Compiling the π-calculus into a Multithreaded Typed Assembly Language

Tiago Cogumbreiro
Francisco Martins
Vasco T. Vasconcelos

DI–FCUL TR–08–13

May 2008

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Compiling the π-calculus into a Multithreaded Typed Assembly Language

Tiago Cogumbreiro Francisco Martins
Vasco T. Vasconcelos

May 2008

Abstract

Current trends in hardware made available multi-core CPU systems to ordinary users, challenging researchers to devise new techniques to bring software into the multi-core world. However, shaping software for multi-cores is more envolving than simply balancing workload among cores. In a near future (in less than a decade) Intel prepares to manufacture and ship 80-core processors; programmers must perform a paradigm shift from sequential to concurrent programming and produce software adapted for multi-core platforms.

In the last decade, proposals have been made to compile formal concurrent and functional languages, notably the π-calculus, typed concurrent objects, and the λ-calculus, into assembly languages. The last work goes a step further and presents a series of type-preserving compilation steps leading from System F to a typed assembly language. Nevertheless, all theses works are targeted at sequential architectures.

This paper proposes a type-preserving translation from the π-calculus into MIL, a multithreaded typed assembly language for multi-core/multi-processor architectures. We start from a simple asynchronous typed version of the π-calculus and translate it into MIL code that is then linked to a run-time library (written in MIL) that provides support for implementation of the π-calculus primitives (*e.g.*, queuing messages and processes). In short, we implement a message-passing paradigm in a shared-memory architecture.

Contents

Chapter 1

Introduction

Physical and electrical constrains are limiting the increase of frequency of each processing unit of a processor, thus the top speed of each processing unit is not expected to increase much more in near future. Instead manufactures are augmenting the number of processing units in each processor (multi-core processors) to continue delivering performance gains. The industry is making big investments in projects, such as RAMP [21] and BEE2 [1], that enable emulation of multi-core architectures, showing interest in supporting the foundations for software research that targets these architectures.

To take advantage of multi-core architectures, parallel and concurrent programming needs to be mastered [19]. 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 multi-core platforms.

The MDA (Model-Driven Architecture) / MDE (Model-Driven Engineering) methodologies are being widely used for software system development [18, 9]. However, these methodologies have informal specification languages and lack semantic foundations. The concurrency theory results (*e.g.* operational semantics, or axiomatic semantics) might enhance these methodologies [5].

In the last decade, some proposals have been made to compile concurrent and functional languages, notably the π-calculus [23], typed concurrent objects [11], and the λ-calculus [17], into assembly languages. The last work goes a step further and presents a series of type preserving compilation steps leading from System F [6] to a typed assembly language [17]. Yet, all the works are targeted for systems with a single core CPU architecture.

We propose a typed preserving translation from the π-calculus into MIL, a multithreaded typed assembly language aiming at multi-core/multi-processor architectures. We depart from a simple asynchronous typed version of the π-calculus [2, 8, 16] and translate it into MIL, based on a run-time library (written in MIL) that provides support (*e.g.* queueing of messages and processes) for implementation the π-calculus primitives.

The run-time library defines channels and operations on channels to send and to receive messages. Messages are buffered in the channel they are sent to, until a process request for a message in the channel. The reverse is also true to processes: a process requesting for a message gets blocked until another one sends a message, by storing their state and their code in the channel.

The run-time library and the translation function have an intertwined design, although there is a clear separation of concerns between them. The concept of a process is traversal and defined at different levels of abstraction in both parts.

The concurrency is also preserved to a certain extent (limited only by the number of available processors): π processes are represented by MIL threads of execution. The concurrent architecture of the target language is, therefore, extensively explored, resulting in highly parallel programs free of race-conditions.

This paper is divided into seven chapters. Chapter 2 describes the π-calculus, the source language. Chapter 3 presents our target language, MIL. Chapter 4 and Chapter 5 discusses the translation from the π-calculus into MIL. Finally, in Chapter 6, we summarize our work and hint at future directions.

Chapter 2

The π-Calculus

The π-calculus, developed by Robin Milner, Joachim Parrow, and David Walker [16], is a process algebra for describing *mobility*. The π-calculus is used to model a network of interconnected processes interacting through connection links (ports) by sending and receiving references to other processes, thus allowing the dynamic reconfiguration of the network.

As an example, consider a process that bounces every message received in a port. Figure 2.1 depicts such an interaction. The client sends message *msg* and a reply channel, where the server should echo the message to the client. Afterwards, the server sends the message *msg* back to the client.

In this chapter we present an overview of the π-calculus syntax and semantics.

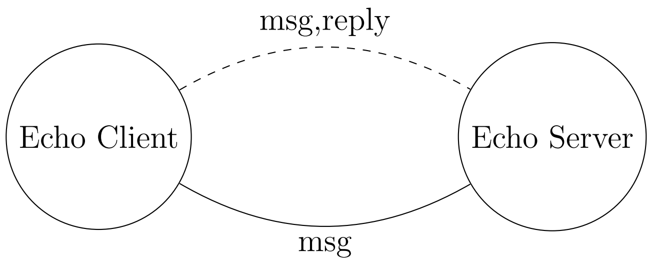

Figure 2.1: The server echoing the received message back to the client. The dashed line represents communication from the client to the server. The full line represents communication from the server to the client.

$$\begin{array}{rcll@{\qquad}rcll}
 & & & \textit{Processes} & & & & \textit{Values} \\
P, Q & ::= & \mathbf{0} & \text{nil} & v & ::= & x, y & \text{name} \\
 & \mid & \overline{x}\langle\vec{v}\rangle & \text{output} & & \mid & \mathit{basval} & \text{basic value} \\
 & \mid & x(\vec{y}).P & \text{input} & & & & \\
 & \mid & P \mid Q & \text{parallel} & & & & \\
 & \mid & (\nu\, x \colon (\vec{T}))\, P & \text{restriction} & & & & \\
 & \mid & !P & \text{replication} & & & &
\end{array}$$

The syntax of T is illustrated in Figure 2.3

Figure 2.2: Process syntax

2.1 Syntax

Processes. The adopted π-calculus syntax is based on [15] with extensions presented in [22]: asynchronous, polyadic, and typed.

The syntax, depicted in Figure 2.2, is divided into two categories: *names* and *processes*. Names are ranged over by lower case letters x and y. Values, v, symbolise either names or primitive values. A vector above a symbol abbreviates a possibly empty sequence of these symbols. For example $\vec{x}$ stands for the sequence of names $x_0 \ldots x_n$ with $n \geq 0$. Processes, denoted by upper case letters P and Q, comprise the nil process, $\mathbf{0}$, corresponding to the inactive process; the output process, $\overline{x}\langle\vec{v}\rangle$, outlines the action of sending data, $\vec{v}$, through a channel x; the input process, $x(\vec{y}).P$, that receives a sequence of values via channel x and continues as P, with the received names substituted for the received values; the parallel composition process, $P \mid Q$, represents two active processes running concurrently; the restriction process, $(\nu\, x \colon (\vec{T}))\, P$, that creates a new channel definition local to process P; and finally the replicated process, $!P$, that represents an infinite number of active processes running in parallel.

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

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

This process is ready to receive a message msg and a communication channel

		Types				*Basic value types*
$T, S ::=$	B	basic value type		$B ::=$	*int*	integer type
$\mid$	$(\vec{T})$	link type		$\mid$	*str*	string type

Figure 2.3: Type syntax

reply trough channel *echo*. After receiving the values, it outputs the message through channel *reply*. The process is replicated because is must be able to communicate with multiple clients.

Types. Types are assigned to channels and to basic values. A basic value type is either a string, *str*, or an integer, *int*; the channel type $(\vec{T})$ describes the types of the communicated value $\vec{T}$ through the channel. For example, a possible type for the *echo* channel from Process 2.1 is $(str, (str))$.

2.2 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.

For the sake of clarity, we omit the type from the restriction operator.

Structural Congruence. The *structural congruence* relation, $\equiv$, is the smallest congruence relation on processes closed under rules given in Figure 2.4. Structural congruence identifies processes that represent the same behaviour structure and can be used to reshape process structure to enable reduction. The rules are straightforward. Rule S1 allows for alpha-conversion; Rules S2, S3, and S4 are the standard commutative monoid laws regarding parallel composition, having **0** as neutral element; Rule S5 allows for scope extrusion; Rule S6 garbage collects unused names; Rule S7 states that restriction order is of no importance; and finally Rule S8 allows replication to unfold.

(S1) change of bound names

(S2) $P \mid \mathbf{0} \equiv P, \quad P \mid Q \equiv Q \mid P, \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

(S3) $(\nu\, x\colon (\vec{T}))\,(P \mid Q) \equiv P \mid (\nu\, x\colon (\vec{T}))\, Q \quad \text{if } x \notin \text{fn}(P)$

(S4) $(\nu\, x\colon (\vec{T}))\,\mathbf{0} \equiv \mathbf{0}$

(S5) $(\nu\, x\colon (\vec{T}))\,(\nu\, y\colon (\vec{S}))\, P \equiv (\nu\, y\colon (\vec{S}))\,(\nu\, x\colon (\vec{T}))\, P$, if $x \neq y$

(S6) $!P \equiv P \mid !P$

Figure 2.4: Structural congruence rules

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

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

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

Figure 2.5: Reaction Rules

Bound and free names are defined as usual in the π-calculus, so we omit their formal definitions.

Reduction. The reduction relation $\to$ defined over processes, in Figure 2.5, establishes how a computational step transforms a process [14]. The formula $P \to Q$ means that process P can interact and evolve (reduce) to process Q.

The axiom React is the gist of the reaction rules, representing the communication along a channel [13]. An output process, $\overline{x}\langle\vec{v}\rangle$, can interact with an input process, $x(\vec{y}).P$, if they have the same channel's name, x. The output message, $\vec{v}$, moves along channel x to process P and replaces the entry points, $\vec{y}$, resulting $P\{\vec{v}/\vec{y}\}$. The term $P\{\vec{v}/\vec{y}\}$ means that the names $\vec{y}$, in process P, are to be replaced by the values $\vec{v}$.

$$\frac{\mathit{baseval} \in B}{\Gamma \vdash \mathit{baseval} \colon B}\ \textsc{Tv-Base} \qquad \frac{}{\Gamma, x \colon T \vdash x \colon T}\ \textsc{Tv-Name}$$

$$\frac{}{\Gamma \vdash \mathbf{0}}\ \textsc{Tv-Nil} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ \textsc{Tv-Rep}$$

$$\frac{\Gamma \vdash x \colon (T_0 \ldots T_i) \quad \Gamma, y_0 \colon T_0, \ldots, y_i \colon T_i \vdash P}{\Gamma \vdash x(\vec{y}).P}\ \textsc{Tv-In}$$

$$\frac{\Gamma \vdash x \colon (\vec{T}) \quad \Gamma \vdash v_i \colon T_i \quad \forall i \in I}{\Gamma \vdash \overline{x}\langle \vec{v} \rangle}\ \textsc{Tv-Out}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}\ \textsc{Tv-Par} \qquad \frac{\Gamma, x \colon (\vec{T}) \vdash P}{\Gamma \vdash (\nu\, x \colon (\vec{T}))\, P}\ \textsc{Tv-Res}$$

Figure 2.6: Typing rules for the π-calculus

Rule Par expresses that reduction can appear on the right side of a parallel composition. Res governs reduction inside the restriction operator. Rule Struct brings congruence rules to the reduction relation.

Process

$$!\mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg} \rangle \mid (\nu\, r)\, \overline{\mathit{echo}}\langle '\texttt{hello world!}', r \rangle$$

represents, respectively, the echo server being run concurrently with a client that creates a new name r and sends it, along with a message, through channel *echo*, to the server. The following steps describe the reaction between both processes:

$$!\mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg} \rangle \mid (\nu\, r)\, \overline{r}\langle '\texttt{hello world!}' \rangle$$

Type system. Figure 2.6 presents a standard type system for the π-calculus.

Rule Tv-Base states that primitive values (strings and numbers) are well typed. Rule Tv-Name sets forth that a name is well typed if it is defined in the type environment and the type used matches the name's declaration. The inactive process $\mathbf{0}$ is always well typed, rule Tv-Nil. The process $(\nu\, x \colon (\vec{T}))\, P$

is well typed if, by adding the association between name x and type $(\vec{T})$ to Γ, the contained process P is well typed, rule Tv-Res. Tv-In rules that the input process, $x(\vec{y}).P$, is well typed if the name of the input channel, x, is a link type and if, by mapping each name of the input channel's arguments to the corresponding type of x, the contained process, P, is well typed. The output process, $\overline{x}\langle\vec{v}\rangle$, is well typed if its name, x, is declared as link type and if its arguments are correctly typed, rule Tv-Out. The consistency of the replicated process depends on the consistency of the process being replicated, rule Tv-Rep. The parallel process is well typed if each of the composing processes are well typed, rule Tv-Par.

Now, we show that process

$$(\nu\, \mathit{echo}\colon (\mathit{str}, (\mathit{str})))\, \mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle$$

is well typed. Using rule Tv-Res we derive

$$\dfrac{\emptyset, \mathit{echo}\colon (\mathit{str}, (\mathit{str})) \vdash \mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle}{\emptyset \vdash (\nu\, \mathit{echo}\colon (\mathit{str}, (\mathit{str})))\, \mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle}\ \textsc{Tv-Res}$$

Let $\Gamma' \overset{\text{def}}{=} \emptyset, \mathit{echo}\colon (\mathit{str}, (\mathit{str}))$. We need to prove that the new typing environment, Γ', typifies process

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

Applying rule Tv-In:

$$\dfrac{\Gamma', \mathit{msg}\colon \mathit{str}, \mathit{reply}\colon (\mathit{str}) \vdash \overline{\mathit{reply}}\langle \mathit{msg}\rangle \quad \Gamma' \vdash \mathit{echo}\colon (\mathit{str}, (\mathit{str}))}{\Gamma' \vdash \mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle \mathit{msg}\rangle}\ \textsc{Tv-In}$$

Rule Tv-Name ensures that $\Gamma' \vdash \mathit{echo}\colon (\mathit{str}, (\mathit{str}))$ holds. Now, let

$$\Gamma'' \overset{\text{def}}{=} \Gamma', \mathit{msg}\colon \mathit{str}, \mathit{reply}\colon (\mathit{str})$$

We are left with the second sequent, that also holds

$$\dfrac{\dfrac{}{\Gamma'' \vdash \mathit{reply}\colon (\mathit{str})}\ \textsc{Tv-Name} \qquad \dfrac{}{\Gamma'' \vdash \mathit{msg}\colon \mathit{str}}\ \textsc{Tv-Name}}{\Gamma'' \vdash \overline{\mathit{reply}}\langle \mathit{msg}\rangle}\ \textsc{Tv-Out}$$

Chapter 3

MIL: Multithreaded Typed Assembly Language

MIL [24] combines a typed assembly language (TAL) with multithreaded programming, providing the possibility for "executing trusted code safely and efficiently" [17]. Types ensure that pointers cannot be fabricated or forged and that jumps can only be done to checked code, allowing untrusted compilers to generate a typed assembly language that can be compiled with a single trusted compiler.

Multithreaded programming at assembly level helps structuring inter-thread synchronisation. The type system we provide for the language enforces the absence of race conditions.

3.1 Architecture

MIL envisages an abstract multi-processor with a shared main memory. Each processor consists of registers and of an instruction cache. The main memory is divided into a heap (for storing data and code blocks) and a run pool. Data blocks are represented by tuples and are protected by locks. Code blocks declare the needed registers (including the type for each register), the required locks, and an instruction set. The run pool contains suspended threads waiting for execution. It may happen that there are more threads to be run than the number of processors. Figure 3.1 summarizes the MIL architecture.

