamental de computação. Os cálculos de processos oferecem esquemas de compilação natural que expõe o paralelismo ao nível dos *threads*. Por estas razões, escolhemos o cálculo π simplesmente tipificado como linguagem fonte.

Vasconcelos e Martins propõem o MIL como uma linguagem *assembly* tipificada para arquitecturas *multithreaded*, um modelo que assenta numa máquina abstracta multi-core com memória principal partilhada. Esta linguagem fortemente tipificada oferece as seguintes propriedades de segurança: de memória, de controlo de fluxo e de liberdade de *race conditions*. O MIL contradiz a ideia que considera a associação entre a memória e *locks* uma convenção, ao torná-la explícita na linguagem. O sistema de tipos faz cumprir uma política de utilização de *lock* que inclui: proibir apanhar um *lock* em posse (fechado), proibir libertar um *lock* que não está em posse e faz com que os *threads* não se esqueçam de libertar todos os *locks* no final da sua execução.

Propomos uma tradução que preserva os tipos do cálculo π para o MIL. O cálculo π é uma álgebra de processos para descrever *mobilidade*: uma rede de processos interligados computa comunicando ligações (ou referências a outros processos). No cálculo π, a comunicação reconfigura dinamicamente a rede, fazendo com que os processos passem a estar visíveis a diferentes nós quando o sistema evolui. Ao traduzirmos o cálculo π em MIL, partimos de uma linguagem onde os processos comunicam através de passagem de mensagens e chegamos a uma linguagem onde *threads* comunicam por memória partilhada.

O processo de compilação não é directo nem trivial: certas abstracções, como canais, não têm uma representação complementar no MIL. Para ajudar a tradução, desenvolvemos, na linguagem de destino, uma biblioteca de tampões não limitados e polimórficos que são usados como canais. Estes tampões não limitados são monitores de Hoare, daí introduzirem uma forma de sincronização aos *threads* que estejam a aceder o tampão, e que encapsulam a manipulação directa de *locks*. A disciplina de *locks* do MIL permite representar explicitamente a noção da transferência ininterrupta da região crítica dos monitores—vai do *thread* que assinala a condição, e que termina, para o *thread* que está à espera nessa mesma condição, e que é activado. Os tampões não limitados são uma boa forma de representar canais, o que por sua vez simplifica a tradução, já que enviar uma mensagem corresponde a colocar um elemento no tampão, e que receber uma mensagem equivale a retirar um elemento do tampão. Impomos uma ordem FIFO no tampão, para assegurarmos que as mensagens enviadas têm a oportunidade de serem alguma vez recebidas.

A função de tradução que definimos é uma especificação formal do compilador. A tradução do cálculo π para MIL comporta a tradução de tipos, de valores e de processos. Os compiladores que preservam os tipos dão garantias em termos de segurança (os programas gerados não vão correr mal) e também em termos de correcção parcial (as propriedades semânticas dadas pelos tipos perduram no programa gerado). O nosso resultado principal é, portanto, uma tradução que preserva os tipos: o compilador gera programas MIL correctos ao nível dos tipos para processos π fechados e bem tipificados. Outra preocupação da nossa função de tradução é que o programa gerado tente manter o nível de concorrência do programa fonte, o que inclui a criação dinâmica de *threads* e a sincronização entre *threads*.

As contribuições deste trabalho são:

- Um algoritmo de compilação que preserva os tipos, mostrando a flexibilidade do MIL num ambiente tipificado e *race-free*.

- Exemplos de programação e estruturas de dados feitos em MIL. Mostramos a implementação de tampões polimórficos não limitados sob a forma de monitores, de variáveis de condição genéricas (primitivas dos monitores) e de filas polimórficas. Também descrevemos como codificar monitores na linguagem MIL.

- Ferramentas. Criámos um protótipo para o compilador de π para MIL, o que consiste em: o analisador sintáctico, o analisador semântico (estático) e o gerador de código. Refinámos o protótipo do MIL: adicionámos suporte para tipos universais e existenciais, *locks* de leitores/escritores, locks lineares e tuplos locais. Criámos uma *applet* Java que mostra de uma forma rápida e intuitiva o nosso trabalho sem ser necessário qualquer instalação (desde que o navegador *web* suporte *applets* de Java): podemos gerar código MIL a partir de código π, alterar o código MIL gerado e executá-lo. Os nossos protótipos estão disponíveis *on-line*, em `http://gloss.di.fc.ul.pt/mil/`: a *applet* Java, o código fonte e exemplos π e MIL.

Palavras-chave: cálculo π, multithreading, compilação certificada, compilação conservadora de tipos, compilação orientada a tipos, monitores

Abstract

Physical and electrical constrains are compelling manufactures to augment the number of cores in each die to continue delivering performance gains. To take advantage of the emerging multicore architectures, we need to master parallel and concurrent programming. We encourage empowering languages with adequate concurrency primitives: fine-grained for low-level languages and coarse grained for high-level languages. This way, compilers can reuse fined-grained primitives to encode multiple coarse-grained primitives.

Work in type-directed compilers (*e.g.*, the Typed Intermediate Language for the ML language) showed that using a strongly typed intermediate language increases not only safety but also efficiency. Typed assembly languages (TAL) draw the benefits from having type information from end-to-end, originating type-preserving compilers. Vasconcelos and Martins proposed the Multithreaded Intermediate Language (MIL) as a typed assembly language for multithreaded architectures, featuring an abstract multicore machine with shared memory that is equipped with locks.

We propose a type-preserving translation from a simply typed π-calculus into MIL. Process calculi provide natural compilation schemes that expose thread-level parallelism, present in the target architecture. By translating the π-calculus into MIL, we depart from a language where processes communicate through message-passing and arrive in a language where threads communicate through shared memory.

Our contributions consist of

- a translation function that generates MIL from the π-calculus;
- the translation is type-preserving;
- using an unbounded buffer monitor to encode channels;
- detailed examples in MIL (handling concurrency primitives and control-flow);
- implementations in MIL of condition variables, of polymorphic queues, and of an unbounded buffer;
- a generic encoding of monitors in MIL.

Keywords: the π-calculus, multithreading, certified compiler, type-preserving compilation, type-directed compilation, monitors

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Physical and electrical constrains are limiting the increase of processor's clock speed in such a way that the computing power of each processing unit is not expected to increase much more in a near future. Instead, manufactures are augmenting the number of processing units in each processor (multicore processors) to continue delivering performance gains. The industry is making big investments in projects, such as RAMP [39] and BEE2 [3], that enable the emulation of multicore architectures, showing interest in supporting the foundations for software research that targets these architectures.

To take advantage of multicore architectures, programmers must master parallel and concurrent programming [33]. With the advent of major availability of parallel facilities (from embedded systems, to super-computers), programmers must do a paradigm shift from sequential to parallel programming and produce, from scratch, software adapted for multicore platforms.

Multithreading is one of industry's most accepted answer to write explicitly parallel systems. Locks stand forth as a fundamental mechanism for writing highly-efficient multithreaded code [8]. Holding locks for too much time, misusing readers/writers locks, and choosing the wrong synchronisation mechanisms are all frequent arising problems encountered while developing multithreaded systems. Such problems can be mitigated with the appropriate high-level concurrency abstractions. We need to empower low-level languages with fine-grained concurrency primitives in a way that compilers for high-level concurrent languages can take advantage of both efficiently and safely.

Types and type systems are a simple and effective way to ensure safety [9, 36]. Types are not only used by type systems to free programs from executing errors, but serve as specification (verifiable in an automatic manner) that captures important operational properties, such as memory safety. TIL, a type-directed optimising compiler for ML, showed that using a strongly typed intermediate language could help optimisation [44]. Also, the typechecker for TIL helped in making the development of the compiler faster and safer, since programming errors would be caught early on.

TIL, however, only preserved types through approximately 80% of the compilation process. Loosing type information in the last 20% seems to just move the problem one (translation) step further. TIL was still translating to an untyped language in the end! This thought stemmed the development of Typed Assembly Languages (TAL) [31] and Proof-Carrying Code (PCC) [32], two techniques that introduce *compiler certification*, a formal method that verifies the semantics of the generated code. *Type-preserving* compilers save type information throughout every compilation stage, enabling both safer compilation and better conservation of semantics — the ones captured by types.

We propose a compiler that addresses the issues raised so far.

- the source language must offer concurrency and parallel abstractions;
- the compiler must target a multicore architecture;
- the translation must preserve type information.

Regarding the source language, process calculi arise as an expressive programming model for concurrent and parallel computing. The π-calculus [28], in particular, has well-understood semantics, consists of a small set of primitives and operators, and its fundamental step of computation is communication. Process calculi provide natural compilation schemes that expose thread-level parallelism [25]. We choose the simply typed π-calculus as our source language [5, 21, 28].

The Multithreaded Intermediate Language (MIL) [47] is a multithreaded typed assembly language for multicore architectures that have shared (main) memory. This strongly typed language gives the following safety properties: memory safety, control-flow safety, race-freedom safety, and deadlock-freedom safety [48]. MIL contradicts the statement that "the association between locks and data is established mostly by convention" [43] by making this association explicit in the language. The type system enforces a policy on the lock usage that forbids locking an owned locked and unlocking an open lock, and makes sure that threads do not forget to unlock owned locks.

Another possible choice for the target language of a compiler of a concurrent language is the work from Feng and Shao [12], following the lines of program verification and PCC. They propose a logic "type" system for static verification of concurrent assembly programs, with support for unbounded dynamic thread creation and termination. Their idea is to reflect specifications, defined as high-level logic, at the assembly level. Type systems embed a limited array of security properties as part of its soundness proof. By using types and typing rules you gain succinct but limited safety properties, the proof of the program is generated automatically in the form of types. Frameworks like CAP [49, 50] give more flexibility with the cost of making the proof explicit. We choose the former approach since the source language is already less expressive than MIL in terms of types.

In terms of a type-preserving compilation, there is no advantage in choosing an even more expressive system like the latter.

There are various works in type-preserving compilation. We highlight the seminal work from Morrisett *et al.* [31] that presents a five stage type-preserving compilation, from System F [17] into a (sequential) typed assembly language. Their work shows the various steps necessary to transform a high-level language into the chosen TAL, whilst maintaining type information. Their TAL features high-level abstractions like tuples, polymorphism, and existential packages.

In the context of process calculi compilation, Turner proposes an abstract machine encoded in the π-calculus for running Pict [38] that is then translated into C [45]. The abstract machine does not take types into consideration. C is not expressive enough to retain the type information of the source language (a polymorphic version of the π-calculus). Also, the generated code is sequential and therefore does not take into account multicore architectures.

In the same line of development as Turner's abstract machine, Lopes *et al.* [25] proposed a multithreaded language (TTyCO) and an abstract machine for TyCO [46], an asynchronous process calculus. TTyCO is an intermediate untyped language, featuring threads and message-queues. This intermediate language lays at a higher-level than MIL. In a sense, we implement the higher-level features of TTyCO that lack in MIL using the unbounded buffer monitor. TTyCO gives no safety-properties. Since the target language is untyped, no type-preserving results can be obtained. Our work continues the work done in TTyCO by targeting a multithreaded typed assembly language through a type-preserving compilation.

1.2 Contributions

The present work proposes a type-preserving translation from the π-calculus into MIL. The π-calculus is a process algebra for describing *mobility*: a network of interconnected processes compute by communicating references to processes (links) among them [28]. We choose a simple, simply typed version of the π-calculus [5, 21, 28].

MIL is an assembly language targeted at an abstract multicore machine equipped with a shared main memory. By translating the π-calculus into MIL, we depart from a language where processes communicate through message-passing and arrive in a language where threads communicate through shared memory. The source and target languages as well as their respective underlying models differ substantially. The former is declarative, computation takes place as processes communicate, and synchronisation is made through message-passing. The latter is imperative, the system evolves by evaluating instructions, and synchronisation is made through shared memory.

Translation is not direct. Mapping π-processes into threads follows naturally, *e.g.* we translate two processes running in parallel into two concurrent threads, but there is no correspondence of channels in MIL. A by-product of our work is an unbounded buffer monitor, a variant of Hoare's bounded buffer monitor [20], entirely written in MIL, that is used by our compiler to encode channels. The monitor provides a (polymorphic) FIFO-ordered buffer that mediates the asynchronous communication between producers and consumers.

As a proof of concept, we develop a program in Java that implements the presented π-calculus type system (in the form of a typechecker) and the π-to-MIL translation function (in the form of a compiler). The MIL interpreter (and typechecker), the π-to-MIL compiler, the examples presented in this work, and the code for the unbounded buffer monitor are all available on-line [26].

The summary of our contributions is:

a translation function that maps terms of the π-calculus into MIL terms;

type-preserving compilation the main result is that the translation of typable (closed) π-programs generates well-typed MIL programs;

detailed examples we document lock acquisition and control-flow manipulation in MIL through a series of examples;

a generic encoding of monitors in MIL we precise code and data constituents of monitors as well as give meaning to entering and exiting a monitor procedure;

condition variables we show a MIL implementation of condition variables that features a representation of suspended threads in the form of continuations;

polymorphic queues we present a polymorphic implementation of queues;

polymorphic unbounded buffer monitor we show an implementation of a monitor in MIL that represents an unbounded buffer, shielding the client code (the π-calculus compiler in particular) from the hazardous task of direct lock manipulation;

MIL interpreter we updated the MIL tools (the typechecker and the interpreter);

π-to-MIL compiler we devised a prototype that includes a typechecker for the source language and the π-to-MIL compiler.

1.3 Outline

The current chapter introduces our work. In the closing chapter we summarise our results and outline directions for further investigation. Chapters 2 and 3 describe the source language (π-calculus) and the target language (MIL), respectively, and establish the foundations for our contribution, which is presented in Chapters 4 and 5.

More specifically, in Chapter 2, we present a polyadic, simply typed, and asynchronous π-calculus, by formalising its syntax and semantics. We omit the summation and the matching operators from the calculus. The replication operator is restricted to input processes only. Our goal is to expose the source language of our compiler.

Chapter 3 gives a thorough description of MIL, the target language of the translation, covering both syntax and semantics. We devote one section to exemplify lock acquisition and transfer of control flow, two fundamental programming patterns that we apply extensively in the run-time and in the generated code. The last section of the chapter presents the two main results of MIL: that "well-typed machines do not get stuck" and that well-typed machines do not have races.

Chapter 4 details the supporting code used by the translation, including polymorphic queues, condition variables, and the encoding of Hoare-style monitor in MIL. We employ these generic elements in the implementation of an unbounded (polymorphic) buffer monitor, used to encode π-channels in MIL. This chapter encloses our contribution in MIL programming, yielding three libraries: queues, condition variables, and unbounded buffer monitors.

Chapter 5 formalises the translation from the π-calculus into MIL. The unbounded buffer monitor presented in Chapter 4 simplifies the translation, since sending and receiving messages from a channel may be viewed as appending and removing elements from a buffer. We show the outcome of applying the translation function to a π-process. This section includes a detailed proof of the main result of our work: a type-preserving compilation, asserting that type-correct π-programs compile into type-correct MIL programs.

Chapter 2

The π-Calculus

The π-calculus, developed by Robin Milner, Joachim Parrow, and David Walker [28], is a process algebra for describing *mobility*, in which processes "move" in a virtual space. The underlying model is a network of interconnected processes that interact by communicating connection links (channels). Since connection links can be retransmitted, the π-calculus is able to express dynamic reconfigurations of the network, where resources become accessible to different parties as the system evolves.

As a motivation example, consider process *echo server* that bounces every message received through a channel back to the sender process (the client). Figure 2.1 depicts such an interplay with emphasis on the server. The server is accessible via link $echo$. Before interaction the server maintains two placeholders to be filled upon data arrival: a message msg that is illustrated by a dashed box in the figure (left) and a $reply$ link represented by a circle enclosing a cross. After interaction the server uses the $reply$ channel, represented in Figure 2.1 (right) by a filled circle, to bounce the received value msg back to the client, portrayed in the figure by a box holding two symbols (● ◇). The arrows from msg to $reply$ and from $reply$ to the client show the data-flow of the echoed message.

In this chapter we describe the source language of our compiler: the π-calculus. We begin by depicting the syntax of processes and of types on Section 2.1. Next, we cover the operational semantics of the π-calculus and its type system.

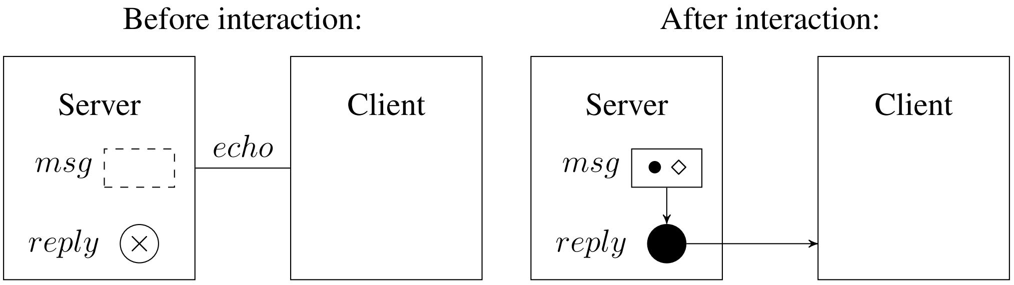

Figure 2.1: The echo server interaction from the server's point of view.

$$
\begin{array}{rcll}
 & & \textit{Values} & \\
v & ::= & x & \text{name} \\
 & | & \mathrm{n} & \text{integer}
\end{array}
\qquad
\begin{array}{rcll}
 & & & \textit{Processes} \\
P, Q & ::= & \mathbf{0} & \text{inactive} \\
 & | & \overline{x}\langle\vec{v}\rangle & \text{output} \\
 & | & x(\vec{y}).P & \text{input} \\
 & | & !x(\vec{y}).P & \text{replicated input} \\
 & | & P \mid Q & \text{parallel} \\
 & | & (\nu\, x\colon \hat{}[\vec{T}])\, P & \text{restriction}
\end{array}
$$

The syntax of T is defined in Figure 2.3.

Figure 2.2: Process syntax.

2.1 Syntax

The adopted π-calculus syntax is based on [27] with extensions presented in [42]: asynchronous [5, 21], meaning that sending a message does not block; polyadic, corresponding to the transmission of sequences of values; and simply typed with constants (integers).

Processes. The syntax, depicted in Figure 2.2, is divided into two categories: *values* ranged over by v that represent names and integers; and *processes* ranged over by $P, Q, \ldots$ that compose the network of processes. A name, ranged over by $x, y, \ldots$, is either a link or a placeholder for values. The value n is a meta-variable over integers that corresponds to any possible integer. A vector above a symbol abbreviates a possibly empty sequence of these symbols. For example $\vec{x}$ stands for the sequence of names $x_1 \ldots x_n$ with $n \geq 0$.

Processes consist of the nil process $\mathbf{0}$, corresponding to the inactive process; the output process $\overline{x}\langle\vec{v}\rangle$ that sends data $\vec{v}$ through a channel x; the input process $x(\vec{y}).P$ that binds a received sequence of values to names $\vec{y}$ in the scope of process P via a channel x; the replicated input process $!x(\vec{y}).P$ that represents an infinite number of input processes running in parallel; the parallel composition of processes $P \mid Q$ that represents two processes running in parallel; and the restriction process $(\nu\, x\colon \hat{}[\vec{T}])\, P$ that creates a new channel definition local to process P.

The following example is a possible implementation of the echo server depicted in Figure 2.1.

$$!echo(msg, reply).\overline{reply}\langle msg\rangle \tag{2.1}$$

The process is ready to receive a message msg and a $reply$ channel via channel $echo$. Upon arrival, the message is bounced back to the server again through the $reply$ link.

Definition 2.1.1. Process P is a *sub-term* of process Q iff

- Q is P,

- or Q is $P' \mid Q'$ and P is a sub-term of P',
- or Q is $P' \mid Q'$ and P is a sub-term of Q',
- or Q is $x(\vec{y}).P'$ and P is a sub-term of P',
- or Q is $!x(\vec{y}).P'$ and P is a sub-term of P',
- or Q is $(\nu\, x\colon \hat{}[\vec{T}])\ P'$ and P is a sub-term of P'.

Types. In the π-calculus a name can assume a range of values during execution. For example, a name can represent a number 1 or some channel. A type defines the set of values a name accepts [9]. By saying that the name is of type integer, we suppose that the given name can only represent integers during every step of execution. We call a language *typed*, when names can be given types, the opposite of untyped languages.

Type systems keep track of types of names and are used to detect execution errors in programs. We quote the definition from Benjamin Pierce [35]:

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

We distinguish two forms of executing errors: *trapped errors* that cause computation to stop immediately and *untrapped errors* that go unnoticed for a certain period of time. For instance, if a machine halts computation when an integer value is used as a channel, then this misuse is a trapped error. A form of an untrapped error is when a machine dwells into an inconsistent state, but does not halt, which could happen upon sending a sequence of three values through a channel that expects a sequence of two values. *Safe languages* are free from untrapped errors, the class of errors harder to track. Languages may enforce safety by performing run-time or compile-time (static) checks. Typed languages usually employ both forms of safety checks. We call *typechecking* to the process that performs the static checks on a typed language. The *typechecker* is the algorithm responsible for typechecking the language.

Which errors should a type system detect? Let*forbidden errors* be the class of possible errors caught by type systems. Forbidden errors should include all untrapped errors and some trapped errors. Type systems therefore target safe languages and additionally catch some trapped errors. We designate programs free from forbidden errors *well-behaved.* A language where all of the (legal) programs are well-behaved is called *strongly checked.*

There are numerous advantages for having types and type systems. For example, a code generator has additional hints for knowing how much memory a value needs. Execution efficiency can be improved, because we can replace run-time checks by static

$$\begin{array}{rcll} T & ::= & \mathsf{int} & \text{integer type} \\ & \mid & \hat{}[\vec{T}] & \text{channel type} \end{array}$$

Figure 2.3: Type syntax.

checks. Small-scale development is hastened by the feedback of the typechecker, reducing the number of debugging sessions. Type information is essential for development tools, including refactoring tools, code analysis, code verification, and integrated development environments. From the knowledge of dependencies between modules given by types, compilers can perform partial compilation that reduces the overall compilation-time. Language features captured by types are usually orthogonal, thus tend to reduce the complexity of programming languages.

The π-calculus version we choose is *monomorphic*, strongly checked, and typed. Monomorphism means that every program fragment (*e.g.*, a value) can only have one type, as opposed to polymorphism, in which a program fragment can have many types. We assign type int to integers. The channel type $\hat{}[\vec{T}]$ describes a link that may communicate a sequence of values each of which assigned to type T_i, respectively.

For example, type $\hat{}[\mathsf{int}, \hat{}[\mathsf{int}]]$ captures the semantics of channel $echo$ from Process 2.1 and corresponds to a channel type with two parameters, the first an integer type and the second a channel type that communicates integers.

2.2 Operational Semantics

The semantics of the π-calculus expresses formally the behaviour of processes. With a rigorous semantics we can identify if two processes have the same structural behaviour, observe how a process evolves as it interacts, and analyse how links move from one process to another.

We begin with name binding. Two constructors bind names.

Definition 2.2.1 (Binding)**.** In $(\nu\, x\colon \hat{}[\vec{T}])\, P$ the displayed occurrence of x is a *binding* with *scope* P. In $x(\vec{y}).P$ each occurrence of y_i is a binding with scope P. An occurrence of a name is *bound* if it is, or lies within the scope of, a binding occurrence of the name. An occurrence of a name in a process is *free* if it is not bound. A process with only bound names is said to be *closed*.

The bound name function can be inductively defined as:

Definition 2.2.2 (The bound name function)**.**

$$\begin{aligned}
\mathrm{bn}(\mathbf{0}) &= \emptyset \\
\mathrm{bn}(\overline{x}\langle\vec{v}\rangle) &= \emptyset \\
\mathrm{bn}(x(\vec{y}).P) &= \{\vec{y}\} \cup \mathrm{bn}(P) \\
\mathrm{bn}(!x(\vec{y}).P) &= \mathrm{bn}(x(\vec{y}).P) \\
\mathrm{bn}(P \mid Q) &= \mathrm{bn}(P) \cup \mathrm{bn}(Q) \\
\mathrm{bn}((\nu\, x \colon \hat{}[\vec{T}])\, P) &= \{x\} \cup \mathrm{bn}(P)
\end{aligned}$$

Contrarily to bound names, free names are globally defined. The free name function can be defined as:

Definition 2.2.3 (The free names function)**.**

$$\begin{aligned}
\mathrm{fn}(\mathbf{0}) &= \emptyset \\
\mathrm{fn}(\overline{x}\langle\vec{v}\rangle) &= \{x\} \cup \mathrm{fn}(\vec{v}) \\
\mathrm{fn}(x(\vec{y}).P) &= \{x\} \cup \mathrm{fn}(P) \setminus \{\vec{y}\} \\
\mathrm{fn}(!x(\vec{y}).P) &= \mathrm{fn}(x(\vec{y}).P) \\
\mathrm{fn}(P \mid Q) &= \mathrm{fn}(P) \cup \mathrm{fn}(Q) \\
\mathrm{fn}((\nu\, x \colon \hat{}[\vec{T}])\, P) &= \mathrm{fn}(P) \setminus \{x\} \\
\mathrm{fn}(v_1 \dots v_n) &= \mathrm{fn}(v_1) \cup \cdots \cup \mathrm{fn}(v_n) \\
\mathrm{fn}(x) &= \{x\} \\
\mathrm{fn}(\mathrm{n}) &= \emptyset
\end{aligned}$$

For example, in process

$$!\mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle$$

name echo is free and names msg and reply are bound.

Notice a name may occur both free and bound in the same expression. In the following example name x appears bound and free:

$$x(y).y(x).\mathbf{0}$$

The first displayed occurrence of name x is free, in the outer input process; the second displayed occurrence of x is bound. In the scope of process **0** the use of name x targets the parameter of link y and not the free name.

Definition 2.2.4. A *substitution* is a function that is the identity except on a finite set, defined from values to names. A formula $v\sigma$ represents the application of substitution σ to value v, where:

$$\mathrm{n}\sigma \overset{\text{def}}{=} \mathrm{n} \qquad\qquad x\sigma \overset{\text{def}}{=} \sigma(x)$$

As for processes, say $P\sigma$, the substitution replaces each free occurrence of name x in process P by $x\sigma$. We define the application of a substitution to processes (see the *name convention* below) as:

$$\begin{aligned}
\mathbf{0}\sigma &\overset{\text{def}}{=} \mathbf{0}\\
(\overline{x}\langle\vec{v}\rangle)\sigma &\overset{\text{def}}{=} \overline{x\sigma}\langle\vec{v}\sigma\rangle\\
(x(\vec{y}).P)\sigma &\overset{\text{def}}{=} (x\sigma)(\vec{y}).P\sigma\\
(!x(\vec{y}).P)\sigma &\overset{\text{def}}{=} !((x(\vec{y}).P)\sigma)\\
(P \mid Q)\sigma &\overset{\text{def}}{=} (P\sigma) \mid (Q\sigma)\\
((\nu\, x\colon \hat{}[\vec{T}])\, P)\sigma &\overset{\text{def}}{=} (\nu\, x\colon \hat{}[\vec{T}])\, P\sigma
\end{aligned}$$

We write $\{\vec{v}/\vec{x}\}$ for the substitution σ such that $x_i\sigma = v_i$ and $y\sigma = y$ for $y \notin \vec{x}$.

For example, substituting z for x in process $\overline{x}\langle 2\rangle \mid x(y).\mathbf{0}$ yields the following result, where $\sigma = \{z/x\}$:

$$\begin{aligned}
&(\overline{x}\langle 2\rangle \mid x(y).\mathbf{0})\sigma\\
=&(\overline{x}\langle 2\rangle)\sigma \mid (x(y).\mathbf{0})\sigma\\
=&\overline{x\sigma}\langle 2\sigma\rangle \mid (x\sigma)(y).\mathbf{0}\sigma\\
=&\overline{\sigma(x)}\langle 2\rangle \mid (\sigma(x))(y).\mathbf{0}\\
=&\overline{z}\langle 2\rangle \mid z(y).\mathbf{0}
\end{aligned}$$

Definition 2.2.5 (Change of bound names)**.** A *change of bound names* in process P is the replacement of a sub-term of P in the form $x(\vec{y}).Q$ by $x(\vec{z}).Q\{\vec{z}/\vec{y}\}$ or in the form $(\nu\, y\colon \hat{}[\vec{T}])\, Q$ by $(\nu\, z\colon \hat{}[\vec{T}])\, Q\{z/y\}$, where z and $\vec{z}$ are not bound nor free in Q.

A *congruence relation* is an equivalence relation[1] that satisfies some algebraic properties. Furthermore, if $\equiv$ is a congruence relation on processes, then

- if $P \equiv Q$, then $x(\vec{y}).P \equiv x(\vec{y}).P$;
- if $P \equiv Q$, then $!x(\vec{y}).P \equiv !x(\vec{y}).P$;
- if $P \equiv Q$, then $P \mid R \equiv Q \mid R$;
- if $P \equiv Q$, then $(\nu\, x\colon \hat{}[\vec{T}])\, P \equiv (\nu\, x\colon \hat{}[\vec{T}])\, Q$;

Process P is *α-congruent* with process Q, notation $P \equiv_\alpha Q$, if Q results from P by successive changes of bound names. For example,

$$\mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \equiv_\alpha \mathit{echo}(x, y).\overline{y}\langle x\rangle$$

[1] Equivalence relations are reflexive, symmetric, and transitive.

(S1) If $P \equiv_\alpha Q$, then $P \equiv Q$.

(S2) The Abelian monoid laws for parallel: commutative $P \mid Q \equiv Q \mid P$, associative $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, and having the inactive process as its neutral element $P \mid \mathbf{0} \equiv P$.

(S3) $!x(\vec{y}).P \equiv x(\vec{y}).P \mid !x(\vec{y}).P$

(S4) The scope extension laws:

$$(\nu\, x\colon \hat{}[\vec{T}])\, \mathbf{0} \equiv \mathbf{0}$$

$$(\nu\, x\colon \hat{}[\vec{T}])\, (P \mid Q) \equiv P \mid (\nu\, x\colon \hat{}[\vec{T}])\, Q, \text{if } x \notin \mathrm{fn}(P)$$

$$(\nu\, x\colon \hat{}[\vec{T}])\, (\nu\, y\colon \hat{}[\vec{S}])\, P \equiv (\nu\, y\colon \hat{}[\vec{S}])\, (\nu\, x\colon \hat{}[\vec{T}])\, P$$

Figure 2.4: Structural congruence rules.

Name convention. For any given mathematical context (*e.g.*, definition, proof), terms are up to α-congruence and assume a convention (Barendregt's variable convention [2]), in which all bound names are chosen to be different from the free names and from each other.

For example, process

$$x(y).y(x).\mathbf{0}$$

breaks the convention, since the displayed occurrence x is both bound and free. The following α-congruent term assumes the name convention:

$$x(y).y(z).\mathbf{0}$$

Structural Congruence. The syntax differentiates processes that intuitively represent the same behaviour. For example, process $P \mid Q$ and process $Q \mid P$ are syntactically different, although our intuition about processes running in parallel is that the syntactic order of these processes is irrelevant. Process $(\nu\, x\colon \hat{}[])\, \overline{y}\langle x\rangle$ and process $(\nu\, z\colon \hat{}[])\, \overline{y}\langle z\rangle$ are also syntactically different, but both represent an output channel sending a private channel, regardless of the different choice of names.

The *structural congruence* relation, $\equiv$, is the smallest congruence relation on processes closed under rules given in Figure 2.4 that capture the intuition about the behaviour of processes. The structural congruence relation divides processes into equivalence classes of terms, simplifying operational semantics rule (*cf.* Figure 2.5). Rule S1 brings the change of bound names into structural congruence; Rule S2 represents the standard commutative monoid laws regarding parallel composition, having **0** as its neutral element; Rule S3 allows replication to fold and unfold. Rule S4 allows for scope

$$\frac{}{x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \to P\{\vec{v}/\vec{a}\}}\ \textsc{React}$$

$$\frac{P \to P'}{P \mid Q \to P' \mid Q}\ \textsc{Par} \qquad \frac{P \to P'}{(\nu\, x\colon \hat{}[\vec{T}])\, P \to (\nu\, x\colon \hat{}[\vec{T}])\, P'}\ \textsc{Res}$$

$$\frac{Q \equiv P \quad P \to P' \quad P' \equiv Q'}{Q \to Q'}\ \textsc{Struct}$$

Figure 2.5: Reaction rules.

extension. Notice that restriction order is of no importance, because terms assume the name convention.

Reduction. The reduction relation → defined over processes establishes how a computational step transforms a process, as defined in Figure 2.5. The formula $P \to Q$ means that process P can interact and evolve (reduce) to process Q. We interpret rules (*e.g.*, Figure 2.5) in form

$$\frac{A \quad B}{C}$$

as A and B imply C. For instance, Rule REACT is an axiom, whereas for Rule PAR we say that if P reduces to P' then $P \mid Q$ reduces to $P' \mid Q$.

Rule REACT is the gist of the reduction rules, representing the communication along a channel. An output process, $\overline{x}\langle\vec{v}\rangle$, can interact with an input process, $x(\vec{y}).P$, since they share link x. Reduction sends output message $\vec{v}$ along channel x, resulting in process $P\{\vec{v}/\vec{y}\}$.

Rule PAR expresses that reduction can appear as the left-hand side of a parallel composition. Rule RES governs reduction inside the restriction operator. With structural congruence, Rule STRUCT, we are able to rearrange processes so that they can react. Structural congruence and process reduction also bring non-determinism to the π-calculus, since we can arrange different processes to react differently.

Consider a copy of the echo server (Process 2.1) running concurrently with a client that sends number 10, as well as a channel for printing integers in the screen, say $printInt$. The order in which these processes are composed in parallel is relevant for reduction. The input must appear as the left-hand side and the output as the right-hand side, thus we rearrange the terms applying rule S2.

$$\begin{aligned} &\overline{echo}\langle 10, printInt\rangle \mid echo(msg, reply).\overline{reply}\langle msg\rangle && \text{S2} \\ \equiv\ &echo(msg, reply).\overline{reply}\langle msg\rangle \mid \overline{echo}\langle 10, printInt\rangle && \textsc{React} \\ \to\ &\overline{printInt}\langle 10\rangle \end{aligned}$$

By placing the process above in parallel with Process 2.1, interactions occur inside the parallel composition (Rule PAR):

$$\begin{aligned}
&\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \mid \overline{\mathit{echo}}\langle 10, \mathit{printInt}\rangle \\
&\mid\, !\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle && \textsc{Par}, \textsc{React} \\
\rightarrow&\overline{\mathit{printInt}}\langle 10\rangle \mid\, !\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle
\end{aligned}$$

With structural congruence and Rule STRUCT, we discard the copy of the echo server, thereby simplifying the term.

$$\begin{aligned}
&\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \mid \overline{\mathit{echo}}\langle 10, \mathit{printInt}\rangle \\
&\mid\, !\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \\
\equiv&\overline{\mathit{echo}}\langle 10, \mathit{printInt}\rangle \mid\, !\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle && \textsc{Struct}, \textsc{Par}, \textsc{React} \\
\rightarrow&\overline{\mathit{printInt}}\langle 10\rangle \mid\, !\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle
\end{aligned}$$

Suppose we have a client that sends number 20 along with a private channel to a copy of the echo server.

$$\begin{aligned}
&\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \mid (\nu\, x)\, (\overline{\mathit{echo}}\langle 20, x\rangle \mid x(y).P) \\
\equiv&(\nu\, x)\, (\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \mid \overline{\mathit{echo}}\langle 20, x\rangle \mid x(y).P) && \textsc{Res}, \textsc{Par}, \textsc{React} \\
\rightarrow&(\nu\, x)\, (\overline{x}\langle 20\rangle \mid x(y).P) && \textsc{Res}, \textsc{React} \\
\rightarrow&P\{20/y\}
\end{aligned}$$

By extending the scope of x to the server, we let interaction evolve inside the restriction and the parallel operators. The server bounces the number 20 back to client $x(y).P$ with Rules RES and REACT. The scope of x is local to the client before reaction, since the scope extension performed on the first step does not make the channel accessible to the server — there are no occurrences of x in the server. After the first reduction, however, the name "moves" to the server. This new scope availability is called *scope extrusion*. The virtual movement from the client to the server is a kind of process mobility. A contrasting form of process mobility is physical mobility, where processes move in a physical space, rather than in a virtual space, for example a mobile phone moving in different cell sites.

Regard how two clients communicate with a copy of the echo server.

$$\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \mid \overline{\mathit{echo}}\langle 10, r\rangle \mid \overline{\mathit{echo}}\langle 20, s\rangle$$

By letting reduction develop as is, the client sending channel r communicates with the server.

$$\begin{aligned}
&\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \mid \overline{\mathit{echo}}\langle 10, r\rangle \mid \overline{\mathit{echo}}\langle 20, s\rangle && \textsc{Par}, \textsc{React} \\
\rightarrow&\overline{r}\langle 10\rangle \mid \overline{\mathit{echo}}\langle 20, s\rangle
\end{aligned}$$

But, if we reorder terms with structural congruence, we observe that the client sending number 20 and channel s can also interact with the server.

$$\begin{aligned}
&\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \mid \overline{\mathit{echo}}\langle 10,r\rangle \mid \overline{\mathit{echo}}\langle 20,s\rangle\\
\equiv&\mathit{echo}(\mathit{msg},\mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle \mid \overline{\mathit{echo}}\langle 20,s\rangle \mid \overline{\mathit{echo}}\langle 10,r\rangle \quad \textsc{Struct}, \textsc{Par}, \textsc{React}\\
\rightarrow&\overline{s}\langle 20\rangle \mid \overline{\mathit{echo}}\langle 10,r\rangle
\end{aligned}$$

This example highlights the non-deterministic nature of the π-calculus. The same process can yield different computational outcomes.

Type system. Before we explain the type system of the π-calculus, we introduce some common grounds.

Definition 2.2.6 (Type environment)**.** The *type assignment*, notation $x\colon T$, assigns a type to a name, where x is the name of the assignment, and T is type of the assignment.

A *type environment*, or *typing*, ranged over by Γ, is an ordered list of type assignments in the form of $\emptyset, x_1\colon T_1,\ldots,x_n\colon T_n$, with $x_1\ldots x_n$ distinct. The notation for the empty type environment is $\emptyset$. Extending a type environment Γ with type assignment $x\colon T$, with name x not mentioned in typing Γ, is of form $\Gamma, x\colon T$. The collection of names $x_1\ldots x_n$ declared in Γ is indicated by $\mathrm{dom}(\Gamma)$.

For example,

$$\emptyset, \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]], \mathit{msg}\colon \mathsf{int}, \mathit{reply}\colon \hat{}[\mathsf{int}]$$

is a type environment with three assignments: a channel echo that transmits an integer and a channels of integers, a name msg of type integer, and a channel reply that communicates integers.

Type judgements are of form $\Gamma \vdash v\colon T$ for values and of form $\Gamma \vdash P$ for processes. The former asserts that a value v has type T. The latter form asserts that process P respects the type assignments in Γ. A typing environment Γ stands as the left-hand side of both forms of judgement and declares the free names of the assertion.

