Universidade dos Açores
Departamento de Matemática

A compiler for the π-Calculus:

the backend

Tiago Cogumbreiro

Orientador
Doutor Francisco Cipriano da Cunha Martins

August 29, 2007

Acknowledgements

I want to express my gratitude to my tutor Professor Francisco Martins. I want to thank you for placing your confidence in me. For the enlightening afternoons we spent working together. For the reassuring words of wisdom that helped me keep focused. I am eager for our next endeavour together!

I am grateful to Professor Vasco Vasconcelos, for the continuous support and especially for the provided chance.

Finally, I wish to thank the Centro de Investigação em Informática e Tecnologias da Informação, of the Universidade Nova de Lisboa, for the financial support.

Abstract

Our work covers the creation of the backend of the compiler for the π-calculus. The backend consists in the translation to an abstract assembly language. The abstract assembly language used in our compiler is MIL, a multi-threaded typed assembly language.

MIL has the concept of types and locks. It uses the concurrency model of shared memory, where tuples are protected by a lock and may be accessed by multiple threads that have to acquire the exclusive right to alter the data. π-Calculus, on the other hand, uses the concurrency model of message passing, where processes communicate through channels passing information amongst each other.

We start by describing the target language's (MIL) syntax, semantics and type discipline, in Chapter 1. Afterwards we show a few usage examples, showing off the basic operations of MIL. Finally we present a runtime library that implements the π-calculus communication in the MIL language.

Compilers need to make various operations over an abstract syntactic tree. The visitor pattern is chosen to solve this problem. During our work in the compiler we envisioned an extension to the classic Visitor pattern. Chapter 2 explains the advantages of using the Extended Visitor pattern versus the usual approach.

The final chapter (Chapter 3) describes the translation process. It starts by explaining the framing step, where variables are arranged and structured according to a certain scope. Afterwards we an overview of the translation to MIL, followed by an architectural in depth analysis of the implementation.

Contents

1 MIL: Multi-threaded Typed Assembly Language **1**
- 1.1 Architecture . . . 2
- 1.2 Syntax . . . 2
- 1.3 Operational Semantics . . . 3
- 1.4 Type Discipline . . . 5
- 1.5 Examples . . . 7
- 1.6 The π-Calculus Runtime . . . 10
  - 1.6.1 Architecture . . . 10
  - 1.6.2 Implementation . . . 11
  - 1.6.3 Usage Examples . . . 12

2 Extending The Visitor Pattern **21**
- 2.1 Intent . . . 21
- 2.2 Motivation . . . 21
- 2.3 Applicability . . . 23
- 2.4 Structure . . . 23
- 2.5 Participants . . . 24
- 2.6 Collaborations . . . 24
- 2.7 Consequences . . . 25
- 2.8 Implementation . . . 25
- 2.9 Sample Code . . . 26
- 2.10 Known Uses . . . 27

3 The Backend **28**
- 3.1 Framing . . . 28
  - 3.1.1 Introduction . . . 28
  - 3.1.2 Implementation . . . 29
- 3.2 Translation . . . 30

3.2.1 Overview 31
3.2.2 Frame Indexing 34
3.2.3 Type Adapting 35
3.2.4 Code Generation Helper 36

4 Conclusion **38**

5 Appendix **39**

Chapter 1

MIL: Multi-threaded Typed Assembly Language

Chip multiprocessors (CMP) are becoming a more realistic choice to the micro processor market. This is because technological advances on single processors are becoming more complex, power hungry and expensive. However, to take full advantage of this new architecture, we need the software industry to adopt the multi-threaded programming paradigm.

MIL [8] proposes a solution to this problem [5] by combining a typed assembly language with multi-threaded programming.

A typed assembly language provides the possibility for "executing trusted code safely and efficiently" [4]. It ensures that foreign code never accesses hidden resources of the host, allowing for programs to provide safe program extensions. It also allows untrusted compilers to generate an assembly code that can be compiled with a single trusted compiler. Because types exist pointers cannot be fabricated or forged and jumps can only be done to checked code. Checks are also made on registers' content before a code block is run.

Multi-threaded programming at assembly level allows to correctly structure inter-thread synchronisation. The type system of this language enforces the absence of race conditions.

1.1 Architecture

MIL envisages an abstract CMP with a shared main memory. Each processor core owns a number of registers and an instruction cache. The main memory is divided into a heap (for storing data and code blocks) and a run pool (for storing suspended threads). Data blocks, kept in the main memory, are represented by tuples and are protected by a lock. Blocks of code define the needed registers (including the type each one needs), a list of required locks, and an instruction set. The run pool contains all the idle threads. It may happen that there are more threads to be run than the number of processors.

1.2 Syntax

The syntax of our language is described by the grammar in Figures 1.1, 1.2, and 1.7. We postpone the exposure of types to Section 1.4. We rely on a set of *heap labels* ranged over l, and a disjoint set of *type variables* ranged over by α, β.

Most of the proposed instructions, represented in Figure 1.1, are standard in assembly languages. Instructions are organised in sequences, ending with a jump or with a yield. The instruction yield frees the processor to execute another thread from the thread pool.

The *abstract machine* is defined by the number of processors available (N) and the number of registers (R), as depicted in Figure 1.2.