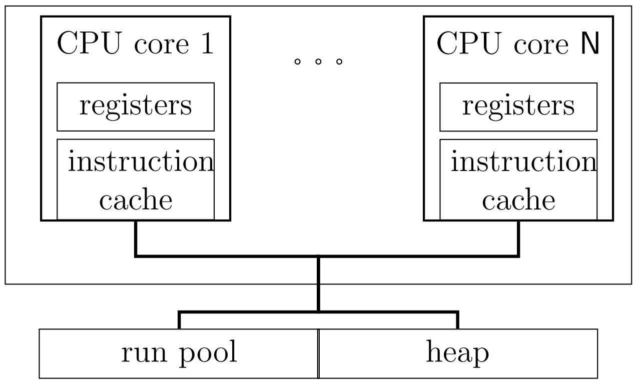

Figure 3.1: The MIL architecture.

3.2 Syntax

The syntax of our language is generated by the grammar in Figures 3.2, 3.3, and 3.9. We postpone the exposure of types to Section 3.4. We rely on a set of *heap labels* ranged over by l, and a disjoint set of *type variables* ranged over by α and β.

Most of the proposed instructions, represented in Figure 3.2, are standard in assembly languages. Instructions are organised in sequences, ending in a jump or in a yield. Instruction yield frees the processor to execute another thread from the thread pool. Our threads are cooperative, meaning that each thread must explicitly release the processor (using the yield instruction).

The *abstract machine*, depicted in Figure 3.3, is parametric in the number of processors available (N) and in the number of registers (R).

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. Heaps are maps from labels into *heap values* that may be tuples or code blocks. *Tuples* are vectors of values protected by some lock. Code blocks comprise a signature and a body. The signature of a code block describes the type of each register used in the body, and the locks held by the processor when jumping to the code block. The body is a sequence of instructions to be executed by a processor.

registers	r	$::=$	$\mathsf{r}_1 \mid \ldots \mid \mathsf{r}_\mathsf{R}$
integer values	n	$::=$	$\ldots \mid$ -1 $\mid 0 \mid 1 \mid \ldots$
lock values	b	$::=$	$\mathbf{-1} \mid \mathbf{0} \mid \mathbf{1} \mid \ldots$
values	v	$::=$	$r \mid n \mid b \mid l \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \mid \mathsf{packL}\ \alpha, v\ \mathsf{as}\ \tau \mid$
			$v[\tau] \mid ?\tau$
instructions	ι	$::=$	
control flow			$r := v \mid r := r + v \mid \mathsf{if}\ r = v\ \mathsf{jump}\ v \mid$
memory			$r := \mathsf{malloc}\ [\vec{\tau}]\ \mathsf{guarded\ by}\ \alpha \mid$
			$r := v[n] \mid r[n] := v \mid$
unpack			$\alpha, r := \mathsf{unpack}\ v \mid$
lock			$\alpha, r := \mathsf{newLock}\ b \mid \alpha := \mathsf{newLockLinear}$
			$r := \mathsf{tslE}\ v \mid r := \mathsf{tslS}\ v \mid \mathsf{unlockE}\ v \mid \mathsf{unlockS}\ v \mid$
fork			$\mathsf{fork}\ v$
inst. sequences	I	$::=$	$\iota; I \mid \mathsf{jump}\ v \mid \mathsf{yield}$

Figure 3.2: Instructions

A thread pool is a multiset of pairs, each of which contains a pointer (*i.e.* a label) to a code block and a register file. A processor array contains N processors, where each is composed of a register file, a set of locks, and a sequence of instructions.

3.3 Operational Semantics

Thread pools are managed by the rules illustrated in Figure 3.4. Rule R-HALT stops the machine when it finds an empty thread pool and, at the same time, all processors are idle, changing the machine state to halt. Otherwise, if there is an idle processor and a thread waiting in the pool, then by Rule R-SCHEDULE the thread is assigned to the idle processor. Rule R-FORK places a new thread in the pool, taking the ownership of locks required by

states	S	$::=$	$\langle H; T; P\rangle \mid \mathsf{halt}$
heaps	H	$::=$	$\{l_1 \colon h_1, \ldots, l_n \colon h_n\}$
heap values	h	$::=$	$\langle v_1 \ldots v_n\rangle^\alpha \mid \tau\{I\}$
thread pool	T	$::=$	$\{\langle l_1, R_1\rangle, \ldots, \langle l_n, R_n\rangle\}$
processors array	P	$::=$	$\{1 \colon p_1, \ldots, \mathsf{N} \colon p_\mathsf{N}\}$
processor	p	$::=$	$\langle R; \Lambda; I\rangle$
register files	R	$::=$	$\{\mathsf{r}_1 \colon v_1, \ldots, \mathsf{r}_\mathsf{R} \colon v_\mathsf{R}\}$
permissions	Λ	$::=$	$(\lambda, \lambda, \lambda)$
lock sets	λ	$::=$	$\alpha_1, \ldots, \alpha_n$

Figure 3.3: Abstract machine

the forked code block.

Operational semantics regarding locks are depicted in Figure 3.5 and in Figure 3.6. The instruction newLock creates a new lock in three possible states, according to its parameter: locked exclusively (when the parameter is -1), locked shared (when the parameter is **1**), and unlocked (when the parameter is **0**). The scope of α is the rest of the code block. A tuple with the value of the parameter of the newLock is allocated in the heap and register r is made to point it. For example, a new lock in the unlocked state allocates the tuple $\langle \mathbf{0}\rangle^\beta$. When the lock is created in the exclusive lock state, the new lock variable β is added to the set of exclusive locks held by the processor. Similarly, when the lock is created in the shared lock state, the new lock variable β is added to the set of shared locks held by the processor, allowing just one reader.

Linear locks are created by newLockLinear. The new lock variable β is added to the set of linear locks.

The *Test and Set Lock*, presented in many machines designed with multiple processes in mind, is an atomic operation that loads the contents of a word into a register and then stores another value in that word. There are two variations of the *Test and Set Lock* in our language: tslE and tslS. When a tslE is applied to an unlocked state, the type variable α is added to the set of exclusive locks and the value becomes $\langle$-1$\rangle^\alpha$. Various threads may

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

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

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

Figure 3.4: Operational semantics (thread pool)

read values from a tuple locked in shared state, hence when tslS is applied to a shared or to an unlocked lock the value contained in the tuple representing the lock is incremented, reflecting the number of readers holding the shared lock, and then the type variable α is added to the set of hold shared locks. When tslS is applied to a lock in the exclusive state, it places a -1 in the target register and the lock is not acquired by the thread issuing the operation.

Shared locks are unlocked with unlockS and the number of readers is decremented. The running processor must hold the shared lock. Exclusive locks are unlocked with unlockE, while the running processor holds the exclusive lock.

Rules related to memory instructions are illustrated in Figure 3.7. Values can be stored in a tuple, when the lock that guards the tuple is hold by the processor in the set of exclusive locks or in the set of linear locks. A value can be loaded from a tuple if the lock guarded by it is hold by the processor in any set of locks. The rule for malloc allocates a new tuple in the heap and makes r point to it. The size of the tuple is that of sequence of types $[\vec{\tau}]$, its values are uninitialised values.

The transition rules for the control flow instructions, illustrated in Figure 3.8, are straightforward [20]. They rely on function $\hat{\mathrm{R}}$ that works on registers or on values, by looking for values in registers, in packs, and in universal concretions.

$$\frac{P(i) = \langle R; \Lambda; (\alpha, r := \mathsf{newLock}\ \mathbf{0}; I)\rangle \qquad l \notin \mathrm{dom}(H) \qquad \beta \notin \Lambda}{\langle H; T; P\rangle \rightarrow \langle H\{l\colon \langle \mathbf{0}\rangle^{\beta}\}; T; P\{i\colon \langle R\{r\colon l\}; \Lambda; I[\beta/\alpha]\rangle\}\rangle} \ (\text{R-NEW-LOCK } \mathbf{0})$$

$$\frac{P(i) = \langle R; \Lambda; (\alpha, r := \mathsf{newLock}\ \mathbf{1}; I)\rangle \qquad l \notin \mathrm{dom}(H) \qquad \beta \notin \Lambda}{\langle H; T; P\rangle \rightarrow \langle H\{l\colon \langle \mathbf{1}\rangle^{\beta}\}; T; P\{i\colon \langle R\{r\colon l\}; (\lambda_E, \lambda_S \uplus \{\beta\}, \lambda_L); I[\beta/\alpha]\rangle\}\rangle} \ (\text{R-NEW-LOCK } \mathbf{1})$$

$$\frac{P(i) = \langle R; \Lambda; (\alpha, r := \mathsf{newLock}\ \mathbf{\text{-}1}; I)\rangle \qquad l \notin \mathrm{dom}(H) \qquad \beta \notin \Lambda}{\langle H; T; P\rangle \rightarrow \langle H\{l\colon \langle \mathbf{\text{-}1}\rangle^{\beta}\}; T; P\{i\colon \langle R\{r\colon l\}; (\lambda_E \uplus \{\beta\}, \lambda_S, \lambda_L); I[\beta/\alpha]\rangle\}\rangle} \ (\text{R-NEW-LOCK } \mathbf{\text{-}1})$$

$$\frac{P(i) = \langle R; \Lambda; (\alpha := \mathsf{newLockLinear}; I)\rangle \qquad \beta \notin \Lambda}{\langle H; T; P\rangle \rightarrow \langle H; T; P\{i\colon \langle R; (\lambda_E, \lambda_S, \lambda_L \uplus \{\beta\}); I[\beta/\alpha]\rangle\}\rangle} \ (\text{R-NEW-LOCKL})$$

Figure 3.5: Operational semantics (lock creation)

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

3.4 Type Discipline

The syntax of types is exposed in Figure 3.9. A type of the form $\langle \vec{\sigma} \rangle^{\alpha}$ describes a tuple that is protected by lock α. Each type $\vec{\sigma}$ is either initialised (τ) or uninitialised ($?\tau$). A type of form $\forall[\vec{\alpha}]\Gamma$ $\mathsf{requires}$ Λ describes a code block; a thread jumping into such a block must instantiate all the universal variables $\vec{\alpha}$; it must also hold a register file type Γ as well as the locks in Λ. The singleton lock type, $\mathsf{lock}(\alpha)$, is used to represent the type of a lock value in the heap. The types $\exists\alpha.\tau$ defines conventional existential type. With type $\exists^{\mathrm{L}}\alpha.\tau$ we are able to use the existential quantification over lock types, by following [4]. The recursive type, where the type may itself be present in the

$$\frac{P(i) = \langle R; \Lambda; (r := \mathsf{tslS}\ v; I)\rangle \qquad \hat{\mathrm{R}}(v) = l \qquad H(l) = \langle b \rangle^\alpha \qquad b \geq \mathbf{0}}{\langle H; T; P\rangle \rightarrow \langle H\{l \colon \langle b + \mathbf{1}\rangle^\alpha\}; T; P\{i \colon \langle R\{r \colon \mathbf{0}\}; (\lambda_E, \lambda_S \uplus \{\alpha\}, \lambda_L); I\rangle\}\rangle} \quad \text{(R-tslS-acq)}$$

$$\frac{P(i) = \langle R; \Lambda; (r := \mathsf{tslS}\ v; I)\rangle \qquad H(\hat{\mathrm{R}}(v)) = \langle \textbf{-1} \rangle^\alpha}{\langle H; T; P\rangle \rightarrow \langle H; T; P\{i \colon \langle R\{r \colon \textbf{-1}\}; \Lambda; I\rangle\}\rangle} \quad \text{(R-tslS-fail)}$$

$$\frac{P(i) = \langle R; \Lambda; (r := \mathsf{tslE}\ v; I)\rangle \qquad \hat{\mathrm{R}}(v) = l \qquad H(l) = \langle \mathbf{0} \rangle^\alpha}{\langle H; T; P\rangle \rightarrow \langle H\{l \colon \langle \textbf{-1}\rangle^\alpha\}; T; P\{i \colon \langle R\{r \colon \mathbf{0}\}; (\lambda_E \uplus \{\alpha\}, \lambda_S, \lambda_L); I\rangle\}\rangle} \quad \text{(R-tslE-acq)}$$

$$\frac{P(i) = \langle R; \Lambda; (r := \mathsf{tslE}\ v; I)\rangle \qquad H(\hat{\mathrm{R}}(v)) = \langle b \rangle^\alpha \qquad b \neq \mathbf{0}}{\langle H; T; P\rangle \rightarrow \langle H; T; P\{i \colon \langle R\{r \colon b\}; \Lambda; I\rangle\}\rangle} \quad \text{(R-tslE-fail)}$$

$$\frac{P(i) = \langle R; (\lambda_E, \lambda_S \uplus \{\alpha\}, \lambda_L); (\mathsf{unlockS}\ v; I)\rangle \qquad \hat{\mathrm{R}}(v) = l \qquad H(l) = \langle b \rangle^\alpha}{\langle H; T; P\rangle \rightarrow \langle H\{l \colon \langle b - \mathbf{1}\rangle^\alpha\}; T; P\{i \colon \langle R; (\lambda_E, \lambda_S, \lambda_L); I\rangle\}\rangle} \quad \text{(R-unlockS)}$$

$$\frac{P(i) = \langle R; (\lambda_E \uplus \{\alpha\}, \lambda_S, \lambda_L); (\mathsf{unlockE}\ v; I)\rangle \qquad \hat{\mathrm{R}}(v) = l \qquad H(l) = \langle \_ \rangle^\alpha}{\langle H; T; P\rangle \rightarrow \langle H\{l \colon \langle \mathbf{0}\rangle^\alpha\}; T; P\{i \colon \langle R; (\lambda_E, \lambda_S, \lambda_L); I\rangle\}\rangle} \quad \text{(R-unlockE)}$$

Figure 3.6: Operational semantics (lock manipulation)

types it is composed by, is defined by $\mu\alpha.\tau$.

The type system is presented in Figures 3.10 to 3.15. Typing for values is illustrated in Figure 3.10. Heap values are distinguished from operands (that include registers as well) by the form of the sequent. Uninitialised value $?\tau$ has type $?\tau$; we use the same syntax for a uninitialised value (at the left of the colon) and its type (at the right of the colon). A formula $\sigma <: \sigma'$ allows to "forget" initialisations.

Instructions are checked against a typing environment Ψ (mapping labels to types, and type variables to the kind Lock: the kind of singleton lock types), a register file type Γ holding the current types of the registers, and

$$\frac{P(i) = \langle R; \Lambda; (r := \textsf{malloc } [\vec{\tau}] \textsf{ guarded by } \alpha; I)\rangle \qquad l \notin \operatorname{dom}(H)}{\langle H; T; P\rangle \to \langle H\{l\colon \langle ?\vec{\tau}\rangle^{\alpha}\}; T; P\{i\colon \langle R\{r\colon l\}; \Lambda; I\rangle\}\rangle} \quad \text{(R-MALLOC)}$$

$$\frac{P(i) = \langle R; \Lambda; (r := v[n]; I)\rangle \quad H(\hat{\mathrm{R}}(v)) = \langle v_1..v_n..v_{n+m}\rangle^{\alpha}}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R\{r\colon v_n\}; \Lambda; I\rangle\}\rangle} \quad \text{(R-LOAD)}$$

$$\frac{\begin{array}{c} P(i) = \langle R; \Lambda; (r[n] := v; I)\rangle \\ R(r) = l \qquad H(l) = \langle v_1..v_n..v_{n+m}\rangle^{\alpha} \end{array}}{\langle H; T; P\rangle \to \langle H\{l\colon \langle v_1..\,\hat{\mathrm{R}}(v)..v_{n+m}\rangle^{\alpha}\}; T; P\{i\colon \langle R; \Lambda; I\rangle\}\rangle} \quad \text{(R-STORE)}$$

Figure 3.7: Operational semantics (memory)

a triple Λ that comprises sets of lock variables (the *permission* of the code block), that are, respectively, the exclusive, the shared, and the linear.

Rule T-YIELD requires that shared and exclusive locks must have been released prior to the ending of the thread. Rule T-FORK splits the permission into the two tuples Λ and Λ': the former is transferred to the forked thread, the latter remains with the current thread, according to the permissions required by the target code block.