A *type system* is a collection of axioms and inference rules. Figure 2.6 presents a standard type system for the π-calculus. Type judgements can be either *valid* or *invalid*. A valid type judgement is one that can be proved with an axiom or inference rule from a given type system. We call the proof of the validity of a type judgement a *typing derivation*. A judgement is invalid if there is no valid judgement to prove that assertion. For instance, $\Gamma \vdash 3\colon \mathsf{int}$ is a valid type judgement, yet $\Gamma \nvdash 3\colon \hat{}[\mathsf{int}]$ is an invalid type judgement. Notice the notation $\nvdash$ for invalid type judgements. A value v is well-typed in Γ, for some Γ, if there is T such that $\Gamma \vdash v\colon T$. Similarly, given a typing Γ, a process P is well-typed under that type environment if $\Gamma \vdash P$.

For typing the π-calculus we are concerned with communication, in particular checking the usage of names. Typing values concerns integers and names. The axiom Tv-Base

$$\frac{}{\Gamma \vdash \mathrm{n}\colon \mathsf{int}}\ \textsc{Tv-Base} \qquad \frac{}{\Gamma, x\colon T \vdash x\colon T}\ \textsc{Tv-Name}$$

$$\frac{}{\Gamma \vdash \mathbf{0}}\ \textsc{Tv-Nil} \qquad \frac{\Gamma \vdash x\colon \hat{}[\vec{T}] \quad \forall v_i \in \vec{v} : \Gamma \vdash v_i\colon T_i}{\Gamma \vdash \overline{x}\langle\vec{v}\rangle}\ \textsc{Tv-Out}$$

$$\frac{\Gamma \vdash x\colon \hat{}[T_1, \ldots, T_n] \quad \Gamma, y_1\colon T_1, \ldots, y_n\colon T_n \vdash P}{\Gamma \vdash x(\vec{y}).P}\ \textsc{Tv-In}$$

$$\frac{\Gamma \vdash x(\vec{y}).P}{\Gamma \vdash !x(\vec{y}).P}\ \textsc{Tv-Rep} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}\ \textsc{Tv-Par} \qquad \frac{\Gamma, x\colon \hat{}[\vec{T}] \vdash P}{\Gamma \vdash (\nu\, x\colon \hat{}[\vec{T}])\, P}\ \textsc{Tv-Res}$$

Figure 2.6: Typing rules for the π-calculus

states that integers are well-typed and assigned to type int. With rule Tv-Name the type assigned to a name is the one found (declared) in the type environment.

As for typing processes, the inactive process $\mathbf{0}$ is well-typed, axiom Tv-Nil. With Rule Tv-Out, a name (of link type) is used as an output process with the arguments of the expected type. Like all composite processes, the typing of the input process $x(\vec{y}).P$ (with rule Tv-In) depends upon the validity of its constituents, namely P. The continuation P is therefore checked under a typing extended with parameters $\vec{y}$ (each y_i is assigned to its respective type T_i). To check a replicated process $!x(\vec{y}).P$ with Rule Tv-Rep it is enough to verify if one of its copies is valid. The validity of the parallel process, checked with Rule Tv-Par, depends upon the validity of its parts. For typing restriction $(\nu\, x\colon \hat{}[\vec{T}])\, P$ (rule Tv-Res) we check the continuation P under a typing augmented with binding x.

We show that environment $\emptyset, \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]]$ is enough to typify the echo server Process 2.1. Using rule Tv-Rep we derive

$$\frac{\dfrac{\dfrac{}{\emptyset, \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]] \vdash \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]]}\ \textsc{Tv-Name} \qquad (1)}{\emptyset, \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]] \vdash \mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle}\ \textsc{Tv-In}}{\emptyset, \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]] \vdash !\mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle}\ \textsc{Tv-Rep}$$

To type the echo server it is enough to type a copy of the server, Rule Tv-Rep. The derivation proceeds by applying rule Tv-In. The sequent for typing name *echo* with rule Tv-Name is direct. We are left with sequent (1) that also holds, where typing $\Gamma \stackrel{\text{def}}{=} \emptyset, \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]], \mathit{msg}\colon \mathsf{int}, \mathit{reply}\colon \hat{}[\mathsf{int}]$

$$\frac{\dfrac{}{\Gamma \vdash \mathit{reply}\colon \hat{}[\mathsf{int}]}\ \textsc{Tv-Name} \qquad \dfrac{}{\Gamma \vdash \mathit{msg}\colon \mathsf{int}}\ \textsc{Tv-Name}}{\Gamma \vdash \overline{\mathit{reply}}\langle \mathit{msg}\rangle}\ \textsc{Tv-Out}$$

Therefore, Process 2.1 is well-typed under typing $\emptyset, \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]]$.

Lemma 2.2.7. *If* $\Gamma \vdash P$*, then* $\mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)$.

Proof. The proof follows by induction in the structure of P.

- Case P is $\mathbf{0}$. By definition of fn, $\mathrm{fn}(\mathbf{0}) = \emptyset$. Hence, $\mathrm{fn}(\mathbf{0}) \subseteq \mathrm{dom}(\Gamma)$.

- Case P is $\overline{x}\langle\vec{v}\rangle$. By definition of fn, $\mathrm{fn}(\overline{x}\langle\vec{v}\rangle) = \{x\} \cup \mathrm{fn}(\vec{v})$. By hypothesis $\Gamma \vdash \overline{x}\langle\vec{v}\rangle$, thus (by rule TV-OUT) $\Gamma \vdash x : \hat{}[\vec{T}]$ and $\forall v_i \in \vec{v}.\Gamma \vdash v_i : T_i$. By rule TV-NAME, if $\Gamma \vdash x : \hat{}[\vec{T}]$, then $x \in \mathrm{dom}(\Gamma)$. By definition of ftv, $\forall v_i \in \vec{v}.\Gamma \vdash v_i : T_i$ if v_i is a base value, then it is not in the free names of $\mathrm{fn}(\vec{v})$ if it is a name, by rule TV-NAME, then $v_i \in \mathrm{dom}(\Gamma)$. Hence, $\{x\} \cup \mathrm{fn}(\vec{v}) \subseteq \mathrm{dom}(\Gamma)$.

- Case P is $x(\vec{y}).Q$. By definition of fn, $\mathrm{fn}(x(\vec{y}).Q) = \{x\} \cup \mathrm{fn}(Q) \setminus \{\vec{y}\}$. By rule TV-IN, we have that $\Gamma \vdash x : \hat{}[\vec{T}]$ and $\Gamma, \vec{y} : \vec{T} \vdash Q$. Since we have $\Gamma \vdash x : \hat{}[\vec{T}]$, then by rule TV-NAME $x \in \mathrm{dom}(\Gamma)$. Since we have $\Gamma, \vec{y} : \vec{T} \vdash Q$, then, by the induction hypothesis, we have that $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma, \vec{y} : \vec{T})$. By applying the set theory and the definition of dom, we have:

$$\begin{aligned}&\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma, \vec{y} : \vec{T})\\ =&\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\vec{y} : \vec{T})\\ =&\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma) \cup \{\vec{y}\}\\ =&\mathrm{fn}(Q) \setminus \{\vec{y}\} \subseteq \mathrm{dom}(\Gamma)\end{aligned}$$

  We have that $x \in \mathrm{dom}(\Gamma)$, hence

$$\begin{aligned}&\mathrm{fn}(Q) \setminus \{\vec{y}\} \subseteq \mathrm{dom}(\Gamma)\\ =&\{x\} \cup \mathrm{fn}(Q) \setminus \{\vec{y}\} \subseteq \mathrm{dom}(\Gamma)\\ =&\mathrm{fn}(x(\vec{y}).Q) \subseteq \mathrm{dom}(\Gamma)\end{aligned}$$

- Case P is $!x(\vec{y}).Q$. By rule TV-REP, we have that $\Gamma \vdash x(\vec{y}).Q$. Then, by the induction hypothesis, we have that $\mathrm{fn}(x(\vec{y}).Q) \subseteq \mathrm{dom}(\Gamma)$. Since $\mathrm{fn}(!x(\vec{y}).Q) = fn(x(\vec{y}).Q)$, then $\mathrm{fn}(!x(\vec{y}).Q) \subseteq \mathrm{dom}(\Gamma)$ holds.

- Case P is $Q_1 \mid Q_2$. By rule TV-PAR, we have that $\Gamma \vdash Q_1$ and $\Gamma \vdash Q_2$. Hence, by the induction hypothesis, we have that $\mathrm{fn}(Q_1) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{fn}(Q_2) \subseteq \mathrm{dom}(\Gamma)$. By applying the set theory, we have

$$\mathrm{fn}(Q_1) \subseteq \mathrm{dom}(\Gamma) = \mathrm{fn}(Q_1) \cup \mathrm{fn}(Q_2) \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{fn}(Q_2)$$

  But $\mathrm{fn}(Q_2) \subseteq \mathrm{dom}(\Gamma)$, hence $\mathrm{dom}(\Gamma) \cup \mathrm{fn}(Q_2) = \mathrm{dom}(\Gamma)$. By definition of fn, we have $\mathrm{fn}(Q_1 \mid Q_2) = \mathrm{fn}(Q_1) \cup \mathrm{fn}(Q_2)$. Hence $\mathrm{fn}(Q_1 \mid Q_2) \subseteq \mathrm{dom}(\Gamma)$.

- Case P is $(\nu\, x\colon T)\, Q$. By hypothesis we have that $\Gamma \vdash (\nu\, x\colon T)\, Q$, then, by rule Tv-Res $\Gamma, x : T \vdash Q$. Hence, by the induction hypothesis, $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma, x : T)$. We apply the definition of function dom and the set theory:

  $$\begin{aligned} &\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma, x : T)\\ =&\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\emptyset, x : T)\\ =&\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}\\ =&\mathrm{fn}(Q) \setminus \{x\} \subseteq \mathrm{dom}(\Gamma)\end{aligned}$$

  By definition of fn, we have that $\mathrm{fn}((\nu\, x\colon T)\, Q) = \mathrm{fn}(Q) \setminus \{x\}$. Thus,

  $$\mathrm{fn}((\nu\, x\colon T)\, Q) \subseteq \mathrm{dom}(\Gamma)$$

□

Corollary 2.2.8. *If* $\emptyset \vdash P$*, then* $\mathrm{fn}(P) = \emptyset$*.*

Chapter 3

A Multithreaded Intermediate Language

Vasconcelos and Martins introduced a multithreaded typed assembly language (MIL), its operational semantics, and a type system that ensures that well-typed programs are free from race conditions [47]. The type system proposed for MIL closely follows the tradition of typed assembly languages [29–31], extended with support for threads and locks, following Flanagan and Abadi [14]. With respect to this last work, however, MIL is positioned at a much lower abstraction level, and faces different challenges inherent to non-lexically scoped languages. Lock primitives have been discussed in the context of concurrent object calculi [13], JVM [15, 16, 22, 23], C [18], C−− [40], but not in the context of typed assembly (or intermediate) languages. In a typed setting, where programs are guaranteed not to suffer from race conditions, MIL

- Syntactically decouples the lock and unlock operations from what one usually finds unified in a single syntactic construct in high-level languages: Birrel's *lock-do-end* construct [4], used under different names (*sync*, *synchronized-in*, *lock-in*) in a number of other works, including the Java programming language [6, 7, 13–16, 18];
- Allows for lock acquisition/release in schemes other than the nested discipline imposed by the *lock-do-end* construct;
- Allows forking threads that hold locks.

MIL is an assembly language targeted at an abstract multi-processor equipped with a shared main memory. Each processor consists of a series of registers and of a local memory for instructions and for local data. The main memory is divided into a heap and a run pool. The heap stores tuples and code blocks. A code block declares the registers types it expects, the required held locks, and an instruction sequence. The run pool contains suspended threads waiting for a free processor. Figure 3.1 summarises the MIL architecture.

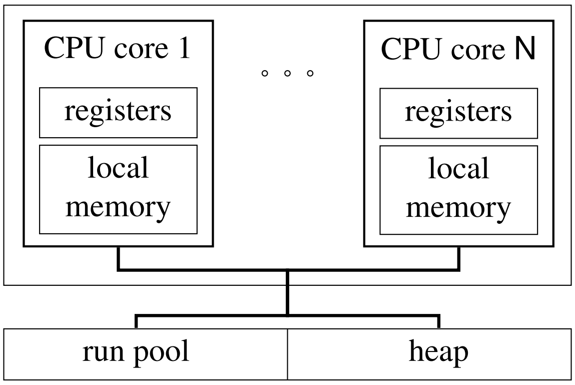

Figure 3.1: The MIL architecture.

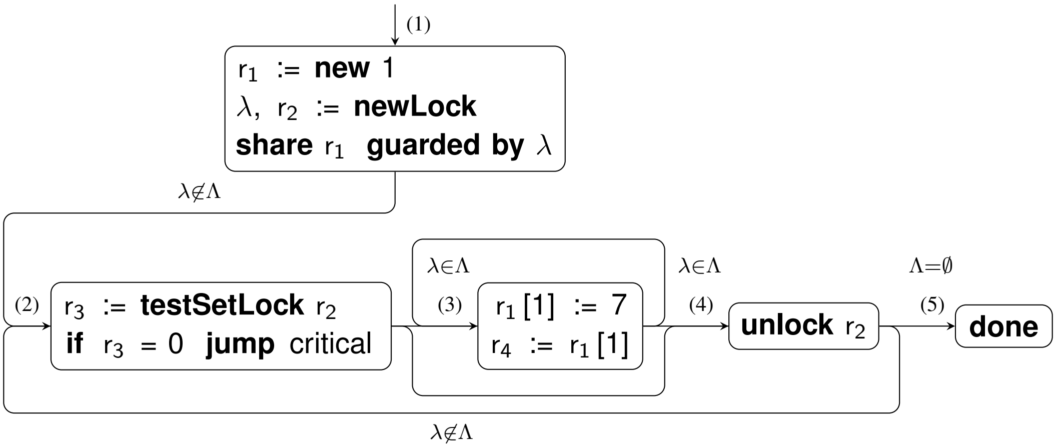

Figure 3.2: The lock discipline.

Lock discipline. We provide two distinct access privileges to tuples shared by multiple processors: read-only and read-write, the latter mediated by locks. A standard *test-and-set-lock* instruction is used to obtain a lock, thus allowing a thread to enter a *critical region*. Processors read and write from the shared heap via conventional load and store instructions. The policy for the usage of locks (enforced by the type system) is depicted in Figure 3.2 (cf. Theorem 3.5.3), where λ denotes a *singleton lock type* and Λ the set of locks held by the processor (the processor's *permission*). Specifically, the lock discipline enforces that:

(1) before lock creation, λ is not a known lock;

(2) before test-and-set-lock, the thread does not hold the lock;

(3) before accessing the heap, the thread holds the lock;

(4) unlocking only in possession of the lock;

(5) thread termination only without held locks.

3.1 Syntax

The syntax of our language is generated by the grammar in Figures 3.3, 3.4, and 3.9. We rely on a set of *heap labels* ranged over by l, a set of *type variables* ranged over by α and β, and a disjoint set of *singleton lock types* ranged over by λ and ρ.

Most of the machine instructions, presented in Figure 3.3, are standard in assembly languages. Instructions are organised in sequences, ending in jump or in done. Instruction jump v transfers the control flow to the code block pointed by value v. Instruction done frees the processor to execute another thread waiting in the thread pool. Our threads are cooperative, meaning that each thread must explicitly release the processor (using the done instruction).

Memory tuples are created locally, directly at processor's registers using the new instruction. To share memory, tuples are transferred to the heap using the share instruction, according to a chosen access policy: read-only or read-write (in which case it must be guarded by a lock). A system implementing MIL must use some scheme of local memory to store local tuples.

The *abstract machine*, generated by the grammar in Figure 3.4, is parametric on the number N of available processors and on the number R of registers per processor. An abstract machine can be in two possible states: halted or running. A running machine comprises a heap, a thread pool, and an array of processors of fixed length N. Heaps are maps from labels into *heap values* that may be tuples or code blocks. *Shared tuples* are

$$\begin{array}{llcl}
\textit{registers} & r & ::= & \mathsf{r}_1 \mid \ldots \mid \mathsf{r}_\mathsf{R} \\
\textit{integer values} & n & ::= & \ldots \mid \text{-}1 \mid 0 \mid 1 \mid \ldots \\
\textit{values} & v & ::= & r \mid n \mid l \mid \textsf{pack}\ \tau, v\ \textsf{as}\ \tau \mid v[\tau] \mid \langle \vec{v} \rangle \\
\textit{authority} & a & ::= & \textsf{read-only} \mid \textsf{guarded by}\ \lambda \\
\textit{instructions} & \iota & ::= & \\
\quad \textit{control flow} & & & r := v \mid r := r + v \mid \textsf{if}\ r = v\ \textsf{jump}\ v \mid \\
\quad \textit{memory} & & & r := \textsf{new}\ n \mid r := v[n] \mid r[n] := v \mid \\
 & & & \textsf{share}\ r\ a \\
\quad \textit{unpack} & & & \omega, r := \textsf{unpack}\ v \mid \\
\quad \textit{lock} & & & \lambda, r := \textsf{newLock} \mid \\
 & & & r := \textsf{testSetLock}\ v \mid \textsf{unlock}\ v \mid \\
\quad \textit{fork} & & & \textsf{fork}\ v \\
\textit{inst. sequences} & I & ::= & \iota; I \mid \textsf{jump}\ v \mid \textsf{done}
\end{array}$$

The syntax of τ, λ, and ω is defined in Figure 3.9.

Figure 3.3: Instructions.

$$\begin{array}{lrcl}
\textit{permissions} & \Lambda & ::= & \lambda_1, \ldots, \lambda_n \\
\textit{access mode} & \pi & ::= & \textsf{ro} \mid \lambda \\
\textit{register files} & R & ::= & \{\mathsf{r}_1 \colon v_1, \ldots, \mathsf{r}_\mathsf{R} \colon v_\mathsf{R}\} \\
\textit{processor} & p & ::= & \langle R; \Lambda; I \rangle \\
\textit{processors array} & P & ::= & \{1 \colon p_1, \ldots, \mathsf{N} \colon p_\mathsf{N}\} \\
\textit{thread pool} & T & ::= & \{\langle l_1[\vec{\tau}_1], R_1 \rangle, \ldots, \langle l_n[\vec{\tau}_n], R_n \rangle\} \\
\textit{heap values} & h & ::= & \langle v_1 \ldots v_n \rangle^\pi \mid \tau\{I\} \\
\textit{heaps} & H & ::= & \{l_1 \colon h_1, \ldots, l_n \colon h_n\} \\
\textit{states} & S & ::= & \langle H; T; P \rangle \mid \textsf{halt}
\end{array}$$

Figure 3.4: Abstract machine.

$$\frac{\forall i.P(i) = \langle \_\,;\_\,;\mathsf{done}\rangle}{\langle \_; \emptyset; P\rangle \rightarrow \mathsf{halt}} \qquad \text{(R-HALT)}$$

$$\frac{P(i) = \langle \_\,;\_\,;\mathsf{done}\rangle \qquad H(l) = \forall[\vec{\omega}].(\_\ \mathsf{requires}\ \Lambda)\{I\}}{\langle H; T \uplus \{\langle l[\vec{\tau}], R\rangle\}; P\rangle \rightarrow \langle H; T; P\{i\colon \langle R; \Lambda; I\{\vec{\tau}/\vec{\omega}\}\rangle\}\rangle} \qquad \text{(R-SCHEDULE)}$$

$$\frac{P(i) = \langle R; \Lambda \uplus \Lambda'; (\mathsf{fork}\ v; I)\rangle \qquad \hat{\mathrm{R}}(v) = l[\vec{\tau}] \qquad H(l) = \forall[\_].(\_\ \mathsf{requires}\ \Lambda)\{\_\}}{\langle H; T; P\rangle \rightarrow \langle H; T \cup \{\langle l[\vec{\tau}], R\rangle\}; P\{i\colon \langle R; \Lambda'; I\rangle\}\rangle} \qquad \text{(R-FORK)}$$

Figure 3.5: Operational semantics (thread pool).

vectors of mutable values protected by some lock λ, or else of constant values (identified by tag ro). Code blocks comprise a signature and a body. The signature of a code block describes the type of the registers and the locks that must be held by the processor when jumping to the code block. The body is a sequence of instructions to be executed by a processor.

A thread pool is a multiset of pairs, each of which contains the address (*i.e.,* a label) of a code block and a register file. Each processor is composed of a register file, a set of locks (the locks held by the thread running at the processor), and a sequence of instructions (the remaining ones for execution). The processor array of the abstract machine contains N processors.

3.2 Operational Semantics

The operational semantics is presented in Figures 3.5 to 3.8. The run pool is managed by the rules in Figure 3.5. Rule R-HALT stops the machine when it finds an empty thread pool and all processors idle, changing the machine state to halt. Otherwise, if there is an idle processor and at least one thread waiting in the pool, then rule R-SCHEDULE assigns a thread to the idle processor. Rule R-FORK places a new thread in the pool; the permissions of the thread are split in two: required by the forked code, and the remaining ones. The thread keeps the latter set.

Operational semantics concerning locks are depicted in Figure 3.6. The instruction newLock creates an open lock ρ whose scope is the rest of the code block, allocates the respective value $\langle 0\rangle^{\rho}$ in the heap, and points register r to that (lock) tuple. A lock is an uni-dimensional tuple holding a *lock value* (an integer), because the machine provides for tuple allocation only; lock ρ is used for type safety purposes, just like all other singleton types. Value $\langle 0\rangle^{\rho}$ represents an open lock, whereas value $\langle 1\rangle^{\rho}$ represents a closed lock.

The *test-and-set* instruction, present in many machines designed for multithreading,

$$\frac{P(i) = \langle R; \Lambda; (\lambda, r := \textsf{newLock}; I)\rangle \quad l \notin \mathrm{dom}(H) \quad \rho \,\mathrm{fresh}}{\langle H; T; P\rangle \rightarrow \langle H\{l\colon \langle 0\rangle^{\rho}\}; T; P\{i\colon \langle R\{r\colon l\}; \Lambda; I[\rho/\lambda]\rangle\}\rangle} \qquad \text{(R-NEWLOCK)}$$

$$\frac{P(i) = \langle R; \Lambda; (r := \textsf{testSetLock}\, v; I)\rangle \quad \hat{\mathrm{R}}(v) = l \quad H(l) = \langle 0\rangle^{\lambda}}{\langle H; T; P\rangle \rightarrow \langle H\{l\colon \langle 1\rangle^{\lambda}\}; T; P\{i\colon \langle R\{r\colon 0\}; \Lambda \uplus \{\lambda\}; I\rangle\}\rangle} \qquad \text{(R-TSL 0)}$$

$$\frac{P(i) = \langle R; \Lambda; (r := \textsf{testSetLock}\, v; I)\rangle \quad H(\hat{\mathrm{R}}(v)) = \langle 1\rangle^{\lambda}}{\langle H; T; P\rangle \rightarrow \langle H; T; P\{i\colon \langle R\{r\colon 1\}; \Lambda; I\rangle\}\rangle} \qquad \text{(R-TSL 1)}$$

$$\frac{P(i) = \langle R; \Lambda \uplus \{\lambda\}; (\textsf{unlock}\, v; I)\rangle \quad \hat{\mathrm{R}}(v) = l \quad H(l) = \langle \_\rangle^{\lambda}}{\langle H; T; P\rangle \rightarrow \langle H\{l\colon \langle 0\rangle^{\lambda}\}; T; P\{i\colon \langle R; \Lambda; I\rangle\}\rangle} \qquad \text{(R-UNLOCK)}$$

Figure 3.6: Operational semantics (locks).

is an atomic operation that loads the contents of a bit into a register and then turns that bit on, *i.e.,* sets the bit to 1. Some architectures, like IBM's z/Architecture, serialise the execution of all the processors upon handling a test-and-set (*e.g.*, by blocking the bus) to ensure the atomicity of the operation.

In MIL, we have the *test-and-set-lock* instruction (testSetLock). The sole difference between test-and-set and test-and-set-lock is the semantics given to the target of the instruction, a bit in the case of the former, a tuple holding an integer in the latter case. A real-world implementation of MIL can represent the testSetLock by the test-and-set instruction, and the uni-dimensional tuple by a bit. The testSetLock, also atomic, stores the enclosed value in the left-hand register and turns the lock pointed by the right-hand side register into $\langle 1\rangle^{\lambda}$. Applying the instruction testSetLock to a lock in the unlocked state $\langle 0\rangle^{\lambda}$ changes the lock into the locked state $\langle 1\rangle^{\lambda}$, loads value 0 to register r, and adds lock λ to the permissions of the processor (Rule R-TSL0). Applying the instruction testSetLock to a closed lock, state $\langle 1\rangle^{\lambda}$, just places a 1 in register r (Rule R-TSL1). Locks are waved using instruction unlock, as long as the thread holds the lock (Rule R-UNLOCK).

An example of creating and then acquiring a lock follows, storing (a reference of) lock λ in register r_1.

```
λ, r1 := newLock
```

Remember the instruction newLock allocates the lock directly in the heap in the unlocked state; register r_1 holds a reference to the enclosed value. Next we try to acquire the open lock $\langle 0\rangle^{\lambda}$ present in register r_1: the value 0 contained in lock is loaded to register r_2.

```
r2 := testSetLock r1
```

Then we test the value in register r_2, jumping to code block criticalRegion if the lock was acquired; otherwise it jumps to code block tryAgain.

$$\frac{P(i) = \langle R; \Lambda; (r := \textsf{new}\; n; I)\rangle \qquad |\vec{0}| = n}{\langle H; T; P\rangle \rightarrow \langle H; T; P\{i\colon \langle R\{r\colon \langle\vec{0}\rangle\}; \Lambda; I\rangle\}\rangle}$$

(R-NEW)

$$\frac{\begin{array}{c} P(i) = \langle R; \Lambda; (\textsf{share}\; r\; a; I)\rangle \qquad R(r) = \langle\vec{v}\rangle \\ l \notin \mathrm{dom}(H) \qquad \pi \text{ is } \lambda \text{ when } a = \textsf{guarded by}\; \lambda \text{ else } \textsf{ro} \end{array}}{\langle H; T; P\rangle \rightarrow \langle H\{l\colon \langle\vec{v}\rangle^{\pi}\}; T; P\{i\colon \langle R\{r\colon l\}; \Lambda; I\rangle\}\rangle}$$

(R-SHARE)

$$\frac{P(i) = \langle R; \Lambda; (r := v[n]; I)\rangle \quad H(\hat{\mathrm{R}}(v)) = \langle v_1..v_n..v_{n+m}\rangle^{\pi} \quad \pi \in \{\textsf{ro}\} \cup \Lambda}{\langle H; T; P\rangle \rightarrow \langle H; T; P\{i\colon \langle R\{r\colon v_n\}; \Lambda; I\rangle\}\rangle}$$

(R-LOADH)

$$\frac{P(i) = \langle R; \Lambda; (r := r'[n]; I)\rangle \quad R(r') = \langle v_1..v_n..v_{n+m}\rangle}{\langle H; T; P\rangle \rightarrow \langle H; T; P\{i\colon \langle R\{r\colon v_n\}; \Lambda; I\rangle\}\rangle}$$

(R-LOADL)

$$\frac{P(i) = \langle R; \Lambda; (r[n] := v; I)\rangle \quad \hat{\mathrm{R}}(v) \neq \langle\_\rangle \quad R(r) = l \quad H(l) = \langle v_1..v_n..v_{n+m}\rangle^{\lambda} \quad \lambda \in \Lambda}{\langle H; T; P\rangle \rightarrow \langle H\{l\colon \langle v_1..\,\hat{\mathrm{R}}(v)..v_{n+m}\rangle^{\lambda}\}; T; P\{i\colon \langle R; \Lambda; I\rangle\}\rangle}$$

(R-STOREH)

$$\frac{P(i) = \langle R; \Lambda; (r[n] := r'; I)\rangle \qquad R(r') \neq \langle\_\rangle \qquad R(r) = \langle v_1..v_n..v_{n+m}\rangle}{\langle H; T; P\rangle \rightarrow \langle H; T; P\{i\colon \langle R\{r\colon \langle v_1..R(r')..v_{n+m}\rangle\}; \Lambda; I\rangle\}\rangle}$$

(R-STOREL)

Figure 3.7: Operational semantics (memory).

```
if r2 = 0 jump criticalRegion   -- λ is locked in code block 'criticalRegion'
jump tryAgain                   -- λ is unlocked in this instruction sequence
```

Control flow instructions are depicted in Figure 3.8 and discussed in the end of this section.

Rules related to memory manipulation are described in Figure 3.7. They rely on the evaluation function $\hat{\mathrm{R}}$ that looks for values in registers, in the pack constructor, and in the application of universal types.

$$\hat{\mathrm{R}}(v) = \begin{cases} R(v) & \text{if } v \text{ is a register} \\ \textsf{pack}\; \tau, \hat{\mathrm{R}}(v')\; \textsf{as}\; \tau' & \text{if } v \text{ is } \textsf{pack}\; \tau, v'\; \textsf{as}\; \tau' \\ \hat{\mathrm{R}}(v')[\tau] & \text{if } v \text{ is } v'[\tau] \\ v & \text{otherwise} \end{cases}$$

Rule R-NEW creates a new tuple in register r, local to some processor, of a given length n; its values are all initialised to zero. Sharing a tuple means transferring it from the processor's local memory into the heap. After sharing the tuple, register r records the fresh location l where the tuple is stored. Depending on the access method, the tuple may be protected by a lock λ, or tagged as read-only (rule R-SHARE). Values may be loaded from a tuple if the tuple is local, if the tuple is shared as a constant, or if the lock guarding the shared tuple is held by the processor. Values can be stored in a tuple when the tuple

$$\frac{P(i) = \langle R; \Lambda; \mathsf{jump}\ v\rangle \qquad H(\hat{\mathrm{R}}(v)) = \_\{I\}}{\langle H;T;P\rangle \rightarrow \langle H;T;P\{i\colon \langle R;\Lambda;I\rangle\}\rangle} \qquad \text{(R-}\textsc{jump}\text{)}$$

$$\frac{P(i) = \langle R; \Lambda; (r := v; I)\rangle \qquad \hat{\mathrm{R}}(v) \neq \langle \_ \rangle}{\langle H;T;P\rangle \rightarrow \langle H;T;P\{i\colon \langle R\{r\colon \hat{\mathrm{R}}(v)\};\Lambda;I\rangle\}\rangle} \qquad \text{(R-}\textsc{move}\text{)}$$

$$\frac{P(i) = \langle R; \Lambda; (r := r' + v; I)\rangle}{\langle H;T;P\rangle \rightarrow \langle H;T;P\{i\colon \langle R\{r\colon R(r') + \hat{\mathrm{R}}(v)\};\Lambda;I\rangle\}\rangle} \qquad \text{(R-}\textsc{arith}\text{)}$$

$$\frac{P(i) = \langle R; \Lambda; (\mathsf{if}\ r = v\ \mathsf{jump}\ v'; \_)\rangle \quad R(r) = v \quad H(\hat{\mathrm{R}}(v')) = \_\{I\}}{\langle H;T;P\rangle \rightarrow \langle H;T;P\{i\colon \langle R;\Lambda;I\rangle\}\rangle} \qquad \text{(R-}\textsc{branch}\text{T)}$$

$$\frac{P(i) = \langle R; \Lambda; (\mathsf{if}\ r = v\ \mathsf{jump}\ \_; I)\rangle \qquad R(r) \neq v}{\langle H;T;P\rangle \rightarrow \langle H;T;\{i\colon \langle R;\Lambda;I\rangle\}\rangle} \qquad \text{(R-}\textsc{branch}\text{F)}$$

$$\frac{P(i) = \langle R; \Lambda; (\omega, r := \mathsf{unpack}\ v; I)\rangle \qquad \hat{\mathrm{R}}(v) = \mathsf{pack}\ \tau, v'\ \mathsf{as}\ \_}{\langle H;T;P\rangle \rightarrow \langle H;T;P\{i\colon \langle R\{r\colon v'\};\Lambda;I[\tau/\omega]\rangle\}\rangle} \qquad \text{(R-}\textsc{unpack}\text{)}$$

Figure 3.8: Operational semantics (control flow).

is held in a register, or the tuple is in shared memory and the lock that guards the tuple is among the processor's permissions.

The reduction rules for control flow, illustrated in Figure 3.8, are straightforward. When the processor executes an instruction $\mathsf{jump}\ v$, rule R-JUMP, the processor transfers the control flow to the code block pointed by $\hat{\mathrm{R}}(v)$. With rule R-MOVE, the processor copies value $\hat{\mathrm{R}}(v)$ to register r. Arithmetic instructions are handled with rule R-ARITH: the processor expects as operands a register r' and a value v, and stores the computation in register r. Rules R-BRANCHT and R-BRANCHF govern both cases of a conditional jump, which test the equality of register r with value v and either jumps to code block pointed by value $\hat{\mathrm{R}}(v')$ upon success, or continues executing instruction I upon failure. Finally, rule R-UNPACK copies the value v' inside the package v into register r.

3.3 Type Discipline

The syntax of types is depicted in Figure 3.9. A type of the form $\langle\vec{\tau}\rangle^{\pi}$ describes a heap allocated tuple: shared and protected by a lock λ if π is λ, or shared and read-only if π is ro (*cf.* Figure 3.4). For example the type $\langle \mathsf{int}, \mathsf{int}\rangle^{\lambda}$ represents a shared tuple type, protected by lock λ, that holds two values of the integer type. A type $\langle\vec{\tau}\rangle$ describes a local tuple

$$\begin{array}{llcl}
\textit{types} & \tau & ::= & \mathsf{int} \mid \langle\vec{\tau}\rangle^{\pi} \mid \langle\vec{\tau}\rangle \mid \forall[\vec{\omega}].(\Gamma\ \mathsf{requires}\ \Lambda) \mid \\
 & & & \lambda \mid \alpha \mid \exists\omega.\tau \mid \mu\alpha.\tau \\
\textit{type variable or lock} & \omega & ::= & \alpha \mid \lambda \\
\textit{register file types} & \Gamma & ::= & \mathsf{r}_1 \colon \tau_1, \ldots, \mathsf{r}_n \colon \tau_n \\
\textit{typing environment} & \Psi & ::= & \emptyset \mid \Psi, l \colon \tau \mid \Psi, \lambda :: \mathsf{Lock} \mid \Psi, \alpha :: \mathsf{TyVar}
\end{array}$$

Figure 3.9: Types.

created directly in a register. A type of the form $\forall[\vec{\omega}].(\Gamma\ \mathsf{requires}\ \Lambda)$ describes a code block: a thread jumping into such a block must instantiate all the universal variables $\vec{\omega}$ (type variables α or singleton lock types λ), it must also hold a register file type Γ, as well as the locks in Λ. We omit the empty universal quantifier abstraction and the empty lock set from code types when necessary, thus $\forall[].(\Gamma\ \mathsf{requires}\ \Lambda) \stackrel{\text{def}}{=} \Gamma\ \mathsf{requires}\ \Lambda$ and $\forall[\vec{\omega}].(\Gamma\ \mathsf{requires}\ \emptyset) \stackrel{\text{def}}{=} \forall[\vec{\omega}].\Gamma$. The heap representation of a lock is an uni-dimensional tuple; the singleton lock type λ is used to describe the type of the value enclosed in a lock.

Types $\exists\alpha.\tau$ are conventional existential types. With type $\exists\lambda.\tau$ we are able to existentially quantify over lock types, following [14]. The scope of a lock extends until the end of the instruction sequence that creates it, *cf.* rule T-NEW. There are situations, however, in which we need to use a lock outside its scope. Consider a thread where $\lambda \in \Psi$. If this thread jumps to label l where $\lambda \notin \Psi$, then it may happen that we need to mention λ again (reintroduce λ in Ψ). For the code block pointed by label l it suffices to know that such a lock exists. Please refer to rule T-UNPACK for more information regarding the introduction of a lock λ in Ψ.

As usual, the recursive type is defined by $\mu\alpha.\tau$. We take an equi-recursive view of types, not distinguishing between a type $\mu\alpha.\tau$ and its unfolding type $\tau[\mu\alpha.\tau/\alpha]$.

There are four bindings:

- in the code type $\forall[\vec{\omega}].(\Gamma\ \mathsf{requires}\ \Lambda)$ each displayed occurrence of ω_i is a binding with scope $(\Gamma\ \mathsf{requires}\ \Lambda)$;
- in the existential type $\exists\omega.\tau\ \ \omega$ is a binding with scope τ;
- in the recursive type $\mu\alpha.\tau$ type variable α is a binding with scope τ;
- finally, in the code fragment $l\ \forall[\vec{\omega}].(\Gamma\ \mathsf{requires}\ \Lambda)\{I\}$ each displayed occurrence of ω_i is a binding with scope $(\Gamma\ \mathsf{requires}\ \Lambda)$ and scope I.

Function $\mathrm{ftv}(\tau)$ denotes the set of free type variables in τ, and $\mathrm{flt}(\tau)$ the set of free singleton lock types in τ.

$$\frac{\forall \omega \in \mathrm{ftv}(\tau).\omega :: \ \mathrm{kind}(\omega) \in \Psi}{\Psi \vdash \tau} \qquad \text{(T-TYPE)}$$

$$\frac{\Psi \vdash \tau_1 <: \tau_2 \quad \Psi \vdash \tau_2 <: \tau_3}{\Psi \vdash \tau_1 <: \tau_3} \qquad \frac{\Psi \vdash \tau_i}{\Psi \vdash \mathsf{r}_1 \colon \tau_1, \ldots, \mathsf{r}_{n+m} \colon \tau_{n+m} <: \mathsf{r}_1 \colon \tau_1, \ldots, \mathsf{r}_n \colon \tau_n}$$

(S-TRANS,S-REGFILE)

$$\frac{\Psi \vdash \tau}{\Psi \vdash \tau <: \tau} \qquad \frac{\Psi, \vec{\omega} :: \ \mathrm{kind}(\vec{\omega}) \vdash \Gamma <: \Gamma'}{\Psi \vdash \forall[\vec{\omega}].(\Gamma \ \mathsf{requires}\ \Lambda) <: \forall[\vec{\omega}].(\Gamma' \ \mathsf{requires}\ \Lambda)}$$

(S-REFLEX,S-CODE)

$$\frac{\Psi \vdash \tau' <: \tau}{\Psi, l \colon \tau'; \Gamma \vdash l \colon \tau} \qquad \Psi; \Gamma \vdash r \colon \Gamma(r) \qquad \Psi; \Gamma \vdash n \colon \mathsf{int} \qquad \text{(T-LABEL,T-REG,T-INT)}$$

$$\frac{\Psi \vdash \tau \quad \Psi; \Gamma \vdash v \colon \tau'[\tau/\omega] \quad \omega \notin \tau, \Psi \quad \tau' \neq \langle \_ \rangle}{\Psi; \Gamma \vdash \mathsf{pack}\ \tau, v\ \mathsf{as}\ \exists \omega.\tau' \colon \exists \omega.\tau'} \qquad \frac{\forall i. \Psi; \Gamma \vdash v_i : \tau_i}{\Psi; \Gamma \vdash \langle \vec{v} \rangle : \langle \vec{\tau} \rangle}$$

(T-PACK,T-TUPLE)

$$\frac{\Psi \vdash \tau \quad \Psi; \Gamma \vdash v \colon \forall[\nu\vec{\omega}].(\Gamma' \ \mathsf{requires}\ \Lambda) \quad \tau \neq \langle \_ \rangle}{\Psi; \Gamma \vdash v[\tau] \colon \forall[\vec{\omega}].(\Gamma'[\tau/\nu] \ \mathsf{requires}\ \Lambda[\tau/\nu])} \qquad \text{(T-VALAPP)}$$

Figure 3.10: Typing rules for values $\boxed{\Psi; \Gamma \vdash v \colon \tau}$, subtyping rules $\boxed{\Psi \vdash \tau <: \tau}$, and typing rules for types $\boxed{\Psi \vdash \tau}$.

$$\Psi; \Gamma; \emptyset \vdash \mathsf{done} \qquad \text{(T-DONE)}$$

$$\frac{\forall i. \Gamma(r_i) \neq \langle \_ \rangle \qquad \Psi; \Gamma \vdash v \colon \forall[].(\Gamma \ \mathsf{requires}\ \Lambda) \qquad \Psi; \Gamma; \Lambda' \vdash I}{\Psi; \Gamma; \Lambda \uplus \Lambda' \vdash \mathsf{fork}\ v; I} \qquad \text{(T-FORK)}$$

$$\frac{\Psi, \lambda :: \mathsf{Lock}; \Gamma\{r \colon \langle \lambda \rangle^{\lambda}\}; \Lambda \vdash I[\lambda/\rho] \qquad \lambda \notin \Psi, \Gamma, \Lambda, I}{\Psi; \Gamma; \Lambda \vdash \rho, r := \mathsf{newLock}; I} \qquad \text{(T-NEWLOCK)}$$

$$\frac{\Psi; \Gamma \vdash v \colon \langle \lambda \rangle^{\lambda} \qquad \Psi; \Gamma\{r \colon \lambda\}; \Lambda \vdash I \qquad \lambda \notin \Lambda}{\Psi; \Gamma; \Lambda \vdash r := \mathsf{testSetLock}\ v; I} \qquad \text{(T-TSL)}$$

$$\frac{\Psi; \Gamma \vdash v \colon \langle \lambda \rangle^{\lambda} \qquad \Psi; \Gamma; \Lambda \vdash I}{\Psi; \Gamma; \Lambda \uplus \{\lambda\} \vdash \mathsf{unlock}\ v; I} \qquad \text{(T-UNLOCK)}$$

$$\frac{\Psi; \Gamma \vdash r \colon \lambda \quad \Psi; \Gamma \vdash v \colon \forall[].(\Gamma \ \mathsf{requires}\ (\Lambda \uplus \{\lambda\})) \quad \Psi; \Gamma; \Lambda \vdash I}{\Psi; \Gamma; \Lambda \vdash \mathsf{if}\ r = 0\ \mathsf{jump}\ v; I} \qquad \text{(T-CRITICAL)}$$

Figure 3.11: Typing rules for instructions (thread pool and locks) $\boxed{\Psi; \Gamma; \Lambda \vdash I}$.

