ELSEVIER

Electronic Notes in Theoretical Computer Science 241 (2009) 57–84

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Compiling the π-calculus into a Multithreaded Typed Assembly Language

Tiago Cogumbreiro, Francisco Martins, Vasco T. Vasconcelos

Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal

Abstract

We extend a previous work on a multithreaded typed assembly language (MIL) targeted at shared memory multiprocessors, and describe the design of a type-preserving compiler from the π-calculus into MIL. The language enforces a policy on lock usage through a typing system that also ensures race-freedom for typable programs, while allowing for typing various important concurrency patterns. Our translation to MIL generates code that is then linked to a library supporting a generic unbounded buffer monitor, variant of Hoare's bounded buffer monitor, entirely written in MIL. Such a monitor shields client code (the π-calculus compiler in particular) from the hazardous task of direct lock manipulation, while allowing for the representation of π-calculus channels. The compiler produces type correct MIL programs from type correct source code, generating low-contention cooperative multithreaded programs.

Keywords: Pi-calculus, multithreaded assembly language, typed assembly language

1 Introduction

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 evolving than simply balancing workload among cores. In a near future (in less than a decade) Intel prepares to manufacture and ship 80-core processors [13]; programmers must perform a paradigm shift from sequential to concurrent programming and produce, from scratch, software adapted for multi-core platforms.

High-level programming languages must as well undergo substantial transformations in order to make available concurrency primitives at an adequate level of abstraction, balancing the added power of concurrent programming with the increase of complexity in applications. Important operational properties of high-level languages, and therefore of programs written in those languages, are captured via types and enforced using type systems. Ultimately, such properties should be present in the running application, since they contribute to certify its correct execution. One approach to express and verify operational properties of actual running code is to

1571-0661 © 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.06.004

equip assembly languages with types, and take advantage of type safety properties enforced by type systems. Building compilers that preserve typings while translating a high-level typed language into a typed assembly language helps in securing properties of the source language. Our contribution towards this end starts not from an high-level programming language, but from the π-calculus, a concise language from channel-based concurrent computations, and arrive at a multithreaded typed intermediate language (MIL), while preserving typability.

The type system we propose for MIL closely follows the tradition of typed assembly languages [20,21,22], extended with support for threads and locks, following Flanagan and Abadi [8]. With respect to this last work, however, our work is positioned at a much lower abstraction level, and faces different challenges inherent to non-lexically scoped languages. Lock primitives have been discussed in the context of concurrent object calculi [7], JVM [9,10,16,17], C [12], C-- [27], but not in the context of typed assembly (or intermediate) languages. In a typed setting, where programs are guaranteed not to suffer from race conditions, we

- Syntactically decouple the lock and unlock operations from what one usually finds unified in a single syntactic construct in high-level languages: Birrel's *lock-do-end* construct [2], used under different names (*sync*, *synchronized-in*, *lock-in*) in a number of other works, including the Java programming language [4,5,7,8,9,10,12];
- Allow for lock acquisition/release in schemes other than the nested discipline imposed by the *lock-do-end* construct;
- Allow forking threads that hold locks.

We previously introduced a multithreaded typed assembly language (MIL) and its operational semantics, together with a type system that ensures that well-typed programs are free from race conditions [30]. The present paper presents a version of the language that includes:

- *Read-only tuples.* We adhere to the continuation-passing [1] style when writing MIL code, since for simplicity MIL does not provide for stack manipulation. It turns out that in many situations closures are immutable. In particular, closures obtained by translating π-calculus programs are constant. In order to help coding this common pattern we introduce read-only tuples that need not be protected by locks, since they introduce no potential races.
- *Tuple creation in registers.* In order to initialize read-only tuples, these are created directly in registers. As long as the tuple remains local to the (single threaded) processor there is no need to protected it with a lock nor to gain permission to manipulate it. If the tuple is to be shared, the `share` instruction allocates memory in the heap, copies the tuple in the register to the heap, while protecting it with a lock or marking as read-only.
- *Polymorphic types.* We discussed how to introduce universal and existential types in [30]. Here we incorporate both kinds of polymorphism, in particular existential types over locks. Universal and existential types over locks allow a lock variable to escape the static scope where it was declared.

Compilers for several concurrent programming languages based on process calculi have been proposed, including for the TyCO language [18], the Join calculus [11], Pict [26], and HACL [23]. These works, however, target sequential architectures and are not proved to be type-preserving. A proposal for compiling TyCO, concurrent object-based language, into a multithreaded run-time system [24] was conducted in a untyped setting.

Type-preserving compilation, on the other hand, maintains type information throughout each compilation stage. The work from Morrisett *et al.* [22] presents a five stage type-preserving compilation, from System F into a (sequential) typed assembly language.

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 [3,15] and translate it into MIL code. The translation is proved to preserve typability. A by-product of this work is an unbounded buffer monitor, variant of Hoare's bounded buffer monitor [14], entirely written in MIL. The monitor provides, in addition to creation, procedures to append elements to the buffer and to remove elements from the buffer, shielding client code (the π-calculus compiler in particular) from the hazardous task of direct lock manipulation. A type-checker and an interpreter for MIL, as well as the code for the unbounded buffer monitor and the π-to-MIL compiler are available on-line [19].

The outline of the paper is as follows. Sections 2 and 4 describe the target language (MIL) and the source language (π-calculus), respectively. The section on MIL also shows the main result for the language, in the form of the lock discipline imposed by the type system, and race freedom for typable processes. Section 3 introduces the code for the monitor, showing MIL in action. Section 5 defines the translation function itself, together with the result on type preservation. In the closing section we summarise our results and outline directions for further investigation.

2 Target Language: A Multithreaded Intermediate Language

MIL is an assembly language targeted at an abstract multi-processor equipped with a shared main memory. Each processor consists of a series of registers and of a local memory for instructions. Registers may directly contain tuples, representing data local to the processor. In order to use MIL a real system must implement some scheme of local memory. The main memory is divided into a heap and a run pool. The heap stores data and code blocks. A code block declares the registers types it expects, the required locks, and an instruction sequence. The run pool contains suspended threads waiting for a free processor. Figure 1 summarises the MIL architecture.

This section introduces the lock discipline, syntax and operational and static semantics for MIL, and concludes with the main result of the language.

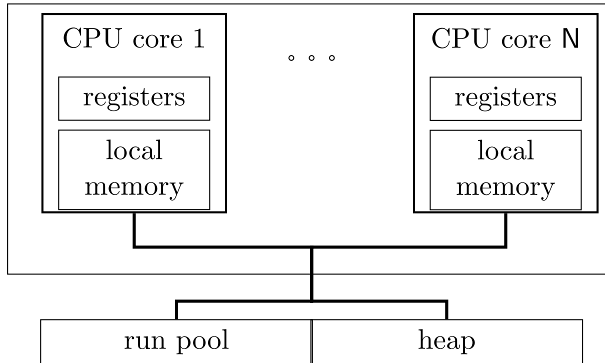

Fig. 1. The MIL architecture.

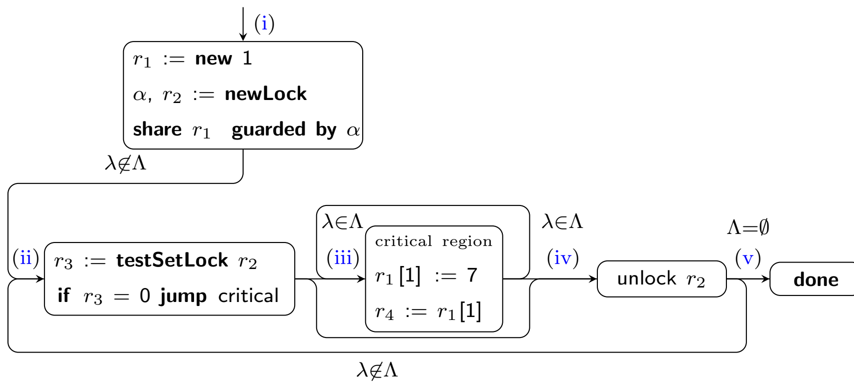

Fig. 2. The lock discipline.