Rules T-NEW-LOCK**1**, T-NEW-LOCK**-1**, and T-NEW-LOCKL each adds the type variable into the respective set of locks. Rules T-NEW-LOCK**0**, T-NEW-LOCK**1**, and T-NEW-LOCK**-1** assign a lock type to the register. Rules T-TSLE and T-TSLS require that the value under test holds a lock, disallowing testing a lock already held by the thread. Rules T-UNLOCKE and T-UNLOCKS make sure that only held locks are unlocked. Finally, the rules T-CRITICALE and T-CRITICALS ensure that the current thread holds the exact number of locks required by the target code block. Each of these rules also adds the lock under test to the respective set of locks of the thread. A thread is guaranteed to hold the lock only after (conditionally) jumping to a critical region. A previous test and set lock instructions may have obtained the lock, but as far as the type system goes, the thread holds the lock after the conditional jump.

The typing rules for memory and control flow are depicted in Figure 3.13 and Figure 3.14. The rule for malloc ensures that allocated memory is pro-

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

$$\frac{P(i) = \langle R; \Lambda; (r := v; I)\rangle}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R\{r\colon \hat{\mathrm{R}}(v)\}; \Lambda; I\rangle\}\rangle} \quad \text{(R-MOVE)}$$

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

$$\frac{\begin{array}{c} P(i) = \langle R; \Lambda; (\mathsf{if}\ r = v\ \mathsf{jump}\ v'; \_)\rangle \\ R(r) = v \qquad H(\hat{\mathrm{R}}(v')) = \_\{I\} \end{array}}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R; \Lambda; I\rangle\}\rangle} \quad \text{(R-BRANCHT)}$$

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

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

$$\frac{P(i) = \langle R; \Lambda; (\alpha, r := \mathsf{unpack}\ v; I)\rangle \qquad \hat{\mathrm{R}}(v) = \mathsf{packL}\ \beta, v'\ \mathsf{as}\ \_}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R\{r\colon v'\}; \Lambda; I[\beta/\alpha]\rangle\}\rangle} \quad \text{(R-UNPACKL)}$$

Figure 3.8: Operational semantics (control flow)

tected by a lock (α) present in the scope. The lock that guards a tuple defines the permissions that affect how the loading and the storing operations work. Holding a lock of any kind enables permission to load a value from a tuple. Only exclusive and linear locks permit storing a value into a tuple.

The rules for typing machine states are illustrated in Figure 3.15. They should be easy to follow. The only remark goes to heap tuples, where we make sure that all locks protecting the tuples are in the domain of the typing environment.

types	τ	$::=$	$\mathsf{int} \mid \langle\vec{\sigma}\rangle^{\alpha} \mid \forall[\vec{\alpha}]\Gamma\ \mathsf{requires}\ \Lambda \mid \mathsf{lock}(\alpha) \mid$ $\mathsf{lockE}(\alpha) \mid \mathsf{lockS}(\alpha) \mid \exists\alpha.\tau \mid \exists^{\mathrm{L}}\alpha.\tau \mid \mu\alpha.\tau \mid \alpha$
init types	σ	$::=$	$\tau \mid ?\tau$
register file types	Γ	$::=$	$\mathsf{r}_1 : \tau_1, \ldots, \mathsf{r}_n : \tau_n$
typing environment	Ψ	$::=$	$\emptyset \mid \Psi, l : \tau \mid \Psi, \alpha :: \mathsf{Lock}$

Figure 3.9: Types

3.5 Examples

We select a case in point of inter-process communication: mutual exclusion. We create a tuple and then start two threads that try to write in the tuple concurrently. A reference for lock α is transferred to each new thread, by instantiating the universal value, since it is not in the scope of the forked threads.

```
main() {
  $\alpha$, $r_1$:= **newLock** $-1$
  $r_2$:= **malloc** [int] **guarded by** $\alpha$
  $r_2$[0] := 0
  **unlockE** $r_1$
  **fork** thread1[$\alpha$]
  **fork** thread2[$\alpha$]
  **yield**
}
```

Each thread competes in acquiring lock α using different strategies. In the first thread (thread1) we use a technique called *spin lock*, where we loop actively, not releasing the processor, until we eventually grab the lock exclusively. After that we jump to the critical region.

```
thread1 $\forall[\alpha]$($r_1$: $\langle$**lock**$(\alpha)\rangle^{\alpha}$, $r_2$: $\langle ?\mathsf{int}\rangle^{\alpha}$) {
  $r_3$:= **tslE** $r_1$-- *exclusive because we want to write*
   **if** $r_3$= 0 **jump** criticalRegion [$\alpha$]
  **jump** thread1[$\alpha$]
}
```

$$\vdash \langle \sigma_1, \ldots, \tau_n, \ldots, \sigma_{n+m}\rangle^\alpha <: \langle \sigma_1, \ldots, ?\tau_n, \ldots, \sigma_{n+m}\rangle^\alpha \qquad \text{(S-UNINIT)}$$

$$\frac{n \leq m}{\vdash \mathsf{r}_0\colon \tau_0, \ldots, \mathsf{r}_m\colon \tau_m <: \mathsf{r}_0\colon \tau_0, \ldots, \mathsf{r}_n\colon \tau_n} \qquad \text{(S-REG-FILE)}$$

$$\vdash \sigma <: \sigma \qquad \frac{\vdash \sigma <: \sigma' \qquad \vdash \sigma' <: \sigma''}{\vdash \sigma <: \sigma''} \qquad \text{(S-REF, S-TRANS)}$$

$$\frac{\vdash \tau' <: \tau}{\Psi, l\colon \tau' \vdash l\colon \tau} \qquad \Psi \vdash n\colon \mathsf{int} \qquad \Psi \vdash b\colon \mathsf{lock}(\alpha) \qquad \Psi \vdash ?\tau\colon ?\tau$$

(T-LABEL,T-INT,T-LOCK,T-UNINIT)

$$\frac{\Psi \vdash v\colon \tau'[\tau/\alpha] \qquad \alpha \notin \tau, \Psi}{\Psi \vdash \mathsf{pack}\ \tau, v\ \mathsf{as}\ \exists\alpha.\tau'\colon \exists\alpha.\tau'} \qquad \frac{\Psi \vdash v\colon \tau[\beta/\alpha] \qquad \alpha \notin \beta, \Psi}{\Psi \vdash \mathsf{packL}\ \beta, v\ \mathsf{as}\ \exists^{\mathrm{L}}\alpha.\tau\colon \exists^{\mathrm{L}}\alpha.\tau}$$

(T-PACK,T-PACKL)

$$\Psi; \Gamma \vdash r\colon \Gamma(r) \qquad \frac{\Psi \vdash v\colon \tau}{\Psi; \Gamma \vdash v\colon \tau} \qquad \text{(T-REG,T-VAL)}$$

$$\frac{\Psi; \Gamma \vdash v\colon \forall[\alpha\vec{\beta}]\Gamma'\ \mathsf{requires}\ \Lambda}{\Psi; \Gamma \vdash v[\tau]\colon \forall[\vec{\beta}]\Gamma'[\tau/\alpha]\ \mathsf{requires}\ \Lambda[\tau/\alpha]} \qquad \text{(T-VAL-APP)}$$

Figure 3.10: Typing rules for values $\Psi \vdash v\colon \sigma$ and for operands $\Psi; \Gamma \vdash v\colon \sigma$

In the critical region, the value contained in the tuple is incremented.

```
criticalRegion  ∀[α](r_1: ⟨lock(α)⟩^α, r_2: ⟨?int⟩^α) requires(α;;) {
  r_3:= r_2[0]
  r_3:= r_3+ 1
  r_2[0] := r_3
  unlockE r_1
  yield
}
```

In the second thread (thread2), we opt for a different technique called *sleep lock*. This strategy is cooperative towards other threads, since the control of the processor is not held exclusively until the lock's permission is granted. We try to acquire the lock exclusively. When we do, we jump to the critical region. If we do not acquire the lock, we fork a copy of this thread that will try again later.

```
thread2 ∀[α](r_1: ⟨lock(α)⟩^α, r_2: ⟨int⟩^α) {
  r_3:= tslE r_1−− exclusive because we want to write
```

$$\Psi; \Gamma; (\emptyset, \emptyset, \lambda_L) \vdash \mathsf{yield} \qquad \text{(T-YIELD)}$$

$$\frac{\Psi; \Gamma \vdash v \colon \forall[]\Gamma' \ \mathsf{requires}\ \Lambda \qquad \Psi; \Gamma; \Lambda' \vdash I \qquad \vdash \Gamma <: \Gamma'}{\Psi; \Gamma; \Lambda \uplus \Lambda' \vdash \mathsf{fork}\ v; I} \qquad \text{(T-FORK)}$$

$$\frac{\Psi, \alpha \colon\colon \mathsf{Lock}; \Gamma\{r \colon \langle \mathsf{lock}(\alpha)\rangle^\alpha\}; \Lambda \vdash I \qquad \alpha \notin \Psi, \Gamma, \Lambda}{\Psi; \Gamma; \Lambda \vdash \alpha, r := \mathsf{newLock}\ \mathbf{0}; I} \qquad \text{(T-NEW-LOCK } \mathbf{0}\text{)}$$

$$\frac{\Psi, \alpha \colon\colon \mathsf{Lock}; \Gamma\{r \colon \langle \mathsf{lock}(\alpha)\rangle^\alpha\}; (\lambda_E, \lambda_S \uplus \{\alpha\}, \lambda_L) \vdash I \qquad \alpha \notin \Psi, \Gamma, \Lambda}{\Psi; \Gamma; \Lambda \vdash \alpha, r := \mathsf{newLock}\ \mathbf{1}; I} \qquad \text{(T-NEW-LOCK } \mathbf{1}\text{)}$$

$$\frac{\Psi, \alpha \colon\colon \mathsf{Lock}; \Gamma\{r \colon \langle \mathsf{lock}(\alpha)\rangle^\alpha\}; (\lambda_E \uplus \{\alpha\}, \lambda_S, \lambda_L) \vdash I \qquad \alpha \notin \Psi, \Gamma, \Lambda}{\Psi; \Gamma; \Lambda \vdash \alpha, r := \mathsf{newLock}\ \mathbf{-1}; I} \qquad \text{(T-NEW-LOCK } \mathbf{-1}\text{)}$$

$$\frac{\Psi, \alpha \colon\colon \mathsf{Lock}; \Gamma; (\lambda_E, \lambda_S, \lambda_L \uplus \{\alpha\}) \vdash I \qquad \alpha \notin \Psi, \Gamma, \Lambda}{\Psi; \Gamma; \Lambda \vdash \alpha := \mathsf{newLockLinear}; I} \qquad \text{(T-NEW-LOCKL)}$$

Figure 3.11: Typing rules for instructions (thread pool and lock creation) $\Psi; \Gamma; \Lambda \vdash I$

```
  if $r_3$= 0 jump criticalRegion [$\alpha$]
  fork thread2[$\alpha$]
  yield
}
```

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). A short coming of the spin lock in machines with only one processor is demonstrated in this example:

```
main () {
  $\alpha$, $r_1$:= newLock $-1$
  fork release [$\alpha$]
  jump spinLock[$\alpha$]
}
release $\forall[\alpha]$ ($r_1$: $\langle$lock($\alpha$)$\rangle^\alpha$) requires ($\alpha$;;) {
```

$$\frac{\Psi;\Gamma \vdash v\colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha} \qquad \Psi;\Gamma\{r\colon \mathsf{lockS}(\alpha)\};\Lambda \vdash I \qquad \alpha \notin \Lambda}{\Psi;\Gamma;\Lambda \vdash r := \mathsf{tslS}\ v; I} \quad (\text{T-TSLS})$$

$$\frac{\Psi;\Gamma \vdash v\colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha} \qquad \Psi;\Gamma\{r\colon \mathsf{lockE}(\alpha)\};\Lambda \vdash I \qquad \alpha \notin \Lambda}{\Psi;\Gamma;\Lambda \vdash r := \mathsf{tslE}\ v; I} \quad (\text{T-TSLE})$$

$$\frac{\Psi;\Gamma \vdash v\colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha} \qquad \alpha \in \lambda_S \qquad \Psi;\Gamma;(\lambda_S \setminus \{\alpha\},\lambda_E,\lambda_L) \vdash I}{\Psi;\Gamma;(\lambda_S,\lambda_E,\lambda_L) \vdash \mathsf{unlockS}\ v; I}$$
(T-UNLOCKS)

$$\frac{\Psi;\Gamma \vdash v\colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha} \qquad \alpha \in \lambda_E \qquad \Psi;\Gamma;(\lambda_S,\lambda_E \setminus \{\alpha\},\lambda_L) \vdash I}{\Psi;\Gamma;(\lambda_S,\lambda_E,\lambda_L) \vdash \mathsf{unlockE}\ v; I}$$
(T-UNLOCKE)

$$\frac{\begin{array}{ll}\Psi;\Gamma \vdash r\colon \mathsf{lockS}(\alpha) & \Psi;\Gamma \vdash v\colon \forall[]\Gamma' \ \mathsf{requires}\ (\lambda_E, \lambda_S \uplus \{\alpha\}, \lambda'_L) \\ \Psi;\Gamma;\Lambda \vdash I & \vdash \Gamma <: \Gamma' \qquad \lambda'_L \subseteq \lambda_L\end{array}}{\Psi;\Gamma;(\lambda_E,\lambda_S,\lambda_L) \vdash \mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v; I}$$
(T-CRITICALS)

$$\frac{\begin{array}{ll}\Psi;\Gamma \vdash r\colon \mathsf{lockE}(\alpha) & \Psi;\Gamma \vdash v\colon \forall[]\Gamma' \ \mathsf{requires}\ (\lambda_E \uplus \{\alpha\}, \lambda_S, \lambda'_L) \\ \Psi;\Gamma;\Lambda \vdash I & \vdash \Gamma <: \Gamma' \qquad \lambda'_L \subseteq \lambda_L\end{array}}{\Psi;\Gamma;\Lambda \vdash \mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v; I}$$
(T-CRITICALE)

Figure 3.12: Typing rules for instructions (locks) $\Psi;\Gamma;\Lambda \vdash I$

```
  unlockE r1
  yield
}
spinLock ∀[α] (r1: ⟨lock(α)⟩^α) {
  r2:= tslE r1
  if r2= 0
    jump someComputation[α] −− will never happen
  jump spinLock[α]
}
```

The permission of lock α is given to the forked process that will execute the code block release — when a processor is available. Afterwards, a spin lock is performed to obtain lock α. But because the sole processor is busy trying to acquire lock α, the scheduled thread that can release it will never

$$\frac{\Psi, \alpha :: \mathsf{Lock}; \Gamma\{r\colon \langle ?\vec{\tau}\rangle^{\alpha}\}; \Lambda \vdash I \qquad \vec{\tau} \neq \mathsf{lock}(\_), \mathsf{lockS}(\_), \mathsf{lockE}(\_)}{\Psi, \alpha :: \mathsf{Lock}; \Gamma; \Lambda \vdash r := \mathsf{malloc}\ [\vec{\tau}]\ \mathsf{guarded\ by}\ \alpha; I} \quad \text{(T-MALLOC)}$$

$$\frac{\Psi; \Gamma \vdash v\colon \langle \sigma_1..\tau_n..\sigma_{n+m}\rangle^{\alpha} \quad \Psi; \Gamma\{r\colon \tau_n\}; \Lambda \vdash I \quad \tau_n \neq \mathsf{lock}(\_) \quad \alpha \in \Lambda}{\Psi; \Gamma; \Lambda \vdash r := v[n]; I} \quad \text{(T-LOAD)}$$

$$\frac{\begin{array}{c}\Psi; \Gamma \vdash v\colon \tau_n \qquad \Psi; \Gamma \vdash r\colon \langle \sigma_1..\sigma_n..\sigma_{n+m}\rangle^{\alpha} \qquad \tau_n \neq \mathsf{lock}(\_) \\ \Psi; \Gamma\{r\colon \langle \sigma_1..\,\mathrm{type}(\sigma_n)..\sigma_{n+m}\rangle^{\alpha}\}; \Lambda \vdash I \qquad \alpha \in \lambda_E \cup \lambda_L\end{array}}{\Psi; \Gamma; \Lambda \vdash r[n] := v; I} \quad \text{(T-STORE)}$$

$$\frac{\Psi; \Gamma \vdash v\colon \tau \qquad \Psi; \Gamma\{r\colon \tau\}; \Lambda \vdash I}{\Psi; \Gamma; \Lambda \vdash r := v; I} \quad \text{(T-MOVE)}$$

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

where $\mathrm{type}(\tau) = \mathrm{type}(?\tau) = \tau$.