The type system is presented in Figures 3.10 to 3.13. Typing judgements target a typing environment Ψ that maps heap labels to types, type variables to kind TyVar, and singleton lock types to kind Lock. Function $\mathrm{kind}(\cdot)$ returns the kind associated with the given singleton lock type, or type variable.

$$\mathrm{kind}(\omega) = \begin{cases} \mathsf{Lock} & \text{if } \omega \text{ is } \lambda \\ \mathsf{TyVar} & \text{if } \omega \text{ is } \alpha \end{cases}$$

We keep track of type variables and of singleton lock types in the typing environment for checking if types are *well-formed* (*cf.* rule T-TYPE). Instructions are checked against a typing environment Ψ, a register file type Γ holding the current types of the registers, and a set Λ of lock variables: the *permission* of (the processor executing) the code block.

Typing rules for values are illustrated in Figure 3.10. Heap values are distinguished from operands (that include registers as well) by the form of the sequent. A formula $\Gamma <: \Gamma'$ allows "forgetting" registers in the register file type, and is particularly useful in jump instructions where we want the type of the target code block to be more general (ask for less registers) than those active in the current code [31]. As an example, consider the subtyping relation $(\mathsf{r}_1 : \mathsf{int}, \mathsf{r}_2 : \langle\mathsf{int}\rangle^{\lambda}) <: (\mathsf{r}_1 : \mathsf{int})$, where we "forget" register r_2; we expect a register file type with a register r_1 holding an integer type, the remaining registers are disregarded. Rule T-TYPE makes sure types are *well-formed*, which is to only include type variables and singleton lock types present in scope. The rules for value application and for pack values, T-VALAPP and T-PACK, work both with type variables α and with singleton lock types λ, taking advantage of the fact that substitution $\tau'[\tau/\omega]$ is defined only when τ is not a singleton lock type and ω is a type variable, or when both τ and ω are singleton lock types. In either case, type τ must be well-formed.

The rules in Figure 3.11 capture most part of the policy over lock usage. Rule T-DONE requires the release of all locks before terminating the processor terminates the thread. Rule T-FORK splits permissions into sets Λ and Λ': the former is transferred to the forked thread according to the permissions required by the target code block, the latter remains with the processor.

As an example, consider typechecking an instruction sequence $\mathsf{fork}\ l; \mathsf{done}$ that forks label l and concludes. Let the processor permissions be lock λ for this instruction sequence. Furthermore, let $\Psi \stackrel{\text{def}}{=} \lambda :: \mathsf{Lock}, l : \forall[].(\Gamma\ \mathsf{requires}(\lambda))$ and Γ stand for any possible register set that $\mathrm{ftv}(\Gamma) = \emptyset$. Then, the inference tree for $\mathsf{fork}\ l; \mathsf{done}$ is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{ftv}(\Gamma) = \emptyset}{\Psi \vdash \forall[].(\Gamma\ \mathsf{requires}\ (\lambda))}\ \text{T-TYPE}}{\Psi \vdash \forall[].(\Gamma\ \mathsf{requires}\ (\lambda)) <: \forall[].(\Gamma\ \mathsf{requires}\ (\lambda))}\ \text{S-REFLEX}}{\Psi; \Gamma \vdash l : \forall[].(\Gamma\ \mathsf{requires}\ (\lambda))}\ \text{T-LABEL} \qquad \dfrac{}{\Psi; \Gamma; \emptyset \vdash \mathsf{done}}\ \text{T-DONE}}{\Psi; \Gamma; \lambda \vdash \mathsf{fork}\ l; \mathsf{done}}\ \text{T-FORK}$$

$$\frac{\Psi;\Gamma\{r\colon\langle\vec{\mathsf{int}}\rangle\};\Lambda\vdash I \qquad |\vec{\mathsf{int}}|=n}{\Psi;\Gamma;\Lambda\vdash r:=\mathsf{new}\ n;I}\tag{T-NEW}$$

$$\frac{\Psi\vdash\lambda \quad \Psi;\Gamma\vdash r\colon\langle\vec{\tau}\rangle \quad \Psi;\Gamma\{r\colon\langle\vec{\tau}\rangle^{\lambda}\};\Lambda\vdash I}{\Psi;\Gamma;\Lambda\vdash\mathsf{share}\ r\ \mathsf{guarded\ by}\ \lambda;I}\tag{T-SHAREL}$$

$$\frac{\Psi;\Gamma\vdash r\colon\langle\vec{\tau}\rangle \quad \Psi;\Gamma\{r\colon\langle\vec{\tau}\rangle^{\mathsf{ro}}\};\Lambda\vdash I}{\Psi;\Gamma;\Lambda\vdash\mathsf{share}\ r\ \mathsf{read\text{-}only};I}\tag{T-SHARER}$$

$$\frac{\Psi;\Gamma\vdash v\colon\langle\tau_1..\tau_{n+m}\rangle^{\pi} \quad \Psi;\Gamma\{r\colon\tau_n\};\Lambda\vdash I \quad \tau_n\neq\lambda \quad \pi\in\Lambda\cup\{\mathsf{ro}\}}{\Psi;\Gamma;\Lambda\vdash r:=v[n];I}\tag{T-LOADH}$$

$$\frac{\Psi;\Gamma\vdash v\colon\langle\tau_1..\tau_{n+m}\rangle \quad \Psi;\Gamma\{r\colon\tau_n\};\Lambda\vdash I \quad \tau_n\neq\lambda}{\Psi;\Gamma;\Lambda\vdash r:=v[n];I}\tag{T-LOADL}$$

$$\frac{\Psi;\Gamma\vdash v:\tau_n \quad \Psi;\Gamma\vdash r:\langle\tau_1..\tau_{n+m}\rangle^{\lambda} \quad \Psi;\Gamma\{r:\langle\tau_1..\tau_{n+m}\rangle\};\Lambda\vdash I \quad \tau_n\neq\lambda,\langle\_\rangle \quad \lambda\in\Lambda}{\Psi;\Gamma;\Lambda\vdash r[n]:=v;I}\tag{T-STOREH}$$

$$\frac{\Psi;\Gamma\vdash v\colon\tau \quad \Psi;\Gamma\vdash r\colon\langle\tau_1..\tau_n..\tau_{n+m}\rangle \quad \Psi;\Gamma\{r\colon\langle\tau_1..\tau..\tau_{n+m}\rangle\};\Lambda\vdash I \quad \tau\neq\lambda,\langle\_\rangle}{\Psi;\Gamma;\Lambda\vdash r[n]:=v;I}\tag{T-STOREL}$$

$$\frac{\Psi;\Gamma\vdash v\colon\tau \qquad \Psi;\Gamma\{r\colon\tau\};\Lambda\vdash I \qquad \tau\neq\langle\_\rangle}{\Psi;\Gamma;\Lambda\vdash r:=v;I}\tag{T-MOVE}$$

$$\frac{\Psi;\Gamma\vdash r'\colon\mathsf{int} \qquad \Psi;\Gamma\vdash v\colon\mathsf{int} \qquad \Psi;\Gamma\{r\colon\mathsf{int}\};\Lambda\vdash I}{\Psi;\Gamma;\Lambda\vdash r:=r'+v;I}\tag{T-ARITH}$$

$$\frac{\Psi;\Gamma\vdash v\colon\exists\omega.\tau \qquad \Psi,\omega\colon\colon\ \mathrm{kind}(\omega);\Gamma\{r\colon\tau\};\Lambda\vdash I \qquad \omega\notin\Psi,\Gamma,\Lambda}{\Psi;\Gamma;\Lambda\vdash\omega,r:=\mathsf{unpack}\ v;I}\tag{T-UNPACK}$$

$$\frac{\Psi;\Gamma\vdash r\colon\mathsf{int} \quad \Psi;\Gamma\vdash v\colon\mathsf{int} \quad \Psi;\Gamma\vdash v\colon\forall[].(\Gamma\ \mathsf{requires}\ \Lambda) \quad \Psi;\Gamma;\Lambda\vdash I}{\Psi;\Gamma;\Lambda\vdash\mathsf{if}\ r=v\ \mathsf{jump}\ v;I}\tag{T-BRANCH}$$

$$\frac{\Psi;\Gamma\vdash v\colon\forall[].(\Gamma\ \mathsf{requires}\ \Lambda)}{\Psi;\Gamma;\Lambda\vdash\mathsf{jump}\ v}\tag{T-JUMP}$$

Figure 3.12: Typing rules for instructions (memory and control flow) $\boxed{\Psi;\Gamma;\Lambda\vdash I}$.

Notice that after forking the permission over lock λ is lost, thereby allowing thread termination.

Rules T-NEWLOCK, T-TSL, T-UNLOCK, and T-CRITICAL are about lock manipulation and therefore constitute part of the enforced lock discipline. Rule T-NEWLOCK assigns type $\langle\lambda\rangle^\lambda$ to the register r that holds a reference to the newly created lock. The new singleton lock type λ is recorded in Ψ, so that it may be used in the rest of the instructions I. Rule T-TSL requires that the value under test is a lock in the heap (of type $\langle\lambda\rangle^\lambda$) and records the type of the lock value λ in register r. This rule also disallows testing a lock already held by the processor. Rule T-UNLOCK makes sure that only held locks are unlocked. Rule T-CRITICAL ensures that the processor holds the exact locks required by the target code block, including the lock under test. A processor is guaranteed to hold the tested lock only after (conditionally) jumping to the critical region. A previous test-and-set-lock instructions may have obtained the lock, but the type system records that the processor holds the lock only after the conditional jump. As an example consider a processor that tries to acquire a lock λ and jumps to a label l if successful. Let $\Psi \overset{\text{def}}{=} \lambda{::}\,\mathsf{Lock}, l : (\mathsf{r}_1 : \langle\lambda\rangle^\lambda)\ \mathsf{requires}\ (\lambda)$.

$$\dfrac{\dfrac{}{\Psi; (\mathsf{r}_1 : \langle\lambda\rangle^\lambda) \vdash \mathsf{r}_1 : \langle\lambda\rangle^\lambda}\ \text{T-REG} \qquad (1) \quad \lambda \notin \emptyset}{\Psi; (\mathsf{r}_1 : \langle\lambda\rangle^\lambda); \emptyset \vdash \mathsf{r}_2 := \mathsf{testSetLock}\ \mathsf{r}_1; \mathsf{if}\ \mathsf{r}_2 = 0\ \mathsf{jump}\ l; I}\ \text{T-TSL}$$

The inference for (1) is as follows.

$$\dfrac{\dfrac{}{\Psi; (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda) \vdash \mathsf{r}_2 : \lambda}\ \text{T-REG} \qquad (2) \quad \Psi; (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda); \emptyset \vdash I}{\Psi; (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda); \emptyset \vdash \mathsf{if}\ \mathsf{r}_2 = 0\ \mathsf{jump}\ l; I}\ \text{T-CRITICAL}$$

The typing judgement above is valid if $\Psi; (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda); \emptyset \vdash I$. Finally, the inference for (2), $\Psi; (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda) \vdash l : (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda)\ \mathsf{requires}\ (\lambda)$, concludes the example by showing the subtyping.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{ftv}(\langle\lambda\rangle^\lambda) = \{\lambda\} \qquad \Psi(\lambda) = \lambda{::}\,\mathsf{Lock}}{\Psi \vdash \langle\lambda\rangle^\lambda}\ \text{T-TYPE}}{\Psi \vdash (\mathsf{r}_1 : \langle\lambda\rangle^\lambda) <: (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda)}\ \text{S-REGFILE}}{\Psi \vdash (\mathsf{r}_1 : \langle\lambda\rangle^\lambda)\ \mathsf{requires}\ (\lambda) <: (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda)\ \mathsf{requires}\ (\lambda)}\ \text{S-CODE}}{\Psi \vdash l : (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda)\ \mathsf{requires}\ (\lambda)}\ \text{T-LABEL}$$

Notice the application of subtyping to the target of instruction jump, where the code type assigned to l is $(\mathsf{r}_1 : \langle\lambda\rangle^\lambda)\ \mathsf{requires}\ (\lambda)$, the register file of the processor at that point is $(\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda)$, and $\Psi \vdash (\mathsf{r}_1 : \langle\lambda\rangle^\lambda) <: (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda)$. We "forget" the lock value in register r_2. Rule S-CODE expects the same permission set (λ) in either code type of the subtyping relation. If the subtyping relation would allow $\Psi \vdash (\mathsf{r}_1 : \langle\lambda\rangle^\lambda) <: (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda)\ \mathsf{requires}\ (\lambda)$, then we would be able to "forget" held locks, permitting a thread to finish without unlocking all previously held locks. If the subtyping relation would admit $\Psi \vdash (\mathsf{r}_1 : \langle\lambda\rangle^\lambda)\ \mathsf{requires}\ (\lambda) <: (\mathsf{r}_1 : \langle\lambda\rangle^\lambda, \mathsf{r}_2 : \lambda)$, then we would be able add a lock to the permissions set without using testSetLock, enabling race conditions and unlocking open locks. Both of these cases invalidate the lock policy, thus they do not typecheck.

The typing rules for memory and control flow are depicted in Figure 3.12. Operations for loading from (T-LOADH), and for storing into (T-STOREH), shared tuples require that the processor hold the right permissions (the locks for the tuples it reads from, or writes to).

For example, consider a lock λ known to the processor, let the register file type of that processor be $(\mathsf{r}_1 : \langle\mathsf{int}\rangle^\lambda)$, and let us assume that the processor is trying to load the integer contained in the tuple without having lock λ held. The following judgement does not hold because lock λ is not in the lock set.

$$\dfrac{\dfrac{}{\lambda :: \mathsf{Lock}; (\mathsf{r}_1 : \langle\mathsf{int}\rangle^\lambda) \vdash \mathsf{r}_1 : \langle\mathsf{int}\rangle^\lambda}\,\textsc{T-reg} \qquad \mathsf{int} \neq \lambda' \quad \boxed{\lambda \notin \emptyset}}{\lambda :: \mathsf{Lock}; (\mathsf{r}_1 : \langle\mathsf{int}\rangle^\lambda); \emptyset \nvdash \mathsf{r}_2 := \mathsf{r}_1[1]; I}\,\textsc{T-loadH}$$

Notice that $\mathsf{int} \neq \lambda'$ avoids the direct manipulation of locks.

A subtlety of rule T-STOREH is that subtyping applies to the value being placed in the tuple, which allows to "forget" registers of stored code blocks. In the following example we store a code block in a shared tuple. Consider a lock λ and a label l known to a thread, $\Psi \overset{\text{def}}{=} \lambda :: \mathsf{Lock}, l : (\mathsf{r}_1 : \mathsf{int}, \mathsf{r}_2 : \mathsf{int})$, and let the register file of the thread be $\Gamma \overset{\text{def}}{=} \mathsf{r}_1 : \langle(\mathsf{r}_1 : \mathsf{int})\rangle^\lambda$. Although the code type $(\mathsf{r}_1 : \mathsf{int}, \mathsf{r}_2 : \mathsf{int})$ pointed by l is different from the expected type $(\mathsf{r}_1 : \mathsf{int})$ to be stored in the tuple, $(\mathsf{r}_1 : \mathsf{int})$ is more general than $(\mathsf{r}_1 : \mathsf{int}, \mathsf{r}_2 : \mathsf{int})$, thus the following type judgement holds.

$$\dfrac{(1) \quad \Psi; \Gamma \vdash \langle(\mathsf{r}_1 : \mathsf{int})\rangle^\lambda \quad \Psi; \Gamma; \lambda \vdash I \quad (\mathsf{r}_1 : \mathsf{int}) \neq \lambda', \langle\_\rangle \quad \lambda \in \lambda}{\Psi; \Gamma; \lambda \vdash \mathsf{r}_1[1] := l; I}\,\textsc{T-storeH}$$

The relevant part is the sequent (1), where we observe subtyping:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{ftv}(\mathsf{int}) = \emptyset}{\Psi \vdash \mathsf{int}}\,\textsc{T-type}}{\Psi \vdash (\mathsf{r}_1 : \mathsf{int}, \mathsf{r}_2 : \mathsf{int}) <: (\mathsf{r}_1 : \mathsf{int})}\,\textsc{S-regFile}}{\Psi \vdash (\mathsf{r}_1 : \mathsf{int}, \mathsf{r}_2 : \mathsf{int}) <: (\mathsf{r}_1 : \mathsf{int})}\,\textsc{S-code}}{\lambda :: \mathsf{Lock}, l : (\mathsf{r}_1 : \mathsf{int}, \mathsf{r}_2 : \mathsf{int}); \Gamma \vdash l : (\mathsf{r}_1 : \mathsf{int})}\,\textsc{T-label}$$

Local tuples are affected by rules T-NEW, T-SHARER, T-SHAREL, T-LOADL, and T-STOREL. Upon checking a processor that allocates a local tuple of size n in register r, we record, in the register file type, that register r is of a local tuple type that consists of n integer types. Storing a value in, or loading a value from, a local tuple requires no particular permission, rules T-LOADL and T-STOREL. Like shared tuples, it is disallowed to store/load lock values in/from local tuples. Shared tuples and local tuples have a big difference: the type of each component of a shared tuple is constant, whereas the type of each component of a local tuple changes after each store. For example, consider that we want to store a label l in a local tuple of type $\langle \mathsf{int} \rangle$ contained in register r_1. Let $\Psi \overset{\text{def}}{=} l : (\mathsf{r}_1 : \mathsf{int})$. The following judgement holds.

$$\frac{\Psi; (\mathsf{r}_1 : \langle \mathsf{int} \rangle) \vdash l : (\mathsf{r}_1 : \mathsf{int}) \quad \Psi; (\mathsf{r}_1 : \langle \mathsf{int} \rangle) \vdash \mathsf{r}_1 : \langle \mathsf{int} \rangle \quad (1) \quad (\mathsf{r}_1 : \mathsf{int}) \neq \lambda, \langle \_ \rangle}{\Psi; (\mathsf{r}_1 : \langle \mathsf{int} \rangle); \emptyset \vdash \mathsf{r}_1[1] := l; \mathsf{done}} \text{ T-STOREL}$$

Where (1) is $\Psi; (\mathsf{r}_1 : \langle (\mathsf{r}_1 : \mathsf{int}) \rangle); \emptyset \vdash \mathsf{done}$ that holds, by T-DONE.

The type system enforces that a local tuple is stored in only one register, special care is therefore taken to disallow duplications and aliasing of local tuples, via the various premises $\tau \neq \langle \_ \rangle$ in the rules. With local tuples in only one place (a register), rules T-SHAREL and T-SHARER transform only one local tuple type into a shared tuple type, not having to keep track of other references to that value. We outline an example of the application of rule T-MOVE that disallows aliasing a local tuple from register r_1 to register r_2. The following instruction sequence does not typecheck.

$$\frac{\emptyset; (\mathsf{r}_1 : \langle \mathsf{int} \rangle) \vdash \mathsf{r}_1 : \langle \mathsf{int} \rangle \quad \boxed{\langle \mathsf{int} \rangle = \langle \_ \rangle}}{\emptyset; (\mathsf{r}_1 : \langle \mathsf{int} \rangle); \emptyset \nvdash \mathsf{r}_2 := \mathsf{r}_1; I} \text{ T-MOVE}$$

Restricting the abstraction of local types is fundamental to eliminate alias, see rules T-PACK and T-VALAPP. Consider code block move that moves the contents of register r_1 to register r_2 and then jumps to a continuation code block in register r_3.

```
move ∀[τ] (r1:τ, r3:(r1:τ,r2:τ)) {
  r2 := r1
  jump r3
}
```

Local tuple types must be restricted from the instantiation in the universal operator, otherwise a thread jumping to code block move could send a local tuple in register r_1, which would be then copied to register r_2. That is the reason why the following typing judgement fails, where $\Psi \overset{\text{def}}{=} \mathsf{move} : \forall[\tau].(\mathsf{r}_1 : \tau,\ \mathsf{r}_3 : (\mathsf{r}_1 : \tau, \mathsf{r}_2 : \tau)),\ l : (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : \langle \mathsf{int} \rangle)$

$$\frac{\Psi; (\mathsf{r}_1 : \langle \mathsf{int} \rangle) \vdash l : (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : \langle \mathsf{int} \rangle) \quad \boxed{(1)} \quad (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : \langle \mathsf{int} \rangle) \neq \langle \_ \rangle}{\Psi; (\mathsf{r}_1 : \langle \mathsf{int} \rangle); \emptyset \nvdash \mathsf{r}_3 := l; \mathsf{jump}\ \mathsf{move}[\langle \mathsf{int} \rangle]} \text{ T-MOVE}$$

Let $\Gamma \overset{\text{def}}{=} (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : \langle \mathsf{int} \rangle))$. The judgement above fails because the sequent (1) fails:

$$\cfrac{\cfrac{\cfrac{\mathrm{ftv}(\langle \mathsf{int} \rangle) = \emptyset}{\Psi \vdash \langle \mathsf{int} \rangle} \text{ T-TYPE} \quad \Psi; (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : \langle \mathsf{int} \rangle)) \vdash \mathsf{move} : \_ \quad \boxed{\langle \mathsf{int} \rangle = \langle \_ \rangle}}{\Psi; (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : \langle \mathsf{int} \rangle)) \nvdash \mathsf{move}[\langle \mathsf{int} \rangle] : \_} \text{ T-VALAPP}}{\Psi; (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : (\mathsf{r}_1 : \langle \mathsf{int} \rangle, \mathsf{r}_2 : \langle \mathsf{int} \rangle)); \emptyset \nvdash \mathsf{jump}\ \mathsf{move}[\langle \mathsf{int} \rangle] : \_} \text{ T-JUMP}$$

$$\frac{\forall i.\Psi \vdash \Gamma(\mathsf{r}_i) \qquad \Psi;\emptyset \vdash R(\mathsf{r}_i)\colon \Gamma(\mathsf{r}_i)}{\Psi \vdash R\colon \Gamma} \qquad (\text{reg file, } \boxed{\Psi \vdash R\colon \Gamma})$$

$$\frac{\forall i.\Psi \vdash P(i)}{\Psi \vdash P} \qquad \frac{\Psi \vdash R\colon \Gamma \qquad \Psi;\Gamma;\Lambda \vdash I}{\Psi \vdash \langle R;\Lambda;I\rangle} \qquad (\text{processors, } \boxed{\Psi \vdash P})$$

$$\frac{\forall i.\Psi \vdash l_i\colon \forall[\vec{\omega}_i].(\Gamma_i \textsf{ requires } \_) \qquad \Psi \vdash R_i\colon \Gamma_i\{\vec{\tau}_i/\vec{\omega}_i\}}{\Psi \vdash \{\langle l_1[\vec{\tau}_1], R_1\rangle, \dots, \langle l_n[\vec{\tau}_n], R_n\rangle\}} \qquad (\text{thread pool, } \boxed{\Psi \vdash T})$$

$$\frac{\tau = \forall[\vec{\omega}].(\Gamma \textsf{ requires } \Lambda) \quad \Psi, \vec{\omega} :: \text{kind}(\vec{\omega}); \Gamma; \Lambda \vdash I}{\Psi \vdash \tau\{I\}\colon \tau} \qquad \frac{\forall i.\Psi;\emptyset \vdash v_i\colon \tau_i \quad r, \langle\_\rangle \notin v_i}{\Psi \vdash \langle\vec{v}\rangle^{\pi}\colon \langle\vec{\tau}\rangle^{\pi}}$$

$$(\text{heap value, } \boxed{\Psi \vdash h\colon \tau})$$

$$\frac{\forall l.\Psi \vdash H(l)\colon \Psi(l)}{\Psi \vdash H} \qquad (\text{heap, } \boxed{\Psi \vdash H})$$

$$\vdash \textsf{halt} \qquad \frac{\Psi \vdash H \qquad \Psi \vdash T \qquad \Psi \vdash P}{\vdash \langle H;T;P\rangle} \qquad (\text{state, } \boxed{\vdash S})$$

Figure 3.13: Typing rules for machine states.

Rule T-UNPACK unpacks either a conventional or a lock existential type. A new entry $\alpha :: \textsf{TyVar}$ or $\lambda :: \textsf{Lock}$ is added to Ψ, according to the nature of ω. The new type variable or singleton lock type may then be used in the rest of the instructions I.

The rules for typing machine states are illustrated in Figure 3.13. The heap value rule for code blocks places each type variable and singleton type $\vec{\omega}$ of the universal abstraction in Ψ, so that they may be used in the rest of the instructions I. The heap value rule for shared tuples restricts the contents of a shared tuple to integers, labels, packed values, and instantiations of types, thereby leaving out registers and local tuples. For example, the following heap value does not typecheck, because there are registers and local tuples in the heap value $\langle \mathsf{r}_1, \langle 2\rangle\rangle^{\mathsf{ro}}$:

$$\frac{r, \langle\_\rangle \in \langle \mathsf{r}_1, \langle 2\rangle\rangle^{\mathsf{ro}}}{\Psi \nvdash \langle \mathsf{r}_1, \langle 2\rangle\rangle^{\mathsf{ro}} : \_} \text{heap value}$$

The remaining rules are straightforward.

3.4 MIL programming examples

We introduce MIL programming by showing common patterns that are used throughout the design of the supporting code and in the translation function. We show how to acquire locks and how to compose code blocks.

Acquiring locks. To illustrate lock manipulation we select a case in point of interprocess communication: mutual exclusion. Two threads compete to enter a critical region to write in a common tuple. In this example, each thread uses a different algorithm to acquire the lock protecting the shared data: one uses spin-lock, another one uses sleep-lock. Code block main initialises the tuple and launches both threads, referred by labels sleep and spin.

```
main() {
  λ, r1 := newLock  −− create the lock
  r2 := malloc 1    −− allocate a tuple holding an integer
  r2[1] := 0        −− initialise the tuple with a zero
  share r2 guarded by λ
  fork sleep[λ]     −− start 'sleep' in a new thread
  fork spin[λ]      −− start 'spin' in a new thread
  done
}
```

In code block spin, we observe a standard technique for acquiring a lock called *spin-lock*, where the processor actively tries to acquire the lock (busy wait). A processor spin-locking cannot execute another thread until the lock is acquired.

```
spin ∀[λ](r1: ⟨λ⟩^λ, r2: ⟨int⟩^λ) {
  r3 := testSetLock r1                −− try to acquire the lock
  if r3 = 0 jump criticalRegion[λ]    −− when successful enter the critical region
  jump spin[λ] }                      −− otherwise try again
```

In the critical region, the threads increment the integer in the tuple.

```
criticalRegion ∀[λ](r1: ⟨λ⟩^λ, r2: ⟨int⟩^λ) requires (λ) {
  r3 := r2[1]     −− load the value of the tuple
  r3 := r3 + 1    −− increment it
  r2[1] := r3     −− and store the result
  unlock r1
  done
}
```

In code block sleep, we notice another well-known technique for acquiring a lock called *sleep-lock*. This strategy is cooperative towards other threads, because each time instruction **testSetLock** fails, the thread releases the processor and tries again later.

```
sleep ∀[λ] (r1: ⟨λ⟩^λ, r2: ⟨int⟩^λ) {
  r3 := testSetLock r1                −− try to acquire the lock
  if r3 = 0 jump criticalRegion[λ]    −− when successful enter the critical region
  fork sleep[λ]                       −− otherwise create a thread to try again
  done                                −− and terminates this thread
}
```

These two techniques have advantages over each other. A spin-lock is faster. A sleep-lock is fairest to other threads. When there is a reasonable expectation that the lock will be available (with exclusive access) in a short period of time it is more appropriate to use a spin-lock. The sleep-lock technique, however, does context switching, which is an expensive operation (*i.e.,* degrades performance).

We demonstrate a deadlock that arises from using spin-lock in machines with only one processor in the following example.

```
 start ∀[λ] (r1:⟨λ⟩^λ) requires (λ) {
   fork release[λ]     -- release takes the lock
   jump spinLock[λ]   -- spin-lock no longer holds λ
}
release ∀[λ] (r1:⟨λ⟩^λ) requires (λ) {
   unlock r1
   done
}
spinLock ∀[λ] (r1:⟨λ⟩^λ) {
   r2 := testSetLock r1
   if r2 = 0
     jump someComputation[λ] -- will never happen
   jump spinLock[λ]
}
```

Code block main cedes its permission over λ after forking code block release. Since there is only one processor and the spin-lock is not a cooperative algorithm, thread release is never executed, therefore leaving the system in a dead-lock. In case of a dual-core processor, the program terminates smoothly.

Continuation-passing style. *Continuations* [41] represent a state of control flow, a suspended computation. Continuations permit transfer of control, allowing programs to store, or activate, “the meaning of the rest of the program”. These suspended computations encompass a static part and a dynamic part, for example, a label to a code block and a snapshot of the processor’s registers. A MIL thread pool item is a representation of a continuation.

The continuation-passing style (CPS) [1] is a style of programming in which operations parametrise their return point through the use of continuations. This programming model is used as an intermediate language in various compilers for functional languages [1, 19, 31]. In MIL, a continuation consists of a label to a code block and the *environment*, which is the data that the continuation has direct access to (via registers). In the following examples we represent the environment as a single value of an abstract type. We show how to abstract the type of the environment with the universal quantification and with the existential quantification.

As an example using the universal type, we implement the addition and the multiplication of integers following the CPS. The continuation is composed by registers r_1 and r_4, representing the environment and the continuation label, respectively. The result of the addition is stored in register r_2. The abstracted environment remains untouched throughout computation.

```
add ∀[τ](r1 :τ,  r2 : int ,  r3 : int ,  r4 :( r1 :τ,r2 : int ))  {
  r2  :=  r3  + r2    −− perform computation
  jump r4             −− deliver the  result
}
mul ∀[τ](r1 :τ,  r2 : int ,  r3 : int ,  r4 :( r1 :τ,r2 : int ))  {
  r2  :=  r2  ∗  r3   −− perform computation
  jump r4             −− deliver the  result
}
```

The composition of two operations is exemplified by the implementation of expression $3 + 5 * 7$ as a CPS operation. The operation is divided into two steps. First, we perform the multiplication of $5 * 7$; second, we add the result to 3. Before jumping to the multiplication, sum creates a new continuation for operation mul that proceeds in sum_2 and accesses the continuation of sum.

```
sum ∀[τ](r1 :τ,  r4 :( r1 :τ,r2 : int ))  {
  r2  :=  new 2  −− create a new environment
  r2 [1]  :=  r1
  r2 [2]  :=  r4
  share r2 read-only
  r1  :=  r2        −− set the new environment
  r2  :=  5         −− the left  operand
  r3  :=  7         −− the right operand
  r4  :=  sum_2  −− the continuation label
  jump mul[⟨τ,(r1 :τ,r2 : int ) ⟩^ro ]  }
```

In the second step of the computation we calculate $3 + 35$; since value 35 is present in register r_2, we only need to move value 1 to register r_4. The fundamental part of the composition of CPS operations is passing the top-level continuation. The first two lines of sum_2 restore the top-level continuation passed by code block sum (through code block mul). The top-level continuation is then passed to add, which is activated upon performing the addition of integers.

```
sum_2 (r1 :⟨τ,( r1 :τ,r2 : int ) ⟩^ro ,  r2 : int )  {
  −− restore the continuation of  'sum'
  r4  :=  r1 [2]
  r1  :=  r1 [1]
  −− r2  holds 35
  r3  :=  3
  jump add[int]
}
```

A program using code block main is sketched as follows:

```
main ()  {
  r1  :=  0                 −− a bogus environment
  r4  :=  exit              −− the continuation label
  jump sum[int]
}
exit  ( r2 : int )  {
  external printInt r2 -- outputs 38 to the screen
  done
}
```

As an example of using CPS with existential types, we rewrite code block sum for using the existential quantifier over the environment, instead of the universal.

```
add (r1 : ∃ α.⟨(r1 : α, r1 : int), α⟩^ro, r2 : int, r3 : int) {
  r2 := r2 + r3
  τ, r1 := unpack r1    -- unpack the closure
  r3 := r1[1]           -- load the continuation
  r1 := r1[2]           -- load the environment
  jump r3               -- jump to the continuation
}
```

Passing a continuation of this form is simpler, since it is represented by a single value. The creation and activation (of continuations), however, entails packing and unpacking the tuple. We show the implementation of the expression $35 + 3$ using add.

```
main () {
  r1 := new 2
  r1[1] := 0     -- a bogus environment
  r1[2] := finish
  share r1 read-only
  r1 := pack int, r1 as ∃ α.⟨(r1 : α, r1 : int), α⟩^ro
  r2 := 35
  r3 := 3
  jump add
}
finish (r1 : int, r2 : int) {
  external printInt r2
}
```

3.5 Types against races

We split the results in three categories: the standard "well-typed machines do not get stuck", the lock discipline, and races. We omit the proofs for the following results since they are easily adapted from [47], which targets a previous, simpler version of MIL. The first result is divided into theorems of progress and of preservation.

Theorem 3.5.1 (Progress). *If* $\vdash S$*, then either* S *is* halt *or else* $S \rightarrow S'$*.*

Theorem 3.5.2 (Preservation). *If* $\vdash S$ *and* $S \rightarrow S'$*, then* $\vdash S'$*.*

The lock discipline is embodied in the following theorem (*cf.* Figure 3.2).

Theorem 3.5.3 (Lock discipline). *Let* $\Psi \vdash H$ *and* $\Psi \vdash \langle R; \Lambda; (\iota; \_)\rangle$*.*

1. *If* ι *is* $\lambda, \_ :=$ newLock*, then* $\lambda \notin \mathrm{dom}(\Psi)$*.*

2. *If ι is $\_ :=$ $\textsf{testSetLock}\, v$ and $H(\hat{\mathrm{R}}(v)) = \langle \_ \rangle^\lambda$, then $\lambda \notin \Lambda$.*

3. *If ι is $v[\_] := \_$ or $\_ := v[\_]$, and $H(\hat{\mathrm{R}}(v)) = \langle \_ \rangle^\lambda$, then $\lambda \in \Lambda$.*

4. *If ι is $\textsf{unlock}\, v$ and $H(\hat{\mathrm{R}}(v)) = \langle \_ \rangle^\lambda$, then $\lambda \in \Lambda$.*

5. *If ι is $\textsf{done}$, then $\Lambda = \emptyset$.*

For races we follow Flanagan and Abadi [14]. We start by defining the *set of permissions* of a machine state, by gathering the permissions of the running threads with those in the run pool, and with the set of unlocked locks in the heap. Remember that a permission is a set of locks, denoted by Λ.

Definition 3.5.4 (State permissions)**.**

$$\begin{aligned}
\mathcal{L}_P &= \{\Lambda \mid i \in [1..\mathsf{R}] \text{ and } P(i) = \langle \_; \Lambda; \_ \rangle\} \\
\mathcal{L}_T &= \{\Lambda \mid \langle l, \_ \rangle \in T \text{ and } H(l) = \forall[\_].(\_ \ \textsf{requires}\ \Lambda)\{\_\}\} \\
\mathcal{L}_H &= \{\{\lambda \mid l \in \mathrm{dom}(H) \text{ and } H(l) = \langle 0 \rangle^\lambda\}\} \\
\mathcal{L}_{\langle H;T;P\rangle} &= \mathcal{L}_P \cup \mathcal{L}_T \cup \mathcal{L}_H \\
\mathcal{L}_{\textsf{halt}} &= 2^{2^L}
\end{aligned}$$

We are interested only in *mutual exclusive states*, that is, states whose permissions do not “overlap.” Also, we say that a state has a *race condition* if it contains two processors trying to access the heap at the same shared location.

Definition 3.5.5. Mutual exclusive states. $\textsf{halt}$ is mutual exclusive; $S \neq \textsf{halt}$ is *mutual exclusive* when $i \neq j$ implies $\Lambda_i \cap \Lambda_j = \emptyset$, for all $\Lambda_i, \Lambda_j \in \mathcal{L}_S$.

Accessing the shared heap. A processor of the form $\langle R; \_; (\iota; \_) \rangle$ *accesses the shared heap H at location l*, if ι is of the form $v[\_] := \_$ or of the form $\_ := v[\_]$, $l = \hat{\mathrm{R}}(v)$, and $H(l) = \langle \_ \rangle^\lambda$, for some λ.

Race condition. A state S has a *race condition* if $S = \langle H; \_; P \rangle$ and there exist i and j distinct such that $P(i)$ and $P(j)$ both access the shared heap H at some location l.

Notice that the definition above allows two threads to access the heap at the same read-only location, since λ in $\langle \_ \rangle^\lambda$ denotes a lock (and not $\textsf{ro}$). We can show that typable mutual exclusive states do not have races.

Theorem 3.5.6. *If S is a mutual exclusive typable state, then S does not have a race condition.*

Also, typability and mutual exclusion are two properties of states preserved by reduction.