Lock discipline

We provide two distinct access privileges to shared tuples: read-only and read-write, the latter mediated by locks. A standard *test and set lock* instruction is used to obtain a lock, thus allowing a thread to enter a *critical region*. Threads read and write from the shared heap via conventional load and store instructions. The policy for the usage of locks (enforced by the type system) is depicted in Figure 2 (cf. Theorem 2.1), where λ denotes a *singleton lock type* and Λ the set of locks held by the thread (the thread's *permission*). Specifically, the lock discipline enforces that:

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

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

(iii) before accessing the heap, the thread holds the lock (the thread has entered a critical region);

(iv) unlocking can happen only when the lock is held;

(v) thread termination only without held locks.

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

Fig. 3. Instructions.

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

Fig. 4. Abstract machine.

Syntax

The syntax of our language is generated by the grammar in Figures 3, 4, and 9. We rely on a set of *heap labels* ranged over by l, a set of *type variables* ranged over by α and β, and a disjoint set of *singleton lock types* ranged over by λ.

Most of the machine instructions, presented in Figure 3, are standard in assembly languages. Instructions are organised in sequences, ending in jump or in done. Instruction done frees the processor to execute another thread waiting in the thread pool. Our threads are cooperative, meaning that each thread must explicitly release the processor (using the done instruction).

Memory tuples are created locally, directly at processor's registers using the new

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

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

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

Fig. 5. Operational semantics (thread pool).

instruction. To share memory, tuples are transferred to the heap using the share instruction, according to a chosen access policy: read-only or read-write (in which case it must be guarded by a lock).

The *abstract machine*, generated by the grammar in Figure 4, is parametric on the number of available processors, N, and on the number of registers per processor, 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 of fixed length N. Heaps are maps from labels into *heap values* that may be tuples or code blocks. *Tuples* are vectors of mutable values protected by some lock λ, or else of constant values (identified by tag ro). Code blocks comprise a signature and a body. The signature of a code block describes the type of the registers and the locks that must be held by the thread when jumping to the code block. The body is a sequence of instructions to be executed by a processor.

A thread pool is a multiset of pairs, each of which contains the address (*i.e.,* a label) of a code block and a register file. A processor array contains N processors, each of which is composed of a register file, a set of locks (the locks held by the thread running at the processor), and a sequence of instructions (the instructions that remain to execute).

Operational Semantics

The operational semantics is presented in Figures 5 to 8. The run pool is managed by the rules in Figure 5. Rule R-HALT stops the machine when it finds an empty thread pool and all processors idle, changing the machine state to halt. Otherwise, if there is an idle processor and a thread waiting in the pool, then rule R-SCHEDULE assigns the thread to the idle processor. Rule R-FORK places a new thread in the pool; the permissions of the thread are split in two—those required by the forked code, and the remaining ones—the thread keeps the latter set.

Operational semantics concerning locks are depicted in Figure 6. The instruction newLock creates a new lock ρ ready to be acquired, whose scope is the rest of the code block. A tuple $\langle \mathbf{0}\rangle^{\rho}$, representing the value of the lock, is allocated in the heap and register r is made to point it. The lock value is within a uni-dimensional tuple because the machine provides for tuple allocation only; lock ρ is used for type safety purposes (just like all other singleton types). The *test and set lock*

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

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

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

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

Fig. 6. Operational semantics (locks).

$$\frac{P(i) = \langle R; \Lambda; (r := \mathsf{new}\ n; I)\rangle \qquad |\vec{0}| = n}{\langle H; T; P\rangle \to \langle H; T; P\{i\colon \langle R\{r\colon \langle \vec{0}\rangle\}; \Lambda; I\rangle\}\rangle} \quad \text{(R-NEW)}$$

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

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

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

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

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

Fig. 7. Operational semantics (memory).

instruction, present in many machines designed for multi-threading, is an atomic operation that loads the contents of a word into a register and then stores another value in that word. When a testSetLock is applied to an unlocked state, the type variable λ is added to the permissions of the processor and its value becomes $\langle \textbf{-}1\rangle^{\lambda}$. Locks are waved using instruction unlock, as long as the thread holds the lock.

Rules related to memory manipulation are described in Figure 7. They rely on the evaluation function $\hat{\mathrm{R}}$ that looks for values in registers, in the pack constructor, and in the application of universal types.

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

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

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

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

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

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

Fig. 8. Operational semantics (control flow).

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

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

The transition rules for control flow, illustrated in Figure 8, are straightforward [25].

Type Discipline

The syntax of types is depicted in Figure 9. A type of the form $\langle\vec{\tau}\rangle^{\pi}$ describes a heap allocated tuple: shared and protected by a lock λ if π is λ, or shared and read-only if π is ro (*cf.* Figure 4). A type $\langle\vec{\tau}\rangle$ describes a tuple directly created in a register. A type of the form $\forall[\vec{\omega}].(\Gamma\ \mathsf{requires}\ \Lambda)$ describes a code block: a thread jumping into such a block must instantiate all the universal variables $\vec{\omega}$ (type variables α or singleton lock types λ), it must also hold a register file type Γ,

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

Fig. 9. Types.

as well as the locks in Λ. The singleton lock type λ is used to represent the type of a lock value in the heap.

Types $\exists\alpha.\tau$ are conventional existential types. With type $\exists\lambda.\tau$ we are able to existentially quantify over lock types, following [8]. The scope of a lock extends until the end of the code block that creates it. However, there are situations in which we need to use a lock outside its scope. For instance, in Section 3 we present a MIL implementation of Hoare-like monitors, associating a lock with each monitor. The implementation we present allocates tuples that must be protected by the monitor's lock, which is out of scope. (The scope of the lock is the code block that creates the monitor.) In fact, for sharing a tuple protected by the monitor's lock, it is sufficient to know that such a lock exists. Our implementation stores the lock in an existential lock value and uses the abstracted singleton lock type when protecting tuples, after unpacking.

As usual, the recursive type is defined by $\mu\alpha.\tau$. We take an equi-recursive view of types, not distinguishing between a type $\mu\alpha.\tau$ and its unfolding $\tau[\mu\alpha.\tau/\lambda]$. Recursive, universal, and existential type constructors introduce bindings on type variables and singleton lock variables as usual. $\mathrm{ftv}(\tau)$ denotes the set of free type variables in τ, and $\mathrm{flt}(\tau)$ the set of free singleton lock types in τ.

The type system is presented in Figures 10 to 13. Instructions are checked against a typing environment Ψ (mapping heap labels to types, type variables to kind TyVar, and singleton lock types to kind Lock), a register file type Γ holding the current types of the registers, and a set Λ of lock variables: the *permission* of the code block.

Typing rules for values are illustrated in Figure 10. Heap values are distinguished from operands (that include registers as well) by the form of the sequent. A formula $\Gamma <: \Gamma'$ allows forgetting registers in the register file type, and is particularly useful in jump instructions where we want the type of the target code block to be more general (ask for less registers) than those active in the current code [22]. Rule T-TYPE makes sure types are *well-formed*, that is do not include type variables or singleton lock types not in scope. The rules for value application and for pack values, T-VALAPP and T-PACK, work both with type variables α and singleton lock types λ, taking advantage of the fact that substitution $\tau'[\tau/\omega]$ is defined only when τ is not a singleton lock type and ω is a type variable, or when both τ and ω are singleton lock types. In either case, type τ must be well-formed.