Figure 3.13: Typing rules for instructions (memory) $\Psi; \Gamma; \Lambda \vdash I$

be executed.

The continuation passing style [7] suits programs written in MIL, since they are executed by a stack-less machine. In this programming model the *user* passes a continuation (a label) to a code block that proceeds in the continuation label after it is executed (either by forking or by jumping). It may be useful to pass *user data* to the continuation code (as one of its parameters). Existential types enable abstracting the type of the user data. Let $\mathsf{UserContinuation}$ stands for

$$\forall[\alpha](r_1 : \langle ?\mathsf{int}\rangle^{\alpha})\ \mathsf{requires}\ (;;\alpha)$$

Let $\mathsf{PackedUserData}$ stands for

$$\exists\beta.\langle \forall[\gamma](r_1 : \beta)\ \mathsf{requires}\ (;;\gamma), \beta\rangle^{\alpha}$$

The type $\mathsf{PackedUserData}$ is a pair that is divided into the continuation of type

$$\forall[\gamma](r_1 : \beta)\ \mathsf{requires}\ (;;\gamma)$$

$$\frac{\Psi;\Gamma \vdash v\colon \exists\alpha.\tau \qquad \Psi;\Gamma\{r\colon\tau\};\Lambda \vdash I \qquad \alpha \notin \Psi,\Gamma,\Lambda}{\Psi;\Gamma;\Lambda \vdash \alpha, r := \textsf{unpack}\ v; I} \quad (\text{T-UNPACK})$$

$$\frac{\Psi;\Gamma \vdash v\colon \exists^{\mathsf{L}}\alpha.\tau \qquad \Psi,\beta :: \textsf{Lock};\Gamma\{r\colon\tau\};\Lambda \vdash I \qquad \alpha \notin \Psi,\Gamma,\Lambda}{\Psi;\Gamma;\Lambda \vdash \alpha, r := \textsf{unpack}\ v; I} \quad (\text{T-UNPACKL})$$

$$\frac{\begin{array}{ccc}\Psi;\Gamma \vdash r\colon \textsf{int} & \Psi;\Gamma \vdash v\colon \forall[]\Gamma'\ \textsf{requires}\ (\lambda_E,\lambda_S,\lambda'_L) & \\ \Psi;\Gamma;\Lambda \vdash I & \vdash \Gamma <: \Gamma' & \lambda'_L \subseteq \lambda_L\end{array}}{\Psi;\Gamma;\Lambda \vdash \textsf{if}\ r = 0\ \textsf{jump}\ v; I} \quad (\text{T-BRANCH})$$

$$\frac{\Psi;\Gamma \vdash v\colon \forall[]\Gamma'\ \textsf{requires}\ (\lambda_E,\lambda_S,\lambda'_L) \qquad \vdash \Gamma <: \Gamma' \qquad \lambda'_L \subseteq \lambda_L}{\Psi;\Gamma;\Lambda \vdash \textsf{jump}\ v} \quad (\text{T-JUMP})$$

where $\text{type}(\tau) = \text{type}(?\tau) = \tau$.

Figure 3.14: Typing rules for instructions (memory and control flow) $\Psi;\Gamma;\Lambda \vdash I$

and the user data of type β, respectively. The type of the user data and the type of register r_1, present in the continuation, is the same.

The pattern of usage for a value of type PackedUSerData is to unpack the continuation and the user data, and "apply" the former to the latter — by moving the user data into register r_1, and then jumping to the continuation. The user data is an *opaque* value to the block of code manipulating the pair. The advantages of an opaque user data is twofold. Firstly, the user data is unaltered by the manipulator. Secondly, the code using the pair is not bound to the type of the user data.

Next is an example of how to make a call to a code block that uses the continuation passing style:

```
main() {
  α := newLockLinear
  r2:= malloc[int] guarded by α
  r1:= malloc[ContinuationType, ⟨?int⟩^α] guarded by α
  r1[0] := continuation
  r1[1] := r2
  r1:= pack r1, ⟨?int⟩^α as PackedUserData
  jump library [α]
```

$$\frac{\forall i.\Psi \vdash R(r_i) : \Gamma(r_i)}{\Psi \vdash R : \Gamma} \qquad \text{(reg file, } \boxed{\Psi \vdash R : \Gamma}\text{)}$$

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

$$\frac{\forall i.\Psi \vdash l_i : \forall[\vec{\alpha}_i]\Gamma_i \ \mathsf{requires}\ \_\{\_\} \qquad \Psi \vdash R_i : \Gamma_i[\vec{\beta}_i/\vec{\alpha}_i]}{\Psi \vdash \{\langle l_1, R_1\rangle, \ldots, \langle l_n, R_n\rangle\}}$$

$$\text{(thread pool, } \boxed{\Psi \vdash T}\text{)}$$

$$\frac{\Psi, \vec{\alpha} :: \mathsf{Lock}; \Gamma; \Lambda \vdash I}{\Psi \vdash \forall[\vec{\alpha}]\Gamma \ \mathsf{requires}\ \Lambda\{I\} : \forall[\vec{\alpha}]\Gamma \ \mathsf{requires}\ \Lambda} \qquad \frac{\forall i.\Psi, \alpha :: \mathsf{Lock} \vdash v_i : \sigma_i}{\Psi, \alpha :: \mathsf{Lock} \vdash \langle \vec{v}\rangle^\alpha : \langle \vec{\sigma}\rangle^\alpha}$$

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

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

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

Figure 3.15: Typing rules for machine states

```
}
 library [α](r₁: PackedUserData) requires (;;α) {
  -- do some computation...
  x, r₁:= unpack r₁-- we do not need the packed type here
  r₂:= r₁[0]          -- the continuation
  r₁:= r₁[1]          -- the user data
  jump r₂[α]
}
continuation  ContinuationType {
  -- do some work
}
```

The code block main allocates the user data of type $\langle ?\mathsf{int}\rangle^\alpha$. Afterwards, the user data is stored into a tuple, along with the label pointing to the continuation. The tuple is then packed and passed to the library, which eventually calls the continuation by unpacking the packed data and jumping to the stored label.

Chapter 4

The π-Calculus Run-time

In this chapter we describe a library, written in MIL, that implements the primitives of the π-calculus used to support the generated code, following the design of Lopes et al. [12].

We implement a message passing paradigm in a shared memory architecture. Because communication in the π-calculus version we select is asynchronous, the output process may be represented (in this run-time) by the message being transmitted itself, which is buffered until delivery in the representation of the transmitting channel. We also define a mechanism to *schedule* a process waiting for a message to be delivered, by blocking its execution, and then resuming it, when a message arrives. This mechanism is used to represent the input process and the replicated input process.

The *closure* of a process consists of an *environment* (the variables known by the process itself) and a *continuation* (a pointer to a code block that embodies the process). We store the closure of processes waiting for a message to be delivered in the target channel. When messages are delivered, we recover the state of the process — by applying the message and the environment into the continuation — and resume its execution.

We present a set of macros used to abstract the definition of types and of operations. For each introduced data structure, we define macros for describing its type, for allocating it, and for accessing the values that compose it. We build our library on top of queues defined in the Appendix A. On Section 4.1, we implement channels. Next we describe operations to send through, and to receive from, a channel. Lastly, we extend the definition of channels, associating a channel with its lock in a tuple (Section 4.3).

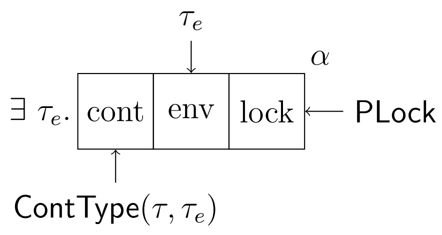

Figure 4.1: The closure of a process

4.1 Channels Queues

In this section, we define the type of a process and the type of a channel queue. A *channel queue* is used for asynchronous message passing between processes, which consists of two pools, one for storing messages and another for storing processes. The continuation of a process is the code block that represents a process, expecting the message being delivered, the environment of the process, and a lock protecting the environment. A process is, therefore, composed by the continuation, the environment, and the environment's lock.

The continuation of a process may be defined by the parametrised type:

$$\mathsf{ContType}(\tau_m,\tau_e) \overset{\mathsf{def}}{=} [\alpha](r_1\colon \tau_m,\ r_2\colon \tau_e,\ r_3\colon \langle\textbf{lock}(\alpha)\rangle^{\alpha})\ \textbf{requires}\ (;\alpha;)$$

Register r_1 (of type τ_m) holds the received message, register r_2 holds the environment of the process (of type τ_e), and register r_3 holds the lock (of type $\langle\mathsf{lock}(\alpha)\rangle^{\alpha}$) that may protect the environment. Lock α is abstracted by the universal operator. Notice that the code block only has permission to read values from the environment, because the environment of a process is a immutable structure, as can be observed in Chapter 5.

The type of the closure of a process, sketched by Figure 4.1, may be defined by:

$$\mathsf{ClosureType}(\tau,\alpha) \overset{\mathsf{def}}{=} \exists\ \tau_e.\langle\mathsf{ContType}(\tau,\tau_e),\ \tau_e,\ \mathsf{PLock}()\rangle^{\alpha}$$

The existential value that abstracts the environment (of type τ_e) of the process (allowing environments of different types) holds a tuple that consists of the continuation, the environment, and the lock type that protects the environment. The continuation process is of type $\mathsf{ContType}(\tau,\tau_e)$, communicating messages of type τ and holding an environment of type τ_e. The lock of the environment is defined by:

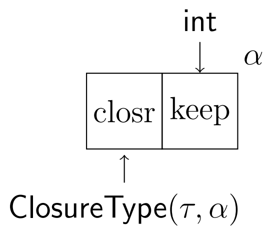

Figure 4.2: A process

$$\mathsf{PLock}() \overset{\text{def}}{=} \exists^{\mathsf{L}}\beta.\langle \mathbf{lock}(\beta)\rangle^{\beta}$$

and is an existential lock value abstracting lock β. Observe that we have extended the definition of closure, by adding the lock of the environment. This extension grants access to data stored inside the environment. The macros for handling closures are:

$$\begin{aligned}
\mathsf{ClosureAlloc}(\tau_m,\tau_e,\alpha) &\overset{\text{def}}{=} \mathbf{malloc}[\mathsf{ContType}(\tau_m,\tau_e),\ \tau_e,\ \mathsf{PLock}()]\ \mathbf{guarded\ by}\ \alpha \\
\mathsf{ClosureCont}(r) &\overset{\text{def}}{=} r[0] \\
\mathsf{ClosureEnv}(r) &\overset{\text{def}}{=} r[1] \\
\mathsf{ClosureLock}(r) &\overset{\text{def}}{=} r[2]
\end{aligned}$$

We may define a process (illustrated by Figure 4.2) as a pair containing the closure and the *keep in channel* flag (of type int):

$$\mathsf{ProcType}(\tau,\alpha) \overset{\text{def}}{=} \langle \mathsf{ClosureType}(\tau,\alpha),\ \mathsf{int}\rangle^{\alpha}$$

The flag is necessary for replicated input processes: after reduction, these processes remain in the channel queue, waiting for more messages. Macros to manipulate processes are:

$$\begin{aligned}
\mathsf{ProcAlloc}(\tau,\alpha) &\overset{\text{def}}{=} \mathbf{malloc}[\mathsf{ClosureType}(\tau,\alpha),\ \mathsf{int}]\ \mathbf{guarded\ by}\ \alpha \\
\mathsf{ProcClosure}(r) &\overset{\text{def}}{=} r[0] \\
\mathsf{ProcKeep}(r) &\overset{\text{def}}{=} r[1]
\end{aligned}$$

The creation of a process is illustrated in the following example, where c has type $\mathsf{ContType}(\mathsf{int},\langle\rangle^{\beta})$ and r_1 has type $\langle \mathsf{lock}(\beta)\rangle^{\beta}$:

```
r3:= malloc [] guarded by β        -- an empty environment is created
r2:= ClosureAlloc( int , ⟨⟩^β, α)   -- alloc the closure of a process
ClosureCont(r2) := c               -- set the continuation to 'c'
```

```
ClosureEnv($r_2$) := $r_3$                 -- set the environment to the one in $r_3$
ClosureLock($r_2$) := $r_1$                -- set the lock of the environment as $\beta$
$r_2$:= pack $\langle\rangle^\beta$, $r_2$ as ClosureType(int,$\alpha$) -- abstract the environment's type
$r_1$:= ProcAlloc(int,$\alpha$)            -- alloc the process itself
ProcClosure($r_1$) := $r_2$                -- set the closure as the one we created
ProcKeep($r_1$) := 0                       -- do not keep this process in the channel
```

In order to create a queue of processes, we need to define a sentinel process. Consider sink:

```
sink [$\alpha$,$\tau_m$,$\tau_e$]($r_1$:$\tau_m$, $r_2$:$\tau_e$, $r_3$:$\langle$lock($\alpha$)$\rangle^\alpha$) requires (;$\alpha$;) {
  unlockS $r_3$
  yield
}
```

Instantiating the universal value sink with τ_m and τ_e (sink[τ_m][τ_e]) results in a value of type ContType(τ_e,τ_m). We may use this continuation in sentinel processes.

The creation of a sentinel process may be defined by macro ProcCreate-Sentinel(r,r_t,τ,α), where:

- r is the register that will refer the process of type ProcType(τ,α);
- r_s is the register that will refer the closure of type ClosureType(τ,α);
- τ is the type of the messages being transmitted;
- α is the lock protecting the channel.