Theorem 3.5.7. *Let $S \to S'$. Then,*

1. *If $\vdash S$, then $\vdash S'$;*

2. *If S is mutual exclusive, then so is S'.*

The proof of each result is by a conventional case analysis on the reduction rules. For the second, we note that the rules that manipulate locks (R-FORK, R-NEWLOCK, R-TSL 0, and R-UNLOCK) all preserve the disjointedness of state permissions.

Corollary 3.5.8 (Types against races)**.** *If S is a mutual exclusive typable state and $S \to^* S'$, then S' does not have a race condition.*

Chapter 4

An Unbounded Buffer Monitor in MIL

The *monitor* presented in [20] is a module of a concurrent system with a specific interface, consisting of shared data and procedures. Monitors represent a shared resource protected by a form of mediator that synchronises concurrent accesses to the data. The only way threads access the shared data of a monitor is through the procedures of that module, thus providing information hiding for code. We say that a thread is *inside* or *within* a monitor if the thread is executing a monitor's procedure, and that a thread is *outside* a monitor if the thread is not executing code of a monitor.

Monitors provide two forms of synchronisation for threads accessing shared data, through *mutual exclusion* and through *cooperation*. The first form, mutual exclusion, is enforced by the policy of monitor access: there may exist at most one thread within a monitor. Concurrent calls to procedures of the same monitor are therefore serialised.

Condition variables represent the availability of a resource. Threads operate these variables from within a monitor: primitive *wait* for suspending a thread and primitive *signal* for resuming a thread. We say that condition variables are a cooperative synchronisation form, because scheduling is performed by the threads themselves with the common goal of optimising effective execution.

Figure 4.1 describes an implementation of an unbounded buffer monitor, inspired by the example of the bounded buffer monitor found in [20, p. 552]. The buffer is unbounded because there is no limit in the number of stored elements. We designate by *producers* the threads placing elements in the buffer and by *consumers* the threads removing elements from the buffer. The condition variable nonempty represents the existence of elements in the queue. Whenever the buffer is empty, as in its initial state, the consumer waits (suspends itself) until the producer has made the queue nonempty. The producer cooperates with the suspended consumer by signalling (issuing a signal on) the condition variable that represents the element availability, thereby reactivating the consumer.

There is an invariant that threads invoking a signal must abide, *i.e.,* a thread may only signal when the predicate that embodies the condition is true. Issuing a signal in the remove procedure would invalidate the invariant of the nonempty condition. A unbounded

```
unbounded buffer:monitor
  begin buffer:queue of Element;
        nonempty:condition;
    procedure append(x:Element);
      begin buffer.enqueue(x);
            nonempty.signal
      end append;
    procedure remove(result x:Element)
      begin if buffer.isEmpty then nonempty.wait;
            x := buffer.dequeue
      end remove;
    buffer := createQueue;
  end unbounded buffer;
```

Figure 4.1: An Hoare-style unbounded buffer monitor.

buffer monitor may only signal the condition nonempty when there is at least one element in the buffer queue, thus after enqueuing an element.

O-J. Dahl gives an overview of the different semantics of the signal primitive in various implementations of monitors [11]. Hoare's own proposal is referred as *signal and urgent wait*. The thread invoking a signal blocks while the thread that got reactivated finishes execution. By issuing a signal the thread blocks, yields exclusive control over the shared data temporarily, and then resumes execution, with exclusive access over the resource. If there are no threads waiting on a condition variable upon invoking a signal, the signaller thread proceeds execution.

Figure 4.2 illustrates the execution of two threads accessing an unbounded buffer as described by Hoare. Thread a tries to remove an element from an empty buffer at time t_1 and is put to wait. Thread b adds an element to the buffer at time t_2 and invokes a signal, getting held up. At the same time thread a resumes execution and terminates at time t_3. Thread b is reactivated at time t_3 and exits the remove procedure.

A possible implementation of this kind of monitor—that opts for the signal and urgent wait regime—uses two queues for storing suspended threads. One queue holds processes trying to enter the monitor to enforce mutual exclusion. Another queue stores threads suspended by wait. The scheduler places the continuation of a thread issuing a signal in the queue of threads trying to enter the monitor with the highest priority and executes the next thread held up in the wait queue.

Another regime O-J. Dahl characterises is *signal and exit*, in which the threads leave the monitor after issuing a signal, *i.e.,* the last instruction of a procedure. The critical region is taken away from the thread issuing a signal and given to a thread held up by the condition variable (exclusively).

Figure 4.3 illustrates the regime for signal and exit. The notable difference is that

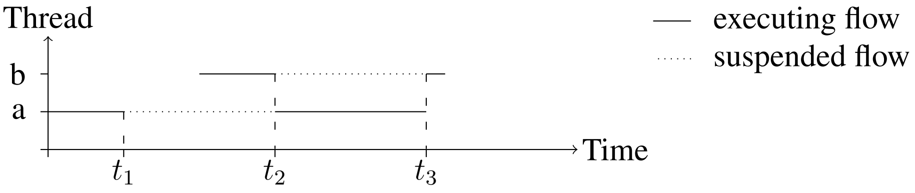

Figure 4.2: Execution view of two threads accessing the same monitor.

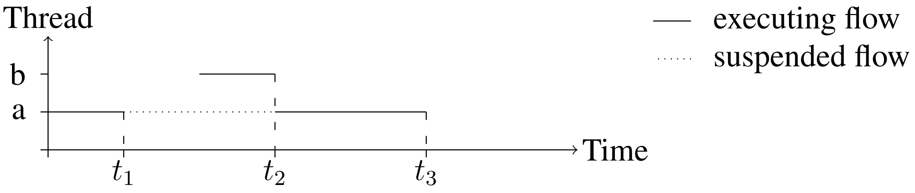

Figure 4.3: Execution view of two threads accessing the same monitor, following the signal and exit regime.

thread b terminates after issuing a signal at time t_2. Since signal is the last operation of the append procedure, the semantics of the monitor present in Figure 4.1 is the same with both regimes. The control flow of thread a remains the same as in Figure 4.2.

Our MIL version of the unbounded buffer monitor exports one code block for creating the monitor and two other code blocks that encode the procedures of the monitor. The signatures of the code blocks are listed bellow.

createBuffer $\forall$[Element] (r_1 :createContinuation(Element))
    append $\forall$[Element] (r_1 :BufferMonitor(Element), r_2 :Element)
    remove $\forall$[Element] (r_1 :BufferMonitor(Element), r_2 :removeContinuation(Element))

Element is the type of the elements in the monitor, and BufferMonitor is the type of the unbounded buffer monitor. Code blocks createBuffer and remove follow the continuation-passing style. Henceforth we follow these conventions: the continuation is a pair consisting of a code block and an environment; the type of the environment is abstracted (from the continuation) by the existential type, and the register file of the continuation expects the environment in register r_1. Let the type of the continuations of createBuffer and remove be:

$$\mathsf{createContinuation(Element)} \overset{\mathsf{def}}{=} \exists\alpha.\langle(\mathsf{r}_1 : \alpha,\ \mathsf{r}_2 : \mathsf{BufferMonitor(Element)}),\ \alpha\rangle^{\mathsf{ro}}$$
$$\mathsf{removeContinuation(Element)} \overset{\mathsf{def}}{=} \exists\alpha.\langle(\mathsf{r}_1 : \alpha,\ \mathsf{r}_2 : \mathsf{Element}),\ \alpha\rangle^{\mathsf{ro}}$$

The continuation of the creation operation expects the newly created monitor in register r_2. The continuation of the remove procedure expects the removed element in register r_2.

We illustrate the usage of the unbounded buffer monitors with a traditional producer/-consumer example. Code block main creates a monitor for a buffer of integers and starts three threads (code block launchThreads): a producer (code block producer) that repeatedly appends integers to the buffer, and two identical consumers defined by code block consumer. Repeatedly, each consumer removes and processes an integer from the buffer.

```
main() {
  -- create the base (empty) environment
  r2 := new 0
  share r2 read-only          -- $\langle\rangle^{ro}$
  -- create the continuation
  r1 := new 2
  r1 [1] := launchThreads     -- set the code block
  r1 [2] := r2                -- set the environment
  share r1 read-only          -- $\langle(r_1{:}\langle\rangle^{ro},\ r_2{:}\mathit{BufferMonitor(int)}),\ \langle\rangle^{ro}\rangle^{ro}$
  r1 := pack $\langle\rangle^{ro}$, r1 as $\exists\,\alpha.\langle(r_1{:}\alpha, r_2{:}\mathrm{BufferMonitor(int)}), \alpha\rangle^{ro}$
  jump createBuffer[int]      -- instantiate the type of the messages (int)
}
launchThreads(r1:$\langle\rangle^{ro}$, r2:BufferMonitor( int )) {
  -- the environment is the monitor
  r1 := r2
  fork producer
  fork consumer
  fork consumer
  done
}
consumer(r1:BufferMonitor(int)) {
  -- create the continuation
  r2 := new 2
  r2 [1] := consumeNext -- set the code block
  r2 [2] := r1          -- the environment is the monitor
  share r2 read-only    -- $\langle(r_1{:}\mathit{BufferMonitor(int)},\ r_2{:}\ \mathit{int}),\ \mathit{BufferMonitor(int)}\rangle^{ro}$
  r2 := pack BufferMonitor(int), r2 as $\exists\,\alpha.\langle(r_1{:}\alpha, r_2{:}\ \mathrm{int}), \alpha\rangle^{ro}$
  fork remove[int]
  done
}
consumeNext(r1:BufferMonitor(int), r2 : int ) {
  -- process the element in r2
  jump consumer
}
producer(r1:BufferMonitor( int )) {
  -- set the element
  r2 := 2               -- 2 as an example
  -- produce the element
  fork append[int]
  jump producer
}
```

The three operations (monitor creation, append, and remove) described above is all we need in order to compile the π-calculus as presented in Chapter 5. The rest of this chapter is organised in three parts. The first part introduces the operations related to the unbounded buffer; the second part presents the wait and signal operations; the last part shows polymorphic queues we use to implement the buffer and condition variables.

4.1 The monitor

Monitors provide a way to unify a synchronisation form (mutual exclusion), shared data, and the body of code that performs the accesses in a module. Monitors also provide condition variables. To encode these module in MIL we consider its three constituents: mutual exclusion, shared data, and code. MIL enforces race freedom by imposing mutual exclusion in the access of shared data. This form of synchronisation is also enforced by monitors. MIL establishes an association between data and locks, since every memory region (in the heap) is protected by a lock. The type of MIL code that accesses shared data is identifiable by requiring exclusive access to a lock. Our encoding of a monitor is delimited by the lock: all the data protected by lock λ and all the code that requires lock λ constitute a monitor, which we may identify by λ. In contrast to a code block that requires a specific lock λ, a code block that requires a lock of any (singleton) type is considered code that may be used by any monitor. Acquiring a lock through testSetLock corresponds to threads trying to enter in a monitor.

We represent the data of the unbounded buffer as a tuple that holds the attributes (fields) of the monitor present in Figure 4.1: the buffer, the nonempty condition variable, and the monitor's lock. Type BufferMonitor(Element) applies the existential quantification over the monitor's lock, thereby allowing the processor to manipulate the monitor without "knowing" its lock, hence $\lambda \notin \Psi$. We can insert the lock of the monitor in the scope of the processor by unpacking the monitor of type BufferMonitor(Element) (*cf.* rule T-UNPACK), which yields type UnpackedBufferMonitor(λ,Element). Code blocks of the monitor, requiring permission for the lock of the monitor, expect a monitor value to be unpacked. The references to the constituents of the monitor are immutable throughout the life-cycle of the monitor, thus we choose the tuple to be read-only. To enable introducing the lock of the monitor into the scope of a thread, we abstract the singleton lock type as an existential type.

$$\mathsf{BufferMonitor(Element)} \stackrel{\text{def}}{=} \exists\ \lambda.\mathsf{UnpackedBufferMonitor}(\lambda,\mathsf{Element})$$

$$\mathsf{UnpackedBufferMonitor}(\lambda,\mathsf{Element}) \stackrel{\text{def}}{=} \langle \mathsf{Queue}(\lambda,\mathsf{Element}),\mathsf{Condition}(\lambda),\langle\lambda\rangle^{\lambda}\rangle^{\mathsf{ro}}$$

$$\mathsf{Condition}(\lambda) \stackrel{\text{def}}{=} \mathsf{Queue}(\lambda,\mathsf{waitContinuation}(\lambda))$$

$$\mathsf{waitContinuation}(\lambda) \stackrel{\text{def}}{=} \exists\ \alpha.\langle(\mathsf{r}_1\!:\!\alpha)\ \textbf{requires}\ (\lambda),\alpha\rangle^{\mathsf{ro}}$$

Notice that a type Queue(λ,α) represents a queue storing elements of type α protected by lock λ. By choosing queues to the elements of the unbounded buffer, we impose a FIFO order upon the elements therein, guaranteeing that all requests will be processed. In our implementation the condition is technically a queue of continuations.

Creating unbounded buffers. For creating the monitor the thread needs to create a tuple and store therein two queues and a lock. This cannot be performed in one step (in a single code block), because the operation for creating queues expects a continuation. The operation for creating queues is divided into code blockscreateBuffer, createBufferCondition, and initBuffer.

The thread executing the first code block creates the monitor's lock and prepares the registers for creating a queue of elements: the environment in register r_2 and the continuation code block in register r_3.

```
createBuffer ∀[Element] (r1 :createContinuation(Element)) {
  λ, r3 := newLock -- create the lock of the monitor
  -- create the environment for createQueue
  r2 := new 2
  r2 [1] := r1          -- store the continuation of 'createBuffer'
  r2 [2] := r3          -- store the monitor's lock
  share r2 read-only
  r3 := createBufferCondition[λ,Element]
  jump createQueue[λ,Element,BufferEnv(λ,Element)]
}
```

The environment for code block createBufferCondition consists of a read-only tuple holding the continuation for code block createBuffer and the monitor's lock $\langle\lambda\rangle^{\lambda}$, respectively.

$$\mathsf{BufferEnv}(\lambda,\mathsf{Element}) \stackrel{\mathsf{def}}{=} \langle \mathsf{createContinuation}(\mathsf{Element}),\langle\lambda\rangle^{\lambda}\rangle^{\mathbf{ro}}$$

In the second code block (createBufferCondition), the thread prepares the registers for creating a queue of continuations, which represents the condition variable nonempty. Upon the creation of the queue, the thread continues executing in code block initBuffer .

```
createBufferCondition ∀[λ,Element] (r1 :Queue(λ,Element),
                                    r2 :BufferEnv(λ,Element)) {
  r3 := new 3                  -- allocate the new environment
  r3 [1] := r1                 -- store the buffer in the internal environment
  r1 := r2 [1]                 -- load the continuation of 'createBuffer'
  r3 [2] := r1                 -- store the continuation of 'createBuffer'
  r1 := r2 [2]                 -- load the monitor's lock
  r3 [3] := r1                 -- store the monitor's lock
  share r3 read-only
  r2 := r3
  r3 := initBuffer [λ,Element] -- set the code block of the continuation
  jump createQueue[λ,waitContinuation(λ),BufferEnv2(λ,Element)]
}
```

The environment for the continuation code block initBuffer is a tuple divided into a queue of elements, the continuation for createBuffer, and the monitor's lock.

$$\mathsf{BufferEnv2}(\lambda,\mathsf{Element}) \stackrel{\mathsf{def}}{=} \langle \mathsf{Queue}(\lambda,\mathsf{Element}),\mathsf{createContinuation}(\mathsf{Element}),\langle\lambda\rangle^{\lambda}\rangle^{\mathbf{ro}}$$

Code block initBuffer stands for the third and final stage of the monitor construction, where the thread allocates and initialises the tuple with the buffer queue, the condition, and the lock of the monitor. Afterwards, the thread unpacks and activates the continuation of code block createBuffer, which expects the unbounded buffer monitor in register r_2.

```
initBuffer ∀[λ,Element] (r1:Condition(λ),r2:BufferEnv2(λ,Element)) {
  r3 := new 3              −− allocate the monitor
  r3[2] := r1              −− store the condition variable 'nonempty'
  r1 := r2[1]              −− load 'buffer' from the internal environment
  r3[1] := r1              −− store 'buffer' in the monitor
  r1 := r2[3]              −− load the lock of the monitor
  r3[3] := r1              −− store the lock
  share r3 read-only       −− ⟨Queue(λ,Element),Condition(λ),⟨λ⟩^λ⟩^ro
  r4 := r2[2]              −− load the continuation of 'createBuffer'
  r2 := pack λ, r3 as BufferMonitor(Element) −− abstract the monitor's lock
  λ,r4 := unpack r4        −− unpack the continuation
  r1 := r4[2]              −− load the environment
  r4 := r4[1]              −− load the code block
  jump r4                  −− proceed
}
```

Appending elements. The monitor operation append places one element in the buffer and signals the nonempty condition variable. The operation accepts elements in register r_2 of type Element, and a monitor in register r_1. The thread starts by unpacking the monitor, thereby introducing the lock in the scope (*cf.* rule T-UNPACK). In code block appendAcquire, the thread spin-locks to acquire exclusive access to the monitor's lock, jumping to code block appendEnqueue on success. This pattern, which consists of the first three code blocks, is repeated for the remove procedure.

```
append ∀[Element] (r1:BufferMonitor(Element), r2:Element) {
  λ,r1 := unpack r1                         −− unpack the monitor's lock
  r3 := r1[3]                               −− load the lock
  jump appendAcquire[λ,Element]             −− try to acquire exclusive access
}
appendAcquire ∀[λ,Element] (r1:UnpackedBufferMonitor(λ,Element),
                                  r2:Element,r3:⟨λ⟩^λ) {
  r4 := testSetLock r3                      −− try to acquire the lock
  if r4 = 0 jump appendEnqueue[λ,Element]   −− lock acquired, continue
  jump appendAcquire[λ,Element]             −− otherwise, repeat
}
```

The two following code blocks implement the body of the append procedure (*vide* Figure 4.1). The thread enqueues the element in the buffer (code block appendAcquire) and sets as continuation code block appendSignal, where the thread issues a signal.

```
appendEnqueue ∀[λ,Element] (r1:UnpackedBufferMonitor(λ,Element),
                                  r2:Element) requires (λ) {
  r3 := new 2                          −− allocate the continuation
  r3[1] := appendSignal[λ,Element]     −− store the code block
  r3[2] := r1                          −− store the environment
  share r3 read-only
  r3 := pack UnpackedBufferMonitor(λ,Element),r3 as enqueueContinuation(λ)
  r1 := r1[1]                          −− load the buffer
  jump enqueue[λ,Element]
}
appendSignal ∀[λ,Element] (r1:UnpackedBufferMonitor(λ,Element)) requires (λ) {
  r2 := r1 [2]                         −− load the condition variable
  r1 := r1 [3]                         −− load the monitor's lock
  jump signal[λ]
}
```

Removing elements. Operation remove takes an element from the buffer of a monitor that is present in register r_1 and transfers it to the continuation in register r_2. In the same way as for appending elements in the unbounded buffer, the first two code blocks try to enter the monitor. Code block remove unpacks the packed monitor and places the lock of the monitor in scope. Then, the thread jumps to removeAcquire, where it tries to acquire the monitor's lock, and then tries to enter the monitor by continuing to code block testCondition.

```
remove ∀[Element] (r1:BufferMonitor(Element),r2:removeContinuation(Element)) {
  λ, r1 := unpack r1                 −− unpack the monitor's lock
  r3 := r1 [3]                       −− load the lock
  jump removeAcquire[λ,Element]      −− acquire the monitor's lock
}
removeAcquire ∀[λ,Element] (r1:UnpackedBufferMonitor(λ,Element),
                             r2 :removeContinuation(Element), r3:⟨λ⟩^λ) {
  r4 := testSetLock r3
  if r4 = 0 jump testCondition[λ,Element]
  jump removeAcquire[λ,Element]
}
```

Code block testCondition maps directly into the line of code

```
if buffer.isEmpty then nonempty.wait
```

present in procedure remove of Figure 4.1. Before checking if there are elements in the queue (instruction **if** r_5 = 0 **jump** wait[λ]) the thread primes the suspended thread as a continuation in case the queue is empty, thus preparing the registers for code block wait, which continues on code block dequeueWaitCont. If the test fails (*i.e.,* there are elements in the queue), then the thread restores the environment of the thread to prepare the registers for code block dequeueWaitCont.

```
testCondition ∀[λ,Element] (r1 :UnpackedBufferMonitor(λ,Element),
                            r2 :removeContinuation(Element),
                            r3 :⟨λ⟩^λ) requires (λ) {
  −− create an environment for the continuation
  r4 := new 2
  r4 [1] := r2   −− store the continuation of operation 'remove'
  r4 [2] := r1   −− store the unpacked monitor
  share r4 read-only
  −− r4: ⟨removeContinuation(Element),UnpackedBufferMonitor(λ,Element)⟩^ro
  r2 := r1 [2]   −− load the condition
  r5 := r1 [1]   −− load the buffer
  r5 := r5 [2]   −− load the buffer’s size
  r1 := r3       −− set the lock of the monitor for the wait operation
  −− create the continuation for wait
  r3 := new 2
  r3 [1] := dequeueWaitCont[λ,Element] −− set the continuation of wait
  r3 [2] := r4                         −− store the environment
  share r3 read-only
  −− r3: ⟨(r1 :RemvEnv(λ,Element)) requires (λ),RemvEnv(λ,Element)⟩^ro
  r3 := pack RemvEnv(λ,Element),r3 as waitContinuation(λ)
  if r5 = 0 jump wait[λ]               −− wait until an element arrives
  r1 := r4                             −− restore the environment
  jump dequeueWaitCont[λ,Element]      −− otherwise dequeue and activate cont.
}
```

The thread proceeds to code block dequeueWaitCont, which maps to code

```
x := buffer.dequeue
```

(the second line of procedure remove present in Figure 4.1). In MIL, the thread sets up the registers for code block dequeue, with the queue of elements in register r_1 and the continuation in register r_2. The continuation proceeds in code block activateWaitCont and holds an environment of type RemvEnv(λ,Element).

```
dequeueWaitCont ∀[λ,Element] (r1:RemvEnv(λ,Element)) requires (λ) {
  r2 := new 2
  r2 [1] := activateWaitCont[λ,Element]
  r2 [2] := r1
  share r2 read−only
  −− r2: ⟨(r1 :RemvEnv(λ,Element),r2:Element) requires(λ),RemvEnv(λ,Element)⟩^ro
  r2 := pack RemvEnv(λ,Element),r2 as dequeueContinuation(λ,Element)
  r1 := r1 [2]   −− load the monitor
  r1 := r1 [1]   −− load the buffer
  jump dequeue[λ,Element]
}
```

The type of the environment of code block dequeueWaitCont is a read-only tuple that is divided into the continuation of code block remove (in the first position) and the unpacked unbounded buffer (in the second position).

```
 RemvEnv(λ,Element) ≝ ⟨removeContinuation(Element),
                       UnpackedBufferMonitor(λ,Element)⟩^ro
```

After dequeuing the element from the buffer and placing it in register r_2, the thread running code block activateWaitCont exits the monitor, by releasing the lock of the monitor and by activating the continuation of code block remove.

```
activateWaitCont ∀[λ,Element] (r1 :RemvEnv(λ,Element),
                              r2 :Element) requires (λ) {
  r4 := r1 [2]   −− load the monitor
  r4 := r4 [3]   -- load the monitor's lock
  unlock r4      -- unlock the monitor's lock
  r3 := r1 [1]   -- load the continuation
  λ, r3 := unpack r3 -- unpack the continuation
  r1 := r3 [2]   -- load the environment
  r3 := r3 [1]   -- load the code fragment
  jump r3        -- continue
}
```

4.2 Wait and Signal

We represent a condition variable in MIL as a (initially empty) queue of continuations that are waiting on that condition. Manipulating the condition's queue is performed through the usual operations wait and signal. A wait operation is issued from inside a monitor and causes the calling thread to suspend itself until a signal operation occurs. Waiting on a condition (in register r_2) amounts to enqueuing the continuation of the wait operation (in register r_3) in the condition's queue. Notice that the lock (in register r_1) protecting the queue is the same lock used to enforce mutual exclusion access to the monitor operations. Also notice that the lock is released after enqueuing the continuation, allowing other threads to use the monitor, and that the thread terminates (*vide* code block release).

```
wait ∀[λ] (r1 :⟨λ⟩^λ, r2 :Condition(λ), r3 :waitContinuation(λ)) requires (λ) {
  -- the continuation is released
  r4 := new 2
  r4 [1] := release[λ]
  r4 [2] := r1
  share r4 read-only -- ⟨(r1:⟨λ⟩^λ) requires (λ),⟨λ⟩^λ⟩^ro
  r4 := pack ⟨λ⟩^λ,r4 as waitContinuation(λ)
  r1 := r2    -- set the queue
  r2 := r3    -- set the element to enqueue
  r3 := r4    -- set the continuation
  jump enqueue[λ,waitContinuation(λ)]
}
release ∀[λ] (r1 :⟨λ⟩^λ) requires (λ) {
  unlock r1  -- release the monitor's lock
  done       -- terminate the thread
}
```

The type of the continuation for code block wait is a read-only pair holding a code block and an environment, as usual. The code block of the continuation requires exclusive access to the lock of the monitor, since the blocked thread that is embodied by this continuation is inside the monitor (in the critical region). The register file of the continuation's code block expects the environment in register r_1.

$$\mathsf{waitContinuation}(\lambda) \overset{\text{def}}{=} \exists\, \alpha.\langle (\mathsf{r}_1 : \alpha)\ \textbf{requires}\ (\lambda), \alpha\rangle^{\textbf{ro}}$$

We opt for signal and exit regime. There is an implicit notion of an uninterrupted transfer of ownership of the monitor's lock that goes from the signalling thread that finishes, to the thread waiting on a condition that resumes execution, which fits nicely in MIL's lock discipline. Execution also shifts from the signal operation to a thread waiting on the target condition variable (represented by a continuation). The transference of lock permission and execution is carried out by jumping to the code block of the continuation without releasing the lock of the monitor. By enforcing that no other instruction follows a signal primitive, we reuse the processor of a signalling thread to execute a delayed thread, both underlining this idea of transfer of control and simplifying the implementation.

O-J. Dahl shows that, according to the criteria chosen in [11], the regime we choose is a (programming) restriction of the signal and urgent wait. The motivation for our choice is twofold. First, the programming restriction does not affect the implementation of the unbounded buffer. Second, the implementation in MIL becomes simpler, since we do not need to worry about the continuation of operation signal.

The signal code block first checks if there are no delayed threads to signal, in which case terminates. Otherwise, the thread dequeues a suspended thread and proceeds to code block signalDequeue.

```
signal ∀[λ] (r1 :⟨λ⟩^λ, r2 :Condition(λ)) requires (λ) {
  r3 := r2[2]                  −− load the length of the queue
  if r3 = 0 jump release[λ]    −− no continuations to signal, finish
  r1 := r2                     −− set the queue of continuations
  r2 := new 2
  r2[1] := signalDequeue[λ]
  r2[2] := 0                   −− an empty environment
  share r2 read-only
  −− r2: ⟨(r1 : int, r2 :waitContinuation(λ)) requires(λ), int⟩
  r2 := pack int, r2 as dequeueContinuation(λ,waitContinuation(λ))
  jump dequeue[λ,waitContinuation(λ)]
}
```

In code block signalDequeue, the thread activates the continuation of code block wait.

```
signalDequeue ∀[λ] (r1: int, r2 :waitContinuation(λ)) requires (λ) {
  λ, r2 := unpack r2           −− unpack the continuation
  r3 := r2[1]                  −− the code fragment
  r1 := r2[2]                  −− the environment
  jump r3                      −− proceed
}
```

4.3 Polymorphic Queues

Queues are the foundational data structure for the implementation of monitors, used for holding elements in the unbounded buffer and for serving as condition variables. A simple

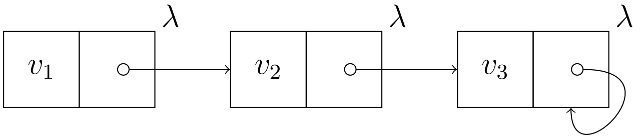

Figure 4.4: Three nodes connected linearly. Each node, guarded by the same lock λ, holds a value v_i.

representation for queues is using a linked-list divided into nodes that are interconnected linearly, as depicted by Figure 4.4. Each node holds a value and a reference to the next element in the queue.

$\mathsf{Node}(\lambda,\alpha) \overset{\mathsf{def}}{=} \langle\alpha,\mathsf{Node}(\lambda,\alpha)\rangle^{\lambda}$

Consider that a queue is a tuple holding a reference for the first node of the queue. How could we represent an empty queue? We would need a witness value for every type, which is impossible if we want a general propose implementation of queues whose values may be of any type. For typing proposes we design queues as an encoding of simple objects, using the existential type. Pierce and Turner proposed Object-Oriented Programming (OOP) using the existential type [37]. We follow Morrisett's encoding of simple objects in typed assembly languages [29]. Encoding queues as simple objects provides encapsulation of the representation, which in turn enables constructing queues without access to a witness value. In our implementation, a queue has two stages in its life-cycle. An initial stage S1 when it is created. A secondary stage S2 after the first value is enqueued, in which the queue is represented by a linked-list.

A queue is a shared tuple divided into its *representation* and its length (the number of values enqueued).

$\mathsf{Queue}(\lambda,\alpha) \overset{\mathsf{def}}{=} \mu\,\mathsf{Q}.\langle\mathsf{QueueRep}(\mathsf{Q},\lambda,\alpha),\mathsf{int}\rangle^{\lambda}$

The representation of a queue is an existential value that encapsulates the implementation details of a given stage.

$\mathsf{QueueRep}(\mathsf{Q},\lambda,\alpha) \overset{\mathsf{def}}{=} \exists\mathsf{S}.\langle\mathsf{S},\mathsf{enqueueHandler}(\mathsf{Q},\lambda,\mathsf{S},\alpha),\mathsf{dequeueHandler}(\mathsf{Q},\lambda,\mathsf{S},\alpha)\rangle^{\mathbf{ro}}$

An implementation of a queue's stage consists of the state of the queue S and two code blocks (which we designate *handlers*), one for enqueuing, and another one for dequeuing. Each of the handlers is specialised for state S.

The operation to enqueue elements in values of type $\mathsf{Queue}(\lambda,\alpha)$ is listed below. The typechecking rule T-UNPACK ensures that the type S of the state can be used abstractly in the remaining of the code block. This way the state can only be passed to the enclosed code blocks (the handlers). The enqueue operation unpacks the representation of the queue and passes the state to the respective handling code block (present in the second position of the tuple).

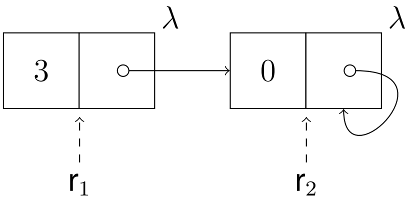

Figure 4.5: One node connected to another node.

```
enqueue ∀[λ,α] (r1:Queue(λ,α), r2:α, r3:enqueueContinuation(λ)) requires (λ) {
  r4 := r1[1]          −− load the representation of the stage of the queue
  S, r4 := unpack r4   −− unpack the representation
  r5 := r4[2]          −− load the handler for enqueue
  r4 := r4[1]          −− load the state of the queue
  jump r5              −− proceed in the handler for enqueue
}
```

$$\text{enqueueContinuation}(\lambda) \overset{\text{def}}{=} \exists\alpha.\langle(\text{r}_1{:}\alpha)\ \textbf{requires}\ (\lambda),\alpha\rangle^{\textbf{ro}}$$

We omit the operation to dequeue elements from values of type Queue(λ,α), because the only difference is loading the code block handling dequeue that is held in third position of the stage tuple, instead of the enqueue handler (in the second position).

On stage S2 the queue is represented by a linked-list that is composed by *nodes*, as portrayed by Figure 4.4. Each node pairs a value picked from α with a node. The recursive type follows.

$$\text{Node}(\lambda,\alpha) \overset{\text{def}}{=} \langle\alpha,\text{Node}(\lambda,\alpha)\rangle^{\lambda}$$

We show an example of the creation of two nodes that is illustrated by Figure 4.5. Consider that the processor holds lock λ.

```
r1 := new 2 −− allocate node 1
share r1 guarded by λ
r2 := new 2 −− allocate node 2
share r2 guarded by λ
r1[1] := 3   −− set the value of node 1 as 3
r1[2] := r2  −− link node 1 to node 2
r2[1] := 0   −− set the value of node 2 as 0
r2[2] := r2  −− link node 2 to itself
```

The node holding 3 is pointed by register r_1. The node holding 0 is pointed by register r_2. The first node is connected to the second, which is connected to itself.

Linked-lists consist of two entry points for accessing a series of linearly interconnected nodes, the first and last nodes of the list. The type of linked-lists

$$\text{LinkedList}(\lambda,\alpha) \overset{\text{def}}{=} \langle\text{Node}(\lambda,\alpha),\text{Node}(\lambda,\alpha)\rangle^{\lambda}$$

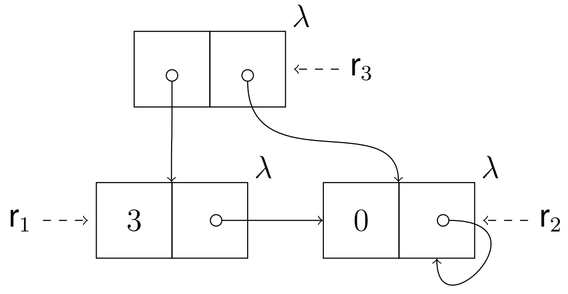

Figure 4.6: A linked-list with one node and one sentinel.

corresponds to a pair of nodes protected by lock λ. The last node is a sentinel, that allows for algorithmic simplifications. We choose double-ended lists because they enable fast adds and fast removes.

Consider the nodes of the previous example, referred by registers r_1 and r_2. The following example, illustrated by Figure 4.6, shows the creation of a linked-list, holding the number 3:

```
r3 := new 2 −− allocate the list
r3 [1] := r1  −− store to the first node (the head of the list )
r3 [2] := r2  −− store to the sentinel (the tail of the list )
share r3 guarded by λ
```

Code blocks handling the enqueue operation for a specific stage are of the abstract type

$$\mathsf{enqueueHandler}(\mathsf{Q},\lambda,\mathsf{S},\alpha) \overset{\mathsf{def}}{=} (\mathsf{r}_1{:}\mathsf{Q},\ \mathsf{r}_2{:}\alpha,\ \mathsf{r}_3{:}\mathsf{enqueueContinuation}(\lambda),\ \mathsf{r}_4{:}\mathsf{S})\ \textbf{requires}\ (\lambda)$$

where the continuation is of type

$$\mathsf{enqueueContinuation}(\lambda) \overset{\mathsf{def}}{=} \exists\ \alpha.\langle(\mathsf{r}_1{:}\alpha)\ \textbf{requires}\ (\lambda),\alpha\rangle^{\textbf{ro}}$$

The handler expects a queue in register r_1, the element to enqueue in register r_2, a continuation in register r_3, the state of the handler in register r_4, and the lock λ held.

The enqueue handler for stage S2 is listed below. Notice that the type of the state for this stage is LinkedList(λ,α), present in register r_4. Figure 4.7 illustrates what happens to the list when a value is enqueued, the expected algorithm for enqueuing in linked-lists. First, the thread creates and initialises a sentinel. Then, the thread stores the new value in the last node and connects that node to the sentinel. Lastly, the thread stores the sentinel node in the linked-list and jumps to code block enqueueFinish.

```
enqueueNormal ∀[λ,α] (r1:Queue(λ,α), r2:α, r3:enqueueContinuation(λ),
                      r4:LinkedList(λ,α)) requires (λ) {
  r6 := new 2 −− create the new sentinel
  share r6 guarded by λ
```

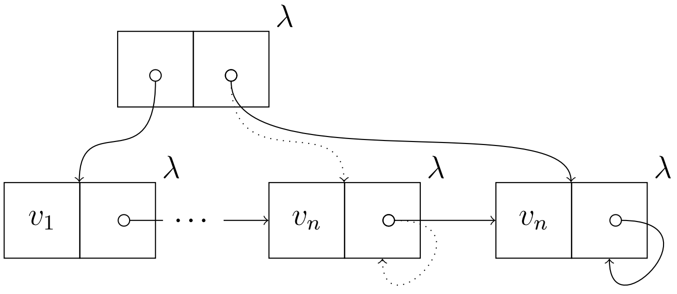

Figure 4.7: Enqueuing the n-th value to a linked-list.

```
  r6 [1] := r2    −− copy the value to the sentinel as well
  r6 [2] := r6    −− link new node to itself
  r5 := r4 [2]    −− the sentinel will become the last valid node
  r5 [1] := r2    −− set the value of the last node
  r5 [2] := r6    −− point to the sentinel
  r4 [2] := r6    −− store the new sentinel
  jump enqueueFinish[λ,α]
}
```

In the following code block, the thread increments the count of elements in the queue and activates the continuation in r_3.

```
enqueueFinish ∀[λ,α] (r1:Queue(λ,α), r3:enqueueContinuation(λ)) requires (λ) {
  r2 := r1 [2]
  r2 := r2 + 1           −− increment the count of elements
  r1 [2] := r2
  α,r2 := unpack r3      −− unpack the continuation
  r1 := r2 [2]           −− load the environment
  r2 := r2 [1]           −− load the code block
  jump r2                −− proceed
}
```

Code blocks are of type dequeueHandler(Q,λ,S,α) for handling the dequeue operation expect a queue in register r_1, a continuation in register r_2, and the state of their representation in register r_3.

$$\text{dequeueHandler(Q,}\lambda\text{,S,}\alpha\text{)} \overset{\text{def}}{=} (\mathsf{r}_1\text{:Q, } \mathsf{r}_2\text{:dequeueContinuation(}\lambda,\alpha\text{), } \mathsf{r}_3\text{:S) } \textbf{requires } (\lambda)$$

The continuation of the dequeue code block is of type dequeueContinuation(λ,α), requires exclusive access to lock λ, and expects the dequeued element in register r_2.

$$\text{dequeueContinuation(}\lambda,\alpha\text{)} \overset{\text{def}}{=} \exists\, \alpha_e.\langle(\mathsf{r}_1{:}\alpha_e,\mathsf{r}_2{:}\alpha)\ \textbf{requires}(\lambda),\alpha_e\rangle^{\textbf{ro}}$$

The dequeue handler for stage S2 performs the usual algorithm for dequeuing in linked-lists, as illustrated by Figure 4.8: simply sets the head of the linked-list to the second node. Afterwards, the thread executing the handler decrements the elements count and proceeds with the continuation.

```
dequeueNormal ∀[λ,α] (r1:Queue(λ,α), r2:dequeueContinuation(λ,α),
                              r3 :LinkedList(λ,α)) requires (λ) {
  r4 := r1 [2]
  r4 := r4 − 1          −− decrement the count of nodes
  r1 [2] := r4
  α,r4 := unpack r2 −− unpack the continuation
  r5 := r3 [1]          −− load the first node
  r2 := r5 [1]          −− load the value to be passed to the continuation
  r5 := r5 [2]          −− load the second node
  r3 [1] := r5          −− point the head of the queue to the second node
  r1 := r4 [2]          −− load the environment
  r4 := r4 [1]          −− load the code block
  jump r4               −− jump to the continuation
}
```

We are left with the representation for stage S1. Our idea is to delay the creation of the linked-list until the first element is enqueued. The enqueue handler for this stage installs the representation for stage S2. The thread executing this handler creates the linked-list that denotes the new state, prepares the tuple for the new representation, and jumps to enqueueFinish, incrementing the count of elements and activates the continuation.

```
enqueueInitial ∀[λ,α] (r1 :Queue(λ,α), r2:α,
                             r3 :enqueueContinuation(λ), r4:int) requires (λ) {
  r6 := new 2 −− allocate the sentinel
  share r6 guarded by λ
  r6 [1] := r2   −− set the (arbitrary) value for the sentinel
  r6 [2] := r6   −− link to itself
  r5 := new 2 −− allocate the first node
  r5 [1] := r2   −− store the value
  r5 [2] := r6   −− link to the sentinel
  share r5 guarded by λ
  r4 := new 2 −− allocate the linked− list
  r4 [1] := r5   −− store the first node
  r4 [2] := r6   −− store the sentinel
  share r4 guarded by λ
  r2 := new 3
  r2 [1] := r4   −− store the implementation (linked−list)
  r2 [2] := enqueueNormal[λ,α]  −− store the enqueue method
  r2 [3] := dequeueNormal[λ,α]  −− store the dequeue method
  share r2 read-only
  r1 [1] := pack LinkedList(λ,α), r2 as QueueRep(Queue(λ,α),λ,α)
  jump enqueueFinish[λ,α]
}
```

The dequeue method of the initial stage is undefined and implemented as an endless loop, because the invariant of "no value is dequeued from an empty queue" is preserved throughout the supporting code.

```
dequeueInitial ∀[λ,α] (r1 :Queue(λ,α),
                            r2 :dequeueContinuation(λ,α), r3:int) requires (λ) {
```

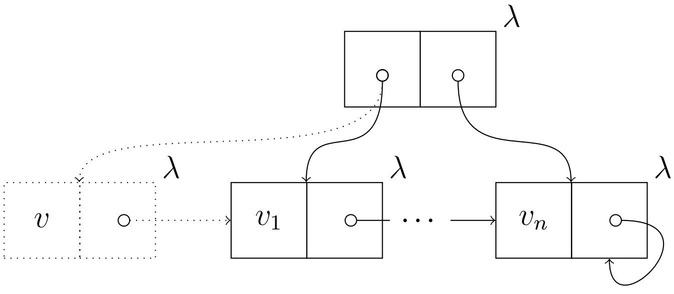

Figure 4.8: Dequeuing the first node of a linked-list.

```
  jump dequeueInitial[λ,α]
}
```

The creation of a queue prepares stage S1, delaying the creation of a linked-list until the first value is enqueued. This stage has no state, thus we represent it with an arbitrary integer, *e.g.*, number 0. The operation assembles the initial representation for this stage and initialises the count of elements in the queue to zero.

```
createQueue ∀[λ,α,αe] (r2:αe, r3 :( r1 :Queue(λ,α),r2:αe)) {
  r4 := new 3
  r4 [1] := 0                        −− store the state
  r4 [2] := enqueueInitial [λ,α]    −− store a label to the enqueue method
  r4 [3] := dequeueInitial [λ,α]    −− store a label to the dequeue method
  share r4 read-only
  r4 := pack int, r4 as QueueRep(Queue(λ,α),λ,α)
  r1 := new 2                        −− create the queue
  r1 [1] := r4                       −− store the implementation
  r1 [2] := 0                        −− the queue is empty
  share r1 guarded by λ
  jump r3                            −− jump to the continuation
}
```

```
 QueueCreateCont(λ,α,αe) ≝ (r1:Queue(λ,α), r2:αe)
```

4.4 Discussion

We are able to divide the code into three separated modules: unbounded buffer monitor, condition variables, and queues. The queues module, for example, may be used without requiring the other modules. The source code consists of 20 type declarations and 24 code blocks, totalling 259 lines of code.