Rule T-DONE in Figure 11 requires that locks must have been released prior to

$$\frac{\emptyset \vdash \tau_i}{\vdash \mathsf{r}_1 : \tau_1, \ldots, \mathsf{r}_{n+m} : \tau_{n+m} <: \mathsf{r}_1 : \tau_1, \ldots, \mathsf{r}_n : \tau_n} \quad \text{(S-regFile)}$$

$$\vdash \tau <: \tau \qquad \frac{\vdash \Gamma <: \Gamma'}{\vdash \forall[\vec{\omega}].(\Gamma \text{ requires } \Lambda) <: \forall[\vec{\omega}].(\Gamma' \text{ requires } \Lambda)} \quad \text{(S-reflex,S-code)}$$

$$\frac{\alpha \in \operatorname{ftv}(\tau) \Rightarrow \alpha :: \mathsf{TyVar} \in \Psi \qquad \lambda \in \operatorname{flt}(\tau) \Rightarrow \alpha :: \mathsf{Lock} \in \Psi}{\Psi \vdash \tau} \quad \text{(T-type)}$$

$$\frac{\vdash \tau' <: \tau}{\Psi, l : \tau' \vdash l : \tau} \qquad \Psi \vdash n : \mathsf{int} \qquad \Psi \vdash b : \lambda \quad \text{(T-label,T-int,T-lock)}$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash v : \tau'[\tau/\omega] \quad \omega \notin \tau, \Psi \quad \tau' \neq \langle _ \rangle}{\Psi \vdash \mathsf{pack}\ \tau, v \ \mathsf{as}\ \exists \omega.\tau' : \exists \omega.\tau'} \quad \text{(T-pack)}$$

$$\frac{\Psi \vdash \tau \quad \Psi; \Gamma \vdash v : \forall[\nu\vec{\omega}].(\Gamma' \text{ requires } \Lambda)}{\Psi; \Gamma \vdash v[\tau] : \forall[\vec{\omega}].(\Gamma'[\tau/\nu] \text{ requires } \Lambda[\tau/\nu])} \qquad \frac{\Psi \vdash v : \tau}{\Psi; \Gamma \vdash v : \tau} \qquad \Psi; \Gamma \vdash r : \Gamma(r)$$

(T-valApp,T-val,T-reg)

Fig. 10. Typing rules for values $\boxed{\Psi \vdash v : \tau}$ and for operands $\boxed{\Psi; \Gamma \vdash v : \tau}$.

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

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

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

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

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

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

Fig. 11. Typing rules for instructions (thread pool and locks) $\boxed{\Psi; \Gamma; \Lambda \vdash I}$.

terminating the thread. Rule T-fork splits permissions into sets Λ and Λ': the former is transferred to the forked thread according to the permissions required by the target code block, the latter remains with the current thread.

Rule T-newLock assigns a lock type $\langle \lambda \rangle^\lambda$ to the register. The new singleton lock type is recorded in Ψ, so that it may be used in the rest of the instructions I. Rules T-tsl requires that the value under test holds a lock, disallowing testing a lock already held by the thread. Rule T-unlock makes sure that only held locks are unlocked. Finally, rule T-critical ensures that the current thread holds the exact number of locks required by the target code block and adds the lock under test to the set of locks of the thread. A thread is guaranteed to hold the lock

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

$$\frac{\Psi \vdash \lambda \quad \Psi; \Gamma \vdash r\colon \langle\vec{\tau}\rangle \quad \Psi; \Gamma\{r\colon \langle\vec{\tau}\rangle^{\lambda}\}; \Lambda \vdash I \quad \pi \text{ is } \lambda \text{ when } a = \mathsf{guarded\ by}\ \lambda \text{ else } \mathsf{ro}}{\Psi; \Gamma; \Lambda \vdash \mathsf{share}\ r\ a; I} \quad \text{(T-SHARE)}$$

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

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

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

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

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

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

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

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

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

Fig. 12. Typing rules for instructions (memory and control flow) $\boxed{\Psi; \Gamma; \Lambda \vdash I}$.

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 only after the conditional jump.

The typing rules for memory and control flow are depicted in Figure 12. The rule for sharing a mutable tuple under lock λ makes sure that the lock is in lexical scope ($\Psi \vdash \lambda$). Operations for loading from and for storing into tuples require that the processor hold the right permissions (the locks for the tuples it reads from or writes to). Local tuples require no permission for its manipulation, however, special care is taken to disallow its duplications or aliasing, via the various $\tau \neq \langle_\rangle$ in the rules. Rule T-UNPACK unpacks either a conventional or a lock existential type. A new entry $\alpha\colon\colon \mathsf{TyVar}$ or $\omega\colon\colon \mathsf{Lock}$ is added to Ψ, according to the nature of ω. The new type variable or singleton lock type may then be used in the rest of the instructions I.

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

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

$$\frac{\forall i.\Psi \vdash l_i\colon \forall[_].(\Gamma_i \text{ requires } _) \qquad \Psi \vdash R_i\colon \Gamma_i}{\Psi \vdash \{\langle l_1, R_1\rangle, \ldots, \langle l_n, R_n\rangle\}} \qquad \text{(thread pool, } \boxed{\Psi \vdash T}\text{)}$$

$$\frac{\Psi, \vec{\omega} :: \text{kind}(\vec{\omega}); \Gamma; \Lambda \vdash I}{\Psi \vdash \forall[\vec{\omega}].(\Gamma \text{ requires } \Lambda)\colon \tau} \qquad \frac{\forall i.\Psi \vdash v_i\colon \tau_i \quad \Psi \vdash \langle\vec{\tau}\rangle^\pi}{\Psi \vdash \langle\vec{v}\rangle^\pi\colon \langle\vec{\tau}\rangle^\pi} \qquad \text{(heap value, } \boxed{\Psi \vdash h\colon \tau}\text{)}$$

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

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

Fig. 13. Typing rules for machine states.

The rules for typing machine states are illustrated in Figure 13. The rules for code blocks in the heap records in Ψ the type variables and the singleton lock types present in the abstraction, so that they may be used in the rest of the instructions I.

For an extended example of MIL in action, refer to Section 3 and to references [19,30].

Types against races

We split the results in three categories: the standard "well-typed machines do not get stuck" (which we omit altogether), the lock discipline, and races. The lock discipline is embodied in the following theorem (cf. Figure 2).

Theorem 2.1 (Lock discipline) *Let* $\Psi \vdash H$ *and* $\Psi \vdash \langle R; \Lambda; (\iota; _)\rangle$.

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

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

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

(iv) *If* ι *is* $\textsf{unlock}\ v$ *and* $H(\hat{\text{R}}(v)) = \langle_\rangle^\lambda$, *then* $\lambda \in \Lambda$.

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

For races we follow Flanagan and Abadi [8]. We start by defining the *set of permissions* of a machine state, by gathering the permissions of the running threads with those in the run pool, and with the set of unlocked locks in the heap. Remember that a permission is a set of locks, denoted by Λ.

Definition 2.2 [State permissions.]

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

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

Definition 2.3 Mutual exclusive states. halt is mutual exclusive; $S \neq \mathsf{halt}$ is *mutual exclusive* when $i \neq j$ implies $\Lambda_i \cap \Lambda_j = \emptyset$, for all $\Lambda_i, \Lambda_j \in \mathcal{L}_S$.

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

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

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

Theorem 2.4 *If S is a mutual exclusive typable state, then S does not have a race condition.*

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

Theorem 2.5 *Let $S \rightarrow S'$. Then,*

(i) *If $\vdash S$, then $\vdash S'$;*

(ii) *If S is mutual exclusive, then so is S'.*

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

Corollary 2.6 (Types against races) *If S is a mutual exclusive typable state and $S \rightarrow^* S'$, then S' does not have a race condition.*

3 An Unbounded Buffer Monitor in MIL

```
unbounded buffer: monitor
begin
  buffer : Queue {of Element}
  nonEmpty: condition
  procedure append(m: Element)
  begin
    buffer .enqueue(m)
    nonEmpty.signal
  end append
  procedure remove(c: Continuation)
  begin
    if buffer .isEmpty() then nonEmpty.wait
    fork c( buffer .dequeue())
  end remove
  buffer = createQueue()
end unbounded buffer
```

Fig. 14. An Hoare-style unbounded buffer monitor.

This section describes an Hoare-style implementation of an unbounded buffer monitor. The buffer is accessible to programs running in parallel: *producers* update the buffer by appending a new element at the end; *consumers* update the buffer by removing the first element. Figure 14 describes the monitor we are interested in.