Defined by:

```
ProcCreateSentinel($r$,$r_s$,$\tau$,$\alpha$) $\stackrel{\text{def}}{=}$
    $\delta$,$r$ := newLock 0                  -- create a dummy lock
    $r$ := packL $\delta$,$r$ as PLock()         -- abstract the lock
    $r_s$:= ClosureAlloc($\tau$, int, $\alpha$)    -- alloc the closure of the process
    ClosureCont($r_s$) := sink[int][$\tau$]  -- set the continuation to 'sink'
    ClosureEnv($r_s$) := 0                  -- set value 0 as the environment
    ClosureLock($r_s$) := $r$                -- set the lock of the environment as $r$
    $r_s$:= pack int,$r_s$ as ClosureType($\tau$,$\alpha$) -- abstract the environment
    $r$ := ProcAlloc($\tau$,$\alpha$)              -- alloc the sentinel process
    ProcClosure($r$) := $r_s$                -- set the closure as the one in $r_s$
    ProcKeep($r$) := 0                      -- do not keep this process in the queue
```

Channel queues (Figure 4.3) may be defined by:

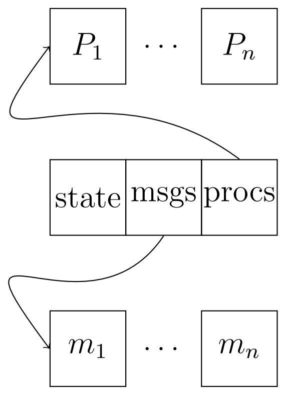

Figure 4.3: A channel queue

$$\mathsf{ChannelQueueType}(\tau,\alpha) \overset{\mathsf{def}}{=} \langle \mathsf{int},\ \mathsf{QueueType}(\tau,\alpha),\ \mathsf{QueueType}(\mathsf{ProcType}(\tau,\alpha),\alpha)\ \rangle^{\alpha}$$

representing a tuple, protected by lock α, that holds a flag (of type int) marking the kind of contents of the channel queue, a queue of messages (of type $\mathsf{QueueType}(\tau,\alpha)$), and a queue of processes (of type $\mathsf{QueueType}(\mathsf{ProcType}(\tau,\alpha),\alpha)$). The flag can assume one of three different values: 0 indicates the channel queue is empty; 1 indicates at least one message enqueued; and 2 represents at least one enqueued process. The macros for manipulating channel queues are:

$$\mathsf{ChannelQueueAlloc}(\tau,\alpha) \overset{\mathsf{def}}{=} \textbf{malloc}[\mathsf{int},\ \mathsf{QueueType}(\tau,\alpha),\ \mathsf{QueueType}(\mathsf{ProcType}(\tau,\alpha),\ \alpha)]\ \textbf{guarded by}\ \alpha$$

$$\begin{aligned}
\mathsf{ChannelQueueState}(r) &\overset{\mathsf{def}}{=} r[0]\\
\mathsf{ChannelQueueMsgs}(r) &\overset{\mathsf{def}}{=} r[1]\\
\mathsf{ChannelQueueProcs}(r) &\overset{\mathsf{def}}{=} r[2]\\
\mathsf{CHANNEL\_QUEUE\_EMPTY} &\overset{\mathsf{def}}{=} 0\\
\mathsf{CHANNEL\_QUEUE\_WITH\_MSGS} &\overset{\mathsf{def}}{=} 1\\
\mathsf{CHANNEL\_QUEUE\_WITH\_PROCS} &\overset{\mathsf{def}}{=} 2
\end{aligned}$$

The creation of a channel queue, having register r_1 type $\mathsf{ProcType}(\mathsf{int},\alpha)$ (as in the result of the previous example), is:

```
$r_4$:= $r_1$                              -- we want to use $r_1$ afterwards
$r_1$:= ChannelQueueAlloc(int, $\alpha$)      -- alloc the channel
-- mark the channel as empty:
ChannelQueueState($r_1$) := CHANNEL_QUEUE_EMPTY
QueueCreateEmpty($r_2$, $r_3$, 0, int, $\alpha$)  -- create an empty queue of integers
ChannelQueueMsgs($r_1$) := $r_2$            -- set the queue of messages as $r_2$
-- create an empty queue of processes; the sentinel is $r_4$:
QueueCreateEmpty($r_2$, $r_3$, $r_4$, ProcType(int,$\alpha$), $\alpha$)
ChannelQueueProcs($r_1$) := $r_2$           -- set the queue of processes as $r_2$
```

The last definition in this section, ChannelQueueCreate(r,r_m,r_t,τ,α), is the creation of a channel queue, where:

- r is the register that will refer the new empty channel of type ChannelQueueType(τ,α);
- r_q is the register that will hold the queue of processes of type QueueType(ProcType(τ,α),α);
- r_e is the register that will point to the sentinel element of the queue of processes;
- v is a sentinel message of type τ;
- τ is the type of the transmitted messages;
- α is the lock protecting the channel.

```
ChannelQueueCreate($r$,$r_q$,$r_e$,$v$,$\tau$,$\alpha$) $\overset{\text{def}}{=}$
    $r$ := ChannelQueueAlloc($\tau$,$\alpha$)      -- create the new channel
    -- flag it as empty:
    ChannelQueueState($r$) := CHANNEL_QUEUE_EMPTY
    -- create an empty queue of messages, where $v$ is the sentinel :
    QueueCreateEmpty($r_q$,$r_e$,$v$,$\tau$,$\alpha$)
    ChannelQueueMsgs($r$) := $r_q$          -- set the channel of messages
    ProcCreateSentinel($r_q$,$r_e$,$\tau$,$\alpha$)     -- create a dummy process
    -- create a queue of processes, where the sentinel is in $r_q$:
    QueueCreateEmpty($r_q$,$r_e$,$r_q$,ProcType($\tau$,$\alpha$),$\alpha$)
    ChannelQueueProcs($r$) := $r_q$         -- set the queue of processes
```

Notice that value v is unaltered in this macro. Furthermore, the value v may only be register r_q.

4.2 Communication

In this section we list the two operations responsible for communications using channels: sendMessage and receiveMessage. The former sends a message through a channel. The latter expects a continuation, where the message will be received, and an environment (user data that is available in the continuation). In either case, when a thread jumps to any of these operations, it is not guaranteed that the processor will be yield, or continue executing. Both operations require exclusive access to the lock of the channel, since they alter its internal state, thus representing a point of contention (in regards to the protecting lock).

Send message. We define an operation to send a message through a channel queue (depicted in Figure 4.4):

```
sendMessage [$\alpha$,$\tau$] -- the protecting lock and the message's type are abstracted
                    ($r_1$: $\tau$,                        -- the message being sent
                     $r_2$: ChannelQueueType($\tau$,$\alpha$),  -- the target channel
                     $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$)              -- the channel's lock
                     requires ($\alpha$;;) {               -- exclusive access to the
                         channel
  $r_4$:= ChannelQueueState($r_2$) -- get the state of the channel
  -- verify if there are process waiting for a message
  if $r_4$= CHANNEL_QUEUE_WITH_PROCS
    -- when there are, deliver the message:
    jump sendMessageReduce[$\tau$][$\alpha$]
  -- else no processes in the channel. enqueue the message
  --
  -- flag the channel as containing messages:
  ChannelQueueState($r_2$) := CHANNEL_QUEUE_WITH_MSGS
  $r_2$:= ChannelQueueMsgs($r_2$) -- get the queue of messages
  QueueAdd($r_2$,$r_1$,$r_4$,$r_1$,$\tau$,$\alpha$)   -- put the message ($r_1$) in the queue
  unlockE $r_3$                 -- unlock the channel's lock
  yield                         -- release control of the processor
}
```

This operation either jumps to the code block sendMessageReduce, when there are processes waiting for a message to arrive, or yields the processor's control, after placing the message in the channel.

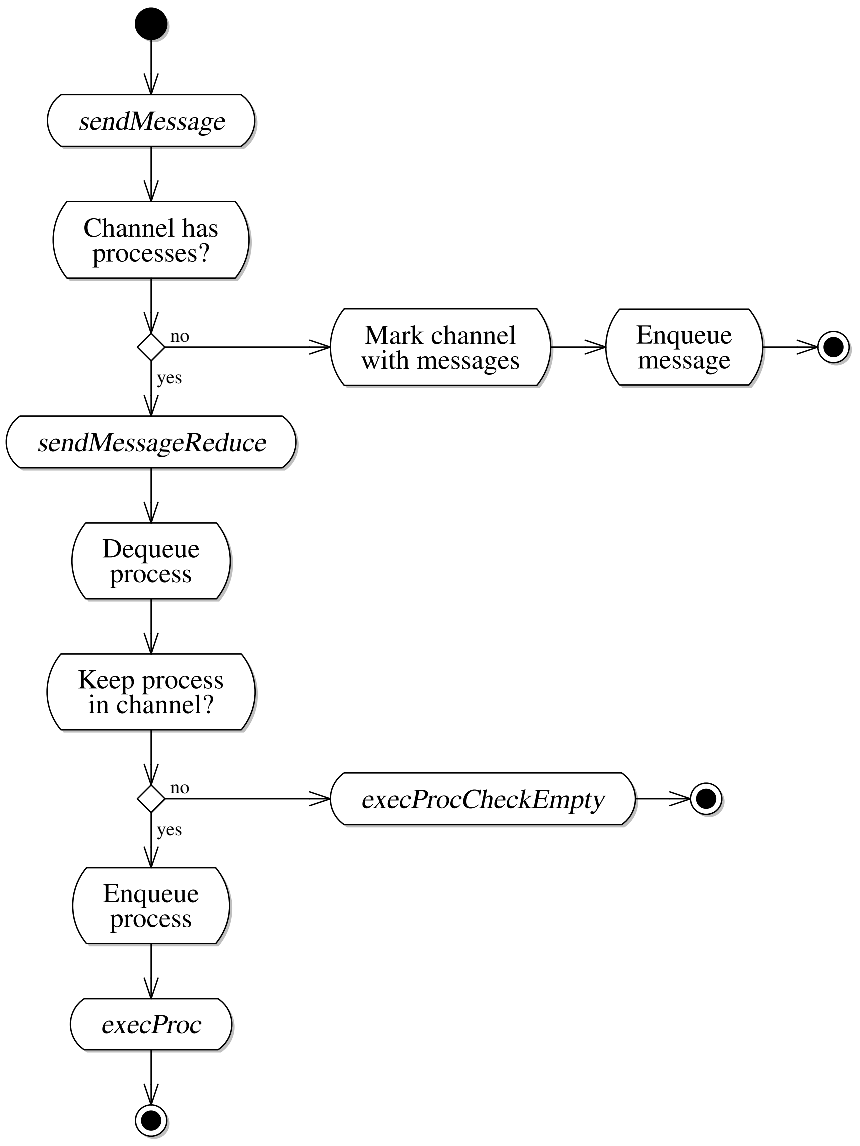

Figure 4.4: The activity diagram outlines the execution of sendMessage, where activities in italic represent code blocks (identified by their labels).

In the following, we list the code block sendMessageReduce. This code block assumes that there is at least one processes waiting for a message to arrive.

```
sendMessageReduce[$\alpha$,$\tau$] -- the protecting lock and the message's type are
    abstracted
                  ($r_1$: $\tau$,                       -- the message being sent
                   $r_2$: ChannelQueueType($\tau$,$\alpha$), -- the target channel
                   $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$)       -- the channel's lock
                   requires ($\alpha$;;) {           -- exclusive access to $\alpha$
  $r_6$:= ChannelQueueProcs($r_2$) -- moves the queue of processes to $r_6$
  -- remove a process from the queue, and move it into $r_4$
  -- $r_5$ holds the number of processes in the queue:
  QueueRemove($r_6$,$r_5$,$r_4$)
  $r_7$:= ProcKeep($r_4$)            -- check if the process is to be kept
  if $r_7$= 0
    -- do not keep it in channel, then deliver the message
    jump execProcCheckEmpty[$\tau$][$\alpha$]
  -- keep the process in the channel after delivery
  $r_7$:= $r_4$                       -- move the message into $r_7$
  -- add process the back into the queue:
  QueueAdd($r_6$,$r_7$,$r_8$,$r_4$,ProcType($\tau$,$\alpha$),$\alpha$)
  -- deliver the message to the process (in $r_4$)
  -- we are certain that the state of the channel is unaltered (with processes)
  unlockE $r_3$                      -- no exclusive access needed
  jump execProc[$\tau$][$\alpha$]            -- deliver the process
}
```

We begin by removing the process from the channel. After that, we verify if we need to put the process back in the channel, by verifying its keep in channel flag. We then jump to execProcCheckEmpty.

Next, is the code block execProcCheckEmpty:

```
execProcCheckEmpty [$\alpha$,$\tau$] -- the protecting lock and the message's type are
    abstracted
                  ($r_1$: $\tau$,                       -- the message being sent
                   $r_2$: ChannelQueueType($\tau$,$\alpha$), -- the target channel
                   $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$,        -- the channel's lock
                   $r_4$: ProcType($\tau$,$\alpha$),         -- the receiving process
                   $r_5$: int)                   -- the remaining processes
                   requires ($\alpha$;;) {           -- exclusive access to $\alpha$

```

```
  -- verify if the channel became empty:
  if $r_5$= 0
    -- the channel became empty, mark and then reduce:
    jump execProcFirstEmpty[$\tau$][$\alpha$]  -- mark it as empty and continue delivery
  -- no need to change the channel's state , continue with the reduction
  --
  -- we don't need access to the channel to deliver the message
  unlockE $r_3$
  jump execProc[$\tau$][$\alpha$]
}
```

The code block checks if the channel needs to be marked as empty, before delivering the message to the process (in code block code block execProc).

We depict code block execProcFirstEmpty:

```
execProcFirstEmpty [$\alpha$,$\tau$] -- the protecting lock and the message's type are
    abstracted
                  ($r_1$: $\tau$,                          -- the message being sent
                   $r_2$: ChannelQueueType($\tau$,$\alpha$),  -- the target channel
                   $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$,              -- the channel's lock
                   $r_4$: ProcType($\tau$,$\alpha$))          -- the receiving process
                   requires ($\alpha$;;) {              -- exclusive access to the
                       channel
  -- mark the channel as empty:
  ChannelQueueState($r_2$) := CHANNEL_QUEUE_EMPTY
  unlockE $r_3$ -- we don't need access to the channel to deliver the message
  jump execProc[$\tau$][$\alpha$]
}
```

where the state of the channel is marked as empty and then continues the delivery.

We present code block

```
execProc [$\alpha$,$\tau$] -- the protecting lock and the message's type are abstracted
                  ($r_1$: $\tau$,                          -- the message being sent
                   $r_2$: ChannelQueueType($\tau$,$\alpha$),  -- the target channel
                   $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$,              -- the channel's lock
                   $r_4$: ProcType($\tau$,$\alpha$)){         -- the receiving process
  -- spin lock to acquire the channel's lock:
  $r_5$:= tslS $r_3$ -- we only need read access to the channel
  if $r_5$= 0
    -- we have access, continue with the delivery :
```

```
    jump execContinue[$\tau$][$\alpha$]
  -- try again:
  jump execProc[$\tau$][$\alpha$]
}
```

The code block spins to get shared access to the channel's lock. Upon success it jumps to execContinue.

Following is a list of code block execContinue:

```
execContinue[$\alpha$,$\tau$] -- the protecting lock and the message's type are abstracted
                  ($r_1$: $\tau$,                          -- the message being sent
                   $r_2$: ChannelQueueType($\tau$,$\alpha$), -- the target channel
                   $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$,          -- the channel's lock
                   $r_4$: ProcType($\tau$,$\alpha$))         -- the receiving process
                   requires (;$\alpha$;) {               -- shared access to the channel
  $r_4$:= ProcClosure($r_4$)  -- get the closure of the process
  $\tau_e$, $r_4$:= unpack $r_4$      -- unpack it and get the type of the environment ($\tau_e$)
  $r_5$:= $r_3$                 -- move channel's lock into $r_5$
  $r_3$:= ClosureLock($r_4$)  -- get the packed lock of the environment
  $\beta$, $r_3$:= unpack $r_3$      -- unpack the lock of the environment
  $r_2$:= ClosureEnv($r_4$)   -- move the environment into $r_2$
  $r_4$:= ClosureCont($r_4$)  -- move the continuation into $r_4$
  unlockS $r_5$             -- we don't need more access to the channel's lock
  -- try to get the lock of the environment, then continue delivery :
  jump execProcGrabLock[$\tau_e$][$\tau$][$\beta$]
}
```

The closure is unpacked, as well as the associated lock, and the lock of the channel released. After that, it jumps to execProcGrabLock, to get the lock of the environment.

We show the code block

```
execProcGrabLock [$\alpha$,$\tau_m$,$\tau_e$] -- the protecting lock, the message's type, and the
    type of the environment are abstracted
                          ($r_1$: $\tau_m$,                -- the message
                           $r_2$: $\tau_e$,                -- the environment
                           $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$,       -- the environment's lock
                           $r_4$: ContType($\tau_m$,$\tau_e$)) { -- the continuation
  -- spin lock to get the shared access:
  $r_5$:= tslS $r_3$
  if $r_5$= 0

```

```
    -- we have shared access, jump to continuation:
    jump $r_4[\alpha]$
  -- try again:
  jump execProcGrabLock$[\tau_e][\tau_m][\alpha]$
}
```

which is a spin lock (for a shared lock) that jumps to the continuation when it is successful.