Turner [45] uses a single queue to hold both the messages and the input processes waiting for messages, taking advantage of an invariant by which queues never contain both messages and input processes simultaneously. Our initial attempt for the supporting code followed a Turner-like version. The data structure implementing π-channels contained addresses of two different queues (one for output messages and another for input processes). The version we present in this thesis is the monitor-based implementation, discussed in the current chapter, also uses two queues: one to hold the messages in the monitor's buffer, the other to implement the condition variable, which, remarkably are instances of the same abstract data type: Queue(λ,Element) and Queue(λ,WaitContinuation(λ)).

An improvement over our previous implementation is the smaller code size and the clarity of the code. The monitor-based version has less 60 lines of code than the Turner-like version, a decrement of about 20%. One reason for the decrease of the code size in the monitor-based version is that we were able pull the process replication out of the supporting code and into the translation.

Chapter 5

Compiling π into MIL

This chapter presents a translation of the simply typed pi-calculus extended with integer values (described in Chapter 2) into the multithreaded intermediate language (described in Chapter 3). The translation is extremely simplified by using the unbounded buffer monitor (described in Chapter 4) to manage channels. We first present the translation, then the main result of the translation—type preservation—, and finally we discuss the choices made.

5.1 The translation function

Translation functions map terms of the source language into terms of the target language. These functions are typically recursive, because they are defined concerning the grammar of the source language, which is also usually recursive. To generate MIL code from the π-calculus we must produce a program that simulates the source π-process.

The translation from the π-calculus into MIL comprises the translation $\mathcal{T}[\![\cdot]\!]$ of types, $\mathcal{V}^{\vec{x}}[\![\cdot]\!]$ of values, and $\mathcal{P}[\![\cdot]\!]$ of programs (closed π-processes). Our main result is a type-preserving [31] translation, meaning that we assert that the type information in the source language is preserved and that the generated programs will not get stuck. We use the unbounded buffer monitor to act a channel, hence the type of a channel is translated into the type of an unbounded buffer. The translation of values includes memory allocation and register manipulation. The translation of processes entails dynamic creation of threads and communication through shared memory.

Types of the π-calculus have a direct representation in the supporting library, thus the translation is straightforward.

$$\begin{aligned}\mathcal{T}[\![\mathsf{int}]\!] &\stackrel{\text{def}}{=} \mathsf{int}\\ \mathcal{T}[\![\hat{}[\vec{T}]]\!] &\stackrel{\text{def}}{=} \mathsf{BufferMonitor}(\langle\mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}})\end{aligned}$$

We write $\mathcal{T}[\![T_1 \ldots T_n]\!]$, or $\mathcal{T}[\![\vec{T}]\!]$, for the sequence of types $\mathcal{T}[\![T_1]\!], \ldots, \mathcal{T}[\![T_n]\!]$. The integer type of the π-calculus is translated in the corresponding type of MIL. A π-channel is translated into an unbounded buffer monitor whose elements are read-only tuples of values: integer values or unbounded buffers (channels). As an example, consider the translation of type $\hat{}[\mathsf{int}]$:

$$\begin{aligned}\mathcal{T}[\![\hat{}[\mathsf{int}]]\!] &= \mathsf{BufferMonitor}(\langle \mathcal{T}[\![\mathsf{int}]\!]\rangle^{\mathsf{ro}})\\ &= \mathsf{BufferMonitor}(\langle \mathsf{int}\rangle^{\mathsf{ro}})\end{aligned}$$

The translation $\mathcal{V}^{\vec{x}}[\![\cdot]\!]$ of values loads into register r_3 a value from the *environment* $\vec{x}$ (addressed by register r_1), or moves into the same register an integer literal. The environment $\vec{x}$ is a sequence of names $x_1 \dots x_n$. The base empty environment is $\emptyset$. By augmenting environment $\vec{x}$ with name y we get environment $\vec{x}y$. By concatenating environment $\vec{x}$ with environment $\vec{y}$ we get environment $\vec{x}\vec{y}$, or $x_1 \dots x_n y_1 \dots x_m$.

$$\mathcal{V}^{\vec{x}}[\![v]\!] \stackrel{\text{def}}{=} \begin{cases} \mathsf{r}_3 := \mathsf{r}_1[i] & \text{if } v \in \vec{x} \\ \mathsf{r}_3 := v & \text{otherwise} \end{cases}$$

The translation of a π-program $\mathcal{P}[\![P]\!]$ yields a heap, containing several code blocks, among which we find main.

$$\begin{aligned}\mathcal{P}[\![P]\!] &\stackrel{\text{def}}{=} \mathsf{main}()\{\mathcal{E}^{\emptyset}(\emptyset,\emptyset); I\} \uplus H\\ &\quad \text{where } (H, I) = \mathcal{P}^{\emptyset,\emptyset}[\![P]\!]\end{aligned}$$

Operator $\uplus$ is the disjoint union of sets. Block main prepares an empty environment, $\mathcal{E}^{\emptyset}(\emptyset,\emptyset)$, for the top level process, which is then translated by $\mathcal{P}^{\emptyset,\emptyset}[\![P]\!]$. In all cases register r_1 contains the current environment, the address of a read-only tuple containing the free names in the process. Instructions therefore expect a register file in which register r_1 has type $\langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}$, where $\vec{x}$ is the environment of process P.

Function $\mathcal{E}^{\Gamma}(\vec{x},\vec{y})$ generates an instruction sequence that creates a new environment as a copy of the current environment $\vec{x}$ (in register r_1 of type $\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}$ where $\vec{T}$ are the types of $\vec{x}$) extended with environment $\vec{y}$ in register r_2 (of type $\langle \mathcal{T}[\![\vec{T'}]\!]\rangle^{\mathsf{ro}}$), leaving the newly created environment in register r_1. By $\Gamma(v)$ we mean T where $\Gamma \vdash v \colon T$ (*cf.* Figure 2.6); in other words, T when $v \colon T \in \Gamma$, or int if v is an integer literal.

$$\begin{aligned}\mathcal{E}^{\Gamma}(\vec{x},\vec{y}) \stackrel{\text{def}}{=} (&\mathsf{r}_3 := \mathsf{new}\ |\vec{x}\vec{y}|\\ &\forall\, i \in \{1,\dots,|\vec{x}|\} \begin{cases}\mathsf{r}_4 := \mathsf{r}_1[i]\\ \mathsf{r}_3[i] := \mathsf{r}_4\end{cases}\\ &\forall\, i \in \{1,\dots,|\vec{y}|\} \begin{cases}\mathsf{r}_4 := \mathsf{r}_2[i]\\ \mathsf{r}_3[i + |\vec{x}|] := \mathsf{r}_4\end{cases}\\ &\mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only} \qquad \text{---}\langle \mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle^{\mathsf{ro}}\\ &\mathsf{r}_1 := \mathsf{r}_3)\end{aligned}$$

First, a new environment is allocated, as a local tuple, and filled with the elements from environments $\vec{x}$ and $\vec{y}$. Next, the tuple is made shared for reading, allowing multiple threads to access the environment without contention. Lastly, the address of the newly created environment is copied to register r_1, as required by the continuation code. For example, the instructions generated by $\mathcal{E}^{\Gamma}(printInt; echo, msg; reply)$ are:

```
r3 := new 3        -- |printInt ;echo;msg;reply|
r4 := r1 [1]       -- i = 1
r3 [1] := r4
r4 := r1 [2]       -- i = 2
r3 [2] := r4
r4 := r2 [1]       -- i = 1
r3 [3] := r4
r4 := r2 [2]       -- i = 2
r3 [4] := r4
share r3 readonly
r1 := r3
```

A process P is translated by function $\mathcal{P}^{\vec{x},\Gamma}[\![P]\!]$, parametric on a sequence of names $\vec{x}$ and on a π-calculus typing environment Γ, where $\Gamma \vdash P$ and $\mathrm{fn}P \subseteq \{\vec{x}\}$. The result of the translation is a pair composed by a heap H and a sequence of instructions I. This function is defined by cases.

The translation of the inactive process is direct: the thread is terminated and an empty heap produced.

$$\mathcal{P}^{\vec{x},\Gamma}[\![\mathbf{0}]\!] \stackrel{\text{def}}{=} (\emptyset, \mathsf{done})$$

For example, the translation of the program represented by inactive process $\mathcal{P}[\![\mathbf{0}]\!]$ generates the following MIL code.

```
main () {
  -- $\mathcal{E}^{\Gamma}(\emptyset, \emptyset)$
  r3 := new 0 -- |$\emptyset$|
  share r3 read-only
  r1 := r3
  -- $\mathcal{P}^{\emptyset,\emptyset}[\![\mathbf{0}]\!]$
  done
}
```

For the translation of the output process, the registers are laid out as expected by code block append in the monitor of Chapter 4: register r_1 contains the (address of) channel x_i (that is, the monitor), and register r_2 contains (the address of) the tuple with values $\vec{v}$ (that is, the element to append to the buffer in the monitor). The control is then transferred to code block append.

$$\begin{aligned}
\mathcal{P}^{\vec{x},\Gamma}[\![\overline{x_i}\langle\vec{v}\rangle]\!] &\stackrel{\text{def}}{=} (\emptyset, I) \text{ where} \\
I &= (\mathsf{r}_2 := \textsf{new } |\vec{v}| \\
&\quad \forall\, j \in \{1, \ldots, |\vec{v}|\} \begin{cases} \mathcal{V}^{\vec{x}}[\![v_j]\!] \\ \mathsf{r}_2[j] := \mathsf{r}_3 \end{cases} \\
&\quad \textsf{share } \mathsf{r}_2 \textsf{ read-only } {-}{-}\langle\mathcal{T}[\![\hat{\Gamma}(\vec{v})]\!]\rangle^{\mathsf{ro}} \\
&\quad \mathsf{r}_1 := \mathsf{r}_1[i] \\
&\quad \textsf{jump append}[\langle\mathcal{T}[\![\hat{\Gamma}(\vec{v})]\!]\rangle^{\mathsf{ro}}]
\end{aligned}$$

Definition 5.1.1. An evaluation type function $\hat{\Gamma}(v)$ is defined for well-typed values as

$$\hat{\Gamma}(v) = \begin{cases} \Gamma(v) & v \in \mathrm{dom}(\Gamma) \\ \mathsf{int} & \text{otherwise} \end{cases}$$

We use the notation $\hat{\Gamma}(v_1 \ldots v_n)$ for the sequence $\hat{\Gamma}(v_1), \ldots, \hat{\Gamma}(v_n)$.

Let the π-calculus typing

$$\Gamma = \emptyset, \mathit{printInt}\colon \hat{}[\mathsf{int}], \mathit{echo}\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]], \mathit{msg}\colon \mathsf{int}, \mathit{reply}\colon \hat{}[\mathsf{int}]$$

and let the environment $\vec{x} = \mathit{printInt}; \mathit{echo}; \mathit{msg}; \mathit{reply}$. Consider the translation of the output process $\mathcal{P}^{\vec{x},\Gamma}[\![\overline{\mathit{reply}}\langle\mathit{msg}\rangle]\!]$. The generated instruction sequence is:

```
  -- ...
  r2 := new 1
  r3 := r1[3]          -- $\mathcal{V}^{\vec{x}}[\![msg]\!]$
  r2[1] := r3
  share r2 read-only  -- $\langle int\rangle^{ro}$
  r1 := r1[4]          -- load 'reply'
  jump append[$\langle$int$\rangle^{ro}$]
}
```

The generated instruction sequence expects a register r_1 of type $\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}$. The translation of the environment's type is:

$$\begin{aligned}
\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}} &= \langle\mathcal{T}[\![\Gamma(\mathit{echo})]\!], \mathcal{T}[\![\Gamma(\mathit{msg})]\!], \mathcal{T}[\![\Gamma(\mathit{reply})]\!]\rangle^{\mathsf{ro}} \\
&= \langle\mathcal{T}[\![\hat{}[\mathsf{int}, \hat{}[\mathsf{int}]]]\!], \mathcal{T}[\![\mathsf{int}]\!], \mathcal{T}[\![\hat{}[\mathsf{int}]]\!]\rangle^{\mathsf{ro}} \\
&= \langle\mathsf{BufferMonitor}(\langle\mathsf{int}, \mathsf{BufferMonitor}(\langle\mathsf{int}\rangle^{\mathsf{ro}})\rangle^{\mathsf{ro}}), \\
&\qquad \mathsf{int}, \mathsf{BufferMonitor}(\langle\mathsf{int}\rangle^{\mathsf{ro}})\rangle^{\mathsf{ro}}
\end{aligned}$$

The translation of an input process is as follows.

$$\begin{aligned}
\mathcal{P}^{\vec{x},\Gamma}[\![x_i(\vec{y}).P]\!] &\stackrel{\text{def}}{=} (H \uplus H', \textsf{jump } l) \text{ where} \\
\hat{}[\vec{T}] &= \Gamma(x_i) \\
\Gamma' &= \Gamma, \vec{y}\colon \vec{T} \\
(H', I') &= \mathcal{P}^{\vec{x}\vec{y},\Gamma'}[\![P]\!]
\end{aligned}$$

```
$H = l\ (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}})$ {
        $\mathsf{r}_2 := \mathsf{new}\ 2$
        $\mathsf{r}_2[1] := l'$
        $\mathsf{r}_2[2] := \mathsf{r}_1$
        share $\mathsf{r}_2$ read-only
        $\mathsf{r}_2 := \mathsf{pack}\ \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \mathsf{removeContinuation}(\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}})$
        $\mathsf{r}_1 := \mathsf{r}_1[i]$
        jump $\mathsf{remove}[\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}]$
    }
    $l'\ (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}})\ \{\mathcal{E}^{\Gamma'}(\vec{x}, \vec{y}); I'\}$
```

Labels l and l' are fresh. The resulting instruction jump l executes code block l in heap H, which prepares registers r_1 and r_2, and then transfers control to the monitor's code block remove. Channel x_i (that is, the monitor) is loaded in register r_1; the continuation for P is loaded at register r_2, as witnessed by type $\langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}$. The code for remove transfers the control back to l' (in heap H), where the current environment is again in register r_1, and the values that replace $\vec{y}$ (that is, the element removed from the monitor's buffer) is in register r_2. The current environment $\vec{x}$ is then extended with $\vec{y}$ and process P is executed.

For example, consider the translation of a copy of the echo server, where the typing $\Gamma = \emptyset, echo\colon$ ^[int, ^[int]].

$$\mathcal{P}^{echo,\Gamma}[\![echo(msg, reply).\overline{reply}\langle msg\rangle]\!]$$

The generated code is:

```
  -- ...
  jump l5
}
l5 (r1 :Env) {
  r2 := new 2
  r2 [1] := l6
  r2 [2] := r1
  share r2 read-only
  r2 := pack Env, r2 as removeContinuation(⟨int,BufferMonitor(⟨int⟩ro ) ⟩ro )
  r1 := r1 [2] -- load 'echo'
  jump remove[⟨int,BufferMonitor(⟨int⟩ro ) ⟩ro ]
}
l6 (r1 :Env, r2 :⟨ int ,BufferMonitor(⟨ int ⟩ro ) ⟩ro ) {
  -- E^Γ(printInt; echo, msg; reply)
  r3 := new 4           -- |printInt ;echo;msg;reply|
  r4 := r1 [1]          -- i = 1
  r3 [1] := r4
  r4 := r1 [2]          -- i = 2
  r3 [2] := r4
  r4 := r2 [1]          -- i = 1
  r3 [3] := r4
  r4 := r2 [2]          -- i = 2
  r3 [4] := r4
  share r3 read-only
  r1 := r3
  -- $\mathcal{P}^{\vec{x},\Gamma_1}[\![\overline{reply}\langle msg\rangle]\!]$
  r2 := new 1
  r3 := r1 [3]          -- $\mathcal{V}^{\vec{x}}[\![msg]\!]$
  r2 [1] := r3
  share r2 read-only    -- ⟨int⟩ro
  r1 := r1 [4]          -- load 'reply'
  jump append[⟨int⟩ro]
}
```

— *The type of the environment*

$$\mathsf{Env} \overset{\mathsf{def}}{=} \langle \mathsf{BufferMonitor}(\langle \mathsf{int} \rangle^{\mathsf{ro}}), \mathsf{BufferMonitor}(\langle \mathsf{int}, \mathsf{BufferMonitor}(\langle \mathsf{int} \rangle^{\mathsf{ro}}) \rangle^{\mathsf{ro}}) \rangle^{\mathsf{ro}}$$

where the π-calculus typing $\Gamma_1 = \Gamma, msg\colon \mathsf{int}, reply\colon \hat{}[\mathsf{int}]$ and the environment $\vec{x} = echo; msg; reply$, both related to the output process $\overline{reply}\langle msg\rangle$.

The translation of the replicated input process is identical, except for the code block

$$l' \; (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}) \; \{\mathsf{fork}\; l; \mathcal{E}^{\Gamma'}(\vec{x},\vec{y}); I'\}$$

that starts the continuation of the removed element by forking a copy of the translation of the input process at code block l, thus unfolding one copy of the replicated input. For example, consider the translation of the echo server. This translation corresponds to the replicated version of the previous example (translating a copy of the echo server). The difference in the output is code block l_2, which we rewrite accordingly:

```
l6 (r1 :Env, r2 :⟨ int ,BufferMonitor(⟨ int ⟩ro ) ⟩ro ) {
  fork l5 -- $\mathcal{P}^{\vec{x},\Gamma}[\![!echo(msg, reply).\overline{reply}\langle msg\rangle]\!]$
  -- $\mathcal{E}^{\Gamma}(printInt; echo, msg; reply)$
  r3 := new 4           -- |printInt ;echo;msg;reply|
  r4 := r1 [1]          -- i = 1
  r3 [1] := r4
  r4 := r1 [2]          -- i = 2
  r3 [2] := r4
  r4 := r2 [1]          -- i = 1
  r3 [3] := r4
  r4 := r2 [2]          -- i = 2
  r3 [4] := r4
  share r3 read-only
  r1 := r3
  -- $\mathcal{P}^{\vec{x},\Gamma_1}[\![\overline{reply}\langle msg\rangle]\!]$
  r2 := new 1
  r3 := r1 [3]          -- $\mathcal{V}^{\vec{x}}[\![msg]\!]$
  r2 [1] := r3
  share r2 read-only  -- ⟨int⟩^ro
  r1 := r1[4]          -- load 'reply'
  jump append[⟨int⟩^ro]
}
```

The parallel process $P \mid Q$ is translated by forking the execution of P and of Q, whilst (read-only) environment $\vec{x}$ is shared by both new threads (executing the sub-processes).

$$\mathcal{P}^{\vec{x},\Gamma}[\![P \mid Q]\!] \stackrel{\text{def}}{=} (H \uplus H_P \uplus H_Q, I)$$

where

$$\begin{aligned}
(H_P, I_P) &= \mathcal{P}^{\vec{x},\Gamma}[\![P]\!] \\
(H_Q, I_Q) &= \mathcal{P}^{\vec{x},\Gamma}[\![Q]\!] \\
H &= \{l_P\colon (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}})\{I_P\}\} \uplus \{l_Q\colon (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}})\{I_Q\}\} \uplus H_P \uplus H_Q \\
I &= (\mathsf{fork}\ l_P; \mathsf{fork}\ l_Q; \mathsf{done})
\end{aligned}$$

Labels l_P and l_Q are fresh. For illustrating an example of the translation of the parallel composition, we translate the communication between a client that sends the number 10 and a channel $printInt$ and the echo server:

$$\mathcal{P}^{\vec{x},\Gamma}[\![\overline{echo}\langle 10, printInt\rangle \mid !echo(msg, reply).\overline{reply}\langle msg\rangle]\!]$$

Let $\Gamma = \emptyset, printInt\colon \hat{}[\mathsf{int}], echo\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]]$ and $\vec{x} = printInt; echo$. An excerpt of the generated MIL code follows.

```
  -- ...
  -- $\mathcal{P}^{\vec{x},\Gamma}[\![\overline{echo}\langle 10, printInt\rangle \mid !echo(msg, reply).\overline{reply}\langle msg\rangle]\!]$
  fork l3
  fork l4
  done
}
l3 (r1 :Env) {
  -- $\mathcal{P}^{\vec{x},\Gamma}[\![\overline{echo}\langle 10, printInt\rangle]\!]$
  r2 := new 2
  r3 := 10         -- $\mathcal{V}^{\vec{x}}[\![10]\!]$
  r2[1] := r3
  r3 := r1[1]      -- $\mathcal{V}^{\vec{x}}[\![printInt]\!]$
  r2[2] := r3
  share r2 read-only
  r1 := r1[2]
  jump append[⟨int, BufferMonitor(⟨int⟩^ro)⟩^ro]
}
l4 (r1 :Env) {
  -- $\mathcal{P}^{\vec{x},\Gamma}[\![!echo(msg, reply).]\!]\overline{reply}\langle msg\rangle$
  jump l5
}
-- ...
```

In the translation of scope restriction, a new monitor is created for channel y and added to the current environment. In instruction sequence I, register r_1 is loaded with the continuation, and control transferred to operation createBuffer. The code for createBuffer transfers the control back to l, where the current environment is again in register r_1 and the newly created monitor is in register r_2. Before proceeding with the code for P, this channel (monitor) must be appended to the current environment: in register r_2 we create a one-place environment containing the channel, which is then concatenated to the current environment via instructions prescribed by $\mathcal{E}^{\Gamma'}(\vec{x}, y)$.

$$\begin{aligned}
\mathcal{P}^{\vec{x},\Gamma}[\![(\nu\, y\colon T)\, P]\!] &\stackrel{\text{def}}{=} (H \uplus H', I) \text{ where}\\
T &= \hat{}[\vec{T}]\\
\Gamma' &= \Gamma, y\colon T\\
(H', I') &= \mathcal{P}^{\vec{x}y,\Gamma'}[\![P]\!]\\
H &= l\ (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \mathcal{T}[\![T]\!])\ \{\\
&\qquad \mathsf{r}_3 := \mathsf{new}\ 1\\
&\qquad \mathsf{r}_3[1] := \mathsf{r}_2\\
&\qquad \mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only} \qquad -\!-\langle \mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}}\\
&\qquad \mathsf{r}_2 := \mathsf{r}_3\\
&\qquad \mathcal{E}^{\Gamma'}(\vec{x}, y)\\
&\qquad I'\}\\
I &= (\mathsf{r}_2 := \mathsf{new}\ 2\\
&\quad \mathsf{r}_2[1] := l\\
&\quad \mathsf{r}_2[2] := \mathsf{r}_1\\
&\quad \mathsf{share}\ \mathsf{r}_2\ \mathsf{read\text{-}only}\\
&\quad \mathsf{r}_1 := \mathsf{pack}\ \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \mathsf{createContinuation}(\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}})\\
&\quad \mathsf{jump}\ \mathsf{createBuffer}[\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}])
\end{aligned}$$

Label l is fresh.

Consider the process representing an echo client communicating with the echo server

$$(\nu\, echo\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]])\ (\overline{echo}\langle 10, printInt\rangle \mid !echo(msg, reply).\overline{reply}\langle msg\rangle)$$

Let $\Gamma = \emptyset, printInt\colon \hat{}[\mathsf{int}], echo\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]]$. By applying $\mathcal{P}^{printInt,\Gamma}[\![\cdot]\!]$ to the process above we get the following instruction sequence.

```
−− ...
r2    :≝ new 2
r2 [1] :≝ l2
r2 [2] :≝ r1
share r2 as read-only
```

```
  r1 :≝ pack EnvInit as createContinuation(⟨int,BufferMonitor(⟨ int ⟩^ro ) ⟩^ro )
  jump createBuffer[⟨int,BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ]
}
l2 (r1 :EnvInit, r2 :BufferMonitor(⟨ int ,BufferMonitor(⟨ int ⟩^ro ) ⟩^ro )) {
  r3 :≝ new 1
  r3 [1] :≝ r2
  share r3 read-only
  r2 :≝ r3
  −− ℰ^Γ(printInt, echo)
  r3 :≝ new 2
  r4 :≝ r1 [1]
  r3 [1] :≝ r4
  r4 :≝ r2 [1]
  r3 [2] :≝ r4
  share r3 read-only
  r1 :≝ r3
  −− 𝒫^{x⃗,Γ}⟦!echo(msg, reply).⟧ ̅r̅e̅p̅l̅y̅⟨msg⟩
  −− ...
}
−− ...
−− The type of the environments
EnvInit ≝ ⟨BufferMonitor(⟨ int ⟩^ro ) ⟩^ro
```

Appendix A encloses the full listing of the translation

$$\mathcal{P}[\![(\nu\, printInt\colon \hat{}[\mathsf{int}])\,(\nu\, echo\colon \hat{}[\mathsf{int}, \hat{}[\mathsf{int}]])$$
$$(\overline{echo}\langle 10, printInt\rangle \mid !echo(msg, reply).\overline{reply}\langle msg\rangle)]\!]$$

5.2 Results

The main result of our compiler states that our translation produces type correct MIL programs from type correct π-terms (closed processes).

For the remaining of this chapter let heap H_0 stands for the supporting code and let Ψ_0 be a type environment such that $\Psi_0 \vdash H_0$. We have not attempted to hand-check the typability of the 250-plus lines of H_0; instead we have run it through the MIL type checker [26], which implements the type system described in Chapter 3 and that has been used to type check various non-trivial programs. Furthermore, let H stands for the hypothesis.

First we introduce Propositions 5.2.1 and 5.2.2, whose proofs we omit since they are standard and can easily be found in the literature (for instance [42]). We present proofs for the remaining propositions and lemmas. The main result is Theorem 5.2.11, that ensures the type preserving translation of our compiler.

Proposition 5.2.1. *If* $\Psi; \Gamma; \Lambda \vdash I$*,* Ψ' *is well-formed,* $\forall l \in \mathrm{dom}(\Psi).\Psi(l) = \Psi'(l)$*, and* $\forall \omega \in \mathrm{dom}(\Psi).\Psi(\omega) = \Psi'(\omega)$*, then* $\Psi'; \Gamma; \Lambda \vdash I$*.*

Proposition 5.2.2. *If* $\Psi; \Gamma; \Lambda \vdash I$*,* Γ' *is well-formed, and* $\Gamma <: \Gamma'$*, then* $\Psi; \Gamma'; \Lambda \vdash I$*.*

Proposition 5.2.3 is useful for checking that a buffered monitor type is well-formed, with rule T-TYPE. This proposition is used in the proof of Proposition 5.2.4 and in the proof of Lemma 5.2.10.

Proposition 5.2.3. *The free type variables of a buffer monitor type is the free type variables of its parameter:* $\mathrm{ftv}(\mathsf{BufferMonitor}(\tau)) = \mathrm{ftv}(\tau)$.

Proof. The proof is done from bottom-up, meaning that we start by getting the free names of the type definitions present in BufferMonitor and finish getting the free names of $\mathsf{BufferMonitor}(\tau)$.

By the definition of enqueueContinuation, we have

$$\mathrm{ftv}(\mathsf{enqueueContinuation}(\lambda)) = \mathrm{ftv}(\exists\,\alpha.\langle(\mathsf{r}_1 : \alpha)\ \textsf{requires}\ (\lambda), \alpha\rangle^{\mathsf{ro}})$$

By the definition of ftv and using simple operations of set theory:

$$\begin{aligned}\mathrm{ftv}(\exists\,\alpha.\langle(\mathsf{r}_1 : \alpha)\ \textsf{requires}\ (\lambda), \alpha\rangle^{\mathsf{ro}}) &= \mathrm{ftv}(\langle(\mathsf{r}_1 : \alpha)\ \textsf{requires}\ (\lambda), \alpha\rangle^{\mathsf{ro}} \setminus \{\alpha\} \\ &= (\mathrm{ftv}((\mathsf{r}_1 : \alpha)\ \textsf{requires}\ (\lambda)) \cup \mathrm{ftv}(\alpha)) \setminus \{\alpha\} \\ &= (\mathrm{ftv}(\alpha) \cup \mathrm{ftv}(\lambda) \cup \mathrm{ftv}(\alpha)) \setminus \{\alpha\} \\ &= (\{\alpha\} \cup \{\lambda\} \cup \{\alpha\}) \setminus \{\alpha\} \\ &= \{\lambda\}\end{aligned}$$

Hence, $\mathrm{ftv}(\mathsf{enqueueContinuation}(\lambda)) = \{\lambda\}$.

By definition of enqueueHandler:

$$\begin{aligned}\mathrm{ftv}(\mathsf{enqueueHandler}(\alpha, \lambda, \beta, \tau)) = \mathrm{ftv}((&\mathsf{r}_1 : \alpha, \mathsf{r}_2 : \tau, \\ &\mathsf{r}_3 : \mathsf{enqueueContinuation}(\lambda), \\ &\mathsf{r}_4 : \beta)\ \textsf{requires}\ (\lambda))\end{aligned}$$

By applying the definition of ftv, we have

$$\begin{aligned}&\mathrm{ftv}((\mathsf{r}_1 : \alpha, \mathsf{r}_2 : \tau, \mathsf{r}_3 : \mathsf{enqueueContinuation}(\lambda), \mathsf{r}_4 : \beta)\ \textsf{requires}\ (\lambda)) \\ =&\,\mathrm{ftv}(\alpha) \cup \mathrm{ftv}(\tau) \cup \mathrm{ftv}(\mathsf{enqueueContinuation}(\lambda)) \cup \mathrm{ftv}(\beta) \cup \{\lambda\} \\ =&\{\alpha\} \cup \mathrm{ftv}(\tau) \cup \mathrm{ftv}(\mathsf{enqueueContinuation}(\lambda)) \cup \{\beta\} \cup \{\lambda\}\end{aligned}$$

But we have that $\mathrm{ftv}(\mathsf{enqueueContinuation}(\lambda)) = \{\lambda\}$, hence,

$$\begin{aligned}&\{\alpha\} \cup \mathrm{ftv}(\tau) \cup \mathrm{ftv}(\mathsf{enqueueContinuation}(\lambda)) \cup \{\beta\} \cup \{\lambda\} \\ =&\{\alpha\} \cup \mathrm{ftv}(\tau) \cup \{\lambda\} \cup \{\beta\} \cup \{\lambda\}\end{aligned}$$

Therefore,

$$\mathrm{ftv}(\mathsf{enqueueHandler}(\alpha, \lambda, \beta, \tau)) = \{\alpha, \lambda, \beta\} \cup \mathrm{ftv}(\tau)$$

By definition of dequeueContinuation we have

$$\mathrm{ftv}(\mathsf{dequeueContinuation}(\lambda, \tau)) = \mathrm{ftv}(\exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \tau) \textsf{ requires } (\lambda), \beta\rangle^{\mathsf{ro}})$$

By applying the definition of ftv we obtain

$$\begin{aligned}&\mathrm{ftv}(\exists\alpha.\langle(\mathsf{r}_1 : \alpha, \mathsf{r}_2 : \tau) \textsf{ requires } (\lambda), \alpha\rangle^{\mathsf{ro}})\\ =&\,\mathrm{ftv}(\langle(\mathsf{r}_1 : \alpha, \mathsf{r}_2 : \tau) \textsf{ requires } (\lambda), \alpha\rangle^{\mathsf{ro}}) \setminus \{\alpha\}\\ =&(\mathrm{ftv}((\mathsf{r}_1 : \alpha, \mathsf{r}_2 : \tau) \textsf{ requires } (\lambda)) \cup \mathrm{ftv}(\alpha) \setminus \{\alpha\}\\ =&((\mathrm{ftv}(\alpha) \cup \mathrm{ftv}(\tau) \cup \mathrm{ftv}(\lambda)) \cup \{\alpha\}) \setminus \{\alpha\}\\ =&((\{\alpha\} \cup \mathrm{ftv}(\tau) \cup \{\lambda\}) \cup \{\alpha\}) \setminus \{\alpha\}\end{aligned}$$

Next we use the set theory to simplify the expression

$$((\{\alpha\} \cup \mathrm{ftv}(\tau) \ \cup \{\lambda\}) \cup \{\alpha\}) \setminus \{\alpha\} = \{\lambda\} \cup \mathrm{ftv}(\tau)$$

Therefore, $\mathrm{ftv}(\mathsf{dequeueContinuation}(\lambda, \tau)) = \{\lambda\} \cup \mathrm{ftv}(\tau)$.

By applying the definition of dequeueHandler, we get

$$\begin{aligned}&\mathrm{ftv}(\mathsf{dequeueHandler}(\alpha, \lambda, \beta, \tau))\\ =&\,\mathrm{ftv}((\mathsf{r}_1 : \alpha, \mathsf{r}_2 : \mathsf{dequeueContinuation}(\lambda, \tau), \mathsf{r}_3 : \beta) \textsf{ requires } (\lambda))\end{aligned}$$

Next, we use the definition of function ftv

$$\begin{aligned}&\mathrm{ftv}((\mathsf{r}_1 : \alpha, \mathsf{r}_2 : \mathsf{dequeueContinuation}(\lambda, \tau), \mathsf{r}_3 : \beta) \textsf{ requires } (\lambda))\\ =&\,\mathrm{ftv}(\alpha) \cup \mathrm{ftv}(\mathsf{dequeueContinuation}(\lambda, \tau)) \cup \mathrm{ftv}(\beta) \cup \mathrm{ftv}(\lambda)\\ =&\{\alpha\} \cup \mathrm{ftv}(\mathsf{dequeueContinuation}(\lambda, \tau)) \cup \{\beta\} \cup \{\lambda\}\end{aligned}$$

Since $\mathrm{ftv}(\mathsf{dequeueContinuation}(\lambda, \tau)) = \{\lambda\} \cup \mathrm{ftv}(\tau)$, hence,

$$\begin{aligned}&\{\alpha\} \cup \mathrm{ftv}(\mathsf{dequeueContinuation}(\lambda, \tau)) \cup \{\beta\} \cup \{\lambda\}\\ =&\{\alpha\} \cup (\{\lambda\} \cup \mathrm{ftv}(\tau)) \cup \{\beta\} \cup \{\lambda\}\end{aligned}$$

Simplifying the expression, we obtain

$$\{\alpha\} \cup (\{\lambda\} \cup \mathrm{ftv}(\tau)) \cup \{\beta\} \cup \{\lambda\} = \{\lambda, \alpha, \beta\} \cup \mathrm{ftv}(\tau)$$

Thus, $\mathrm{ftv}(\mathsf{dequeueHandler}(\alpha, \lambda, \beta, \tau)) = \{\lambda, \alpha, \beta\} \cup \mathrm{ftv}(\tau)$.