Whenever the buffer is empty, as in the initial state of the monitor, the operation that removes the first element from the queue is undefined. We make sure that the consumer waits until the producer has made the queue nonempty. The queue is assumed to be infinitely large, thus there is always room in the queue for producers to append new elements. For this reason, and contrasting with removing from the buffer, appending does not block.

Our monitor exports three operations—create, append, and remove—as MIL code, with the following signatures,

```
createMonitor ∀[Element] (r1: CreateContinuation(Element))
       append ∀[Element] (r1: Monitor(Element), r2: Element)
       remove ∀[Element] (r1: Monitor(Element), r2: RemoveContinuation(Element))
```

where Element is a given type, the type of the elements in the monitor, and Monitor is an abstract type. By using continuation-passing style [1], operations createMonitor and remove also receive a closure containing the continuation. In either closure, the continuation code expects an environment of type α in r_1. The continuation of the create procedure, however, expects in r_2 the newly created monitor, whereas that for the remove procedure expects an element of the buffer.

```
def CreateContinuation(Element) = ∃α.⟨(r1: α, r2: Monitor(Element)), α⟩^ro
def RemoveContinuation(Element) = ∃α.⟨(r1: α, r2: Element), α⟩^ro
```

We illustrate the usage of monitors with a traditional producer/consumer example. Code block main creates a monitor for a buffer of integers and starts three threads (code block producerConsumer): a producer (code block producer) that appends integers to the buffer, and two identical consumers defined by code block consumer. The element removed from the buffer is delivered at code block consumeNext that "consumes" the element and asks for another one by jumping back to the consumer code block.

```
main() {
```

```
  -- create the base (empty) environment
  r2 := new 0
  share r2 read-only  -- ⟨⟩^ro
  -- create the closure
  r1 := new 2
  r1[1] := producerConsumer -- set the continuation
  r1[2] := r2    -- set the environment
  share r1 read-only  -- ⟨(r1:⟨⟩^ro, r2:Monitor(int)), ⟨⟩^ro⟩
  r1 := pack ⟨⟩^ro, r1 as CreateContinuation(int)
  jump createMonitor[int]
}
producerConsumer(r1:⟨⟩^ro, r2:Monitor(int)) {
  -- the environment is the monitor
  r1 := r2
  fork producer
  fork consumer
  fork consumer
  done
}

consumer(r1:Monitor(int)) {
  -- create the closure
  r2 := new 2
  r2[1] := consumeNext -- set the continuation
  r2[2] := r1          -- the environment is the monitor
  share r2 read-only   -- ⟨(r1:Monitor(int), r2:int), Monitor(int)⟩
  r2 := pack Monitor(int), r2 as RemoveContinuation(int)
  fork remove[int]
  done
}
consumeNext(r1:Monitor(int), r2:int) {
  -- process the element in r2
  jump consumer
}

producer(r1:Monitor(int)) {
  -- set the element
  r2 := 2
  -- produce the element
  fork append[int]
  jump producer
}
```

The three monitor operations described above is all we need in order to compile the π-calculus in Section 5. The rest of this section is organised in three parts. The first introduces the three monitor operations. The second presents the wait and signal operations, and the last the queues of generic elements, used to implement both the buffer and the monitor's condition.

The monitor

When implementing monitors in MIL, we associate a lock with each monitor to ensure mutual exclusion for the monitor's operations. Each operation needs to acquire the monitor's lock before executing its body and release the lock upon exit. Monitor's data is stored in a tuple containing the references for both the buffer queue and the nonEmpty condition.

```
def Monitor(Element) = ∃λ.UnpackedMonitor(λ,Element)
def UnpackedMonitor(λ,Element) = ⟨Queue(λ,Element),Condition(λ),⟨λ⟩^λ⟩^ro
def Condition(λ) = Queue(λ,WaitContinuation(λ))
```

For creating the monitor we generate a lock, a buffer queue, and a condition queue. After that we assemble a tuple with the monitor's local data and pack it together with the monitor's lock. The resulting value is a monitor of type Monitor(Element), with Element being the type of the elements to store in the monitor's buffer.

Code block createMonitor is the entry point for the creation of a monitor, expecting a continuation in register r_1 of type CreateContinuation(Element), defined above. Operation createMonitor is divided into three code blocks, since we adhere to the CPS style. The first code block creates the monitor's lock, allocates an environment for storing intermediate data, recording the operation continuation closure and the monitor's lock, and finally issues the creation of the buffer queue.

```
createMonitor ∀[Element] (r1:CreateContinuation(Element)) {
  λ, r3 := newLock -- create the lock of the monitor
  -- create the environment for createQueue
  r2 := new 3 -- ⟨CreateContinuation(Element),⟨λ⟩^λ,Queue(λ,Element)⟩
  r2[1] := r1 -- store the continuation closure
  r2[2] := r3 -- store the monitor's lock
  r3 := createMonitorCondition[λ,Element] -- set the continuation of createQueue
  jump createQueue[λ,Element,MonitorEnv(λ,Element)]
}

def MonitorEnv(λ,Element) = ⟨CreateContinuation(Element),⟨λ⟩^λ,Queue(λ,Element)⟩
```

The second code block (createMonitorCondition) stores the buffer queue (just created and passed in r_1) in the intermediate environment, jumping to the creation of the queue supporting the non empty condition.

```
createMonitorCondition ∀[λ,Element] (r1:Queue(λ,Element), r2:MonitorEnv(λ,Element)) {
  r2[3] := r1 -- store the buffer in the internal environment
  r3 := initMonitor [λ,Element] -- set the continuation of createCondition
  jump createQueue[λ,WaitContinuation(λ),MonitorEnv2(λ,Element)]
}

def MonitorEnv2(λ,Element) = ⟨CreateContinuation(Element),⟨λ⟩^λ,Queue(λ,Element)⟩
```

Code block initMonitor implements the third and final stage of the monitor construction by allocating and initialising the monitor's data: the buffer queue, the condition, and the monitor's lock. Then the thread processing this code block unpacks the closure and passes the new monitor (in register r_1) to the continuation.

```
initMonitor ∀[λ,Element] (r1:Condition(λ),r2:MonitorEnv2(λ,Element)) {
  -- allocate the monitor
  r3 := new 3
  r3[2] := r1                -- store the condition variable
  r1 := r2[3]                -- load the buffer from the internal environment
  r3[1] := r1                -- store the buffer in the monitor
  r1 := r2[2]                -- load the lock of the monitor
  r3[3] := r1                -- store the lock
  share r3 read-only         -- ⟨Queue(λ,Element),Condition(λ),⟨λ⟩^λ⟩
  r4 := r2[1]                -- load the closure
  α,r4 := unpack r4          -- unpack the closure
  r1 := r4[2]                -- load the environment
  r4 := r4[1]                -- load the continuation
  r2 := pack λ, r3 as Monitor(Element) -- abstract the monitor's lock
  jump r4                    -- execute the continuation
}
```

The monitor operation append places one element in the buffer and signals the non empty condition variable. The operation accepts elements of type Element in register r_2, and a monitor of type Monitor(Element) in register r_1. The operation starts by unpacking the monitor for accessing the monitor's lock. Code block appendAcquire then spin-locks to acquire exclusive access to the lock, jumping to code block appendEnqueue on success. This pattern is repeated for all monitor procedures.