Receive a message. We describe receiveMessage, another core operation, where we schedule a process to receive a message. The outline of the algorithm, depicted by Figure 4.5, is to place the process in the buffer, if there are no messages to be delivered. Otherwise, we execute the process. When a process has the flag to keep in the channel queue set, all the messages in the channel queue are consumed by that process.

```
receiveMessage [$\alpha$,  -- the lock of the channel,
                $\beta$,   -- the lock of the environment,
                $\tau_m$,  -- the type of the message,
                $\tau_e$]  -- and the type of the environment, are abstracted
                    ($r_1$: ContType($\tau_m$,$\tau_e$),           -- the continuation
                     $r_2$: ChannelQueueType($\tau_m$,$\alpha$), -- the target channel
                     $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$,             -- the channel's lock
                     $r_4$: $\tau_e$,                     -- the environment
                     -- the environment's lock:
                     $r_5$: $\langle$lock($\beta$)$\rangle^\beta$,
                     -- the keep in channel flag:
                     $r_6$:int)
                     requires ($\alpha$;;) {
  $r_5$:= packL $\beta$,$r_5$ as PLock()  -- pack the environment's lock
  $r_7$:= ClosureAlloc($\tau_m$,$\tau_e$,$\alpha$) -- alloc the closure
  ClosureCont($r_7$) := $r_1$ -- set the continuation
  ClosureEnv($r_7$) := $r_4$  -- set the environment
  ClosureLock($r_7$) := $r_5$ -- set the packed lock
  $r_7$:= pack $\tau_e$,$r_7$ as ClosureType($\tau_m$,$\alpha$) -- abstract the environment of closure
  $r_4$:= ProcAlloc($\tau_m$,$\alpha$) -- alloc a process
  ProcClosure($r_4$) := $r_7$ -- set the packed closure of the process
  ProcKeep($r_4$) := $r_6$   -- set the keep on channel flag
  -- test the state of the channel:
  $r_1$:= ChannelQueueState($r_2$)
```

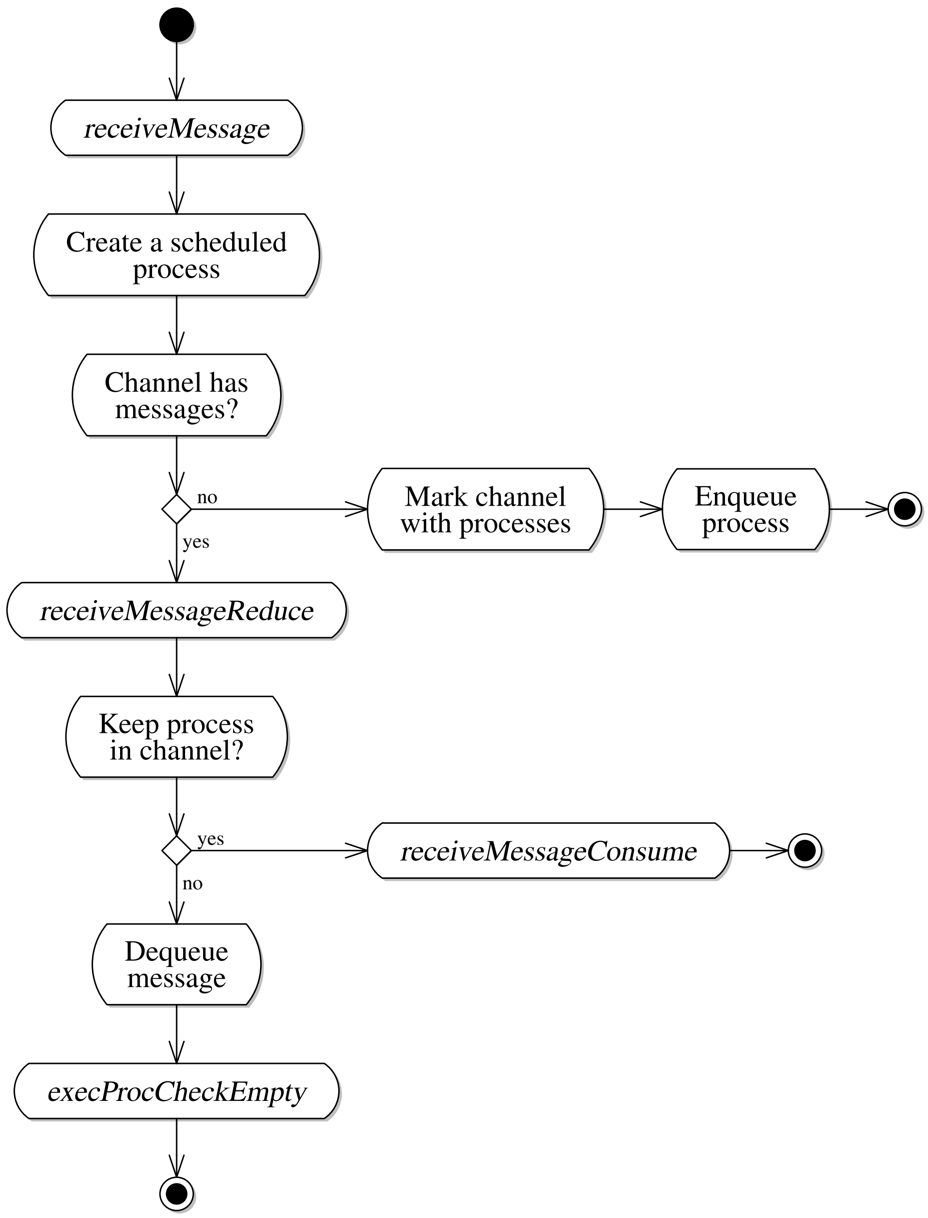

Figure 4.5: The activity diagram outlines the execution of receiveMessage, where activities in italic represent code blocks (identified by their labels).

```
    if $r_1$= CHANNEL_QUEUE_WITH_MSGS
      −− when there are messages, consume the message:
      jump receiveMessageReduce[$\tau_m$][$\alpha$]
    −− otherwise enqueue the process
    −− mark the channel with procs:
    ChannelQueueState($r_2$) := CHANNEL_QUEUE_WITH_PROCS
    $r_1$:= ChannelQueueProcs($r_2$)  −− get the queue of processes
    −− place the process in the queue:
    QueueAdd($r_1$,$r_4$,$r_5$,$r_4$,ProcType($\tau_m$,$\alpha$),$\alpha$)
    unlockE $r_3$                     −− we don't need to alter the channel anymore
    yield                             −− give the control of the processor back
}
```

Next, we present the code block receiveMessageReduce. In Figure 4.6 we show a flow-chart that depicts the execution of the following block of code.

```
receiveMessageReduce [$\alpha$,$\tau$] −− the protecting lock and the message's type are
      abstracted
                    ($r_2$: ChannelQueueType($\tau$,$\alpha$),  −− the target channel
                     $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$,                 −− the channel's lock
                     $r_4$: ProcType($\tau$,$\alpha$))          −− the receiving process
                     requires ($\alpha$;;) {            −− exclusive access
    −− first, we check if the process stays in the channel after delivery :
    $r_1$:= ProcKeep($r_4$)
    if $r_1$= 1
      −− if so consume all messages in the channel:
      jump receiveMessageConsume[$\tau$][$\alpha$]
    −− otherwise remove a message from the channel and proceed with delivery
    −− get the queue of messages:
    $r_6$:= ChannelQueueMsgs($r_2$)
    −− remove one message from the queue, keeping it in $r_1$
    QueueRemove($r_6$,$r_5$,$r_1$)
    −− verify if the channel is empty and then reduce:
    jump execProcCheckEmpty[$\tau$][$\alpha$]
}
```

Next, we depict the code block receiveMessageConsume. This code block just prepares the registers for code block receiveAllMessages.

```
receiveMessageConsume[$\alpha$,$\tau$] −− the protecting lock and the message's type are
      abstracted
```

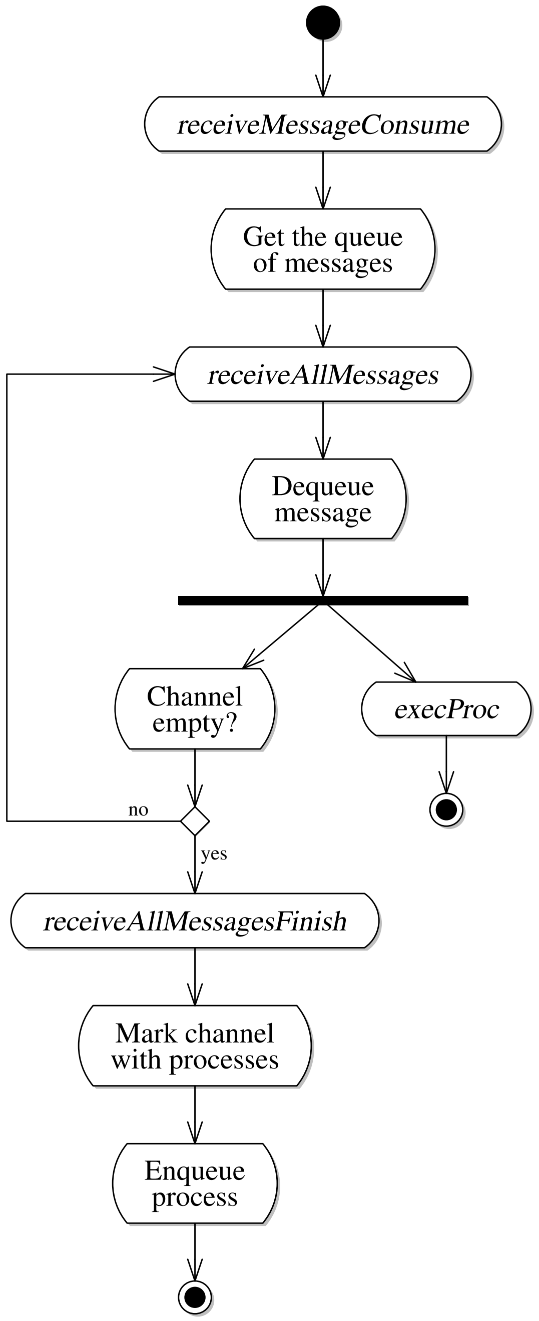

Figure 4.6: The activity diagram outlines the execution of receiveMessage-Consume, where activities in italic represent code blocks (identified by their labels).

```
                    ($r_2$: ChannelQueueType($\tau$,$\alpha$),  -- the target channel
                     $r_3$: ⟨lock($\alpha$)⟩$^\alpha$,          -- the channel's lock
                     $r_4$: ProcType($\tau$,$\alpha$))          -- the receiving process
                     requires ($\alpha$;;) {                    -- exclusive access
  $r_5$:= ChannelQueueMsgs($r_2$) -- move the queue of messages into $r_5$
  -- start the consuming loop:
  jump receiveAllMessages[$\tau$][$\alpha$]
}
```

Following, we show the code block receiveAllMessages, which consumes all messages that exist in the queue, while forking the deliveries.

```
receiveAllMessages [$\alpha$,$\tau$] -- the protecting lock and the message's type are
    abstracted
                    ($r_2$: ChannelQueueType($\tau$,$\alpha$),  -- the target channel
                     $r_3$: ⟨lock($\alpha$)⟩$^\alpha$,          -- the channel's lock
                     $r_4$: ProcType($\tau$,$\alpha$),          -- the receiving process
                     $r_5$: QueueType($\tau$,$\alpha$))         -- the queue of messages
                     requires ($\alpha$;;) {                    -- exclusive access
  QueueRemove($r_5$,$r_6$,$r_1$)  -- remove one message, moving it to $r_1$
  -- deliver the removed message:
  fork execProc[$\tau$][$\alpha$]      -- start the process in a different thread
  if $r_6$= 0                   -- check if the channel is empty
    -- channel is empty, finish loop:
    jump receiveAllMessagesFinish [$\tau$][$\alpha$]
  -- continue looping:
  jump receiveAllMessages[$\tau$][$\alpha$]
}
```

Finally, we describe the code block receiveAllMessagesFinish that places the process in the channel and updates its state.

```
receiveAllMessagesFinish [$\alpha$,$\tau$](-- the target channel:
                                  $r_2$: ChannelQueueType($\tau$,$\alpha$),
                                  $r_3$: ⟨lock($\alpha$)⟩$^\alpha$,       -- the channel's lock
                                  $r_4$: ProcType($\tau$,$\alpha$)       -- the replicated process
                                  ) requires ($\alpha$;;) {   -- exclusive access
  -- mark the channel with processes:
  ChannelQueueState($r_2$) := CHANNEL_QUEUE_WITH_PROCS
  $r_1$:= ChannelQueueProcs($r_2$)                 -- get the queue of processes
  QueueAdd($r_1$,$r_4$,$r_2$,$r_4$,ProcType($\tau$,$\alpha$),$\alpha$)  -- add the process to the queue

  unlockE r3                          -- unlock the channel's lock
  yield                               -- stop execution
}
```

4.3 Channels

We extend our run-time to include support for channels with *private* locks (*i.e.* one lock per channel). We define one more type that embodies this concept. We also define two operations analogous to sendMessage and to receiveMessage, but operate on channels with a private lock. The idea is to store a channel queue and the protecting channel in a tuple, while abstracting the lock name with the existential lock value. Every channel in the system is then protected by the same *global lock*, which protects these new data structures.

We distinguish channel queues from channels with private locks: code that targets the former must know the lock that protects it, whereas code that targets the latter is not aware of the lock that protects the channel, but knows the global lock instead. Channel queues may share a lock amongst each other. Channels, however, do not; each channel has a private lock, thus reducing contention.

We may define a *channel* type as

$$\mathsf{ChannelType}(\tau,\alpha) \overset{\mathsf{def}}{=} \exists^{\mathrm{L}}\beta.\langle \mathsf{ChannelQueueType}(\tau,\beta), \langle \mathbf{lock}(\beta)\rangle^{\beta}\rangle^{\alpha}$$

a tuple that includes a channel and the lock that protects it. Parameter τ is the type of the messages being transmitted. Parameter α is the global lock that protects the tuple holding the channel queue and its lock.

We define macros for handling packed channels:

$$\begin{aligned} \mathsf{ChannelAlloc}(\tau,\beta,\alpha) &\overset{\mathsf{def}}{=} \mathbf{malloc}[\mathsf{ChannelQueueType}(\tau,\beta), \langle \mathbf{lock}(\beta)\rangle^{\beta}] \\ &\qquad\qquad \mathbf{guarded\ by}\ \alpha \\ \mathsf{ChannelChannelQueue}(r) &\overset{\mathsf{def}}{=} r[0] \\ \mathsf{ChannelLock}(r) &\overset{\mathsf{def}}{=} r[1] \end{aligned}$$

The initialisation of this structure may be defined by macro Channel-CreateEmpty(r,r_l,r_c,r_q,r_e,v,τ,α), where:

- r is the register that will refer the channel of type ChannelType(τ, α);

- r_l is the register that will refer the abstracted lock protecting the channel queue of type $\exists^{\mathrm{L}}\beta.\ \langle \textbf{lock}(\beta)\rangle^{\beta}$;
- r_c is the register that will point to the channel queue of type ChannelQueueType(τ, β);
- r_q is the register that will hold the queue of processes of type QueueType(ProcType(τ, β), β);
- r_e is the register that will point to the sentinel of the queue of processes;
- v is the sentinel message of type τ;
- τ is the type of the transmitted messages;
- α is the global lock.

```
ChannelCreateEmpty($r$,$r_l$,$r_c$,$r_q$,$r_e$,$v$,$\tau$,$\alpha$) $\overset{\mathsf{def}}{=}$
    $\beta$, $r_l$ := **newLock** −1                  −− *create the channel's lock*
    ChannelQueueCreate($r_c$,$r_q$,$r_e$,$v$,$\tau$,$\beta$)  −− *create the channel queue, protected*
        *by* $\beta$
    **unlockE** $r_l$                             −− *we don't need access to the channel*
    $r$ := ChannelAlloc($\tau$,$\beta$,$\alpha$)            −− *alloc the channel*
    ChannelChannelQueue($r$) := $r_c$          −− *set the channel queue*
    ChannelLock($r$) := $r_l$                  −− *set the private lock*
    −− *abstract the private lock:*
    $r$ := **packL** $\beta$, $r$ **as** ChannelType($\tau$,$\alpha$)
```

Keep in mind that we need exclusive access to the global lock α in order to use this macro. Also that v is unaltered after the expansion of this macro. Notice that v may only be registers r and r_q.

We extend the two operations sendMessage and receiveMessage, by acquiring the global lock with shared access in order to unpack the channel, and then acquiring exclusive access to the channel queue. Finally, the code block jumps to the actual operation. We only list the extension of the operation to send a message, because the extension of receiveMessage is very similar (only changing the register used for temporary operations and the labels of the code blocks).