We apply the definition of QueueRep.

$$\begin{aligned}&\mathrm{ftv}(\mathsf{QueueRep}(\alpha, \lambda, \tau))\\ =&\,\mathrm{ftv}(\exists\ \beta.\langle\beta, \mathsf{enqueueHandler}(\alpha, \lambda, \beta, \tau), \mathsf{dequeueHandler}(\alpha, \lambda, \beta, \tau)\rangle^{\mathsf{ro}})\end{aligned}$$

Again, we apply the definition of ftv.

$$\begin{aligned}
&\mathrm{ftv}(\exists\, \beta.\langle \beta, \mathsf{enqueueHandler}(\alpha,\lambda,\beta,\tau), \mathsf{dequeueHandler}(\alpha,\lambda,\beta,\tau)\rangle^{\mathsf{ro}})\\
=&\,\mathrm{ftv}(\langle \beta, \mathsf{enqueueHandler}(\alpha,\lambda,\beta,\tau), \mathsf{dequeueHandler}(\alpha,\lambda,\beta,\tau)\rangle^{\mathsf{ro}}) \setminus \{\beta\}\\
=&(\mathrm{ftv}(\beta) \cup \mathrm{ftv}(\mathsf{enqueueHandler}(\alpha,\lambda,\beta,\tau))\cup\\
&\quad \mathrm{ftv}(\mathsf{dequeueHandler}(\alpha,\lambda,\beta,\tau))) \setminus \{\beta\}\\
=&(\{\beta\} \cup \mathrm{ftv}(\mathsf{enqueueHandler}(\alpha,\lambda,\beta,\tau))\cup\\
&\quad \mathrm{ftv}(\mathsf{dequeueHandler}(\alpha,\lambda,\beta,\tau))) \setminus \{\beta\}
\end{aligned}$$

But we have

$$\begin{aligned}
&\mathrm{ftv}(\mathsf{enqueueHandler}(\alpha,\lambda,\beta,\tau)) = \{\lambda,\alpha,\beta\} \cup \mathrm{ftv}(\tau) \text{ and}\\
&\mathrm{ftv}(\mathsf{dequeueHandler}(\alpha,\lambda,\beta,\tau)) = \{\lambda,\alpha,\beta\} \cup \mathrm{ftv}(\tau).
\end{aligned}$$

Hence,

$$\begin{aligned}
&(\{\beta\} \cup \mathrm{ftv}(\mathsf{enqueueHandler}(\alpha,\lambda,\beta,\tau))\cup\\
&\quad \mathrm{ftv}(\mathsf{dequeueHandler}(\alpha,\lambda,\beta,\tau))) \setminus \{\beta\}\\
=&(\{\beta\} \cup (\{\lambda,\alpha,\beta\} \cup \mathrm{ftv}(\tau)) \cup (\{\lambda,\alpha,\beta\} \cup \mathrm{ftv}(\tau))) \setminus \{\beta\}\\
=&\{\lambda,\alpha\} \cup \mathrm{ftv}(\tau)
\end{aligned}$$

Therefore, $\mathrm{ftv}(\mathsf{QueueRep}(\alpha,\lambda,\tau)) = \{\lambda,\alpha\} \cup \mathrm{ftv}(\tau)$.

By the definition of Queue, we get

$$\mathrm{ftv}(\mathsf{Queue}(\lambda,\tau)) = \mathrm{ftv}(\mu\, \alpha.\langle \mathsf{QueueRep}(\alpha,\lambda,\tau), \mathsf{int}\rangle^{\lambda})$$

By the definition of ftv, we have

$$\begin{aligned}
&\mathrm{ftv}(\langle \mathsf{QueueRep}(\alpha,\lambda,\tau), \mathsf{int}\rangle^{\lambda}) \setminus \{\alpha\}\\
=&(\mathrm{ftv}(\mathsf{QueueRep}(\alpha,\lambda,\tau) \cup \mathrm{ftv}(\mathsf{int}) \cup \{\lambda\}) \setminus \{\alpha\}\\
=&(\mathrm{ftv}(\mathsf{QueueRep}(\alpha,\lambda,\tau) \cup \emptyset \cup \{\lambda\}) \setminus \{\alpha\}\\
=&\,\mathrm{ftv}(\mathsf{QueueRep}(\alpha,\lambda,\tau) \setminus \{\alpha\} \cup \{\lambda\}
\end{aligned}$$

But we know that $\mathrm{ftv}(\mathsf{QueueRep}(\alpha,\lambda,\tau)) = \{\lambda,\alpha\} \cup \mathrm{ftv}(\tau)$, thus we get

$$\mathrm{ftv}(\mathsf{QueueRep}(\alpha,\lambda,\tau) \setminus \{\alpha\} \cup \{\lambda\} = (\{\lambda,\alpha\}\, \mathrm{ftv}(\tau)) \setminus \{\alpha\} \cup \{\lambda\}$$

Again, simplifying the expression.

$$(\{\lambda,\alpha\} \cup \mathrm{ftv}(\tau)) \setminus \{\alpha\} \cup \{\lambda\} = \mathrm{ftv}(\tau) \cup \{\lambda\}$$

Hence, $\mathrm{ftv}(\mathsf{Queue}(\lambda, \tau)) = \{\lambda\} \cup \mathrm{ftv}(\tau)$.

By the definition of waitContinuation, we obtain

$$\mathrm{ftv}(\mathsf{waitContinuation}(\lambda)) = \mathrm{ftv}(\exists\alpha.\langle(\mathsf{r}_1 : \alpha) \textsf{ requires } (\lambda), \alpha\rangle^{\mathsf{ro}})$$

By definition of ftv we have

$$\begin{aligned}
&\mathrm{ftv}(\exists\alpha.\langle(\mathsf{r}_1 : \alpha) \textsf{ requires } (\lambda), \alpha\rangle^{\mathsf{ro}})\\
=&\,\mathrm{ftv}(\langle(\mathsf{r}_1 : \alpha) \textsf{ requires } (\lambda), \alpha\rangle^{\mathsf{ro}}) \setminus \{\alpha\}\\
=&(\mathrm{ftv}((\mathsf{r}_1 : \alpha) \textsf{ requires } (\lambda)) \cup \mathrm{ftv}(\alpha)) \setminus \{\alpha\}\\
=&((\mathrm{ftv}(\alpha) \cup \mathrm{ftv}(\lambda)) \cup \{\alpha\}) \setminus \{\alpha\}\\
=&((\{\alpha\} \cup \{\lambda\}) \cup \{\alpha\}) \setminus \{\alpha\}\\
=&\{\lambda\}
\end{aligned}$$

Therefore, $\mathrm{ftv}(\mathsf{waitContinuation}(\lambda)) = \{\lambda\}$.

By definition of Condition.

$$\mathrm{ftv}(\mathsf{Condition}(\lambda)) = \mathrm{ftv}(\mathsf{Queue}(\lambda, \mathsf{waitContinuation}(\lambda)))$$

We have that $\mathrm{ftv}(\mathsf{Queue}(\lambda, \tau)) = \{\lambda\} \cup \mathrm{ftv}(\tau)$, thus

$$\mathrm{ftv}(\mathsf{Queue}(\lambda, \mathsf{waitContinuation}(\lambda))) = \{\lambda\} \cup \mathrm{ftv}(\mathsf{waitContinuation}(\lambda))$$

We also have $\mathrm{ftv}(\mathsf{waitContinuation}(\lambda)) = \{\lambda\}$, therefore,

$$\{\lambda\} \cup \mathrm{ftv}(\mathsf{waitContinuation}(\lambda)) = \{\lambda\} \cup \{\lambda\}$$

Hence, $\mathrm{ftv}(\mathsf{Condition}(\lambda)) = \{\lambda\}$.

By definition of UnpackedBufferMonitor we have

$$\mathrm{ftv}(\mathsf{UnpackedBufferMonitor}(\lambda, \tau)) = \mathrm{ftv}(\langle\mathsf{Queue}(\lambda, \tau), \mathsf{Condition}(\lambda), \langle\lambda\rangle^{\lambda}\rangle^{\mathsf{ro}})$$

By the definition of ftv we get

$$\begin{aligned}
&\mathrm{ftv}(\langle\mathsf{Queue}(\lambda, \tau), \mathsf{Condition}(\lambda), \langle\lambda\rangle^{\lambda}\rangle^{\mathsf{ro}})\\
=&\,\mathrm{ftv}(\mathsf{Queue}(\lambda, \tau)) \cup \mathrm{ftv}(\mathsf{Condition}(\lambda)) \cup \mathrm{ftv}(\langle\lambda\rangle^{\lambda})\\
=&\,\mathrm{ftv}(\mathsf{Queue}(\lambda, \tau)) \cup \mathrm{ftv}(\mathsf{Condition}(\lambda)) \cup (\mathrm{ftv}(\lambda) \cup \mathrm{ftv}(\lambda))\\
=&\,\mathrm{ftv}(\mathsf{Queue}(\lambda, \tau)) \cup \mathrm{ftv}(\mathsf{Condition}(\lambda)) \cup (\{\lambda\} \cup \{\lambda\})
\end{aligned}$$

Since $\mathrm{ftv}(\mathsf{Queue}(\lambda, \tau)) = \{\lambda\} \cup \mathrm{ftv}(\tau)$ and $\mathrm{ftv}(\mathsf{Condition}(\lambda)) = \{\lambda\}$, hence,

$$\begin{aligned}
&\mathrm{ftv}(\mathsf{Queue}(\lambda, \tau)) \cup \mathrm{ftv}(\mathsf{Condition}(\lambda)) \cup (\{\lambda\} \cup \{\lambda\})\\
=&(\{\lambda\} \cup \mathrm{ftv}(\tau)) \cup \{\lambda\} \cup (\{\lambda\} \cup \{\lambda\})\\
=&\{\lambda\} \cup \mathrm{ftv}(\tau)
\end{aligned}$$

So, $\mathrm{ftv}(\mathsf{UnpackedBufferMonitor}(\lambda, \tau)) = \{\lambda\} \cup \mathrm{ftv}(\tau)$.

Finally, by the definition of BufferMonitor, we have

$$\mathrm{ftv}(\mathsf{BufferMonitor}(\tau)) = \mathrm{ftv}(\exists\lambda.\mathsf{UnpackedBufferMonitor}(\lambda, \tau))$$

We apply the definition of function ftv and get

$$\mathrm{ftv}(\exists\lambda.\mathsf{UnpackedBufferMonitor}(\lambda, \tau)) = \mathrm{ftv}(\mathsf{UnpackedBufferMonitor}(\lambda, \tau)) \setminus \{\lambda\}$$

But we know that $\mathrm{ftv}(\mathsf{UnpackedBufferMonitor}(\lambda, \tau)) = \{\lambda\} \cup \mathrm{ftv}(\tau)$, hence,

$$\mathrm{ftv}(\mathsf{UnpackedBufferMonitor}(\lambda, \tau)) \setminus \{\lambda\} = (\{\lambda\} \cup \mathrm{ftv}(\tau)) \setminus \{\lambda\}$$

Thus, we have that $\mathrm{ftv}(\mathsf{BufferMonitor}(\tau)) = \mathrm{ftv}(\tau)$.

□

Proposition 5.2.4 is helpful for checking that a translated type is well-formed (and closed) and that is not a local tuple type or a singleton lock type (needed to store and move translated values). It is used in proof of Lemma 5.2.6, of Lemma 5.2.9, and of Lemma 5.2.10.

Proposition 5.2.4. *Function $\mathcal{T}[\![T]\!]$ generates a closed type ($\mathrm{ftv}(\mathcal{T}[\![T]\!]) = \emptyset$) either of the form $\exists\lambda.\tau$ (the translation of a channel type $\hat{}[\vec{T}]$) or of the form int (the translation of an integer type int).*

Proof. The proof follows by induction on the definition of function $\mathcal{T}[\![\cdot]\!]$.

- Case T is int. By definition of the translation, $\mathcal{T}[\![\mathsf{int}]\!] = \mathsf{int}$. We have that $\mathrm{ftv}(\mathsf{int}) = \emptyset$, by the definition of ftv.

- Case T is $\hat{}[T_1, \ldots, T_n]$, we have the translation

  $$\mathcal{T}[\![\hat{}[T_1, \ldots, T_n]]\!] = \mathsf{BufferMonitor}(\langle \mathcal{T}[\![T_1, \ldots, T_n]\!]\rangle^{\mathsf{ro}})$$

  By Proposition 5.2.3, we have that

  $$\mathrm{ftv}(\mathsf{BufferMonitor}(\langle \mathcal{T}[\![T_1, \ldots, T_n]\!]\rangle^{\mathsf{ro}})) = \mathrm{ftv}(\langle \mathcal{T}[\![T_1, \ldots, T_n]\!]\rangle^{\mathsf{ro}})$$

  By definition of ftv we have

  $$\mathrm{ftv}(\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}) = \bigcup_i \mathrm{ftv}(\mathcal{T}[\![T_i]\!])$$

  By induction hypothesis $\mathcal{T}[\![T_1]\!], \ldots, \mathcal{T}[\![T_n]\!]$ are closed types, thus

  $$\bigcup_i \mathrm{ftv}(\mathcal{T}[\![T_i]\!]) = \emptyset$$

  Hence, $\mathcal{T}[\![\hat{}[T_1, \ldots, T_n]]\!]$ is closed.

  Type $\mathcal{T}[\![\hat{}[T_1, \ldots, T_n]]\!]$ is of the form $\exists\lambda.\tau$, because, by definition of $\mathcal{T}[\![\cdot]\!]$ and of BufferMonitor, we have:

  $$\mathcal{T}[\![\hat{}[T_1, \ldots, T_n]]\!] = \exists\lambda.\mathsf{UnpackedBufferMonitor}(\langle \mathcal{T}[\![T_1]\!], \ldots, \mathcal{T}[\![T_n]\!]\rangle^{\mathsf{ro}})$$

□

We use Proposition 5.2.5 in the proof of Lemma 5.2.10 to assert that environments are closed.

Proposition 5.2.5. *Type $\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}$ is closed.*

Proof. The proof is straightforward. By definition of ftv, we have $\mathrm{ftv}(\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}) = \bigcup_i \mathrm{ftv}(\mathcal{T}[\![T_i]\!])$. By Proposition 5.2.4, $\mathrm{ftv}(\mathcal{T}[\![T_i]\!]) = \emptyset$, thus, $\mathrm{ftv}(\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}) = \emptyset$. □

Lemma 5.2.6 shows that the translation of a value $\mathcal{V}^{\vec{x}}[\![v]\!]$ is type-preserving, thus the translation of the value's type is assigned to register r_3. This lemma is used in the proof of Proposition 5.2.7.

Lemma 5.2.6. *If*

1. $\Gamma \vdash v : T$,

2. $\Gamma_1(r_1) = \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{ro}$,

3. $\mathrm{fn}(v) \subseteq \vec{x}$, *and*

4. $\Psi; \Gamma_1\{r_3 \colon \mathcal{T}[\![T]\!]\}; \emptyset \vdash I$,

then, $\Psi; \Gamma_1; \emptyset \vdash (\mathcal{V}^{\vec{x}}[\![v]\!]; I)$.

Proof. A value can either be a name or a base value (an integer). The proof is done by inspection on the structure of value v.

Case v is n. By hypothesis, we have $\Gamma \vdash \mathrm{n} : \mathsf{int}$, $\Gamma_1(\mathsf{r}_1) = \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}$, $\mathrm{fn}(\mathrm{n}) \subseteq \{\vec{x}\}$. By definition of $\mathcal{V}^{\cdot}[\![\cdot]\!]$, we have $\mathcal{V}^{\vec{x}}[\![\mathrm{n}]\!] = (\mathsf{r}_3 := \mathrm{n})$. Our goal is to prove that $\Psi; \Gamma_1; \emptyset \vdash (\mathsf{r}_3 := \mathrm{n}; I)$.

$$\dfrac{\dfrac{}{\Psi; \Gamma_1 \vdash \mathrm{n} : \mathsf{int}}\ \textsc{T-Int} \qquad \Psi; \Gamma_1\{\mathsf{r}_3 \colon \mathsf{int}\}; \emptyset \vdash I \quad \mathsf{int} \neq \langle \_ \rangle}{\Psi; \Gamma_1; \emptyset \vdash (\mathsf{r}_3 := \mathrm{n}; I)}\ \textsc{T-Move}$$

By the definition of $\mathcal{T}[\![\cdot]\!]$, we have $\mathcal{T}[\![\mathsf{int}]\!] = \mathsf{int}$, thus, $\Psi; \Gamma_1\{\mathsf{r}_3 \colon \mathsf{int}\}; \emptyset \vdash I$ is given by hypothesis.

Case v is x_i. By hypothesis we have that $\Gamma \vdash x_i : T$, $\Gamma_1(\mathsf{r}_1) = \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}$, $\mathrm{fn}(x_i) \subseteq \vec{x}$, and $\Psi; \Gamma_1\{\mathsf{r}_3 \colon \mathcal{T}[\![T]\!]\}; \emptyset \vdash I$.

By definition of $\mathcal{V}^{\vec{x}}[\![\cdot]\!]$, we have $\mathcal{V}^{\vec{x}}[\![x_i]\!] = (\mathsf{r}_3 := \mathsf{r}_1[i])$. We want to prove that judgement $\Psi; \Gamma_1; \emptyset \vdash (\mathsf{r}_3 := \mathsf{r}_1[i]; I)$ holds. Since $x_i \in \vec{x}$ and $\Gamma_1(\mathsf{r}_1) = \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}$, then

$$\begin{aligned}\Gamma_1(\mathsf{r}_1) &= \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\\ &= \langle \mathcal{T}[\![\Gamma(x_1)]\!], \ldots, \mathcal{T}[\![\Gamma(x_i)]\!], \ldots, \mathcal{T}[\![\Gamma(x_n)]\!]\rangle^{\mathsf{ro}}\\ &= \langle \mathcal{T}[\![\Gamma(x_1)]\!], \ldots, \mathcal{T}[\![T]\!], \ldots, \mathcal{T}[\![\Gamma(x_n)]\!]\rangle^{\mathsf{ro}}\end{aligned}$$

Let $\langle\vec{\tau}\rangle^{\mathsf{ro}} = \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}$, where $\forall\tau_j \in \vec{\tau}.\tau_j = \mathcal{T}\llbracket\Gamma(x_j)\rrbracket$. In particular we have that

$$\tau_i = \mathcal{T}\llbracket\Gamma(x_i)\rrbracket = \mathcal{T}\llbracket T\rrbracket$$

Then, the following judgement tree holds:

$$\dfrac{\dfrac{\Gamma_1(\mathsf{r}_1) = \langle\vec{\tau}\rangle^{\mathsf{ro}}}{\Psi;\Gamma_1 \vdash \mathsf{r}_1 : \langle\vec{\tau}\rangle^{\mathsf{ro}}}\,\text{T-REG} \qquad \Psi;\Gamma_1\{\mathsf{r}_3\colon \tau_i\};\emptyset \vdash I \quad \dfrac{}{\tau_i \neq \lambda}\,\text{Prop. 5.2.4} \qquad \mathsf{ro} \in \{\mathsf{ro}\}}{\Psi;\Gamma_1;\emptyset \vdash \mathsf{r}_3 := \mathsf{r}_1[i]; I}\,\text{T-LOADH}$$

Where $\Psi;\Gamma_1\{\mathsf{r}_3\colon \mathcal{T}\llbracket T\rrbracket\};\emptyset \vdash I$ is given by hypothesis. □

With Proposition 5.2.7, we have that copying a range of values $\vec{v}$ into a local tuple (present in register r_2) fills the local tuple with the translation of each π-value. This proposition is used in the proof of Lemma 5.2.9 and in the proof of Lemma 5.2.10.

Proposition 5.2.7. *If we have a sequence of values $\vec{v}$ ($|\vec{v}| = n$), a sequence of names $\vec{x}$, a π-typing Γ, a MIL-typing Ψ, an instruction sequence I, and $\Gamma_1 = (r_1 \colon \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{ro}, r_2 \colon \langle\tau_1..\tau_m\rangle)$ where*

1. $m >= n$,

2. $\mathrm{fn}(\vec{v}) \subseteq \vec{x}$,

3. $\forall v_i \in \vec{v}.\Gamma \vdash v_i : T_i$, *and*

4. $\Psi; (r_1 \colon \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{ro}, r_2 \colon \langle\mathcal{T}\llbracket\hat{\Gamma}(\vec{v})\rrbracket, \tau_{n+1}..\tau_m\rangle); \emptyset \vdash I$,

then $\Psi;\Gamma_1;\emptyset \vdash \forall\, j \in \{1,\ldots,n\}(\mathcal{V}^{\vec{x}}\llbracket v_j\rrbracket; r_2[j] := r_3); I$.

Proof. We want to prove

$$\Psi;\Gamma_1;\emptyset \vdash \forall\, j \in \{1,\ldots,n\}(\mathcal{V}^{\vec{x}}\llbracket v_j\rrbracket; \mathsf{r}_2[j] := \mathsf{r}_3); I$$

We prove by induction on n.

Base case ($n = 0$). Our goal becomes $\Psi;\Gamma_1;\emptyset \vdash I$.

By hypothesis we have $\Psi;\Gamma_1\{\mathsf{r}_2\colon \langle\tau_1,\ldots,\tau_m\rangle\};\emptyset \vdash I$ and $\Gamma_1(\mathsf{r}_2) = \langle\tau_1,\ldots,\tau_m\rangle$. But $\Gamma_1 = \Gamma_1\{\mathsf{r}_2\colon \langle\tau_1,\ldots,\tau_m\rangle\}$, thus the proof for our goal is given by hypothesis.

Induction step. We prove the induction step $n = k + 1$. First we get to the induction hypothesis. Let $\vec{v} = \vec{w}v_{k+1}$ and $\Gamma_3 = (\mathsf{r}_1 : \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle\mathcal{T}\llbracket\hat{\Gamma}(\vec{w})\rrbracket, \tau_{k+1}..\tau_m\rangle)$. By hypothesis, we have $\Gamma \vdash v_{k+1} : T_{k+1}$, $\Gamma_3(\mathsf{r}_1) = \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}$, and $\mathrm{fn}(v_{k+1}) \subseteq \{\vec{x}\}$, then

$$\dfrac{\Gamma \vdash v_{k+1} : T_{k+1} \quad \Gamma_3(\mathsf{r}_1) = \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}} \quad \mathrm{fn}(v_{k+1}) \subseteq \{\vec{x}\} \quad (1)}{\Psi;\Gamma_3;\emptyset \vdash (\mathcal{V}^{\vec{x}}\llbracket v_{k+1}\rrbracket; \mathsf{r}_2[k+1] := \mathsf{r}_3; I)}\,\text{Lem. 5.2.6}$$

Sequent (1) is $\Psi;\Gamma_{3.1};\emptyset \vdash (\mathsf{r}_2[k+1] := \mathsf{r}_3; I)$, where

$$\Gamma_{3.1} = \Gamma_3\{\mathsf{r}_3 \colon \mathcal{T}\llbracket T_{k+1}\rrbracket\} = (\mathsf{r}_1 \colon \langle \mathcal{T}\llbracket \Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}, \mathsf{r}_2 \colon \langle \mathcal{T}\llbracket\hat{\Gamma}(\vec{w})\rrbracket, \tau_{k+1}..\tau_m\rangle, \mathsf{r}_3 \colon \mathcal{T}\llbracket T_{k+1}\rrbracket)$$

We have that $\mathcal{T}\llbracket T_{k+1}\rrbracket = \mathcal{T}\llbracket\hat{\Gamma}((v_{k+1}))\rrbracket$. Sequent (3) follows.

$$\dfrac{\dfrac{\Gamma_{3.1}(\mathsf{r}_3) = \mathcal{T}\llbracket\hat{\Gamma}(v_{k+1})\rrbracket}{\Psi;\Gamma_{3.1} \vdash \mathsf{r}_3 : \mathcal{T}\llbracket\hat{\Gamma}(v_{k+1})\rrbracket}\,\text{T-REG} \qquad (2) \quad (3) \quad \dfrac{}{\mathcal{T}\llbracket\hat{\Gamma}(v_{k+1})\rrbracket \neq \lambda, \langle \_ \rangle}\,\text{Prop. 5.2.4}}{\Psi;\Gamma_{3.1};\emptyset \vdash (\mathsf{r}_2[k+1] := \mathsf{r}_3; I)}\,\text{T-STOREL}$$

Sequent (2) is $\Psi;\Gamma_{3.1} \vdash \mathsf{r}_3 : \langle \mathcal{T}\llbracket\hat{\Gamma}(\vec{w})\rrbracket, \tau_{k+1}..\tau_m\rangle$ and checked by rule T-REG. Sequent (3) is $\Psi;\Gamma_{3.2};\emptyset \vdash I$, where

$$\begin{aligned}\Gamma_{3.2} &= \Gamma_{3.1}\{\mathsf{r}_2 \colon \langle \mathcal{T}\llbracket\hat{\Gamma}(\vec{w}v_{k+1})\rrbracket, \tau_{k+2}..\tau_m\rangle\}\\ &= (\mathsf{r}_1 \colon \langle \mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}, \mathsf{r}_2 \colon \langle \mathcal{T}\llbracket\hat{\Gamma}(\vec{v})\rrbracket, \tau_{k+2}..\tau_m\rangle, \mathsf{r}_3 \colon \mathcal{T}\llbracket T_{k+1}\rrbracket)\end{aligned}$$

By hypothesis we have that $\Psi;(\mathsf{r}_1 \colon \langle \mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}, \mathsf{r}_2 \colon \langle \mathcal{T}\llbracket\hat{\Gamma}(\vec{v})\rrbracket, \tau_{k+2}..\tau_m\rangle);\emptyset \vdash I$, therefore. by Proposition 5.2.2, we have that $\Psi;\Gamma_{3.2};\emptyset \vdash I$ holds. Hence, by induction hypothesis, we prove our case:

$$\Psi;\Gamma_1;\emptyset \vdash (\forall\, j \in \{1,\ldots,k+1\}(\mathcal{V}^{\vec{x}}\llbracket v_j\rrbracket; \mathsf{r}_2[j] := \mathsf{r}_3); I)$$

□

Proposition 5.2.8 asserts that copying a range of values from a read-only tuple into a local tuple stores each type in the appropriate location (as expected). This proposition is used in the proof of Lemma 5.2.9.

Proposition 5.2.8. *If*

- $\Psi;\Gamma \vdash v : \langle \tau'_1..\tau'_n\vec{\tau}'\rangle^{ro}$,
- $\forall \tau \in \tau'_1..\tau'_n.\tau \neq \lambda, \langle\_\rangle$, $\Psi;\Gamma \vdash r : \langle\vec{\tau}\tau_1..\tau_m\rangle$, $r_t \notin \mathrm{dom}(\Gamma)$*, and*
- $\Psi;\Gamma\{r \colon \langle\vec{\tau}\tau'_1..\tau'_n..\tau_m\rangle\};\Lambda \vdash I$ *with* $m >= n$*,*

then, $\Psi;\Gamma;\Lambda \vdash \forall j \in \{1\ldots n\}.(r_t := v[j]; r[j+|\vec{\tau}|]); I.$

Proof. The proof is by induction on n.

Base case ($n = 0$). Our goal is to prove that the following judgement holds.

$$\Psi;\Gamma;\Lambda \vdash \forall j \in \emptyset.(r_t := v[j]; r[j+|\vec{\tau}|]); I \quad \text{or simply} \quad \Psi;\Gamma;\Lambda \vdash I$$

Since $\Gamma_1\{r \colon \langle\vec{\tau}\tau_1..\tau_m\rangle\}$ is Γ_1, then $\Psi;\Gamma;\Lambda \vdash I$ is $\Psi;\Gamma_1\{r \colon \langle\vec{\tau}\tau_1..\tau_m\rangle\};\Lambda \vdash I$. Hence, this case is proved by hypothesis.

Induction step ($n > 0$). Our goal is to check that

$$\Psi; \Gamma_1; \Lambda \vdash \forall j \in \{1 \ldots n+1\}.(r_t := v[j]; r[j + |\vec{\tau}|]); I$$

First we want to prove the induction hypothesis. Let $I_1 = (r[n + 1 + |\vec{\tau}|] := r_t; I)$. Let $\Gamma_{1.1} = \Gamma_1\{r\colon \langle\vec{\tau}\tau'_1..\tau'_{n+1}\tau_{n+2}..\tau_m\rangle\}$ and $\Gamma_{1.2} = \Gamma_{1.1}\{r_t\colon \tau'_{n+1}\}$. All premises are given by hypothesis, except the following tree.

$$\dfrac{\dfrac{}{\Psi; \Gamma_{1.1} \vdash v : \langle\tau'_1..\tau'_{n+1}\vec{\tau}'\rangle^{\mathsf{ro}}}\,\text{H} \qquad \overbrace{\Psi; \Gamma_{1.2}; \Lambda \vdash I_1}^{(1)} \qquad \dfrac{}{\tau'_{n+1} \neq \lambda}\,\text{H}}{\Psi; \Gamma_{1.1}; \Lambda \vdash (r_t := v[n+1]; I_1)}\,\text{T-LOADH}$$

Sequent (1) follows.

$$\dfrac{\dfrac{\Gamma_{1.2}(r_t) = \tau'_{n+1}}{\Psi; \Gamma_{1.2} \vdash r_t : \tau'_{n+1}}\,\text{T-REG} \qquad (2) \quad (3) \qquad \dfrac{}{\tau'_{n+1} \neq \lambda, \langle\_\rangle}\,\text{H}}{\Psi; \Gamma_{1.2}; \Lambda \vdash (r[n+1+|\vec{\tau}|] := r_t; I)}\,\text{T-STOREL}$$

Sequent (2) is $\Psi; \Gamma_{1.2} \vdash r : \langle\vec{\tau}\tau'_1..\tau'_n..\tau_m\rangle$ and is given by rule T-REG. Sequent (3) is $\Psi; \Gamma_{1.2}\{r\colon \langle\vec{\tau}\tau_1..\tau_m\rangle\}$. We have that

$$\Gamma_1\{r\colon \langle\vec{\tau}\tau'_1..\tau'_{n+1}..\tau_m\rangle\} <: \Gamma_{1.2}\{r\colon \langle\vec{\tau}\tau'_1..\tau'_{n+1}..\tau_m\rangle\}$$

Therefore, by Proposition 5.2.2 we have

$$\Psi; \Gamma_{1.1}\{r\colon \langle\vec{\tau}\tau'_1..\tau'_{n+1}..\tau_m\rangle\}; \Lambda \vdash I$$

Thus, by induction hypothesis, we have that this case holds.

□

Lemma 5.2.9 shows that merging environments preserves the types in the merged environment. Used in the proof of Lemma 5.2.10.

Lemma 5.2.9. *If* $\Psi; (\mathsf{r}_1 : \langle\mathcal{T}[\![\vec{x}\vec{y}]\!]\rangle^{\mathsf{ro}}); \emptyset \vdash I$*, then* $\Psi; (\mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle\mathcal{T}[\![\Gamma(\vec{y})]\!]\rangle^{\mathsf{ro}}); \emptyset \vdash (\mathcal{E}^{\Gamma}(\vec{x}, \vec{y}); I)$.

Proof. By definition of the environment-merge function we have:

$$\begin{aligned}
\mathcal{E}^{\Gamma}(\vec{x}, \vec{y}) =& (\mathsf{r}_3 := \mathsf{new}\ |\vec{x}\vec{y}|; I_1) \\
\text{with } I_1 =& (\forall\, i \in \{1, \ldots, |\vec{x}|\}.(\mathsf{r}_4 := \mathsf{r}_1[i]; \mathsf{r}_3[i] := \mathsf{r}_4); I_2), \\
I_2 =& (\forall\, i \in \{1, \ldots, |\vec{y}|\}.(\mathsf{r}_4 := \mathsf{r}_2[i]; \mathsf{r}_3[i + |\vec{x}|] := \mathsf{r}_4); I_3), \\
I_3 =& (\mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only}; \mathsf{r}_1 := \mathsf{r}_3; I), \\
n =& |\vec{x}|, \text{ and} \\
m =& |\vec{y}|.
\end{aligned}$$

Let $\Gamma_1 = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\langle\mathcal{T}[\![\Gamma(\vec{y})]\!]\rangle^{\mathsf{ro}})$ and $\Gamma_2 = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\vec{x}\vec{y}]\!]\rangle^{\mathsf{ro}})$.

Our goal is to prove that $\Psi;\Gamma_1;\emptyset\vdash(\mathcal{E}^{\Gamma}(\vec{x},\vec{y}); I)$ holds.

$$\dfrac{\overbrace{\Psi;\Gamma_1\{\mathsf{r}_3\colon\langle\overrightarrow{\mathsf{int}}\rangle\};\emptyset\vdash I_1}^{(1)} \qquad |\vec{x}\vec{y}| = |\overrightarrow{\mathsf{int}}|}{\Psi;\Gamma_1;\emptyset\vdash(\mathsf{r}_3 := \mathsf{new}\ |\vec{x}\vec{y}|; I_1)}\ \text{T-NEW}$$

Let $\Gamma_{1.1} = \Gamma_1\{\mathsf{r}_3\colon\langle\overrightarrow{\mathsf{int}}\rangle\} = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\langle\mathcal{T}[\![\Gamma(\vec{y})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_3\colon\langle\overrightarrow{\mathsf{int}}\rangle)$. If $\Psi;\Gamma_{1.1};\emptyset\vdash \mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}$ (rule T-REG), $\forall\tau\in\mathcal{T}[\![\Gamma(\vec{x})]\!]\neq\lambda,\langle\_\rangle$ (by Proposition 5.2.4), $\Psi;\Gamma_{1.1}\vdash \mathsf{r}_3 : \langle\overrightarrow{\mathsf{int}}\rangle$ (rule T-REG), $\mathsf{r}_4\notin\mathrm{dom}(\Gamma_{1.1})$, sequent (2) $\Psi;\Gamma_{1.1}\{\mathsf{r}_3\colon\langle\Gamma(\vec{x}),\overrightarrow{\mathsf{int}}\rangle\};\emptyset\vdash I_2$, and $|\overrightarrow{\mathsf{int}}| >= |\vec{x}|$, then, by Proposition 5.2.8, sequent (1) holds.

We are left with proving sequent (2) $\Psi;\Gamma_{1.2};\emptyset\vdash I_2$, with

$$\Gamma_{1.2} = \{\mathsf{r}_3\colon\langle\Gamma(\vec{x}),\overrightarrow{\mathsf{int}}\rangle\} = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\langle\mathcal{T}[\![\Gamma(\vec{y})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_3\colon\langle\Gamma(\vec{x}),\overrightarrow{\mathsf{int}}\rangle)$$

If $\Psi;\Gamma_{1.2};\emptyset\vdash \mathsf{r}_2\colon\langle\mathcal{T}[\![\Gamma(\vec{y})]\!]\rangle^{\mathsf{ro}}$ (rule T-REG), $\forall\tau\in\mathcal{T}[\![\Gamma(\vec{y})]\!]\neq\lambda,\langle\_\rangle$ (by Proposition 5.2.4), $\Psi;\Gamma_{1.2}\vdash \mathsf{r}_3 : \langle\Gamma(\vec{x}),\overrightarrow{\mathsf{int}}\rangle$ (rule T-REG), $\mathsf{r}_4\notin\mathrm{dom}(\Gamma_{1.2})$, sequent (3) is

$$\Psi;\Gamma_{1.2}\{\mathsf{r}_3\colon\langle\Gamma(\vec{x}\vec{y})\rangle\};\emptyset\vdash I_3$$

and $|\overrightarrow{\mathsf{int}}| = |\vec{y}|$, then, by Proposition 5.2.8, sequent (2) holds. Let

$$\Gamma_{1.3} = \{\mathsf{r}_3\colon\langle\Gamma(\vec{x}\vec{y})\rangle\} = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\langle\mathcal{T}[\![\Gamma(\vec{y})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_3\colon\langle\Gamma(\vec{x}\vec{y})\rangle)$$

We need to prove that sequent (3) $\Psi;\Gamma_{1.3};\emptyset\vdash I_3$ holds.

$$\dfrac{\dfrac{\Gamma_{1.3}(\mathsf{r}_3) = \langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle}{\Psi;\Gamma_{1.3}\vdash \mathsf{r}_3 : \langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle}\ \text{T-REG} \qquad (4)}{\Psi;\Gamma_{1.3};\emptyset\vdash(\mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only}; \mathsf{r}_1 := \mathsf{r}_3; I)}\ \text{T-SHARER}$$

Let

$$\Gamma_{1.4} = \Gamma_{1.3}\{\langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle^{\mathsf{ro}}\colon =\}(\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\langle\mathcal{T}[\![\Gamma(\vec{y})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_3\colon\langle\Gamma(\vec{x}\vec{y})\rangle^{\mathsf{ro}})$$

Sequent (4) is $\Psi;\Gamma_{1.4};\emptyset\vdash(\mathsf{r}_1 := \mathsf{r}_3; I)$, which we typecheck following.

$$\dfrac{\dfrac{\Gamma_{1.4}(\mathsf{r}_3) = \langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle^{\mathsf{ro}}}{\Psi;\Gamma_{1.4}\vdash \mathsf{r}_3 : \langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle^{\mathsf{ro}}}\ \text{T-REG} \qquad (5) \qquad \langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle^{\mathsf{ro}}\neq\langle\_\rangle}{\Psi;\Gamma_{1.4};\emptyset\vdash(\mathsf{r}_1 := \mathsf{r}_3; I)}\ \text{T-MOVE}$$

Sequent (5) is $\Psi;\Gamma_{1.4}\{\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle^{\mathsf{ro}}\};\emptyset\vdash I$. By hypothesis

$$\Psi;(\mathsf{r}_1\colon\langle\mathcal{T}[\![\vec{x}\vec{y}]\!]\rangle^{\mathsf{ro}});\emptyset\vdash I$$

We have $(\mathsf{r}_1 : \langle\mathcal{T}[\![\vec{x}\vec{y}]\!]\rangle^{\mathsf{ro}}) <: \Gamma_{1.4}\{\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle^{\mathsf{ro}}\}$ (rules S-CODE and S-REGFILE). Hence, by Proposition 5.2.2, we have that (5) holds, thus $\Psi;(\mathsf{r}_3\colon\langle\mathcal{T}[\![\Gamma(\vec{x}\vec{y})]\!]\rangle);\emptyset\vdash I_3$ also holds.

□

Lemma 5.2.10 ensures that the translation function $\mathcal{P}^{\vec{x},\Gamma}[\![P]\!]$ is type-preserving, helpful for proving our main result.

Lemma 5.2.10. *If* $\Gamma \vdash P$*,* $\mathrm{fn}(P) \subseteq \vec{x}$*, and* $\mathcal{P}^{\vec{x},\Gamma}[\![P]\!] = (H, I)$*, then* $\exists\Psi$ *such that*

1. $\forall l \in \mathrm{dom}(\Psi_0).\Psi(l) = \Psi_0(l)$*,*

2. $\Psi \vdash H$*, and*

3. $\Psi; (\mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}); \emptyset \vdash I$*.*

Proof. The proof for this lemma is by induction on the structure of P.

Case P is 0. By definition of the translation function we have

$$\mathcal{P}^{\vec{x},\Gamma}[\![\mathsf{0}]\!] = (\emptyset, \mathsf{done})$$

We need to prove that $\exists\Psi$ such that 1) $\forall l \in \mathrm{dom}(\Psi_0).\Psi(l) = \Psi_0(l)$, 2) $\Psi \vdash \emptyset$, and 3) $\Psi; (\mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}); \emptyset \vdash \mathsf{done}$. Let $\Psi = \Psi_0$. Hence, 1) holds, 2) follows by the typing rule for the heap, and 3) is a direct application of rule T-DONE.

Case P is $\overline{x_i}\langle\vec{v}\rangle$. By definition of the translation function we have that

$$\mathcal{P}^{\vec{x},\Gamma}[\![\overline{x_i}\langle\vec{v}\rangle]\!] = (\emptyset, I)$$

$$\begin{aligned}
\text{with } I &= (\mathsf{r}_2 := \mathsf{new}\ |\vec{v}|; I_1),\\
I_1 &= (\forall\, j \in \{1, \ldots, |\vec{v}|\}(\mathcal{V}^{\vec{x}}[\![v_j]\!]; \mathsf{r}_2[j] := \mathsf{r}_3); I_2),\\
I_2 &= (\mathsf{share}\ \mathsf{r}_2\ \mathsf{read\text{-}only}; \mathsf{r}_1 := \mathsf{r}_1[i]; \mathsf{jump}\ \mathsf{append}[\langle\mathcal{T}[\![\hat{\Gamma}(\vec{v})]\!]\rangle^{\mathsf{ro}}]), \text{ and}\\
\Gamma_1 &= (\mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}).
\end{aligned}$$

By hypothesis, we have that $\Gamma \vdash \overline{x_i}\langle\vec{v}\rangle$ and that $\mathrm{fn}(\overline{x_i}\langle\vec{v}\rangle) = \{x_i\} \cup \mathrm{fn}(\vec{v}) \subseteq \vec{x}$. We need to prove that $\exists\Psi$ such that 1) $\forall l \in \mathrm{dom}(\Psi_0).\Psi(l) = \Psi_0(l)$, 2) $\Psi \vdash \emptyset$, and 3) $\Psi; (\mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}); \emptyset \vdash I$.