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

The two following code blocks implement the body of the append procedure (*vide* Figure 14) by enqueuing the element in the buffer (code block appendAcquire), and by signalling the non empty condition (code block appendSignal), possibly awakening a pending consumer.

```
appendEnqueue ∀[λ,Element] (r1:UnpackedMonitor(λ,Element),r2:Element) requires (λ) {
  r3 := new 2
  r3[1] := appendSignal[λ,Element]   −− set the continuation to the signal operation
  r3[2] := r1             −− store the environment
  share r3 read-only      −− ⟨(r1:UnpackedMonitor(λ,Element)) requires (λ),UnpackedMonitor(λ,Element)⟩
  r3 := pack UnpackedMonitor(λ,Element),r3 as EnqueueContinuation(λ)
  r1 := r1[1]             −− load the buffer
  jump enqueue[λ,Element]
}
appendSignal ∀[λ,Element] (r1:UnpackedMonitor(λ,Element)) requires (λ) {
  r2 := r1[2]   −− load the condition variable
  r1 := r1[3]   −− load the monitor's lock
  jump signal[λ]
}
```

Operation remove takes an element from the buffer of a monitor (in register r_1) and transfers it to a closure of type RemoveContinuation(Element) expected in register r_2. We list the code blocks embodying method remove. Likewise operation append, the first code block unpacks the monitor's lock, and jumps to removeAcquire. The second code block performs a spin-lock to acquire exclusive access to the monitor's lock, continuing to code block testCondition on success.

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

The implementation of the procedure body (*vide* Figure 14): checks for available elements in the buffer (code block testCondition); if the buffer is empty, it suspends itself and waits for a non empty buffer (**if** $r_5 = 0$ **jump** wait[λ]); otherwise, it dequeues

the element (code block exec) and applies the continuation to the dequeued element (code block execRelease). Notice that execRelease frees the monitor's lock before entering the consumer's continuation.

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

The environment storing remove internal data is of type

$$\textbf{def}\ \mathsf{RemvEnv}(\lambda,\mathsf{Element}) = \langle \mathsf{RemoveContinuation}(\mathsf{Element}),\mathsf{UnpackedMonitor}(\lambda,\mathsf{Element})\rangle^{\mathbf{ro}}$$

The following code blocks dequeue an element from the buffer and deliver it to the consumer.

```
exec ∀[λ,Element] (r1:RemvEnv(λ,Element)) requires (λ) {
  r2 := new 2
  r2[1] := execRelease[λ,Element]
  r2[2] := r1
  share r2 read−only −− ⟨(r1:RemvEnv(λ,Element),r2:Element) requires(λ),RemvEnv(λ,Element)⟩
  r2 := pack RemvEnv(λ,Element),r2 as DequeueContinuation(λ,Element)
  r1 := r1[2]   −− load the monitor
  r1 := r1[1]   −− load the buffer
  jump dequeue[λ,Element]
}
execRelease ∀[λ,Element] (r1:RemvEnv(λ,Element), r2:Element) requires (λ) {
  r4 := r1[2]   −− load the monitor
  r4 := r4[3]   −− load the monitor's lock
  unlock r4     −− unlock the monitor's lock
  r3 := r1[1]   −− load the continuation
  α,r3 := unpack r3 −− unpack the closure
  r1 := r3[2]   −− load the environment
  r3 := r3[1]   −− load the continuation
  jump r3       −− jump to the continuation
}
```

Wait and Signal

We represent a *condition variable* in MIL as a (initially empty) queue of closures that are currently waiting on that condition. Manipulating the condition's queue is through the usual operations wait and signal. A wait operation is issued from inside a monitor and causes the calling thread to suspend itself until a signal operation occurs. Waiting on a condition (in register r_2) amounts to enqueuing the continuation of the wait operation (in register r_3) in the condition's queue. Notice that the

lock (in register r_1) protecting the queue is the same lock used to enforce mutual exclusion access to the monitor operations. Also notice that the lock is released after enqueuing the continuation, allowing other threads to use the monitor, and that the thread terminates (*vide* code block release).

```
def WaitContinuation(λ) = ∃α.⟨(r1:α) requires (λ),α⟩^ro


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

Code block signal, also issued from inside a monitor, causes exactly one of the delayed threads to resume immediately. A signal operation must be followed directly by resumption of a delayed thread, without possibility of an intervening procedure call from a third thread. There is an implicit notion of an uninterrupted transfer of ownership that goes from the signalling thread that finishes to the delayed thread that resumes execution, which fits nicely in MIL's lock discipline. The transmission of lock permission is carried out by jumping to the continuation closure without releasing the monitor's lock. We implement O-J. Dahl's variant [6] of the signal operation that should be used as the last operation of a monitor procedure.

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