We list the code block send that tries to acquire the global lock and then jumps to sendUnpack:

```
send [$\alpha$,$\tau$] -- the global lock and the type of the message are abstracted
          ($r_1$: $\tau$,                   -- the message being sent
           $r_4$: ChannelType($\tau$,$\alpha$),  -- the target channel
           $r_5$: $\langle$lock($\alpha$)$\rangle^\alpha$) {        -- the global lock
  -- spin lock to acquire the global lock $\alpha$
  $r_2$:= tslS $r_5$
  if $r_2$= 0
    -- acquired the lock, unpack the channel
    jump sendUnpack[$\tau$][$\alpha$]
  -- try again
  jump send[$\tau$][$\alpha$]
}
```

Next, we show code block sendUnpack that unpacks the channel and loads the channel queue and its lock and then jumps to sendMessageGrabLock:

```
sendUnpack [$\alpha$,$\tau$] -- the global lock and the type of the message are abstracted
          ($r_1$: $\tau$,                       -- the message being sent
           $r_4$: ChannelType($\tau$,$\alpha$),      -- the target channel
           $r_5$: $\langle$lock($\alpha$)$\rangle^\alpha$)             -- the global lock
           requires (;$\alpha$;) {            -- shared access to the global lock
  $\beta$, $r_4$:= unpack $r_4$                 -- unpack the channel
  $r_2$:= ChannelChannelQueue($r_4$)     -- move the channel queue to $r_2$
  $r_3$:= ChannelLock($r_4$)             -- move the channel queue's lock to $r_3$
  unlockS $r_5$                        -- unlock the global lock
  jump sendMessageGrabLock[$\tau$][$\beta$] -- acquire the channel's lock
}
```

Finally, we depict the code block sendMessageGrabLock that spin locks to get exclusive access to the lock protecting the channel and then jumps to sendMessage (after acquiring the lock):

```
sendMessageGrabLock [$\alpha$,$\tau$] -- the lock and the type of the message are
    abstracted
                            ($r_1$: $\tau$,                          -- the message
                             $r_2$: ChannelQueueType($\tau$,$\alpha$),  -- the channel queue
                             $r_3$: $\langle$lock($\alpha$)$\rangle^\alpha$) {             -- the channel's lock
  -- spin lock for exclusive access:
  $r_4$:= tslE $r_3$
  if $r_4$= 0
    -- send the message:
    jump sendMessage[$\tau$][$\alpha$]
  -- try again:
  jump sendMessageGrabLock[$\tau$][$\alpha$]
}
```

Concerning the extension of the operation to receive a message, we show the type of the code block receive:

```
receive  [$\alpha$, -- the global lock,
          $\beta$,   -- the environment's lock,
          $\tau_m$,  -- the type of the message,
          $\tau_e$]  -- and the type of the environment are abstracted
                ($r_1$: ContType($\tau_m$,$\tau_e$),      -- the continuation
                 $r_2$: $\tau_e$,                     -- the environment
                 $r_3$: $\langle$lock($\beta$)$\rangle^\beta$,              -- the environment's lock
                 $r_4$: ChannelType($\tau_m$,$\alpha$),    -- the target channel
                 $r_5$: $\langle$lock($\alpha$)$\rangle^\alpha$,              -- the global lock
                 $r_6$: int)                  -- the flag keep in channel
```

For convention's sake, the label of the code block that unpacks the channel is receiveUnpack; the label of the code block that grabs the lock of the channel queue is named receiveMessageGrabLock. The implementation of the three code blocks is straight forward.

Chapter 5

Translating the π-calculus into MIL

The translation from the π-calculus into MIL comprises three parts: the translation of types with function $\mathcal{T}[\![\cdot]\!](\gamma)$, the translation of values with function $\mathcal{V}[\![\cdot]\!](\vec{x}, r)$, and the translation of processes with function $\mathcal{P}[\![\cdot]\!](\Gamma)$. Our translation functions are conditioned by the π-calculus run-time (Chapter 4).

We begin by defining the function that translates types:

$$\mathcal{T}[\![int]\!](\gamma) \stackrel{\text{def}}{=} \mathsf{int} \qquad \mathcal{T}[\![(T)]\!](\gamma) \stackrel{\text{def}}{=} \mathsf{ChannelType}(\mathcal{T}[\![T]\!](\gamma), \gamma)$$

Parameter γ is the global lock for pairing a channel with its protective lock. Recall that γ is protecting the structure that holds the channel queues, not affecting the operations on channels themselves.

Notwithstanding, two processes running in parallel that are creating each a channel at the same time are serialised because of the global lock. As are two processes wanting to read values from different channels at the same time. Contention is not critical, however, since the creation of channels is less usual than other operations on channels.

For simplicity we create a new environment whenever a new name is defined. The motivation is twofold. First, immutable environments may be shared among threads *without* contention. Second, processors may increase performance, by exploiting the locality of frames [3]. This is the motivation, in the run-time library, for continuations of processes requiring shared access to environments (reflecting its usage in the translation). When we create a

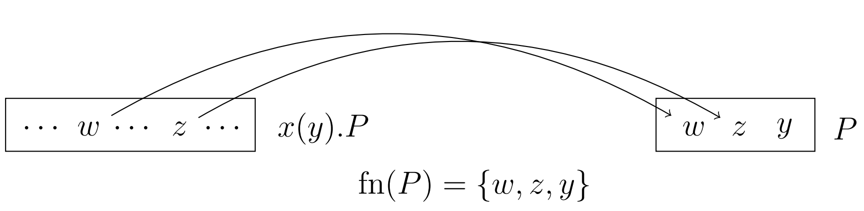

Figure 5.1: Example of the creation of a new environment, based on a old one.

new environment, we only copy the free names of that process (Figure 5.1), therefore attaining an optimised memory usage (in what concerns to possible values to copy).

The macro related to environments may be defined as:

$$\mathsf{EnvType}(\vec{x}, \Gamma, \gamma, \alpha) \overset{\text{def}}{=} \langle \mathcal{T}[\![\Gamma(x_0)]\!](\gamma), \cdots, \mathcal{T}[\![\Gamma(x_n)]\!](\gamma) \rangle^{\alpha}$$
$$\mathsf{EnvAlloc}(\vec{x}, \Gamma, \gamma, \alpha) \overset{\text{def}}{=} \mathsf{malloc}\ [\mathcal{T}[\![\Gamma(x_0)]\!](\gamma), \cdots, \mathcal{T}[\![\Gamma(x_n)]\!](\gamma)]\ \mathsf{guarded\ by}\ \alpha$$

The type of each name is translated into MIL.

The translation of values is straightforward:

$$\mathcal{V}[\![v]\!](\vec{x}, r) \overset{\text{def}}{=} \begin{cases} v & \text{if } v \text{ is of type } \textit{baseval} \\ r[i] & \text{if } v = x_i \text{ where } \vec{x} = x_0 \cdots x_i \cdots x_n \end{cases}$$

Since there exists a one-to-one relation of literals between source and target languages, if it is a literal value, then we use it. If it is a variable, then we must get it from the environment (held in r).

We are now ready to define the translation of processes. The translation becomes simpler because the run-time library supports the channel communication. Yet, we must still maintain the environments of processes.

We start by defining the top-level translation function that defines code block main and code block grabLock and further translates process P:

$\mathcal{P}[\![P]\!](\Gamma) \stackrel{\text{def}}{=}$

$\mathsf{ChannelsLock} = \langle \mathbf{0} \rangle^{\gamma} : \langle \mathsf{lock}(\gamma) \rangle^{\gamma}$
$\mathsf{continuationType}\colon \forall[\alpha, \tau](r_2\colon \tau,\ r_3\colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha})\ \mathsf{requires}\ (;\alpha;)$

```
grabLock $\forall[\alpha,\tau]$(
      $r_2\colon \tau$,
      $r_3\colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha}$,
      $r_4\colon$ continuationType){
   $r_1$ := tslS $r_3$
   if $r_1 = 0$
      jump $r_4[\tau][\alpha]$
   jump grabLock$[\tau][\alpha]$
}
```

```
main () requires ($\gamma$;;) {
   $\alpha, r_3$ := newLock $\mathbf{1}$
   $r_2$ := [] guarded by $\alpha$
   jump $l$
}
```

$\mathcal{P}[\![P]\!](\vec{x}, l, \Gamma, \mathsf{ChannelsLock}, \gamma)$

where $\tau_e = \mathsf{EnvType}(\vec{x}, \Gamma, \gamma, \alpha),\ l$ is fresh

Code block main creates the base (empty) environment and jumps to the translated process pointed by l. Code block (grabLock) is a helper primitive that is used to acquire shared access to the environment. We then begin the actual translation, by providing a fresh label l to the subsequent translation function of processes.

All translated processes share the same type, parametrised by the typing environment Γ, the environment ($\vec{x}$) of the translated process, and by the global lock γ. The register file of the code blocks comprise the environment in r_1 and the lock of the environment in r_3:

$$\begin{aligned}\mathsf{ProcBlock}(\Gamma, \vec{x}, \gamma) \stackrel{\text{def}}{=} \forall[\alpha](r_2\colon &\mathsf{EnvType}(\vec{x}, \Gamma, \gamma, \alpha),\\ &r_3\colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha})\ \mathsf{requires}\ (;\alpha;)\end{aligned} \tag{5.1}$$

The translation of the inactive process is predictable, we just yield the processor's control.

```
$\mathcal{P}[\![\mathbf{0}]\!](\vec{x}, l, \Gamma, g, \gamma) \stackrel{\text{def}}{=}$
$l$ ProcBlock$(\Gamma, \vec{x}, \gamma)$ {
    unlockS $r_3$
    yield
}
```

When translating the output process, we use:

```
$\mathcal{P}[\![\overline{x_i}\langle v\rangle]\!](\vec{x}, l, \Gamma, g, \gamma) \stackrel{\text{def}}{=}$
$l$ ProcBlock$(\Gamma, \vec{x}, \gamma)$ {
    $r_1 := \mathcal{V}[\![v]\!](\vec{x}, r_2)$
    $r_4 := r_2[i]$
    unlockS $r_3$
    $r_5 := g$
    jump send$[\tau][\gamma]$
}
where $\tau = \mathcal{T}[\![\Gamma(v)]\!](\gamma)$, $\vec{x} = x_0 \cdots x_i \cdots x_n$
```

We prepare the registers according to the code block send, by moving the translated message into register r_1 and fetching the channel from the environment into register r_4.

The input is translated by preparing the registers of the code block receive, where we send the environment of the translated process. In the continuation we create a new environment (for P) and copy the free names (like x) that are used in P, if variable x is used at all; otherwise we reuse the environment of the translated process. Finally, we create the new environment and then we proceed with the translation of P. (We list this macro in further detail later in Macro 5.2.)

$\mathcal{P}[\![x(y).P]\!](\vec{x}, l, \Gamma, g, \gamma) \overset{\text{def}}{=}$

```
$l$ ProcBlock$(\Gamma, \vec{x}, \gamma)$ {
    $r_4 := r_2[i]$
    unlockS $r_3$
    $r_5 := g$
    $r_1 := l_1$
    $r_6 := 0$
    jump receive$[\tau_e][\tau][\alpha][\gamma]$
}

$l_1$ ContType$(\tau, \tau_e)$ {
    jump $l_2[\alpha]$
}
CreateEnvAndTranslate$(P, y, \tau, \tau_e, \vec{x}, l_2, \Gamma, g, \gamma)$
```

where $\tau = \mathcal{T}[\![\Gamma(y)]\!](\gamma)$, $\tau_e = \mathsf{EnvType}(\vec{x}, \Gamma, \gamma, \alpha)$, l_1 and l_2 are fresh
$\vec{x} = x_0 \cdots x_i \cdots x_n$

When translating the parallel process, we fork the execution of the translated process on the left, and, because we loose the permission of the environment, we try to acquire it and continue executing the translated process on the right. No contention exists in acquiring the lock of the environment, since all threads have share access.

$\mathcal{P}[\![P \mid Q]\!](\vec{x}, l, \Gamma, g, \gamma) \overset{\text{def}}{=}$

```
$l$ ProcBlock$(\Gamma, \vec{x}, \gamma)$ {
    fork $l_1[\alpha]$
    $r_4 := l_2$
    fork grabLock$[\tau][\alpha]$
    yield
}

$\mathcal{P}[\![P]\!](\vec{x}, l_1, \Gamma, g, \gamma)$
$\mathcal{P}[\![Q]\!](\vec{x}, l_2, \Gamma, g, \gamma)$
```

where $\tau = \mathsf{EnvType}(\vec{x}, \Gamma, \gamma, \alpha)$, l_1 and l_2 are fresh

The translation of the restriction is similar to the input, since there is an environment creation. We begin by creating the new channel, if it is used. After that, we create a new environment, continuing the translation of process P.

$\mathcal{P}[\![(\nu\, x\colon (T))\, P]\!](\vec{x}, l, \Gamma, g, \gamma) \overset{\text{def}}{=}$

If $x \in \text{fn}(P)$:

```
$l$ ProcBlock($\Gamma, \vec{x}, \gamma$) {
    $r_1$ := tslE $g$
    if $r_1 = 0$
        jump $l_1[\alpha]$
    jump $l[\alpha]$
}
$l_1$ ProcBlockCont($\tau_e, \gamma$) {
    ValueInit($r_1, (T), \{r_2, r_3\}$)
    $r_2[i] := r_1$
    unlockE ChannelsLock
    jump $l_1[\alpha]$
}
CreateEnvAndTranslate($P, x, \tau, \tau_e, \vec{x}, l_1, \Gamma, g, \gamma$)
where $\tau = \mathcal{T}[\![\Gamma(x)]\!](\gamma)$, $\tau_e$ = EnvType($\vec{x}, \Gamma, \gamma, \alpha$), $l_1$ and $l_2$ are fresh
```

Otherwise:

$\mathcal{P}[\![P]\!](\vec{x}, l_1, \Gamma, g, \gamma)$

Where ProcBlockCont is defined by:

$$\mathsf{ProcBlockCont}(\tau, \gamma) \overset{\text{def}}{=} \forall[\alpha](r_2\colon \tau,\ r_3\colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha})\ \mathsf{requires}\ (\gamma; \alpha;)$$

The translation of the replicated input process is almost the same as the input, but the flag to keep the process in the channel is turned on.

$\mathcal{P}[\![!x(y).P]\!](\vec{x}, l, \Gamma, g, \gamma) \overset{\text{def}}{=}$

```
$l$ ProcBlock($\Gamma, \vec{x}, \gamma$) {
    $r_4 := r_2[i]$
    unlockS $r_3$
    $r_5 := g$
    $r_1 := l_1$
    $r_6 := 1$
    jump receive$[\tau_e][\tau][\alpha][\gamma]$
}
$l_1$ ContType($\tau, \tau_e$) {
    jump $l_2[\alpha]$
}
CreateEnvAndTranslate($P, y, \tau, \tau_e, \vec{x}, l_2, \Gamma, g, \gamma$)
```

where $\tau = \mathcal{T}[\![\Gamma(y)]\!](\gamma)$, τ_e = EnvType($\vec{x}, \Gamma, \gamma, \alpha$), l_1 and l_2 are fresh
$\vec{x} = x_0 \cdots x_i \cdots x_n$

We present the macro to create new environments and then translate a given process:

```
$\mathsf{CreateEnvAndTranslate}(P, x_i, \tau, \tau_e, \vec{x}, l, \Gamma, g, \gamma) \overset{\text{def}}{=}$
If $x_i \in \vec{y}$:
  $l$ $\mathsf{ContType}(\tau, \tau_e)$ {
    $\beta, r_5 :=$ $\mathsf{newLock}$ $\mathbf{-1}$
    $r_4 := \mathsf{EnvAlloc}(P, \Gamma, \gamma, \beta)$
    $\forall y_j \in \vec{y} \setminus \{x_i\} \begin{cases} r_6 := r_2[j] \\ r_4[k] := r_6, \text{ where } y_j = x_k \text{ and } x_k \in \vec{x} \end{cases}$
    $\mathsf{unlockS}$ $r_3$
    $r_3 := r_5$
    $r_2 := r_4$                                                    (5.2)
    $r_2[i] := r_1$
    $\mathsf{unlockE}$ $r_3$
    $r_4 := l_1$
    $\mathsf{jump}$ $\mathsf{grabLock}[\tau_e'][\beta]$
  }
  where $\tau_e' = \mathsf{EnvType}(\vec{y}, \Gamma, \gamma, \beta)$
  $\mathcal{P}[\![P]\!](\vec{y}, l_1, \Gamma, g, \gamma)$
Otherwise:
  $\mathcal{P}[\![P]\!](\vec{x}, l, \Gamma, g, \gamma)$
where $\vec{y} = \mathrm{fn}(P)$, $l_1$ is fresh
```

There are two possible expansions for macro CreateEnvAndTranslate. One expansion is chosen when the name is not used, in which case we skip environment creation and use the provided label as a parameter of the translation. The other expansion is chosen when x_i is a free name, in which case we need to create a new environment, and then translate P.