Let $\Psi = \Psi_0$. Then 1) and 2) holds as for the previous case. We are left with proving that judgement 3) holds. We check the following judgement.

$$\frac{\Psi; \Gamma_1\{\mathsf{r}_2 \colon \langle\vec{\mathsf{int}}\rangle\}; \emptyset \vdash I_1 \quad |\vec{v}| = n}{\Psi; \Gamma_1; \emptyset \vdash \mathsf{r}_2 := (\mathsf{new}\ |\vec{v}|; I_1)}\ \text{T-NEW}$$

Let

$$\begin{aligned}
\Gamma_{1.1} &= \Gamma_1\{\mathsf{r}_2 \colon \langle\vec{\mathsf{int}}\rangle\} = (\mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle\vec{\mathsf{int}}\rangle) \text{ and}\\
\Gamma_{1.2} &= (\mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle\mathcal{T}[\![\hat{\Gamma}(\vec{v})]\!]\rangle).
\end{aligned}$$

We use Proposition 5.2.7 to prove sequent $\Psi; \Gamma_{1.1}\ \emptyset \vdash I_1$. For that we need to show:

(i) $\forall v_i \in \vec{v} : \Gamma \vdash v_i : T_i$,

(ii) $\mathrm{fn}(\vec{v}) \subseteq \vec{x}$, and

(iii) $\Psi; \Gamma_{1.2}; \emptyset \vdash I_2$.

We have (i) using rule TV-OUT and the hypothesis; for (ii) we use the hypothesis and the definition of fn. So, we prove (iii).

$$\dfrac{\dfrac{\Gamma_{1.2}(\mathsf{r}_2) = \langle \mathcal{T}\llbracket \hat{\Gamma}(\vec{v}) \rrbracket \rangle}{\Psi; \Gamma_{1.2} \vdash \mathsf{r}_2 : \langle \mathcal{T}\llbracket \hat{\Gamma}(\vec{v}) \rrbracket \rangle}\,\text{T-REG} \qquad (1)}{\Psi; \Gamma_{1.2}; \emptyset \vdash (\mathsf{share}\ \mathsf{r}_2\ \mathsf{read\text{-}only}; \mathsf{r}_1 := \mathsf{r}_1[i]; \mathsf{jump}\ \mathsf{append}[\langle \mathcal{T}\llbracket \hat{\Gamma}(\vec{v}) \rrbracket \rangle^{\mathsf{ro}}])}\,\text{T-SHARER}$$

Let $\Gamma_{1.3} = (\mathsf{r}_1 : \langle \mathcal{T}\llbracket \Gamma(\vec{x}) \rrbracket \rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle \mathcal{T}\llbracket \hat{\Gamma}(\vec{v}) \rrbracket \rangle^{\mathsf{ro}})$. Since $x_i \in \vec{x}$, therefore

$$\langle \mathcal{T}\llbracket \Gamma(\vec{x}) \rrbracket \rangle^{\mathsf{ro}} = \langle \mathcal{T}\llbracket \Gamma(x_1) \rrbracket, \ldots \mathcal{T}\llbracket \Gamma(x_i) \rrbracket, \ldots, \mathcal{T}\llbracket \Gamma(x_n) \rrbracket \rangle^{\mathsf{ro}}$$

We are left with proving sequent (1), $\Psi;\Gamma_{1.3};\emptyset \vdash (\mathsf{r}_1 := \mathsf{r}_1[i]; \mathsf{jump}\ \mathsf{append}[\langle \mathcal{T}\llbracket \hat{\Gamma}(\vec{v}) \rrbracket \rangle^{\mathsf{ro}}])$.

$$\dfrac{\dfrac{\Gamma_{1.3}(\mathsf{r}_1) = \langle \mathcal{T}\llbracket \Gamma(\vec{x}) \rrbracket \rangle^{\mathsf{ro}}}{\Psi; \Gamma_{1.3} \vdash \mathsf{r}_1 : \langle \mathcal{T}\llbracket \Gamma(\vec{x}) \rrbracket \rangle^{\mathsf{ro}}}\,\text{T-REG} \qquad (2) \qquad \dfrac{}{\mathcal{T}\llbracket \Gamma(x_i) \rrbracket \neq \lambda}\,\text{Prop. 5.2.4}}{\Psi; \Gamma_{1.3}; \emptyset \vdash (\mathsf{r}_1 := \mathsf{r}_1[i]; \mathsf{jump}\ \mathsf{append}[\langle \mathcal{T}\llbracket \hat{\Gamma}(\vec{v}) \rrbracket \rangle^{\mathsf{ro}}])}\,\text{T-LOADH}$$

Next, we prove judgement (2) $\Psi; \Gamma_4; \emptyset \vdash \mathsf{jump}\ \mathsf{append}[\langle \mathcal{T}\llbracket \hat{\Gamma}(\vec{v}) \rrbracket \rangle^{\mathsf{ro}}]$, where

$$\Gamma_{1.4} = \Gamma_{1.3}\{\mathsf{r}_1 \colon \mathcal{T}\llbracket \Gamma(x_i) \rrbracket\} = (\mathsf{r}_1 : \mathcal{T}\llbracket \Gamma(x_i) \rrbracket, \mathsf{r}_2 : \langle \mathcal{T}\llbracket \hat{\Gamma}(\vec{v}) \rrbracket \rangle^{\mathsf{ro}})$$

By hypothesis and rule TV-OUT $\hat{\Gamma}(\vec{v}) = \vec{T}$; by hypothesis and by definition of $\mathcal{T}\llbracket\cdot\rrbracket$ $\mathcal{T}\llbracket \Gamma(x_i) \rrbracket = \mathsf{BufferMonitor}(\langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}})$. Hence, we rewrite sequent (2) as $\Psi;\Gamma_{1.4};\emptyset \vdash \mathsf{jump}\ \mathsf{append}[\langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}}]$ and $\Gamma_{1.4}$ as $(\mathsf{r}_1 : \mathsf{BufferMonitor}(\langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}}), \mathsf{r}_2 : \langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}})$. Let $\Gamma_{1.5} = (\mathsf{r}_1 : \mathsf{BufferMonitor}(\alpha), \mathsf{r}_2 : \alpha)$. We have that $\Gamma_{1.4} = \Gamma_{1.5}\{\langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}}/\alpha\}$ and $\langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}} \neq \langle \_ \rangle$, for proving rule T-VALAPP.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\alpha) = \{\alpha\}}\,\mathrm{ftv}}{\mathrm{ftv}(\mathsf{BufferMonitor}(\alpha)) = \{\alpha\}}\,\text{Prop. 5.2.3} \qquad \dfrac{}{\mathrm{ftv}(\alpha) = \{\alpha\}}\,\mathrm{ftv} \qquad \{\alpha\} \cup \{\alpha\} = \{\alpha\}}{\mathrm{ftv}(\mathsf{BufferMonitor}(\alpha)) \cup \mathrm{ftv}(\alpha) = \{\alpha\}}\,\cup}{\mathrm{ftv}(\Gamma_{1.5}) = \{\alpha\}}\,\mathrm{ftv} \qquad \{\alpha\} \setminus \alpha = \emptyset}{\mathrm{ftv}(\Gamma_{1.5}) \setminus \{\alpha\} = \emptyset}\,\setminus}{\mathrm{ftv}(\forall[\alpha].(\Gamma_{1.5})) = \emptyset}\,\mathrm{ftv}}{\Psi \vdash \forall[\alpha].(\Gamma_{1.5})}\,\text{T-TYPE}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}}) = \emptyset}\,\text{Prop. 5.2.5}}{\Psi \vdash \langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}}}\,\text{T-TYPE} \qquad \dfrac{\dfrac{\Psi \vdash \forall[\alpha].(\Gamma_{1.5})}{\Psi; \forall[\alpha].(\Gamma_5) <: \forall[\alpha].(\Gamma_{1.5})}\,\text{S-REFLEX}}{\Psi; \Gamma_{1.4} \vdash \mathsf{append} : \forall[\alpha].(\Gamma_{1.5})}\,\text{T-LABEL}}{\Psi; \Gamma_{1.4} \vdash \mathsf{append}[\langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}}]}\,\text{T-VALAPP}}{\Psi; \Gamma_{1.4}; \emptyset \vdash \mathsf{jump}\ \mathsf{append}[\langle \mathcal{T}\llbracket \vec{T} \rrbracket \rangle^{\mathsf{ro}}]}\,\text{T-JUMP}$$

We know that $\Psi_0(\mathsf{append}) = \forall[\alpha].(\Gamma_{1.5}) = \forall[\alpha].(\mathsf{r}_1 : \mathsf{BufferMonitor}(\alpha), \mathsf{r}_2 : \alpha)$.

Case P is $x_i(\vec{y}).Q$. By definition of the translation function, we have:

$$
\begin{aligned}
(H, \mathsf{jump}\ l) =& \mathcal{P}^{\vec{x},\Gamma}[\![x_i(\vec{y}).Q]\!] \\
\text{with } H =& H_Q \uplus H', \\
(H_Q, I_Q) =& \mathcal{P}^{\vec{x}\vec{y},\Gamma'}[\![Q]\!], \\
H' =& \{l\colon \Gamma_1\{I_1\}\} \uplus \{l'\colon \Gamma_2\{I_2\}\}, \\
I_1 =& (\mathsf{r}_2 := \mathsf{new}\ 2; I_{1.1}), \\
I_{1.1} =& (\mathsf{r}_2[1] := l'; I_{1.2}), \\
I_{1.2} =& (\mathsf{r}_2[2] := \mathsf{r}_1; I_{1.3}), \\
I_{1.3} =& (\mathsf{share}\ \mathsf{r}_2\ \mathsf{read\text{-}only}; I_{1.4}), \\
I_{1.4} =& (\mathsf{r}_2 := \mathsf{pack}\ \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \exists\alpha.\langle\Gamma_3, \alpha\rangle^{\mathsf{ro}}; I_{1.5}), \\
I_{1.5} =& (\mathsf{r}_1 := \mathsf{r}_1[i]; I_{1.6}), \\
I_{1.6} =& \mathsf{jump}\ \mathsf{remove}[\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}], \\
I_2 =& (\mathcal{E}^{\Gamma'}(\vec{x}, \vec{y}); I_Q), \\
\Gamma_1 =& (\mathsf{r}_1 \colon \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}), \\
\Gamma_2 =& (\mathsf{r}_1 \colon \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 \colon \langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}), \text{ and} \\
\Gamma_3 =& (\mathsf{r}_1 \colon \alpha, \mathsf{r}_2 \colon \langle \mathcal{T}[\![\Gamma'(\vec{y})]\!]\rangle^{\mathsf{ro}}) = (\mathsf{r}_1 \colon \alpha, \mathsf{r}_2 \colon \langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}).
\end{aligned}
$$

Labels l and l' are fresh.

By hypothesis, we have that $\Gamma \vdash x_i(\vec{y}).Q$ and $\mathrm{fn}(x_i(\vec{y}).Q) \subseteq \vec{x}$. Let $\Gamma(x_i) = \hat{}[\vec{T}]$ and $\Gamma' = \Gamma, \vec{y}\colon \vec{T}$. Our goal is to prove that $\exists \Psi$ such that 1) $\forall l \in \mathrm{dom}(\Psi_0).\Psi(l) = \Psi_0(l)$, 2) $\Psi \vdash H$, and 3) $\Psi; \Gamma_1; \emptyset \vdash \mathsf{jump}\ l$.

Since we have that $\Gamma \vdash x_i(\vec{y}).Q$, then $\Gamma' \vdash Q$ and $\Gamma \vdash x_i : \hat{}[\vec{T}]$, rule TV-IN. By the definition of free names $\mathrm{fn}(x_i(\vec{y}).Q) = (\{x_i\} \cup \mathrm{fn}(Q)) \setminus \{\vec{y}\}$. Simplifying the expression

$$
\begin{aligned}
& (\{x_i\} \cup \mathrm{fn}(Q)) \setminus \{\vec{y}\} \subseteq \{\vec{x}\} \\
=& ((\{x_i\} \cup \mathrm{fn}(Q) \setminus \{\vec{y}\}) \cup \{\vec{y}\}) \subseteq \{\vec{x}\} \cup \{\vec{y}\} \\
=& \{x_i\} \cup \mathrm{fn}(Q) \subseteq \{\vec{x}\vec{y}\} \\
=& \mathrm{fn}(Q) \subseteq \{\vec{x}\vec{y}\}
\end{aligned}
$$

By $\Gamma_Q \vdash Q$, $\mathrm{fn}(Q) \subseteq \{\vec{x}\vec{y}\}$, and the induction hypothesis, we have that $\exists \Psi_Q$ such that

(i) $\forall l \in \mathrm{dom}(\Psi_0).\Psi_Q(l) = \Psi_0(l)$,

(ii) $\Psi_Q \vdash H_Q$, and

(iii) $\Psi_Q; \Gamma_1; \emptyset \vdash I_Q$.

Let $\Psi = \Psi_Q, l\colon \Gamma_1, l'\colon \Gamma_2$, with l and l' fresh. Judgement (i) holds, hence, $\forall l_i \in \mathrm{dom}(\Psi_0).\Psi(l_i) = \Psi_0(l_i)$ also holds. By the heap typing rule, if $\Psi \vdash H_Q$ and $\Psi \vdash H'$, then $\Psi \vdash H_Q \uplus H'$. By Proposition 5.2.1, since $\Psi_Q \vdash H_Q$ and $\Psi = \Psi_Q, l\colon \Gamma_1, l'\colon \Gamma_2$, then $\Psi \vdash H_Q$. We are left with proving that $\Psi \vdash H'$ to prove that $\Psi \vdash H$ holds. First, we prove that $\Psi \vdash l\ \Gamma_1\{I_1\}$. By applying the typing rule for heap values, we get

$$\dfrac{(1.1) \qquad |\mathsf{int}, \mathsf{int}| = 2}{\Psi; \Gamma_1; \emptyset \vdash (\mathsf{r}_2 := \mathsf{new}\ 2; I_{1.1})}\ \text{T-NEW}$$

We are left with sequent (1.1) $\Psi; \Gamma_{1.1}; \emptyset \vdash I_{1.1}$, where

$$\Gamma_{1.1} = \Gamma_1\{\mathsf{r}_2\colon \langle \mathsf{int}, \mathsf{int}\rangle\} = (\mathsf{r}_1 \colon \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 \colon \langle \mathsf{int}, \mathsf{int}\rangle)$$

Let $\Gamma_{1.2} = \Gamma_{1.1}\{\mathsf{r}_2\colon \langle \Gamma_2, \mathsf{int}\rangle\} = (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle \Gamma_2, \mathsf{int}\rangle)$. For rule T-LABEL we have $\Psi(l) = \Gamma_2$. For rule T-STOREL $\Gamma_2 \neq \lambda, \langle \_ \rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overbrace{\mathrm{ftv}(\Gamma_2) = \emptyset}^{(1.2.1)}}{\Psi \vdash \Gamma_2}\ \text{T-TYPE}}{\Psi \vdash \Gamma_2 <: \Gamma_2}\ \text{S-REFLEX}}{\Psi; \Gamma_{1.1} \vdash l' : \Gamma_2}\ \text{T-LABEL} \qquad \dfrac{\Gamma_{1.1}(\mathsf{r}_2) = \langle \mathsf{int}, \mathsf{int}\rangle}{\Psi; \Gamma_{1.1} \vdash \mathsf{r}_2 : \langle \mathsf{int}, \mathsf{int}\rangle}\ \text{T-REG} \qquad \overbrace{\Psi; \Gamma_{1.2}; \emptyset \vdash I_{1.2}}^{(1.2)}}{\Psi; \Gamma_{1.1}; \emptyset \vdash (\mathsf{r}_2[1] := l'; I_{1.2})}\ \text{T-STOREL}$$

Sequent $(1.2.1)$ follows.

$$\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}) = \emptyset}\ \text{Prop. 5.2.5} \qquad \dfrac{}{\mathrm{ftv}(\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}) = \emptyset}\ \text{Prop. 5.2.5}}{\mathrm{ftv}(\langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}) \cup \mathrm{ftv}(\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}) = \emptyset}\ \cup}{\mathrm{ftv}(\Gamma_2) = \emptyset}\ \mathrm{ftv}$$

We typecheck instruction sequence $I_{1.2}$ via judgement (1.2). For rule T-STOREL, we have that $\langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}} \neq \lambda, \langle \_ \rangle$.

$$\dfrac{\dfrac{\Gamma_{1.2}(\mathsf{r}_1) = \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}}{\Psi; \Gamma_{1.2} \vdash \mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}}\ \text{T-REG} \quad \dfrac{\Gamma_{1.2}(\mathsf{r}_2) = \langle \Gamma_2, \mathsf{int}\rangle}{\Psi; \Gamma_{1.2} \vdash \mathsf{r}_2 : \langle \Gamma_2, \mathsf{int}\rangle}\ \text{T-REG} \quad (1.3)}{\Psi; \Gamma_{1.2}; \emptyset \vdash (\mathsf{r}_2[2] := \mathsf{r}_1; I_{1.3})}\ \text{T-STOREL}$$

Let $\Gamma_{1.3} = \Gamma_{1.2}\{\mathsf{r}_2\colon \langle \Gamma_2, \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle\} = (\mathsf{r}_1 \colon \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 \colon \langle \Gamma_2, \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle)$. Next, we prove sequent (1.3) $\Psi; \Gamma_{1.3}; \emptyset \vdash I_{1.3}$.

$$\dfrac{\dfrac{\Gamma_{1.3}(\mathsf{r}_2) = \langle \Gamma_2, \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle}{\Psi; \Gamma_{1.3} \vdash \mathsf{r}_2 : \langle \Gamma_2, \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle}\ \text{T-REG} \qquad (1.4)}{\Psi; \Gamma_{1.3}; \emptyset \vdash (\mathsf{share}\ \mathsf{r}_2\ \mathsf{read\text{-}only}; I_{1.4})}\ \text{T-SHARER}$$

Let $\Gamma_{1.4} = \Gamma_{1.3}\{\mathsf{r}_2\colon \langle \Gamma_2, \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}}\} = (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle \Gamma_2, \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}})$. We continue by proving judgement (1.4) $\Psi; \Gamma_{1.4}; \emptyset \vdash I_{1.4}$.

$$\dfrac{(1.5) \quad (1.6) \quad \exists \alpha.\langle \Gamma_3, \alpha\rangle^{\mathsf{ro}} \neq \langle \_ \rangle}{\Psi; \Gamma_{1.4}; \emptyset \vdash (\mathsf{r}_2 := \mathsf{pack}\ \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \exists \alpha.\langle \Gamma_3, \alpha\rangle^{\mathsf{ro}}; I_{1.5})}\ \text{T-MOVE}$$

Let $\Gamma_{1.5} = \Gamma_{1.4}\{\mathsf{r}_2 \colon \exists\alpha.\langle\Gamma_3,\alpha\rangle^{\mathsf{ro}}\} = (\mathsf{r}_1 : \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \exists\alpha.\langle\Gamma_3,\alpha\rangle^{\mathsf{ro}})$. Now we have to prove two judgements: (1.5) $\Psi;\Gamma_{1.4} \vdash \mathsf{pack}\ \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \exists\alpha.\langle\Gamma_3,\alpha\rangle^{\mathsf{ro}}$ and (1.6) $\Psi;\Gamma_{1.5};\emptyset \vdash I_{1.5}$. First, we address sequent (1.5). For rule T-PACK we have that

- $\alpha \notin \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}$, by Proposition 5.2.4,
- $\alpha \notin \Psi$, by change of bound names, and
- $\langle(\mathsf{r}_1 \colon \alpha, \mathsf{r}_2 \colon \langle\mathcal{T}\llbracket\Gamma(\vec{y})\rrbracket\rangle^{\mathsf{ro}}), \alpha\rangle^{\mathsf{ro}} \neq \langle\_\rangle$.

For sequent (1.5) we have the following induction tree.

$$\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}) = \emptyset}\text{Prop. 5.2.5}}{\Psi \vdash \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}}\text{T-TYPE} \qquad \dfrac{\Gamma_{1.4}(\mathsf{r}_2) = \langle\Gamma_3,\alpha\rangle^{\mathsf{ro}}\{\langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}/\alpha\}}{\Psi;\Gamma_{1.4} \vdash \mathsf{r}_2 : \langle\Gamma_3,\alpha\rangle^{\mathsf{ro}}\{\langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}/\alpha\}}\text{T-REG}}{\Psi;\Gamma_{1.4} \vdash \mathsf{pack}\ \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \exists\alpha.\langle\Gamma_3,\alpha\rangle^{\mathsf{ro}}}\text{T-PACK}$$

With respect to rule T-LOADH, since $x_i \in \vec{x}$ (by hypothesis), then

$$\Gamma(\vec{x}) = \Gamma(x_1), \ldots, \Gamma(x_i), \ldots, \Gamma(x_n)$$

We now address judgement (1.6).

$$\dfrac{\dfrac{\Gamma_{1.5}(\mathsf{r}_1) = \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}}{\Psi;\Gamma_{1.5} \vdash \mathsf{r}_1 : \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}}\text{T-REG} \quad (1.7) \quad \dfrac{}{\mathcal{T}\llbracket\Gamma(x_i)\rrbracket \neq \lambda}\text{Prop 5.2.4}}{\Psi;\Gamma_{1.5};\emptyset \vdash (\mathsf{r}_1 := \mathsf{r}_1[i]; I_{1.6})}\text{T-LOADH}$$

Now, we typecheck instruction sequence $I_{1.6}$, using judgement (1.7) $\Psi;\Gamma_{1.6};\emptyset \vdash I_{1.6}$. Let $\Gamma_{1.6} = \Gamma_{1.5}\{\mathsf{r}_1 \colon \mathcal{T}\llbracket\Gamma(x_i)\rrbracket\} = (\mathsf{r}_1 : \mathcal{T}\llbracket\Gamma(x_i)\rrbracket, \mathsf{r}_2 : \exists\alpha.\langle\Gamma_3,\alpha\rangle^{\mathsf{ro}})$, and $\Gamma_{1.7} = (\mathsf{r}_1 : \mathsf{BufferMonitor}(\beta), \mathsf{r}_2 \colon \exists\alpha.\langle(\mathsf{r}_1 \colon \alpha, \mathsf{r}_2 \colon \beta), \alpha\rangle^{\mathsf{ro}})$. By definition of $\mathcal{T}\llbracket\cdot\rrbracket$, we have $\mathcal{T}\llbracket\Gamma(x_i)\rrbracket = \mathsf{BufferMonitor}(\langle\mathcal{T}\llbracket\vec{T}\rrbracket\rangle^{\mathsf{ro}})$. Notice that $\Gamma_{1.7} = \Gamma_{1.6}\{\langle\mathcal{T}\llbracket\vec{T}\rrbracket\rangle^{\mathsf{ro}}/\beta\}$. By definition of ftv and Proposition 5.2.3, we have that

$$\begin{aligned}
&\mathrm{ftv}(\forall[\beta].\Gamma_{1.7})\\
=&\,\mathrm{ftv}(\Gamma_{1.7}) \setminus \{\beta\}\\
=&(\mathrm{ftv}(\mathsf{BufferMonitor}(\beta)) \cup \mathrm{ftv}(\exists\alpha.\langle(\mathsf{r}_1 \colon \alpha, \mathsf{r}_2 \colon \beta), \alpha\rangle^{\mathsf{ro}})) \setminus \{\beta\}\\
=&(\mathrm{ftv}(\mathsf{BufferMonitor}(\beta)) \cup \mathrm{ftv}(\langle(\mathsf{r}_1 \colon \alpha, \mathsf{r}_2 \colon \beta), \alpha\rangle^{\mathsf{ro}}) \setminus \{\alpha\}) \setminus \{\beta\}\\
=&(\mathrm{ftv}(\mathsf{BufferMonitor}(\beta)) \cup (\mathrm{ftv}((\mathsf{r}_1 \colon \alpha, \mathsf{r}_2 \colon \beta)) \cup \mathrm{ftv}(\alpha)) \setminus \{\alpha\}) \setminus \{\beta\}\\
=&(\mathrm{ftv}(\mathsf{BufferMonitor}(\beta)) \cup ((\mathrm{ftv}(\alpha) \cup \mathrm{ftv}(\beta)) \cup \mathrm{ftv}(\alpha)) \setminus \{\alpha\}) \setminus \{\beta\}\\
=&(\mathrm{ftv}(\mathsf{BufferMonitor}(\beta)) \cup ((\{\alpha\} \cup \{\beta\}) \cup \{\alpha\}) \setminus \{\alpha\}) \setminus \{\beta\}\\
=&\,\mathrm{ftv}(\mathsf{BufferMonitor}(\beta)) \setminus \{\beta\}\\
=&\{\beta\} \setminus \{\beta\}\\
=&\emptyset
\end{aligned}$$

Thus, $\mathrm{ftv}(\forall[\beta].\Gamma_{1.7}) = \emptyset$. For rule T-VALAPP, we have $\langle\mathcal{T}\llbracket\vec{T}\rrbracket\rangle^{\mathsf{ro}} \neq \langle\_\rangle$.

$$\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\langle\mathcal{T}\llbracket\vec{T}\rrbracket\rangle^{\mathsf{ro}}) = \emptyset}\;\text{Prop. 5.2.5}}{\Psi \vdash \langle\mathcal{T}\llbracket\vec{T}\rrbracket\rangle^{\mathsf{ro}}}\;\textsc{T-type} \qquad \dfrac{\dfrac{\mathrm{ftv}(\forall[\beta].(\Gamma_{1.7})) = \emptyset}{\Psi \vdash \forall[\beta].(\Gamma_{1.7}) <: \forall[\beta].(\Gamma_{1.7})}\;\textsc{S-reflex}}{\Psi; \Gamma_{1.6} \vdash \mathsf{remove} : \forall[\beta].(\Gamma_{1.7})}\;\textsc{T-label}}{\dfrac{\Psi; \Gamma_{1.6} \vdash \mathsf{remove}[\langle\mathcal{T}\llbracket\vec{T}\rrbracket\rangle^{\mathsf{ro}}] : \Gamma_{1.6}}{\Psi; \Gamma_{1.6}; \emptyset \vdash \mathsf{jump}\ \mathsf{remove}[\langle\mathcal{T}\llbracket\vec{T}\rrbracket\rangle^{\mathsf{ro}}]}\;\textsc{T-jump}}\;\textsc{T-valApp}$$

For asserting that $\Psi(\mathsf{remove}) = \forall[\beta].(\Gamma_{1.7})$, we follow the scheme on the case for the output process, where we prove that $\Psi(\mathsf{append}) = \forall[\alpha].(\Gamma_5)$. Hence, $\Psi \vdash l\, \Gamma_1\{I_1\}$ holds.

We continue by proving $\Psi \vdash \{l'\colon \Gamma_2\{I_2\}\}$. By induction hypothesis, we have that $\Psi_Q; (\mathsf{r}_1 : \langle\mathcal{T}\llbracket\Gamma'(\vec{x}\vec{y})\rrbracket\rangle^{\mathsf{ro}}); \emptyset \vdash I_Q$. Hence, by Proposition 5.2.1, $\Psi; (\mathsf{r}_1 : \langle\mathcal{T}\llbracket\Gamma'(\vec{x}\vec{y})\rrbracket\rangle^{\mathsf{ro}}); \emptyset \vdash I_Q$ holds. By Lemma 5.2.9, we prove $\Psi; \Gamma_2; \emptyset \vdash (\mathcal{E}^{\Gamma'}(\vec{x}, \vec{y}); I_Q)$, where $(\mathcal{E}^{\Gamma'}(\vec{x}, \vec{y}); I_Q)$ is I_2. Given that $\Psi \vdash H'(l)$ and $\Psi \vdash H'(l')$, hence, $\Psi \vdash H'$ (heap typing rule). Since we have $\Psi \vdash H'$ and $\Psi \vdash H_Q$, then, by the heap typing rule, we have that $\Psi \vdash H$.

Finally, we prove 3) $\Psi; \Gamma_1; \emptyset \vdash \mathsf{jump}\ l$, thereby concluding our proof for this case.

$$\dfrac{\dfrac{\Psi(l) = \Gamma_1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}) = \emptyset}\;\text{Prop. 5.2.5}}{\mathrm{ftv}(\Gamma_1) = \emptyset}\;\mathrm{ftv}}{\Psi \vdash \Gamma_1}\;\textsc{T-type}}{\Psi \vdash \Gamma_1 <: \Gamma_1}\;\textsc{S-reflex}}{\Psi; \Gamma_1 \vdash l : \Gamma_1}\;\textsc{T-label}}{\Psi; \Gamma_1; \emptyset \vdash \mathsf{jump}\ l}\;\textsc{T-jump}$$

Case P is $!x_i(\vec{y}).Q$. The proof is similar to the previous case apart from typechecking the following code block:

$$l'\ \Gamma_2\ \{\mathsf{fork}\ l; \mathcal{E}^{\Gamma'}(\vec{x}, \vec{y}); I_Q\}$$

Where $\mathcal{P}^{\vec{x}\vec{y},\Gamma'}\llbracket Q\rrbracket = (H_Q, I_Q)$ and $\Gamma_2 = (\mathsf{r}_1 : \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle\mathcal{T}\llbracket\vec{T}\rrbracket\rangle^{\mathsf{ro}})$. The type inference for this case is

$$\dfrac{\forall i. \Gamma_2(\mathsf{r}_i) \neq \langle\_\rangle \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}) = \emptyset}\;\text{Prop. 5.2.5}}{\Psi \vdash \langle\mathcal{T}\llbracket\Gamma(\vec{x})\rrbracket\rangle^{\mathsf{ro}}}\;\textsc{T-type}}{\Gamma_1 <: \Gamma_2}\;\textsc{S-regFile}}{\Gamma_1 <: \Gamma_2}\;\textsc{S-code}}{\Psi; \Gamma_2 \vdash l : \forall[].(\Gamma_2)}\;\textsc{T-label} \qquad \Psi; (\mathsf{r}_1\colon \langle\mathcal{T}\llbracket\Gamma'(\vec{x}\vec{y})\rrbracket\rangle^{\mathsf{ro}}); \emptyset \vdash I_Q}{\Psi; \Gamma_2; \emptyset \vdash (\mathsf{fork}\ l; \mathcal{E}^{\Gamma'}(\vec{x}, \vec{y}); I_Q)}\;\textsc{T-fork}$$

By induction hypothesis, we have that $\Psi_Q; (\mathsf{r}_1 : \langle\mathcal{T}\llbracket\Gamma'(\vec{x}\vec{y})\rrbracket\rangle^{\mathsf{ro}}); \emptyset \vdash I_Q$. Hence, by Proposition 5.2.1, $\Psi; (\mathsf{r}_1 : \langle\mathcal{T}\llbracket\Gamma'(\vec{x}\vec{y})\rrbracket\rangle^{\mathsf{ro}}); \emptyset \vdash I_Q$ holds. By Lemma 5.2.9, we prove $\Psi; \Gamma_2; \emptyset \vdash (\mathcal{E}^{\Gamma'}(\vec{x}, \vec{y}); I_Q)$.

Case P is $Q_1 \mid Q_2$. By the definition of the translation function we have:

$$\begin{aligned}
(H, I) &= \mathcal{P}^{\vec{x},\Gamma}[\![Q_1 \mid Q_2]\!] \\
\text{with } H &= \{l_{Q_1} \colon \Gamma_1\{I_{Q_1}\}\} \uplus \{l_{Q_2} \colon \Gamma_1\{I_{Q_2}\}\} \uplus H_{Q_1} \uplus H_{Q_2}, \\
I &= (\mathsf{fork}\ l_{Q_1}; \mathsf{fork}\ l_{Q_2}; \mathsf{done}), \\
(H_{Q_1}, I_{Q_1}) &= \mathcal{P}^{\vec{x},\Gamma}[\![Q_1]\!], \\
(H_{Q_2}, I_{Q_2}) &= \mathcal{P}^{\vec{x},\Gamma}[\![Q_2]\!], \text{ and} \\
\Gamma_1 &= (\mathsf{r}_1 \colon \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}).
\end{aligned}$$

By hypothesis, $\Gamma \vdash Q_1 \mid Q_2$ and $\text{fn}(Q_1 \mid Q_2) \subseteq \vec{x}$. Our goal is to prove that $\exists\Psi$ such that

1. $\forall l \in \text{dom}(\Psi_0).\Psi(l) = \Psi_0(l)$,
2. $\Psi \vdash H$, and
3. $\Psi; \Gamma_1; \emptyset \vdash I$.

Since $\Gamma \vdash Q_1 \mid Q_2$, using rule TV-PAR $\Gamma \vdash Q_1$ and $\Gamma \vdash Q_2$. By definition of function fn, we have that $\text{fn}(Q_1) \cup \text{fn}(Q_2) \subseteq \vec{x}$, therefore, we have $\text{fn}(Q_1) \subseteq \vec{x}$ and $\text{fn}(Q_2) \subseteq \vec{x}$. From $\Gamma \vdash Q_1$, $\Gamma \vdash Q_2$, $\text{fn}(Q_1) \subseteq \vec{x}$, and $\text{fn}(Q_2) \subseteq \vec{x}$, we have, by the induction hypothesis, that $\exists\Psi_{Q_1}, \Psi_{Q_2}$ such that

- $\forall l \in \text{dom}(\Psi).\Psi_{Q_1}(l) = \Psi_{Q_2}(l) = \Psi_0(l)$,
- $\Psi_{Q_1} \vdash H_{Q_1}$,
- $\Psi_{Q_2} \vdash H_{Q_2}$,
- $\Psi_{Q_1}; \Gamma_1; \emptyset \vdash I_{Q_1}$, and
- $\Psi_{Q_2}; \Gamma_1; \emptyset \vdash I_{Q_2}$.

Because $\Psi_{Q_1} \vdash H_{Q_1}$ and $\Psi_{Q_1}; \Gamma_1; \emptyset \vdash I_{Q_1}$, then, the heap values rule assures that we have $\Psi_{Q_1} \vdash H_{Q_1} \uplus \{l_{Q_1} \colon \Gamma_1\{I_{Q_1}\}\}$. Using similar arguments, $\Psi_{Q_2} \vdash H_{Q_2} \uplus \{l_{Q_2} \colon \Gamma_1\{I_{Q_2}\}\}$ follows. By construction of the translation function, the labels in $\text{dom}(H)$ are always fresh, thus $\Psi_{Q_1} \cap \Psi_{Q_2} = \emptyset$. Let $\Psi = \Psi_{Q_1} \cup \Psi_{Q_2}, l_{Q_1} : \Gamma_1, l_{Q_2} : \Gamma_1$, with l and l' fresh. By definition of the typing environment, we have that 1) holds. By Proposition 5.2.1 a) $\Psi \vdash H_{Q_2} \uplus \{l_{Q_1} \colon \Gamma_1\{I_{Q_1}\}\}$ and b) $\Psi \vdash H_{Q_2} \uplus \{l_{Q_2} \colon \Gamma_1\{I_{Q_2}\}\}$, entailing that 2) $\Psi \vdash H$ holds, by the heap values rule.

Next we address 3). Our goal is to prove that

$$\Psi; \Gamma_1; \emptyset \vdash (\mathsf{fork}\ l_{Q_1}; \mathsf{fork}\ l_{Q_2}; \mathsf{done})$$

holds. We have that $\Psi(l_{Q_1}) = \forall[].\Gamma_1$.

$$\dfrac{\dfrac{\Gamma_1(\mathsf{r}_1) = \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}}{\Gamma_1(\mathsf{r}_1) \neq \langle\_\rangle} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}) = \emptyset}\,\text{Prop. 5.2.5}}{\mathrm{ftv}(\Gamma_1) = \emptyset}\,\mathrm{ftv}}{\Psi \vdash \Gamma_1}\,\text{T-TYPE}}{\Psi \vdash \Gamma_1 <: \Gamma_1}\,\text{S-REFLEX}}{\Psi; \Gamma_1 \vdash l_{Q_1} : \forall[].\Gamma_1}\,\text{T-LABEL} \qquad (1)}{\Psi; \Gamma_1; \emptyset \vdash (\mathsf{fork}\ l_{Q_1}; \mathsf{fork}\ l_{Q_2}; \mathsf{done})}\,\text{T-FORK}$$

The second part of this proof follows similarly to the first. We have that $\Psi(l_{Q_2}) = \forall[].\Gamma_1$.

$$\dfrac{\Gamma_1(\mathsf{r}_1) \neq \langle\_\rangle \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}) = \emptyset}\,\text{Prop. 5.2.5}}{\mathrm{ftv}(\Gamma_1) = \emptyset}\,\mathrm{ftv}}{\Psi \vdash \Gamma_1}\,\text{T-TYPE}}{\Psi \vdash \Gamma_1 <: \Gamma_1}\,\text{S-REFLEX}}{\Psi; \Gamma_1 \vdash l_{Q_2} : \forall[].\Gamma_1}\,\text{T-LABEL} \qquad \dfrac{}{\Psi; \Gamma_1; \emptyset \vdash \mathsf{done}}\,\text{T-DONE}}{\Psi; \Gamma_1; \emptyset \vdash (\mathsf{fork}\ l_{Q_2}; \mathsf{done})}\,\text{T-FORK}$$

Case P is $(\nu\, y \colon T)\, Q$. By definition of $\mathcal{P}^{\vec{x},\Gamma}[\![\cdot]\!]$ we have that

$$\begin{aligned}
\mathcal{P}^{\vec{x},\Gamma}[\![(\nu\, y \colon T)\, Q]\!] =& (H, I) \\
\text{with } H =& l\ \Gamma_2\ \{I_2\} \uplus H_Q, \\
T =& \hat{}[\vec{T}], \\
\Gamma' =& \Gamma, y \colon T, \text{ and} \\
(H_Q, I_Q) =& \mathcal{P}^{\vec{x}y,\Gamma'}[\![Q]\!].
\end{aligned}$$

Where

$$\begin{aligned}
I_2 &= (\mathsf{r}_3 := \mathsf{new}\ 1; I_{2.1}), \\
I_{2.1} &= (\mathsf{r}_3[1] := \mathsf{r}_2; I_{2.2}), \\
I_{2.2} &= (\mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only}; I_{2.3}), \\
I_{2.3} &= (\mathsf{r}_2 := \mathsf{r}_3; I_{2.4}), \text{ and} \\
I_{2.4} &= (\mathcal{E}^{\Gamma'}(\vec{x}, y); I_Q).
\end{aligned}$$

And

$$\begin{aligned}
I &= (\mathsf{r}_2 := \mathsf{new}\ 2; I_{1.1}),\\
I_{1.1} &= (\mathsf{r}_2[1] := l; I_{1.2}),\\
I_{1.2} &= (\mathsf{r}_2[2] := \mathsf{r}_1; I_{1.3}),\\
I_{1.3} &= (\mathsf{share}\ \mathsf{r}_2\ \mathsf{read\text{-}only}; I_{1.4}),\\
I_{1.4} &= (\mathsf{r}_1 := \mathsf{pack}\ \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathcal{T}[\![T]\!]), \beta\rangle^{\mathsf{ro}}; I_{1.5}), \text{ and}\\
I_{1.5} &= \mathsf{jump}\ \mathsf{createBuffer}[\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}].
\end{aligned}$$

With $\Gamma_1 = (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}})$, $\Gamma_2 = (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \mathcal{T}[\![T]\!])$, and $\Gamma_3 = (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma'(\vec{x}y)]\!]\rangle^{\mathsf{ro}}$. By hypothesis, $\Gamma \vdash (\nu\, y\colon T)\, Q$ and $\mathrm{fn}((\nu\, y\colon T)\, Q) \subseteq \vec{x}$ hold. Also $\Gamma' \vdash Q$, applying both $\Gamma \vdash (\nu\, y\colon T)\, Q$ and rule TV-RES. By the free names definition we have that

$$\begin{aligned}
&\mathrm{fn}((\nu\, y\colon T)\, Q) \subseteq \{\vec{x}\}\\
=&\mathrm{fn}(Q) \setminus \{y\} \subseteq \{\vec{x}\}\\
=&(\mathrm{fn}(Q) \setminus \{y\}) \cup \{y\} \subseteq \{\vec{x}\} \cup \{y\}\\
=&\mathrm{fn}(Q) \subseteq \{\vec{x}y\}
\end{aligned}$$

Hence, by induction hypothesis, we have that $\exists \Psi_Q$ such that

- $\forall l_i \in \mathrm{dom}(\Psi_0).\Psi_Q(l_i) = \Psi_0(l_i)$
- $\Psi_Q \vdash H_Q$
- $\Psi_Q; \Gamma_3; \emptyset \vdash I_Q$.