```
signal ∀[λ] (r1:⟨λ⟩^λ, r2:Condition(λ)) requires (λ) {
  r3 := r2[2]                   -- load the length of the queue
  if r3 = 0 jump release[λ]     -- no closures to signal, finish
  r1 := r2                      -- set the queue of closure
  r2 := new 2
  r2[1] := signalDequeue[α]
  r2[2] := 0                    -- an empty environment
  share r2 read-only            -- ⟨(r1:int,r2:WaitContinuation(λ)) requires (λ), int⟩
  r2 := pack int,r2 as DequeueContinuation(λ,WaitContinuation(λ))
  jump dequeue[λ,WaitContinuation(λ)]
}
signalDequeue ∀[λ] (r1:int,r2:WaitContinuation(λ)) requires (λ) {
  α,r2 := unpack r2             -- unpack the suspended thread continuation and go
  r3 := r2[1]                   -- the continuation
  r1 := r2[2]                   -- the environment
  jump r3
}
```

An implementation of queues

			Processes				*Values*
P, Q	::=	$\mathbf{0}$	inactive	v	::=	x	name
	|	$\overline{x}\langle\vec{v}\rangle$	output		|	n	integer literal
	|	$x(\vec{y}).P$	input				
	|	$!x(\vec{y}).P$	rep input				*Types*
	|	$P \mid Q$	parallel	T	::=	int	integer type
	|	$(\nu\, x\colon T)\, P$	restriction		|	$[\vec{T}]$	channel type

Fig. 15. Syntax of the π-calculus.

The buffer and the condition variable are implemented as queues. A queue of type

def Queue(λ,α)=$\langle$QueueImpl(λ,α),int$\rangle^{\lambda}$

describes unbounded queues of elements of type α, protected by lock λ. Queues are implemented as tuples protected by a lock λ that hold the implementation of type QueueImpl(λ,α) and an integer representing the length of the queue. The description of type QueueImpl(λ,α) as well as the code for operations on queues can be found in [19]. There are three operations on queues—creation, enqueuing, and dequeuing—with the following signatures:

createQueue $\forall[\lambda,\alpha,\beta]$ (r_2:α, r_3:(r_1:Queue(λ,α), r_2:β))
enqueue $\forall[\lambda,\alpha]$(r_1:Queue(λ,α),r_2:α,r_3:EnqueueContinuation(λ)) **requires** (λ)
dequeue $\forall[\lambda,\alpha]$ (r_1:Queue(λ,α), r_2:DequeueContinuation(λ,α)) **requires** (λ)

For creating queues we provide operation createQueue that expects in register r_2 the environment of any type α, and in register r_3 a code block for passing the new queue (in register r_1). Operation enqueue places the element present in register r_2 at the end of the queue given in register r_1. Afterwards, the thread executing this code block continues by unpacking and processing the closure in register r_3 of type

def EnqueueContinuation(λ)=$\exists\ \alpha.\langle$(r_1:α) **requires** (λ),$\alpha\rangle^{\mathsf{ro}}$

Removing an element from the head of the queue (operation dequeue) that targets a queue in register r_1 and delivers the removed element to the continuation present in register r_2 of type

def DequeueContinuation(λ,α)=$\exists\ \beta.\langle$(r_1:β,r_2:α) **requires** (λ),$\beta\rangle^{\mathsf{ro}}$

Both enqueue and dequeue operations as well as their continuations require exclusive access to the lock of the queue.

4 Source language: the π-calculus

Our starting point is the simple typed asynchronous π-calculus [3,15,28], equipped with integer values, generated by the syntax in Figure 15.

The syntax is divided into three categories: *processes*, *values*, and *types*. Values v are either names or integer values. Names are ranged over by lower case Roman letters and are taken from a denumerable set. The vector notation is used to denote

$$\Gamma \vdash \mathrm{n}\colon \mathsf{int} \qquad \Gamma \vdash x\colon T \qquad \Gamma \vdash \mathbf{0} \qquad \text{(T-INT,T-NAME,T-NIL)}$$

$$\frac{\Gamma \vdash x(\vec{y}).P}{\Gamma \vdash !x(\vec{y}).P} \quad \frac{\Gamma \vdash x\colon [T_0 \dots T_i] \quad \Gamma, y_0\colon T_0, \dots, y_i\colon T_i \vdash P}{\Gamma \vdash x(\vec{y}).P} \qquad \text{(T-REP,T-IN)}$$

$$\frac{\Gamma \vdash x\colon [\vec{T}] \quad \Gamma \vdash v_i\colon T_i \quad \forall i \in I}{\Gamma \vdash \overline{x}\langle \vec{v} \rangle} \qquad \text{(T-OUT)}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash P'}{\Gamma \vdash P \mid P'} \qquad \frac{\Gamma, x\colon [\vec{T}] \vdash P}{\Gamma \vdash (\nu\, x\colon [\vec{T}])\, P} \qquad \text{(T-PAR,T-RES)}$$

Fig. 16. Typing rules for the π-calculus.

a possibly empty sequence of symbols; for example $\vec{x}$ stands for the sequence of names $x_1 \dots x_n$ with $n \geq 0$.

Processes P comprise the inactive process $\mathbf{0}$; the output process $\overline{x}\langle \vec{v} \rangle$ that sends a sequence of values $\vec{v}$ on channel x; the input process $x(\vec{y}).P$ that receives a value via channel x and proceeds as P, after substituting $\vec{v}$ for $\vec{y}$. The parallel composition process running concurrently $P \mid Q$; the restriction process $(\nu\, x\colon T)\, P$ that creates a new channel definition local to process P; and, finally, the replicated input process $!x(\vec{y}).P$ that represents an infinite number of active input processes $x(\vec{y}).P$ running in parallel.

For types T, we have int representing integer values, and $[\vec{T}]$ denoting a channel that can carry a sequence of values of types $\vec{T}$. The operational semantics for the π-calculus is the standard and can be easily found in, *e.g.*, [28].

Figure 16 presents a standard type system for the π-calculus. A typing Γ is a partial function of finite domain from names to types. We write $\mathrm{dom}(\Gamma)$ for the domain of Γ. When $x \notin \mathrm{dom}(\Gamma)$ we write $\Gamma, x\colon T$ for the typing Γ' such that $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = T$, and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$. Type judgements are of two forms: (a) $\Gamma \vdash v\colon T$ means that value v has type T under the assumptions in typing Γ; and (b) $\Gamma \vdash P$ asserts that process P is well typed regarding typing Γ.

The typing rules are straightforward. Rule T-INT states that primitive values are typed under the int type. The type of name is taken from the type environment (Rule T-NAME). The inactive process $\mathbf{0}$ is always well typed (Rule T-NIL). A replicated input process $!x(\vec{y}).P$ is well typed if its non-replicated from is (Rule T-REP). T-IN says that the input process $x(\vec{y}).P$ is well typed if the input channel x is a channel type and if continuation process P is also well typed in an environment extended with the types for the parameters. The output process $\overline{x}\langle \vec{v} \rangle$ is well typed if x is a channel if its arguments are correctly typed, rule T-OUT. The parallel process is well typed if each of its parts are, rule T-PAR. Finally, 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 T-RES.

5 Compiling π into MIL

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

The translation function

The translation from the π-calculus into MIL comprises the translation $\mathcal{T}[\![\cdot]\!]$ of types, $\mathcal{V}^{\vec{x}}[\![\cdot]\!]$ of values, and $\mathcal{P}[\![\cdot]\!]$ of programs (closed π-processes).

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

$$\mathcal{T}[\![\mathsf{int}]\!] \stackrel{\text{def}}{=} \mathsf{int} \qquad \mathcal{T}_{\text{seq}}[\![T_1 \ldots T_n]\!] \stackrel{\text{def}}{=} \langle \mathcal{T}[\![T_1]\!], \ldots, \mathcal{T}[\![T_n]\!] \rangle^{\mathsf{ro}}$$
$$\mathcal{T}[\![[\vec{T}]]\!] \stackrel{\text{def}}{=} \mathsf{Monitor}(\mathcal{T}_{\text{seq}}[\![\vec{T}]\!])$$

The integer type of the π-calculus is translated in the corresponding type of MIL. A π-channel is translated into an unbounded buffer monitor whose elements are read-only tuples of values: integer values or monitors for other channels.