To create the new environment we allocate a tuple and copy each value from the old environment $\vec{x}$ into the new one $\vec{y}$. The new environment is protected by a new lock in order to make access to environments *always unblocked*. Translation then proceeds with the fresh environment.

The initialisation of values is recursively defined by:

$\mathsf{ValueInit}(r, T, R) \overset{\text{def}}{=}$

If T is int:	Otherwise, considering that T is (T'):
$r := 0$	$r_i \notin R$
	$r_j \notin R \cup \{r_i\}$
	$\mathsf{ValueInit}(r_i, T', R \cup \{r_i\})$
	$\mathsf{ChannelCreateEmpty}(r, r_i, r_j, \mathcal{T}[\![T']\!](\gamma), \gamma)$

If it is an integer, we move the value 0 to the target register. Otherwise, we create an empty channel and move it to the target register.

Chapter 6

Conclusion

In our work we show a type-preserving compiler that translates the π-calculus into MIL. The translation process also tries to preserve the semantics, by taking advantage of the multithreaded architecture of the target language. In MIL we have a finite number of processors, where each executes a π process, thus reduction between an active (in a processor) and an inactive (in the thread pool) process is not possible.

As related work, we take in analysis Pict, a compiler that translates from the π-calculus into C; a compiler that targets a typed assembly language; and TyCo, a framework for compiling process calculi. Pict [23] is a compiler from the π-calculus into C. The main difference between Pict and our compiler is the target architecture: the former targets a sequential machine, whereas the latter targets a multithreaded machine. Thus, there are no concerns about concurrency on Pict. 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 has bind variables to registers. On Pict there is no run-time library. The full code of communication is expanded each time it is used, resulting in more code being generated. The π-calculus version of Pict is richer than the one we use, having support for recursive types, polymorphism, and type inference. In Pict there is concerns about memory usage; MIL abstracts these concerns.

The compiler from Greg Morrisett et al. [17] translates from System-F into TAL (a typed assembly language) in 5 compilation stages. The first compilation stage is conversion to CPS. This does not apply to the π-calculus, since it is a CPS-friendly message. The second compilation step makes environments of functions explicit. Both compilers use the existential value to abstract environment. In their work, packing the environment is done in the translation stage. We pack the environments in the run-time library (less code is generated). The third compilation step, hoisting, defines heap values that consist in code blocks (much like MIL's code blocks). The forth compilation step makes memory allocation explicit. The final translation step is not relevant, since it is a mostly a syntactic translation to TAL. The main difference between works is that our source and target languages are concurrent.

The work [10] presents a framework for compilation of process-calculi. The abstract machine that runs the target language is sequential, thus suffers from the same limitations found in Pict. Contrary to MIL, there are no typing rules for the target language of this work.

Further work includes extending MIL and simplifying the run-time library. We are adding support for read-only tuples (in MIL), thus reducing contention and removing locks from the translation of processes. We are also working on simplifying the run-time library, by enabling the channel queue to hold messages and processes in the same data structure (instead of using two queues). Furthermore, we are pursuing a wait-free implementation of the π-calculus, by instrumenting MIL with compare and swap rather than locks.

Appendix A

Queues

We use a double-ended queue instead of a pool to store messages and scheduled processes in channels. The implementation uses a linearly-linked list, composed by elements (nodes) that are connected sequentially. Notice that the FIFO order ensures fairness. Also the queue being double-ended enables fast adds and fast removes.

The type of an element (Figure A.1) may be parametrised by:

$$\mathsf{ElementType}(\tau,\alpha) \overset{\mathsf{def}}{=} \mu\ \beta.\langle\tau,\beta\rangle^{\alpha}$$

An element is a tuple, protected by lock α, that holds three values: the contents of the element (of type τ), a reference to the previous element (of the recursive type β), and a reference to the next element (also of type of the element, β). The macros for accessing this data structure:

$$\mathsf{ElementAlloc}(\tau,\alpha) \overset{\mathsf{def}}{=} \textbf{malloc}[\tau,\ \mathsf{ElementType}(\tau,\alpha)]\ \textbf{guarded by}\ \alpha$$
$$\mathsf{ElementValue}(r) \overset{\mathsf{def}}{=} r[0]$$
$$\mathsf{ElementNext}(r) \overset{\mathsf{def}}{=} r[1]$$

We show an example of the creation of two elements connected:

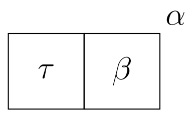

Figure A.1: An element.

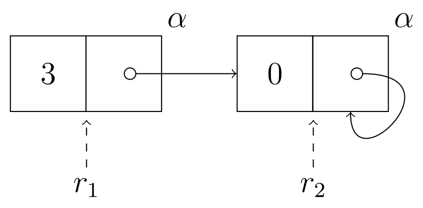

Figure A.2: Two elements connected.

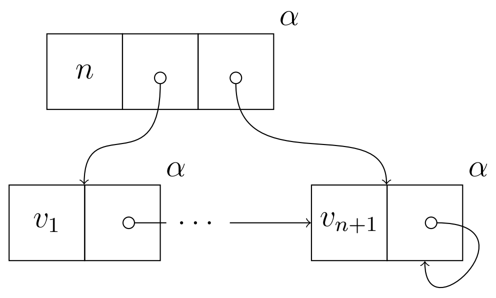

Figure A.3: A queue with n elements.

```
$r_1$:= ElementAlloc(int,$\alpha$)  -- alloc element 1
$r_2$:= ElementAlloc(int,$\alpha$)  -- alloc element 2
ElementValue($r_1$) := 3      -- set the value of element 1 as 3
ElementNext($r_1$) := $r_2$     -- link element 1 to element 2
ElementValue($r_2$) := 0      -- set the value of element 2 as 0
ElementNext($r_2$) := $r_2$     -- link element 2 to  itself
```

Figure A.2 shows the two new elements, which are pointed by registers r_1 and r_2.

We define a queue as

$$\mathsf{QueueType}(\tau,\alpha) \overset{\mathsf{def}}{=} \langle \mathsf{int}, \mathsf{ElementType}(\tau,\alpha), \mathsf{ElementType}(\tau,\alpha)\rangle^{\alpha}$$

The tuple comprises the number of elements in the queue, the first element in the queue, and the last element in the queue, as portrayed by Figure A.3. The associated macros are:

$$\mathsf{QueueAlloc}(\tau,\ \alpha) \overset{\mathsf{def}}{=} \textbf{malloc}\ [\mathsf{int},\ \mathsf{ElementType}(\tau,\ \alpha),\ \mathsf{ElementType}(\tau,\ \alpha)]\ \textbf{guarded by}\ \alpha$$

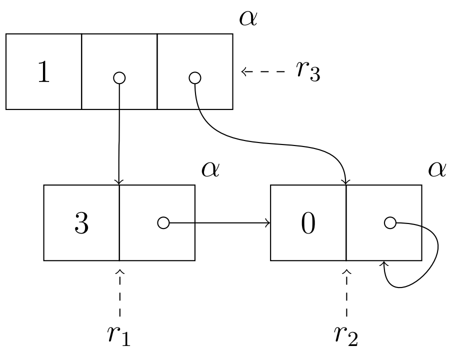

Figure A.4: A queue with one element.

$$\mathsf{QueueLen}(r) \overset{\text{def}}{=} r[0]$$
$$\mathsf{QueueFirst}(r) \overset{\text{def}}{=} r[1]$$
$$\mathsf{QueueLast}(r) \overset{\text{def}}{=} r[2]$$

Consider the elements of the previous example, stored referred by registers r_1 and r_2. The following example, illustrated by Figure A.4, shows the creation of a queue, holding the number 3:

```
1  $r_3$:= QueueAlloc(int,$\alpha$) -- alloc the queue
2  QueueLen($r_3$) := 1         -- set the number of valid elements in the queue
3  QueueFirst($r_3$) := $r_1$       -- point to the first element (the head of the queue)
4  QueueuLast($r_3$) := $r_2$       -- point to the sentinel (the tail of the queue)
```

Our queues have a sentinel element, ensuring that every element has a next value. Thus, even though we have two elements, the second one, pointed by the tail of the queue, does not count as valid.

The initialisation of empty queues (with a sentinel), depicted by Figure A.5 is so common that we also define macro QueueCreateEmpty(r,r_e,v,τ,α), where:

- r is the register that will refer the new queue;
- r_e is the register that will point to the sentinel element;
- v is the value held by the sentinel element (of type τ);
- τ is the type of the contents of the queue (consequently of v as well);

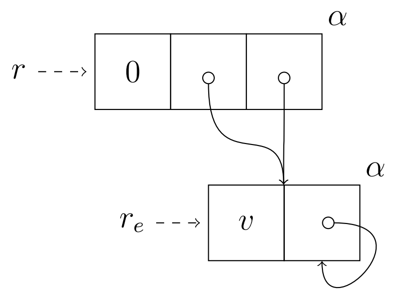

Figure A.5: An empty queue.

- α is the lock protecting the queue.

Defined by:

```
QueueCreateEmpty($r$,$r_e$,$v$,$\tau$,$\alpha$) $\overset{\text{def}}{=}$
    $r_e$:= ElementAlloc($\tau$, $\alpha$)   -- alloc the sentinel
    ElementValue($r_e$) := $v$       -- set the dummy value
    ElementNext($r_e$) := $r_e$       -- link to itself
    $r$ := QueueAlloc($\tau$, $\alpha$)     -- alloc the queue
    QueueLen($r$) := 0             -- this queue is empty
    QueueFirst($r$) := $r_e$          -- point the head to the sentinel
    QueueLast($r$) := $r_e$           -- point the queue to the sentinel
```

Notice that because of the order of the instructions, v can be the same register as r, but not the same as r_e. It is also important to realise that v is not altered after the expansion of this macro.

We define macro QueueAdd(r,r_l,r_e,v,τ,α) to add an element to the queue (delineated by Figure A.6), where:

- r is the register that refers the queue;
- r_l is the register that will store the number of elements of the queue;
- r_e is the register that will refer the sentinel element;
- v is the value to be added to the queue (of type τ);
- τ is the type of the contents of the queue (consequently of v as well);

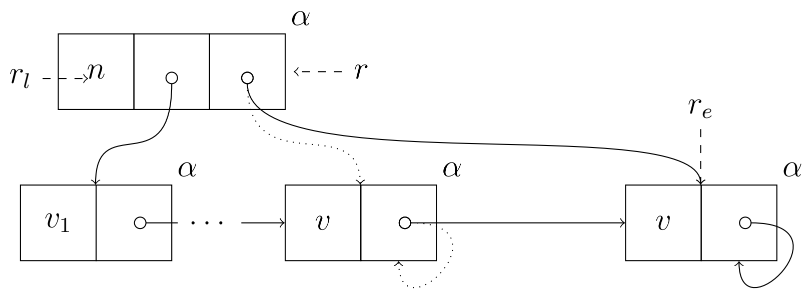

Figure A.6: Adding the n-th value to a queue.

- α is the lock protecting the queue.

Defined by:

```
QueueAdd($r$,$r_l$,$r_e$,$v$,$\tau$,$\alpha$) $\overset{\text{def}}{=}$
    $r_e$:= QueueLast($r$)           -- the sentinel will become the last valid element
    ElementValue($r_e$) := $v$        -- set the value of the last element
    $r_e$:= ElementAlloc($\tau$,$\alpha$)    -- create the new sentinel
    ElementValue($r_e$) := $v$        -- copy the value to the sentinel as well
    ElementNext($r_e$) := $r_e$        -- link new element to itself
    $r_l$:= QueueLast($r$)           -- get the last element again
    ElementNext($r_l$) := $r_e$        -- link it to the sentinel
    QueueLast($r$) := $r_e$           -- point the tail of the queue to the sentinel
    $r_l$:= QueueLen($r$)
    $r_l$:= $r_l$+ 1
    QueueLen($r$) := $r_l$            -- increment the count of elements
```

Notice that, because of the order of the instructions in the macro, the value v may be register r_l but it may not be registers r and r_e.

We now define macro QueueRemove(r,r_l,r_v), illustrated by Figure A.7, to remove a value from a queue, while retaining it in a register. The interface is:

- r is the register that refers the queue;
- r_l is the register that will store the length of the queue;
- r_v is the register that will refer the removed value;

The macro is defined by:

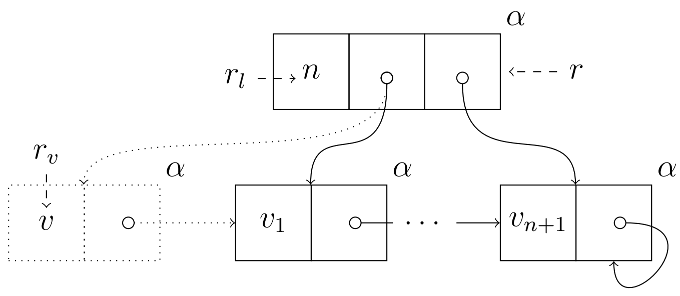

Figure A.7: Removing the first element of a queue.

```
QueueRemove($r$,$r_l$,$r_v$) $\overset{\text{def}}{=}$
    $r_l$ := QueueFirst($r$)       -- *get the first element*
    $r_v$:= ElementValue($r_l$) -- *move removed value to* $r_v$
    $r_l$ := ElementNext($r_l$)   -- *get the second element*
    QueueFirst($r$) := $r_l$      -- *update the first element of the queue*
    $r_l$ := QueueLen($r$)
    $r_l$ := $r_l - 1$
    QueueLen($r$) := $r_l$        -- *increment the count of elements*
```

Bibliography

[1] BEE2 (Berkeley Emulation Engine 2). `http://bee2.eecs.berkeley.edu/`.

[2] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.

[3] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, 1972.

[4] Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *Proceedings of ESOP '99*, volume 1576 of *LNCS*, pages 91–108. Springer, 1999.

[5] Hubert Garavel. Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular. *Electronic Notes in Theoretical Computer Science*, 209:149–164, 2008.

[6] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[7] Jr. Guy Lewis Steele. RABBIT: A Compiler for SCHEME. Master's thesis, MIT AI Lab, 1978.

[8] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pièrre America, editor, *Proceedings of ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[9] Stuart Kent. Model Driven Engineering. In *Proceedings of IFM '02*, pages 286–298. Springer, 2002.

[10] Luís Lopes, Fernando Silva, and Vasco Vasconcelos. Compiling Object Calculi. Technical Report DCC-98-3, University of Oporto, 1998.

[11] 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.

[12] Luís Lopes, Vasco T. Vasconcelos, and Fernando Silva. Fine Grained Multithreading with Process Calculi. *IEEE Transactions on Computers*, 50(9):229–233, 2001.

[13] Francisco Martins. *Controlling Security Policies in a Distributed Environment.* PhD thesis, Faculty of Sciences, University of Lisbon, 2005.

[14] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[15] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[16] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992.

[17] 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.

[18] Jishnu Mukerji and Joaquin Miller eds. *Model Driven Architecture.* Object Management Group, 2001. `http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01`.

[19] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[20] Benjamin C. Pierce. *Advanced Topics In Types And Programming Languages.* MIT Press, 2004.

[21] The RAMP (Research Accelerator for Multiprocessors) project. `http://ramp.eecs.berkeley.edu/`.

[22] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[23] David Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, LFCS, University of Edinburgh, 1996. CST-126-96 (also published as ECS-LFCS-96-345).

[24] Vasco T. Vasconcelos and Francisco Martins. A multithreaded typed assembly language. In *Proceedings of TV '06*, 2006.