Our goal is to prove that $\exists \Psi$ such that 1) $\forall l_j \in \mathrm{dom}(\Psi_0).\Psi(l_j) = \Psi_0(l_j)$, 2) $\Psi \vdash H$, and 3) $\Psi; \Gamma_1; \emptyset \vdash I$. Let $\Psi = \Psi_Q, l\colon \Gamma_2$, with l fresh (by definition of the translation function). Since label l is fresh (by hypothesis) and $\forall l_i \in \mathrm{dom}(\Psi_0).\Psi_Q(l_i) = \Psi_0(l_i)$ (by induction hypothesis), then, by definition of the typing environment, 1) holds.

Now we address 2). By Proposition 5.2.1 and by induction hypothesis, if $\Psi_Q \vdash H_Q$, then $\Psi \vdash H_Q$. Judgement $\Psi \vdash \{l\colon \Gamma_2\{I_2\}\}$ holds if $\Psi; \Gamma_2; \emptyset \vdash I_2$, by the heap value typing rule. Let $\Gamma_{2.1} = \Gamma_2\{\mathsf{r}_3\colon \langle\mathsf{int}\rangle\} = (\mathsf{r}_1 : \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \mathcal{T}[\![T]\!], \mathsf{r}_3 : \langle\mathsf{int}\rangle)$. For rule T-STOREL, we have that $\mathcal{T}[\![T]\!] \neq \lambda, \langle\_\rangle$, by Proposition 5.2.4.

$$\dfrac{\dfrac{\dfrac{\Gamma_{2.1}(\mathsf{r}_2) = \mathcal{T}[\![T]\!]}{\Psi; \Gamma_{2.1} \vdash \mathsf{r}_2 : \mathcal{T}[\![T]\!]}\ \text{T-REG} \qquad \dfrac{\Gamma_{2.1}(\mathsf{r}_3) = \langle\mathsf{int}\rangle}{\Psi; \Gamma_{2.1} \vdash \mathsf{r}_3 : \langle\mathsf{int}\rangle}\ \text{T-REG} \qquad (1)}{\Psi; \Gamma_2\{\mathsf{r}_3\colon \langle\mathsf{int}\rangle\}; \emptyset \vdash (\mathsf{r}_3[1] := \mathsf{r}_2; I_{2.2})}\ \text{T-STOREL}}{\Psi; \Gamma_2; \emptyset \vdash (\mathsf{r}_3 := \mathsf{new}\ 1; I_{2.1})}\ \text{T-NEW}$$

The proof proceeds with sequent (1) $\Psi;\Gamma_{2.1};\emptyset \vdash I_{2.1}$. Let $\Gamma_{2.2} = \Gamma_{2.1}\{\mathsf{r}_3\colon \langle \mathcal{T}[\![T]\!]\rangle\} = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\mathcal{T}[\![T]\!], \mathsf{r}_3\colon\langle\mathcal{T}[\![T]\!]\rangle)$.

$$\dfrac{\dfrac{\Gamma_{2.2}(\mathsf{r}_3)=\langle\mathcal{T}[\![T]\!]\rangle}{\Psi;\Gamma_{2.2}\vdash \mathsf{r}_3:\langle\mathcal{T}[\![T]\!]\rangle}\,\text{T-REG}\qquad \overbrace{\Psi;\Gamma_{2.2}\{\mathsf{r}_3\colon\langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}}\};\emptyset\vdash I_{2.3}}^{(2)}}{\Psi;\Gamma_{2.2};\emptyset\vdash(\mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only};I_{2.3})}\,\text{T-SHARER}$$

Let $\Gamma_{2.3} = \Gamma_{2.2}\{\mathsf{r}_3\colon \langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}}\} = (\mathsf{r}_1:\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2:\mathcal{T}[\![T]\!], \mathsf{r}_3:\langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}})$. We prove judgement (2).

$$\dfrac{\dfrac{\Gamma_{2.3}(\mathsf{r}_3)=\langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}}}{\Psi;\Gamma_{2.3}\vdash \mathsf{r}_3:\langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}}}\,\text{T-REG}\qquad (3)\qquad \langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}}\neq\langle\_\rangle}{\Psi;\Gamma_{2.3};\emptyset\vdash(\mathsf{r}_2:=\mathsf{r}_3;I_{2.4})}\,\text{T-MOVE}$$

We are left with proving judgement (3)

$$\Psi;(\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}},\mathsf{r}_2\colon\langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}},\mathsf{r}_3\colon\langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}});\emptyset\vdash(\mathcal{E}^{\Gamma'}(\vec{x},y);I_Q)$$

By induction hypothesis we have that $\Gamma'\vdash Q$ and that $\Gamma'=\Gamma,y\colon T$, thus $\langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}} = \langle\mathcal{T}[\![\Gamma'(y)]\!]\rangle^{\mathsf{ro}}$ and $\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}} = \langle\mathcal{T}[\![\Gamma'(\vec{x})]\!]\rangle^{\mathsf{ro}}$ hold. So, we can rewrite sequent (3) as

$$\Psi;(\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma'(\vec{x})]\!]\rangle^{\mathsf{ro}},\mathsf{r}_2\colon\langle\mathcal{T}[\![\Gamma'(y)]\!]\rangle^{\mathsf{ro}},\mathsf{r}_3\colon\langle\mathcal{T}[\![\Gamma'(y)]\!]\rangle^{\mathsf{ro}});\emptyset\vdash(\mathcal{E}^{\Gamma'}(\vec{x},y);I_Q)$$

By Lemma 5.2.9, sequent (3) holds if we show $\Psi;\Gamma_3;\emptyset\vdash I_Q$. By induction hypothesis, we have that $\Psi_Q;\Gamma_3;\emptyset\vdash I_Q$. Then, by Proposition 5.2.2, sequent (3) holds. Hence, $\Psi\vdash\{l\colon\Gamma_2\{I_2\}\}$ holds. By the heap typing rule, if $\Psi\vdash H_Q$ and $\Psi\vdash\{l\colon\Gamma_2\{I_2\}\}$, then $\Psi\vdash H$.

We now prove 3). Let $\Gamma_{1.1}=\Gamma_1\{\mathsf{r}_2\colon\langle\mathsf{int},\mathsf{int}\rangle\} = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}},\mathsf{r}_2\colon\langle\mathsf{int},\mathsf{int}\rangle)$. We have that $\Psi(l)=\Gamma_2$. Notice that on rule T-STOREL we have that $\Gamma_2\neq\lambda,\langle\_\rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\mathcal{T}[\![\Gamma(\vec{x})]\!])=\emptyset}\,\text{Prop. 5.2.5}}{\mathrm{ftv}(\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}})=\emptyset}\qquad\dfrac{}{\mathrm{ftv}(\mathcal{T}[\![T]\!])=\emptyset}\,\text{Prop. 5.2.4}}{\mathrm{ftv}(\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}})\cup\mathrm{ftv}(\mathcal{T}[\![T]\!])=\emptyset}\,\cup}{\mathrm{ftv}(\Gamma_2)=\emptyset}\,\mathrm{ftv}}{\Psi\vdash\Gamma_2}\,\text{T-TYPE}}{\Psi\vdash\Gamma_2<:\Gamma_2}\,\text{S-REFLEX}}{\Psi;\Gamma_{1.1}\vdash l:\Gamma_2}\,\text{T-LABEL}\qquad\dfrac{\Gamma_{1.1}(\mathsf{r}_2)=\langle\mathsf{int},\mathsf{int}\rangle}{\Psi;\Gamma_{1.1}\vdash\mathsf{r}_2:\langle\mathsf{int},\mathsf{int}\rangle}\,\text{T-REG}\qquad(4)}{\dfrac{\Psi;\Gamma_1\{\mathsf{r}_2\colon\langle\mathsf{int},\mathsf{int}\rangle\};\emptyset\vdash(\mathsf{r}_2[1]:=l;I_{1.2})}{\Psi;\Gamma_1;\emptyset\vdash(\mathsf{r}_2:=\mathsf{new}\ 2;I_{1.1})}\,\text{T-NEW}}\,\text{T-STOREL}$$

Next, we prove sequent (4) $\Psi;\Gamma_{1.2};\emptyset \vdash I_{1.2}$, where $\Gamma_{1.2} = \Gamma_{1.1}\{\mathsf{r}_2\colon \langle\Gamma_2,\mathsf{int}\rangle\} = (\mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2 \colon \langle\Gamma_2,\mathsf{int}\rangle)$. Again, for rule T-STOREL, we have that $\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}} \neq \lambda, \langle\_\rangle$.

$$\dfrac{\dfrac{\Gamma_{1.2}(\mathsf{r}_1) = \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}}{\Psi;\Gamma_{1.2}\vdash \mathsf{r}_1 : \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}}\,\text{T-REG} \qquad \dfrac{\Gamma_{1.2}(\mathsf{r}_2) = \langle\Gamma_2,\mathsf{int}\rangle}{\Psi;\Gamma_{1.2}\vdash \mathsf{r}_2 : \langle\Gamma_2,\mathsf{int}\rangle}\,\text{T-REG} \qquad (5)}{\Psi;\Gamma_{1.2};\emptyset\vdash (\mathsf{r}_2[2] := \mathsf{r}_1; I_{1.3})}\,\text{T-STOREL}$$

Let

$$\begin{aligned}
\Gamma_{1.3} =& \Gamma_{1.2}\{\mathsf{r}_2\colon \langle\Gamma_2,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle\} = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\langle\Gamma_2,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle)\\
\Gamma_{1.4} =& \Gamma_{1.3}\{\mathsf{r}_2\colon \langle\Gamma_2,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}}\} = (\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\langle\Gamma_2,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}})
\end{aligned}$$

Sequent (5) is $\Psi;\Gamma_{1.3};\emptyset\vdash I_{1.3}$, which we typecheck as follows.

$$\dfrac{\dfrac{\Gamma_{1.3}(\mathsf{r}_2) = \langle\Gamma_2,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle}{\Psi;\Gamma_{1.3}\vdash \mathsf{r}_2 : \langle\Gamma_2,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle}\,\text{T-REG} \qquad \overbrace{\Psi;\Gamma_{1.4};\emptyset\vdash I_{1.4}}^{(6)}}{\Psi;\Gamma_{1.3};\emptyset\vdash (\mathsf{share}\ \mathsf{r}_2\ \mathsf{read\text{-}only}; I_{1.4})}\,\text{T-SHARER}$$

We have that $T = \hat{}[\vec{T}]$. Let $\Gamma_3 = \Gamma_2\{\beta/\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\} = (\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathcal{T}[\![T]\!])$. We are left with sequent (6).

$$\dfrac{(7) \qquad \overbrace{\Psi;\Gamma_{1.5};\emptyset\vdash \mathsf{jump}\ \mathsf{createBuffer}[\langle\mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}]}^{(8)} \qquad \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}} \neq \langle\_\rangle}{\Psi;\Gamma_{1.4};\emptyset\vdash (\mathsf{r}_1 := \mathsf{pack}\ \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \exists\beta.\langle\Gamma_3,\beta\rangle^{\mathsf{ro}}; I_{1.5})}\,\text{T-MOVE}$$

Sequent (7) is $\Psi;\Gamma_{1.4}\vdash \mathsf{pack}\ \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \exists\beta.\langle\Gamma_3,\beta\rangle^{\mathsf{ro}}$. We have that

$$\langle\Gamma_2,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}} \neq \langle\_\rangle$$

and

$$\begin{aligned}
&\langle\Gamma_2,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}} =\\
=&\langle(\mathsf{r}_1\colon\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\colon\mathcal{T}[\![T]\!]), \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}} =\\
=&\langle\underbrace{(\mathsf{r}_1\colon\beta, \mathsf{r}_2\colon\mathcal{T}[\![T]\!])}_{\Gamma_3}, \beta\rangle^{\mathsf{ro}}\{\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}/\beta\}
\end{aligned}$$

Sequent (7) follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\mathcal{T}[\![\Gamma(\vec{x})]\!]) = \emptyset}\,\text{Prop. 5.2.5}}{\mathrm{ftv}(\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}) = \emptyset}\,\mathrm{ftv}}{\Psi\vdash\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}}\,\text{T-TYPE} \qquad \dfrac{\Gamma_{1.4}(\mathsf{r}_2) = \langle\Gamma_1,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}}}{\Psi;\Gamma_{1.4}\vdash \mathsf{r}_2 : \langle\Gamma_1,\langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}}}\,\text{T-REG} \qquad (7.1)}{\Psi;\Gamma_{1.4}\vdash \mathsf{pack}\ \langle\mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \mathsf{r}_2\ \mathsf{as}\ \exists\beta.\langle\Gamma_3,\beta\rangle^{\mathsf{ro}}}\,\text{T-PACK}$$

Sequent (7.1) is $\beta \notin \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}, \Psi$. By Proposition 5.2.4, $\beta \notin \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}$ and by change of bound names, $\beta \notin \Psi$. Let $\Gamma_{1.5}$ and $\Gamma'_{1.5}$ be

$$\begin{aligned}
\Gamma_{1.5} =& \Gamma_{1.4}\{\mathsf{r}_1\colon \exists\beta.\langle \Gamma_3, \beta\rangle^{\mathsf{ro}}\} = (\mathsf{r}_1 : \exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathcal{T}[\![T]\!]), \beta\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle \Gamma_2, \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}})\\
\Gamma'_{1.5} =& \Gamma_{1.5}\{\alpha/\langle\mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}}\}\\
\Gamma'_{1.5} =& (\mathsf{r}_1 : \exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}}, \mathsf{r}_2 : \langle \Gamma_2, \langle \mathcal{T}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\rangle^{\mathsf{ro}})\\
&\quad \text{where } \mathcal{T}[\![T]\!] = \mathsf{BufferMonitor}(\langle \mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}})
\end{aligned}$$

Next, we typecheck sequent (8).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{ftv}(\mathcal{T}[\![\vec{T}]\!]) = \emptyset}\ \text{Prop. 5.2.5}}{\mathrm{ftv}(\langle\mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}) = \emptyset}\ \mathrm{ftv}}{\Psi \vdash \langle\mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}}\ \textsc{T-type} \qquad (9) \qquad \langle\mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}} \neq \langle\_\rangle}{\Psi; \Gamma_{1.5} \vdash \mathsf{createBuffer}[\langle\mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}] : \Gamma_{1.5}}\ \textsc{T-valApp}}{\Psi; \Gamma_{1.5}; \emptyset \vdash \mathsf{jump}\ \mathsf{createBuffer}[\langle\mathcal{T}[\![\vec{T}]\!]\rangle^{\mathsf{ro}}]}\ \textsc{T-jump}$$

We have that $\Psi(\mathsf{createBuffer}) = \forall[\alpha].(\mathsf{r}_1 : \exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}})$. By definition of ftv,

$$\begin{aligned}
&\mathrm{ftv}(\exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}})\\
=& \mathrm{ftv}(\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}}) \setminus \{\beta\}\\
=& (\mathrm{ftv}((\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha))) \cup \mathrm{ftv}(\beta))) \setminus \{\beta\}\\
=& ((\mathrm{ftv}(\beta) \cup \mathrm{ftv}(\mathsf{BufferMonitor}(\alpha))) \cup \mathrm{ftv}(\beta)) \setminus \{\beta\}\\
=& ((\{\beta\} \cup \mathrm{ftv}(\mathsf{BufferMonitor}(\alpha))) \cup \{\beta\}) \setminus \{\beta\}\\
=& \mathrm{ftv}(\mathsf{BufferMonitor}(\alpha))
\end{aligned}$$

By Proposition 5.2.3 and by the definition of ftv, we have that

$$\mathrm{ftv}(\mathsf{BufferMonitor}(\alpha)) = \mathrm{ftv}(\alpha) = \{\alpha\}$$

Hence, $\mathrm{ftv}(\exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}}) = \{\alpha\}$. Finally, we prove sequent (9) $\Psi; \Gamma_{1.5} \vdash \mathsf{createBuffer} : \forall[\alpha].(\Gamma'_{1.5})$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(\Psi, \alpha :: \mathsf{TyVar})(\alpha) = \alpha :: \ \mathrm{kind}(\alpha)\\ \mathrm{ftv}(\exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}}) = \{\alpha\}\end{array}}{\Psi, \alpha :: \mathsf{TyVar} \vdash \exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}}}\ \textsc{T-type}}{\Psi, \alpha :: \mathsf{TyVar} \vdash \mathsf{r}_1 : \exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}} <: \Gamma_{1.5}}\ \textsc{S-regFile}}{\Psi \vdash \forall[\alpha].(\mathsf{r}_1 : \exists\beta.\langle(\mathsf{r}_1 : \beta, \mathsf{r}_2 : \mathsf{BufferMonitor}(\alpha)), \beta\rangle^{\mathsf{ro}}) <: \forall[\alpha].(\Gamma_{1.5})}\ \textsc{S-code}}{\Psi; \Gamma_{1.5} \vdash \mathsf{createBuffer} : \forall[\alpha].(\Gamma'_{1.5})}\ \textsc{T-label}$$

With judgement (9) proved, judgement

$$\Psi; \Gamma_1; \emptyset \vdash I$$

holds, thus we conclude our proof. □

Theorem 5.2.11 is our main result, which warrants that the top-level translation of a well-typed π-process produces a type-correct MIL program.

Theorem 5.2.11. *If* $\emptyset \vdash P$*, then* $\exists \Psi$ *such that* $\forall l_i \in \mathrm{dom}(\Psi_0).\Psi(l_i) = \Psi_0(l_i)$ *and* $\Psi \vdash \mathcal{P}[\![P]\!]$.

Proof. By Corollary 2.2.8, since we have $\emptyset \vdash P$, then $\mathrm{fn}(P) = \emptyset$. By definition of $\mathcal{P}[\![\cdot]\!]$,

$$\begin{aligned} \mathcal{P}[\![P]\!] =& \mathsf{main}()\{\mathcal{E}^{\emptyset}(\emptyset, \emptyset); I_P\} \uplus H_P \\ & \text{where } (H_P, I_P) = \mathcal{P}^{\emptyset,\emptyset}[\![P]\!] \end{aligned}$$

Since we have $\emptyset \vdash P$ and $\mathrm{fn}(P) = \emptyset$, hence, by Lemma 5.2.10, we have that $\exists \Psi$ such that

- $\forall l_j \in \mathrm{dom}(\Psi_0).\Psi_P(l_j) = \Psi_0(l_j)$,
- $\Psi_P \vdash H_P$, and
- $\Psi_P; (\mathsf{r}_1 : \langle\rangle^{\mathsf{ro}}); \emptyset \vdash I_P$.

Let $\Psi = \Psi_P, \mathsf{main} : ()$. We need to prove that 1) $\forall l_i \in \mathrm{dom}(\Psi_0).\Psi(l_i) = \Psi_0(l_i)$ and 2) $\Psi \vdash \mathcal{P}[\![P]\!]$. By construction, we know that $\mathsf{main} \notin \mathrm{dom}(\Psi_P)$, hence, 1) holds. By the typing rule for heap values, if $\Psi \vdash H_P$ and $\Psi \vdash \{\mathsf{main}\colon ()\ \{\mathcal{E}^{\emptyset}(\emptyset, \emptyset); I_P\}\}$, then $\Psi \vdash \mathcal{P}[\![P]\!]$ holds. Since $\Psi_P \vdash H_P$ and $\mathsf{main} \notin \mathrm{dom}(\Psi_P)$, then, by Proposition 5.2.1, $\Psi \vdash H_P$ holds. We just need to show $\Psi \vdash \{\mathsf{main}\colon ()\ \{\mathcal{E}^{\emptyset}(\emptyset, \emptyset); I_P\}\}$. If $\Psi_P; (\mathsf{r}_1 : \langle\rangle^{\mathsf{ro}}); \emptyset \vdash I_P$, then, by Proposition 5.2.1, $\Psi; (\mathsf{r}_1 : \langle\rangle^{\mathsf{ro}}); \emptyset \vdash I_P$. By definition of $\mathcal{E}^{\cdot}(\cdot, \cdot)$, we have $(\mathcal{E}^{\emptyset}(\emptyset, \emptyset); I_P) = (\mathsf{r}_3 := \mathsf{new}\ 0; \mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only}; \mathsf{r}_1 := \mathsf{r}_3; I_P)$, then

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Psi; (\mathsf{r}_3 : \langle\rangle) \vdash \mathsf{r}_3 : \langle\rangle}\;\text{T-REG} \qquad \Psi; (\mathsf{r}_3 : \langle\rangle^{\mathsf{ro}}); \emptyset \vdash I_Q \qquad \langle\rangle^{\mathsf{ro}} \neq \langle\_\rangle}{\Psi; (\mathsf{r}_3 : \langle\rangle^{\mathsf{ro}}); \emptyset \vdash (\mathsf{r}_1 := \mathsf{r}_3; I_P)}\;\text{T-MOVE}}{\Psi; (\mathsf{r}_3 : \langle\rangle); \emptyset \vdash (\mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only}; \mathsf{r}_1 := \mathsf{r}_3; I_P)}\;\text{T-SHARER}}{\Psi; \emptyset; \emptyset \vdash (\mathsf{r}_3 := \mathsf{new}\ 0; \mathsf{share}\ \mathsf{r}_3\ \mathsf{read\text{-}only}; \mathsf{r}_1 := \mathsf{r}_3; I_P)}\;\text{T-NEW}$$

Hence, 2) $\Psi \vdash \mathcal{P}[\![P]\!]$ holds.

□

Chapter 6

Conclusion

Summary. This work presents a type-preserving translation from the π-calculus into MIL that is supported in all its extent by [10]. We depart from the π-calculus, a process algebra where communication is the elementary computation step, and process mobility expresses dynamic reconfiguration of the network of processes. We arrive at (compile into) MIL, a language for an abstract machine that consists of multiple cores (processors) and a main shared memory.

MIL is a multithreaded typed assembly language with a powerful type system that enforces both race-freedom and deadlock-freedom [47, 48]. The threading model is co-operative, meaning that threads must explicitly relinquish the processor resource. The language's primitive synchronisation mechanism is locks. MIL also enforces a safe usage over locks.

The compilation is not direct, nor trivial: certain abstractions, like channels and name binding, have no definite representation in MIL. To support the translation, we devise, in the target language, a library of polymorphic unbounded buffers that act as π-channels. These unbounded buffers are Hoare's monitors [20], which introduce a form of synchronisation for threads accessing the buffer, and encapsulate direct lock manipulation. The lock discipline of MIL allows for a safe transfer of the monitor's critical region that goes from the signalling thread, which finishes, to the thread waiting in a condition, which activates. Unbounded buffers are an effective way to represent channels that simplifies the translation: sending a message corresponds to placing an element in the buffer, and receiving a message amounts to removing a message from the buffer. We impose a FIFO ordering to the buffer, in order to make sure sent messages are eventually received.

The translation function is a formal specification of the compiler. The compilation of the π-calculus into MIL comprises the translation of types, of values, and of processes. Type-preserving compilers give confidence in terms of safety (generated programs will not get stuck) and also in terms of partial correctness (semantic properties given by types are preserved). Our main result is a type-preserving translation [31]: the compiler produces type-correct MIL programs from well-typed (closed) π-processes. Another concern of code generation is ensuring that the translated processes are concurrent, which includes the dynamic creation of threads and the synchronisation between threads.

The contributions of this work are:

- A type-preserving compilation algorithm from the π-calculus into MIL that witnesses the flexibility of the target language in a typed (hence race-free) scenario.
- MIL programming examples and data structures. We show the implementation of polymorphic unbounded buffer monitors, generic condition variables, and polymorphic queues. We also describe how to encode a monitor in MIL.
- Tools. We created a prototype for the π-to-MIL compiler that consists of: the parsers, the typechecker, and the code generator. We refined the MIL typechecker, and refined the MIL interpreter, by adding support for universal and existential types, readers/writers and linear locks, and local tuples. We also implemented a Java applet that quickly showcases our work without installation (on browsers that support Java applets): we can compile our version of the π-calculus into MIL, edit the generated MIL code, and execute it. All of our work is available on-line [26].

Related work. As related work, we take into analysis Pict, a compiler that translates from the π-calculus into C; a type-preserving compiler that targets a typed assembly language; an abstract machine for a process calculus; and a multithreaded virtual machine that is the target of a process calculus.

Pict [45] is a compiler from the π-calculus into C. Turner defines an abstract machine encoded in the π-calculus to represent the run-time behaviour of translated C programs and proves this abstract machine is correct with respect to the source language. Afterwards, Turner refines the abstract machine to improve its efficiency and approximate it to the behaviour of the generated code (*e.g.*, uses an environment machine to represent name binding, instead of using substitution). In Pict, there is no concurrency at run-time. Processes are executed by a sequential scheduler. Contrary to Pict, our compiler produces concurrent (multithreaded) code. Variable binding is also very different: Pict uses the variable binding of C, since there is no support for closures in C, the environment of a process must be manually created. MIL represents name binding in a tuple that indexes all names known by a process at a given time. The π-calculus version of Pict is richer than the one we use, having support for recursive types, polymorphism, and type inference. The type system present in MIL features polymorphism and recursive types, so we believe that a type preserving translation from the polymorphic π-calculus into MIL is possible. The focus of our work is to show a type-preserving translation in a concurrent setting, not to show a translation from a polymorphic language into a typed assembly language, like [31]. In Pict there is concerns about memory usage; MIL abstracts these concerns.

Greg Morrisett *et al.* present a type-preserving translation from System F [17] into TAL (a typed assembly language) in five compilation stages [31]. The first compilation stage converts code into the continuation-passing style (or closure-passing style). In the second compilation stage, called closure conversion, the translation separates program code from data, thereby rewriting functions to expect one additional argument (their free variables). The third compilation step is called hoisting and places nested functions in the root level. The forth compilation step makes memory allocation explicit. The final translation step performs mostly syntactic translations, *e.g.*, converts from variables into registers.

A key difference from [31] to our work is that there is no concurrency in System-F nor in TAL. The compiler we present does not perform the continuation-passing style conversion, since the π-calculus has no call-and-return constructs, *i.e.,* the control flow is already explicitly passed via channels (continuations). The third, forth, and fifth stages are present in the translation from the π-calculus into MIL. For example, we have closure conversion whenever we find threads passing continuations to other threads. We perform explicit memory allocation, like in the forth stage. Our approach towards name binding is to enclose the free names in a tuple. Morrisett's compiler expects unlimited registers and represents each variable with a different register.

Our compiler is inspired by the work of Lopes *et al.* that presents an abstract machine for running multithreaded code [24] (TTyCO) generated from the process calculus TyCO [46]. The target language is register based, multithreaded, and features queues that represent lightweight channels, *i.e.,* a thread may enqueue messages and continuations that are dequeued (reduced) whenever a message and a continuation are present at the same time in the same queue. This data structure is also present in Turner's abstract machine and is called channel queue. There are key differences between the compiler we present and this work. First, they do not present a translation function, instead just sketch a compilation algorithm. Second, TTyCO lays at a higher-level than MIL (*e.g.*, there are no queues in the latter). Third, TTyCO is untyped and with no safety properties, therefore no type-preservation result is possible.

Paulino *et al.* devised a virtual-machine [34] that executed TyCOIL, still an untyped language. This work is based on the specification of the abstract machine described in [24]. One immediate difference between TTyCO and TyCOIL is that the latter features locks as a synchronisation primitive. MIL is the evolution of TyCOIL: queues disappear, frames turn into tuples, and the language becomes strongly typed. Continuing the work on multithreaded intermediate languages, the work by Vasconcelos *et al.* extends MIL's type system to check that programs are exempt from deadlocks at compile time [48].

Future work. We separate future work into two stances: more theoretical results about the translation function and MIL extensions. The type system of the source language is simpler than the target language. We would like to understand how much more could we enrich the type system of the π-calculus without altering MIL's type system or loosing the type-preserving result. We also plan to prove the correctness of translation. We would like to understand the completeness of the translation function, *i.e.,* to figure out which behaviours of π-processes are left out.

Related to extending the target language, we envision two points. First, to develop a version equipped with a compare-and-swap primitive rather than locks, allowing in particular to obtain a wait-free implementation of queues. Second, to elaborate a model that closely adheres more to multicore processors as we know them, in particular adding support for memory hierarchies and forgoing the direct allocation of arbitrary-length tuples directly on registers.

Appendix A

Example of code generation

The following MIL program is the outcome of

$$\begin{aligned}&\mathcal{P}[\![(\nu\, printInt\colon {}^\wedge[\mathsf{int}])\,(\nu\, echo\colon {}^\wedge[\mathsf{int},{}^\wedge[\mathsf{int}]])\\ &\quad(\overline{echo}\langle 10, printInt\rangle \mid !echo(msg,reply).\overline{reply}\langle msg\rangle)]\!] =\end{aligned}$$

```
l1 (r1 :⟨⟩^ro , r2 :BufferMonitor(⟨ int ⟩^ro )) {
  r3 := new 1
  r3 [1] := r2
  share r3 read-only
  r2 := r3
  r3 := new 1 −− E([],[printInt ])
  r4 := r2 [1]
  r3 [1] := r4
  share r3 read-only
  r1 := r3
  r2 := new 2 −− Translation of new echo:ˆ[int, ˆ[ int ]]
              −− (echo⟨10, printInt⟩ | !echo(msg, reply).reply⟨msg⟩)
  r2 [1] := l2
  r2 [2] := r1
  share r2 read-only
  r1 := pack ⟨BufferMonitor(⟨int⟩^ro ) ⟩^ro , r2
          as createContinuation(⟨int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro )
  jump createBuffer[⟨int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ]
}
l2 (r1 :⟨BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ,
    r2 :BufferMonitor(⟨ int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro )) {
  r3 := new 1
  r3 [1] := r2
  share r3 read-only
  r2 := r3
  r3 := new 2 −− E([printInt ],[echo])
  r4 := r1 [1]
  r3 [1] := r4
  r4 := r2 [1]
  r3 [2] := r4
  share r3 read-only
  r1 := r3
  fork l3   -- Translation of echo⟨10, printInt⟩
            -- | !echo(msg, reply).reply⟨msg⟩
  fork l4
  done
}
l3 (r1 :⟨BufferMonitor(⟨ int ⟩^ro ),
        BufferMonitor(⟨ int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ) ⟩^ro ) {
  r2 := new 2 -- Translation of echo⟨10, printInt⟩
  r3 := 10
  r2 [1] := r3
  r3 := r1 [1] -- printInt
  r2 [2] := r3
  share r2 read-only
  r1 := r1 [2]
  jump append[⟨int, BufferMonitor(⟨int ⟩^ro ) ⟩^ro ]
}
l4 (r1 :⟨BufferMonitor(⟨ int ⟩^ro ),
        BufferMonitor(⟨ int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ) ⟩^ro ) {
  jump l5 -- Translation of !echo(msg, reply).reply⟨msg⟩
}
l5 (r1 :⟨BufferMonitor(⟨ int ⟩^ro ),
        BufferMonitor(⟨ int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ) ⟩^ro ) {
  r2 := new 2
  r2 [1] := l6
  r2 [2] := r1
  share r2 read-only
  r2 := pack ⟨BufferMonitor(⟨int ⟩^ro ),
               BufferMonitor(⟨ int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ) ⟩^ro ,
             r2 as removeContinuation(⟨int, BufferMonitor(⟨ int ⟩^ro ) ⟩^ro )
  r1 := r1 [2] -- load 'echo'
  jump remove[⟨int, BufferMonitor(⟨int ⟩^ro ) ⟩^ro ]
}
l6 (r1 :⟨BufferMonitor(⟨ int ⟩^ro ),
        BufferMonitor(⟨ int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ) ⟩^ro ,
    r2 :⟨ int , BufferMonitor(⟨ int ⟩^ro ) ⟩^ro ) {
  fork l5
  r3 := new 4 -- E([printInt, echo],[msg, reply])
  r4 := r1 [1]
  r3 [1] := r4
  r4 := r1 [2]
  r3 [2] := r4
  r4 := r2 [1]
  r3 [3] := r4
  r4 := r2 [2]
  r3 [4] := r4
  share r3 read-only
  r1 := r3
  r2 := new 1 −− Translation of reply⟨msg⟩
  r3 := r1 [3] −− msg
  r2 [1] := r3
  share r2 read-only
  r1 := r1 [4]
  jump append[⟨int⟩^ro]
}
main () {
  r3 := new 0 −− E([],[])
  share r3 read-only
  r1 := r3
  r2 := new 2 −− Translation of new printInt :ˆ[ int ]
              −− new echo:ˆ[int, ˆ[ int ]]
              −− (echo⟨10, printInt⟩ | !echo(msg, reply).reply⟨msg⟩)
  r2 [1] := l1
  r2 [2] := r1
  share r2 read-only
  r1 := pack ⟨⟩^ro, r2 as createContinuation(⟨int⟩^ro )
  jump createBuffer[⟨int⟩^ro ]
}
```

Bibliography

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1991.

[2] Hendrik P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[3] BEE2 Website. `http://bee2.eecs.berkeley.edu/`.

[4] Andrew Birrell. An Introduction to Programming with Threads. Technical Report 35, Digital Systems Research Center, Palo Alto, California, 1989.

[5] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.

[6] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of OOPSLA'02*, pages 211–230. ACM, 2002.

[7] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of OOPSLA'01*, pages 56–69. ACM, 2001.

[8] Bryan Cantrill and Jeff Bonwick. Real-world Concurrency. *Queue*, 6(5):16–25, 2008.

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

[10] Tiago Cogumbreiro, Francisco Martins, and Vasco T. Vasconcelos. Compiling the pi-calculus into a multithreaded typed assembly language. *ENTCS*, 241:57–84, 2009.

[11] Ole-Johan Dahl. A Note on Monitor Versions: an Essay in the Honour of C. A. R. Hoare. In Jim Davis, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, Cornerstones of Computing, pages 91–98. PALGRAVE, 2000.

[12] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proceedings of ICFP'05*, pages 254–267. ACM Press, 2005.

[13] Cormac Flanagan and Martin Abadi. Object Types against Races. In *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 288–303. Springer, 1999.

[14] Cormac Flanagan and Martin Abadi. Types for Safe Locking. In *Proceedings of ESOP'99*, volume 1576 of *LNCS*, pages 91–108. Springer, 1999.

[15] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[16] Cormac Flanagan and Stephen N. Freund. Type Inference Against Races. In *Proceedings of SAS'04*, volume 3148 of *LNCS*, pages 116–132. Springer, 2004.

[17] Jean-Yves Girard. The System F of Variable Types, Fifteen Years Later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[18] Dan Grossman. Type-Safe Multithreading in Cyclone. In *Proceedings of TLDI'03*, volume 38(3) of *SIGPLAN Notices*, pages 13–25. ACM, 2003.

[19] Jr. Guy Lewis Steele. RABBIT: A Compiler for SCHEME. Master's thesis, MIT AI Lab, 1978.

[20] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[21] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In *Proceedings of ECOOP'91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[22] Futoshi Iwama and Naoki Kobayashi. A new type system for JVM lock primitives. In *Proceedings of ASIA-PEPM'02*, pages 71–82. ACM, 2002.

[23] Cosimo Laneve. A type system for JVM threads. *Journal of Theoretical Computer Science*, 290(1):741–778, 2003.

[24] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. Compiling Process Calculi. DCC 98–3, DCC-FC & LIACC, Universidade do Porto, March 1998.

[25] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. A Virtual Machine for the TyCO Process Calculus. In *Proceedings of PPDP'99*, volume 1702 of *LNCS*, pages 244–260. Springer, 1999.

[26] MIL Website. `http://gloss.di.fc.ul.pt/mil/`.

[27] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[28] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I/II. *Journal of Information and Computation*, 100:1–77, 1992.

[29] Greg Morrisett. Typed Assembly Language. In *Advanced Topics in Types and Programming Languages*, pages 137–176. MIT Press, 2005.

[30] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Proceedings of CSSS'99*, pages 25–35, 1999.

[31] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programing Language and Systems*, 21(3):527–568, 1999.

[32] George C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[33] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[34] Hervé Paulino, Pedro Marques, Luís Lopes, Vasco T. Vasconcelos, and Fernando Silva. A Multi-Threaded Asynchronous Language. In *Proceedings of PaCT'03*, volume 2763 of *LNCS*, pages 316–323. Springer, 2003.

[35] Benjamin C. Pierce. *Types And Programming Languages*. MIT Press, 2002.

[36] Benjamin C. Pierce. *Advanced Topics In Types And Programming Languages*. MIT Press, 2004.

[37] Benjamin C. Pierce and David N. Turner. Object-Oriented Programming Without Recursive Types. In *Proceedings of POPL'93*, pages 299–312. ACM Press, 1993.

[38] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing, pages 455–494. MIT Press, 2000.

[39] RAMP Website. `http://ramp.eecs.berkeley.edu/`.

[40] Norman Ramsey and Simon P. Jones. Featherweight concurrency in a portable assembly language, 2000.

[41] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[42] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[43] Nir Shavit. Technical perspective transactions are tomorrow's loads and stores. *Communications of the ACM*, 51(8):90–90, 2008.

[44] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: a type-directed, optimizing compiler for ML. *SIGPLAN Notices*, 39(4):554–567, 2004.

[45] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, LFCS, University of Edinburgh, 1996.

[46] Vasco T. Vasconcelos. Processes, functions, and datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[47] Vasco T. Vasconcelos and Francisco Martins. A Multithreaded Typed Assembly Language. In *Proceedings of TV'06*, pages 133–141, 2006.

[48] Vasco T. Vasconcelos, Francisco Martins, and Tiago Cogumbreiro. Type inference for deadlock detection in a multithreaded typed assembly language. Presented at PLACES'09, 2009.

[49] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proceedings of ESOP'03*, 2003.

[50] Dachuan Yu and Zhong Shao. Verification of safety properties for concurrent assembly code. In *Proceedings of ICFP'04*, 2004.