The translation $\mathcal{V}^{\vec{x}}[\![\cdot]\!]$ of values loads into register r_3 a value from the environment $\vec{x}$ (addressed by register r_1), or moves into the same register an integer literal.

$$\mathcal{V}^{\vec{x}}[\![v]\!] \stackrel{\text{def}}{=} \begin{cases} r_3 := r_1[i] & \text{if } v = x_i \\ r_3 := v & \text{if } v \text{ is an integer literal} \end{cases}$$

The translation of a program P yields a heap, containing several code blocks, among which we find main.

$$\mathcal{P}[\![P]\!] \stackrel{\text{def}}{=} \mathsf{main}()\{\mathcal{E}^{\Gamma}(\emptyset,\emptyset); I\} \uplus H$$
$$\text{where } \langle H, I \rangle = \mathcal{P}^{\emptyset,\emptyset}[\![P]\!]$$

Block main prepares an empty environment, $\mathcal{E}^{\Gamma}(\emptyset,\emptyset)$, for the top level process, which is then translated by $\mathcal{P}^{\emptyset,\emptyset}[\![P]\!]$. In all cases register r_1 contains the current environment, the address of a read-only tuple containing the free names in the process.

Function $\mathcal{E}^{\Gamma}(\vec{x},\vec{y})$ generates an instruction sequence that creates a new environment as a copy of the current environment $\vec{x}$ (in register r_1 of type $\mathcal{T}_{\text{seq}}[\![\vec{T}]\!]$ where $\vec{T}$ are the types of $\vec{x}$) extended with environment $\vec{y}$ in register r_2 (of type $\mathcal{T}_{\text{seq}}[\![\vec{T}']\!]$),

leaving the newly created environment in register r_1.

$$\begin{aligned}\mathcal{E}^{\Gamma}(\vec{x},\vec{y}) \stackrel{\text{def}}{=} (&r_3 := \mathsf{new}\ |\vec{x}\vec{y}| \\ &\forall\ 1 \le i \le |\vec{x}| \begin{cases} r_4 := r_1[i] \\ r_3[i] := r_4 \end{cases} \\ &\forall\ 1 \le j \le |\vec{y}| \begin{cases} r_4 := r_2[j] \\ r_3[j + |\vec{x}|] := r_4 \end{cases} \\ &\mathsf{share}\ r_3\ \mathsf{read\text{-}only} \qquad -- \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x}\vec{y})]\!] \\ &r_1 := r_3)\end{aligned}$$

First, a new environment is allocated, as a local tuple, and filled with the elements from environments $\vec{x}$ and $\vec{y}$. Next, the tuple is made shared for reading, allowing multiple threads to access the environment without contention. Lastly, the address of the newly created environment is copied to register r_1, as required by the continuation code.

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

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

$$\begin{aligned}\mathcal{P}^{\vec{x},\Gamma}[\![P \mid Q]\!] &\stackrel{\text{def}}{=} \langle H \uplus H_P \uplus H_Q, I \rangle \\ &\text{where} \\ \langle H_P, I_P \rangle &= \mathcal{P}^{\vec{x},\Gamma}[\![P]\!] \\ \langle H_Q, I_Q \rangle &= \mathcal{P}^{\vec{x},\Gamma}[\![Q]\!] \\ H &= l_Q\ (r_1 : \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x})]\!])\{I_Q\} \\ I &= (\mathsf{fork}\ l_Q; I_P)\end{aligned}$$

$$\begin{aligned}\mathcal{P}^{\vec{x},\Gamma}[\![\overline{x_i}\langle\vec{v}\rangle]\!] &\stackrel{\text{def}}{=} \langle \emptyset, I \rangle \text{ where} \\ I = (&r_2 := \mathsf{new}\ |\vec{v}| \\ &\forall\ 1 \le j \le |\vec{v}| \begin{cases} \mathcal{V}^{\vec{x}}[\![v_j]\!] \\ r_2[j] := r_3 \end{cases} \\ &\mathsf{share}\ r_2\ \mathsf{read\text{-}only}\ -- \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{v})]\!] \\ &r_1 := r_1[i] \\ &\mathsf{jump}\ \mathsf{append}[\mathcal{T}_{\text{seq}}[\![\Gamma(\vec{v})]\!]])\end{aligned}$$

The translation of the inactive process is direct: the thread is terminated and an empty heap produced. The parallel process $P \mid Q$ is translated by forking the execution of Q and continuing with the execution of P in the current thread, whilst (read-only) environment $\vec{x}$ is shared by both processes. For the output process, the registers are laid out as expected by code block append in the monitor of Section 3: register r_1 contains the (address of) channel x_i (that is, the monitor), and register r_2 contains (the address of) the tuple with values $\vec{v}$ (that is, the element to append to the buffer in the monitor). The control is then transferred to code block append. By $\Gamma(v)$ we mean T where $\Gamma \vdash v : T$ (*cf.* Figure 16); in other words, T when $v : T \in \Gamma$, or int if v is an integer literal.

The translation of an input process is as follows.

$$\begin{aligned}
\mathcal{P}^{\vec{x},\Gamma}[\![x_i(\vec{y}).P]\!] &\stackrel{\text{def}}{=} \langle H \uplus H', \mathsf{jump}\ l\rangle \text{ where}\\
[\vec{T}] &= \Gamma(x_i)\\
\Gamma' &= \Gamma, \vec{y}\colon \vec{T}\\
\langle H', I'\rangle &= \mathcal{P}^{\vec{x}\vec{y},\Gamma'}[\![P]\!]\\
\mathit{contType} &= (r_1 : \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x})]\!], r_2 : \mathcal{T}_{\text{seq}}[\![\vec{T}]\!])
\end{aligned}$$

$$\begin{aligned}
H = {} & l'\ \mathit{contType}\ \{\mathcal{E}^{\Gamma'}(\vec{x},\vec{y}); I'\}\\
& l\ (r_1 : \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x})]\!])\ \{\\
& \quad r_2 := \mathsf{new}\ 2\\
& \quad r_2[1] := l'\\
& \quad r_2[2] := r_1\\
& \quad \mathsf{share}\ r_2\ \mathsf{read\text{-}only} \quad -\!\!-\ \langle \mathit{contType}, \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\\
& \quad r_2 := \mathsf{pack}\ \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x})]\!], r_2\ \mathsf{as}\ \mathsf{RemoveContinuation}(\mathcal{T}_{\text{seq}}[\![\vec{T}]\!])\\
& \quad r_1 := r_1[i]\\
& \quad \mathsf{jump}\ \mathsf{remove}[\mathcal{T}_{\text{seq}}[\![\vec{T}]\!]]\}
\end{aligned}$$

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

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

$$l'\ \mathit{contType}\ \{\mathsf{fork}\ l; \mathcal{E}^{\Gamma'}(\vec{x},\vec{y}); I'\}$$

that starts the continuation of the removed element by forking a copy of the translation of the input process at code block l.

In the translation of scope restriction, a new monitor is created for channel y

and added to the current environment.

$$
\begin{aligned}
\mathcal{P}^{\vec{x},\Gamma}[\![(\nu\, y\colon T)\, P]\!] &\stackrel{\text{def}}{=} \langle H \uplus H', I\rangle \text{ where}\\
T &= [\vec{T}]\\
\Gamma' &= \Gamma, y\colon T\\
\langle H', I'\rangle &= \mathcal{P}^{\vec{x}y,\Gamma'}[\![P]\!]\\
contType &= (r_1 : \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x})]\!], r_2 : \mathcal{T}[\![T]\!])\\
H &= l\ contType\ \{\\
&\qquad r_3 := \mathsf{new}\ 1\\
&\qquad r_3[1] := r_2\\
&\qquad \mathsf{share}\ r_3\ \mathsf{read\text{-}only} \qquad -- \langle \mathcal{T}[\![T]\!]\rangle^{\mathsf{ro}}\\
&\qquad r_2 := r_3\\
&\qquad \mathcal{E}^{\Gamma'}(\vec{x}, y)\\
&\qquad I'\}
\end{aligned}
$$

$$
\begin{aligned}
I = (&r_2 := \mathsf{new}\ 2\\
&r_2[1] := l\\
&r_2[2] := r_1\\
&\mathsf{share}\ r_2\ \mathsf{read\text{-}only} \qquad -- \langle contType, \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x})]\!]\rangle^{\mathsf{ro}}\\
&r_1 := \mathsf{pack}\ \mathcal{T}_{\text{seq}}[\![\Gamma(\vec{x})]\!], r_2\ \mathsf{as\ CreateContinuation}(\mathcal{T}_{\text{seq}}[\![\vec{T}]\!])\\
&\mathsf{jump\ createMonitor}[\mathcal{T}_{\text{seq}}[\![\vec{T}]\!]])
\end{aligned}
$$

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

Applying the translation function to process $(\nu\, x\colon \mathsf{int})\, (x(y).\mathbf{0} \mid \overline{x}\langle 2\rangle))$ yields the code shown in Figure 17.

Results

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

Theorem 5.1 *If $\emptyset \vdash P$, then $\Psi \vdash \mathcal{P}[\![P]\!]$ for some Ψ.*

The proof builds a typing derivation for the MIL program $\mathcal{P}[\![P]\!]$, using the following lemma to construct the derivations for the heap H and the instruction sequence I generated by the translation $\mathcal{P}^{\emptyset,\emptyset}[\![P]\!]$ of process P, where Ψ_0 is the environment that

```
main () {
  −− [[(new x:⟨int⟩ )(x(y).0 | x̄⟨2⟩ )]]
  −− E(0,0)
  r3 := new 0
  share r3 read-only    −− ⟨⟩ro
  r1 := r3
  −− (new x:⟨int⟩ )[[(x(y).0 | x̄⟨2⟩ )]]
  r2 := new 2
  r2[1] := l1
  r2[2] := r1
  share r2 read-only
  −− ⟨(r1:⟨⟩ro ,r2:Monitor(⟨int⟩ro )), ⟨⟩ro ⟩
  r1 := pack ⟨⟩ro ,r2 as CreateContinuation(⟨int⟩ro )
  jump createMonitor[⟨int⟩ro ]
}
l1 (r1:⟨⟩ro , r2:Monitor(⟨int⟩ro )) {
  r3 := new 1
  r3[1] := r2
  share r3 read-only    −− ⟨Monitor(⟨int⟩ro )⟩ro
  r2 := r3
  −− E(0,x)
  r3 := new 1
  r4 := r2[1]
  r3[1] := r4
  share r3 read-only    −− ⟨Monitor(⟨int⟩ro )⟩
  r1 := r3
  −− [[x(y).0]] | [[x̄⟨2⟩ ]]
  fork l4
  −− x(y).[[0]]
  jump l2
}
l2 (r1:⟨Monitor(⟨int⟩ro )⟩ro ) {
  r2 := new 2
  r2[1] := l3
  r2[2] := r1
  share r2 read-only
−− ⟨(r1:⟨Monitor(⟨int⟩ro )⟩ro ,r2:⟨int⟩ro ), ⟨Monitor(⟨int⟩ro )⟩ro ⟩
  r2 := pack ⟨Monitor(⟨int⟩ro )⟩ro ,r2 as RemoveContinuation(⟨int⟩ro )
  r1 := r1[1]
  jump remove[⟨int⟩ro ]
}
l3 (r1:⟨Monitor(⟨int⟩ro )⟩ro ,r2:⟨int⟩ro ) {
  −− E(x,y)
  r3 := new 2
  r4 := r1[1]
  r3[1] := r4
  r4 := r2[1]
  r3[2] := r4
  share r3 read-only −− ⟨Monitor(⟨int⟩ro ), int⟩
  r1 := r3
  −− 0
  done
}
−− x̄⟨2⟩
l4 (r1:⟨Monitor(⟨int⟩ro )⟩ro ) {
  r2 := new 1
  r3 := 2
  r2[1] := r3
  share r2 read-only   −− ⟨int⟩ro
  r1 := r1[1]
  jump append[⟨int⟩ro ]
}
```

Fig. 17. The translation of process $(\nu\, x\colon \mathsf{int})\,(x(y).\mathbf{0} \mid \overline{x}\langle 2\rangle))$ into a MIL program.

types the whole library, and includes entries for code blocks createMonitor, append, and remove with the corresponding types described in Section 3.

Lemma 5.2 *If* $\mathcal{P}^{\vec{x},\Gamma}[\![P]\!] = \langle H, I\rangle$ *with* $\Gamma \vdash P$ *and* $\mathrm{fn}(P) \subseteq \vec{x}$*, then* $\exists \Psi \supseteq \Psi_0$ *such that* $\Psi \vdash H$ *and* $\Psi; (r_2 \colon \mathcal{T}_{seq}[\![\Gamma(\vec{x})]\!]); \emptyset \vdash I$*.*

The proof for this lemma is by induction on the structure of the π-process P, simplified by the fact that each process constructor generates quite a concise code thanks to the library discussed in Section 3. The target code produced by $\mathcal{P}[\![P]\!]$ must be linked to the library H_0. We have not attempted to hand-check the typability of the 250-plus lines of H_0; instead we have run it through the MIL type checker [19], which has been used to type check various non-trivial programs.

Discussion

Turner uses a single queue to hold both the messages and the input processes waiting for messages, taking advantage of an invariant by which queues never con-

tain both messages and input processes simultaneously [29]. Our implementation uses two queues, one to hold the messages in the monitor's buffer, the other to implement the condition variable, which, remarkably are instances of the same abstract data type: Queue(λ,Element) and Queue(λ,WaitContinuation(λ)). We tried a Turner-like implementation where the data structure implementing π-channels contained addresses of two different queues, given that, for typing reasons, we cannot have a same queue containing messages and objects, even if at different times. The monitor-based implementation we propose uses the same amount of memory, yet less lines of code.

6 Conclusions and Further Work

The contributions of this work are twofold: a) the extension of MIL with memory local to a processor, heap allocated read-only tuples that may be shared without contention, polymorphic and lock-existential types, and the corresponding type soundness result, and b) a type-preserving compilation algorithm from the π-calculus into MIL, witnessing the flexibility of the language in a typed (hence race-free) scenario. As a by-product of the translation we showed how to implement a generic unbounded buffer monitor in MIL.

We are currently developing on a version of MIL equipped with a compare-and-swap primitive rather than locks, allowing in particular to obtain a wait-free implementation of queues, hence of the π-calculus and related languages. Future work includes the static detection of deadlocks and developing a model that adheres more closely to multi-core processors as we know them, in particular foregoing the direct allocation of arbitrary-length tuples directly on registers.

References

[1] Appel, A. W., "Compiling with Continuations," Cambridge University Press, 1991.

[2] Birrell, A., *An introduction to programming with threads*, Technical Report 35, Digital Systems Research Center, Palo Alto, California (1989).

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

[4] Boyapati, C., R. Lee and M. Rinard, *Ownership types for safe programming: preventing data races and deadlocks*, in: *Proceedings of OOPSLA '02* (2002), pp. 211–230.

[5] Boyapati, C. and M. Rinard, *A parameterized type system for race-free Java programs*, in: *Proceedings of OOPSLA '01* (2001), pp. 56–69.

[6] Dahl, O. J. and C. A. R. Hoare, "Hierarchical program structures," Academic Press, 1972 pp. 175–220.

[7] Flanagan, C. and M. Abadi, *Object types against races*, in: J. C. M. Baeten and S. Mauw, editors, *Proceedings of CONCUR '99*, LNCS **1664** (1999), pp. 288–303.

[8] Flanagan, C. and M. Abadi, *Types for safe locking*, in: S. D. Swierstra, editor, *Proceedings of ESOP '99*, LNCS **1576** (1999), pp. 91–108.

[9] Flanagan, C. and S. N. Freund, *Type-based race detection for Java*, ACM SIGPLAN Notices **35** (2000), pp. 219–232.

[10] Flanagan, C. and S. N. Freund, *Type inference against races*, in: R. Giacobazzi, editor, *Proceedings of SAS '04*, LNCS **3148** (2004), pp. 116–132.

[11] Fournet, C. and G. Gonthier, *The reflexive chemical abstract machine and the join-calculus*, in: *Proceedings of POPL '96*, ACM Press, 1996, pp. 372–385.

[12] Grossman, D., *Type-safe multithreading in Cyclone*, in: Z. Shao and P. Lee, editors, *Proceedings of TLDI '03*, SIGPLAN Notices **38(3)** (2003), pp. 13–25.

[13] Held, J., J. Bautista and S. Koehl, *From a few cores to many: A tera-scale computing research overview* (2006), white paper.

[14] Hoare, C. A. R., *Monitors: an operating system structuring concept*, Communications of the ACM **17** (1974), pp. 549–557.

[15] Honda, K. and M. Tokoro, *An Object Calculus for Asynchronous Communication*, in: P. America, editor, *Proceedings of ECOOP '91*, LNCS **512** (1991), pp. 133–147.

[16] Iwama, F. and N. Kobayashi, *A new type system for JVM lock primitives*, in: *Proceedings of ASIA-PEPM '02* (2002), pp. 71–82.

[17] Laneve, C., *A type system for JVM threads*, Journal of Theoretical Computer Science **290** (2003), pp. 741–778.

[18] Lopes, L., F. Silva and V. T. Vasconcelos, *A Virtual Machine for the TyCO Process Calculus*, in: G. Nadathur, editor, *Proceedings of PPDP '99*, LNCS **1702** (1999), pp. 244–260.

[19] *MIL website*.
URL http://gloss.di.fc.ul.pt/mil/

[20] Morrisett, G., *Typed assembly language*, in: B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, MIT Press, 2002 pp. 137–176.

[21] Morrisett, G., K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich and S. Zdancewic, *Talx86: A realistic typed assembly language*, in: *Proceedings of WCSSS '99*, 1999, pp. 25–35.

[22] Morrisett, G., D. Walker, K. Crary and N. Glew, *From System F to Typed Assembly Language*, ACM Transactions on Programing Language and Systems **21** (1999), pp. 527–568.

[23] Oyama, Y., K. Taura and A. Yonezawa, *An Efficient Compilation Framework for Languages Based on a Concurrent Process Calculus*, in: C. Lengauer, M. Griebl and S. Gorlatch, editors, *Proceedings of Euro-Par '97*, LNCS **1300** (1997), pp. 546–553.

[24] Paulino, H., P. Marques, L. Lopes, V. T. Vasconcelos and F. Silva, *A multi-threaded asynchronous language*, in: V. Malyshkin, editor, *Proceedings of PaCT '03*, LNCS **2763** (2003), pp. 316–323.

[25] Pierce, B. C., "Advanced Topics In Types And Programming Languages," MIT Press, 2002.

[26] Pierce, B. C. and D. N. Turner, *Pict: A Programming Language Based on the Pi-Calculus*, in: G. Plotkin, C. Stirling and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing (2000), pp. 455–494.

[27] Ramsey, N. and S. P. Jones, *Featherweight concurrency in a portable assembly language* (2001).

[28] Sangiorgi, D. and D. Walker, "The π-calculus: a Theory of Mobile Processes," Cambridge University Press, 2001.

[29] Turner, D. N., "The Polymorphic Pi-Calculus: Theory and Implementation," Ph.D. thesis, LFCS, University of Edinburgh (1996).

[30] Vasconcelos, V. T. and F. Martins, *A multithreaded typed assembly language*, in: G. Gopalakrishnan and J. O'Leary, editors, *Proceedings of TV '06*, 2006, pp. 133–141.