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 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, which is enforced by the type system to be a $\forall[\vec{\alpha}].(\Gamma \text{ requires } \Lambda)$, describes a universal operator $\forall[\vec{\alpha}]$ that abstracts types used in the signature and used in the body, a register file Γ that describes the type of each register, and the locks hold by the processor when jumping to this code block. The body is a sequence of instructions that are executed by a processor.

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.

registers	r	$::=$	$\mathsf{r}_1 \mid \ldots \mid \mathsf{r}_R$
integer values	n	$::=$	$\ldots \mid$ -1 $\mid$ 0 $\mid$ 1 $\mid \ldots$
lock values	b	$::=$	**-1** $\mid$ **0** $\mid$ **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 1.1: Instructions

1.3 Operational Semantics

Thread pools are managed by the rules illustrated in Figure 1.3. 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 pair is in the thread pool, then rule R-SCHEDULE assigns a new thread to the running processor. Rule R-FORK places a new thread in the pool, taking the ownership of locks required by the forked code block.

Operation semantics regarding locks are depicted in Figure 1.4. The newLock creates a new lock in three possible states, according to the 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 sets	λ	$::= \alpha_1, \ldots, \alpha_n$
permissions	Λ	$::= (\lambda, \lambda, \lambda)$
register files	R	$::= \{\mathsf{r}_1 \colon v_1, \ldots, \mathsf{r}_\mathsf{R} \colon v_\mathsf{R}\}$
processor	p	$::= \langle R; \Lambda; I \rangle$
processors array	P	$::= \{1 \colon p_1, \ldots, \mathsf{N} \colon p_\mathsf{N}\}$
thread pool	T	$::= \{\langle l_1, R_1 \rangle, \ldots, \langle l_n, R_n \rangle\}$
heap values	h	$::= \langle v_1 \ldots v_n \rangle^\alpha \mid \tau\{I\}$
heaps	H	$::= \{l_1 \colon h_1, \ldots, l_n \colon h_n\}$
states	S	$::= \langle H; T; P \rangle \mid \mathsf{halt}$

Figure 1.2: Abstract machine

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.

Linear locks are created by newLockLinear. They are initialised in the locked state. The new lock variable β is added to the set of linear locks.

The *Test and Set Lock*, present 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. These two operations (the load and the store) are indivisible. There are two variations of the *Test and Set Lock* in our language: tslE and tslS. When tslE is applied to an unlocked state the type variable α is added to the set of exclusive locks and the value becomes $\langle \text{-}1 \rangle^\alpha$. Various threads may read values from a tuple locked in shared state, hence when tslS is applied to a shared or to an unlocked lock the value of contained in the tuple 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 tslE is applied in a tuple with a number greater than -1, it places a 0 in the target register.

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.

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

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

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

Figure 1.3: Operational semantics (thread pool)

Rules related to memory instructions are illustrated in Figure 1.5. 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 1.6, are straightforward [6]. They rely on the function $\hat{\mathrm{R}}$ that works on registers or on values, by looking for values in registers, in packs, and in universal concretions.

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

1.4 Type Discipline

The syntax of types is exposed in Figure 1.7. A type of the form $\langle \vec{\sigma} \rangle^{\alpha}$ describes a tuple that is protected by lock α, that resides in the heap. Each type $\vec{\sigma}$ is either initialised (τ) or uninitialised ($?\tau$). A type of form

$\forall[\vec{\alpha}].(\Gamma$ requires $\Lambda)$ 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 types $\mathsf{lock}(\alpha)$, $\mathsf{lockE}(\alpha)$, and $\mathsf{lockS}(\alpha)$ describe singleton types, respectively the lock type, the lock type exclusive, and the lock type shared. The types $\exists\alpha.\tau$ and $\exists^{\mathsf{L}}\alpha.\tau$ define the existential operator in [2]. The recursive type $\mu\alpha.\tau$ allows the definition of recursive data structures.

The type system is presented in Figures 1.8 to 1.11. Typing for values is illustrated in Figure 1.8. Heap values are distinguished from operands (that include registers as well) by the form of the sequent. Notice that lock values (**-1**, **0**, and **1**) have any lock type. Also, 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 a tuple Λ that comprises three sets of lock variables (the *permission* of the code block), that are, respectively, exclusive, shared, and linear.

Rule T-YIELD requires that all shared and that all exclusive locks must have been released prior to ending the thread. Only the thread that acquired a lock may release it.

Rule T-FORK splits the permission into two tuples, Λ and Λ': one goes with the forked thread, the other 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 1.10. The rule for malloc makes sure that the lock α is in scope, meaning that it must be preceded by a newLock, in the *same* code block, or that the type variable must be abstracted in the universal value operator. Values can be loaded from tuples if the guarding type variable is in one of the set of locks. Values can be stored in tuples if the guarding type variable is in the set of exclusive locks or in the set of linear locks.

The rules for typing machine states are illustrated in Figure 1.11. They should be easy to understand. 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.

1.5 Examples

To exemplify how MIL works, we show interprocessor communication. We create a tuple of shared memory and then create two threads that try to write in it concurrently. The lock α is passed to each code block, because it is not in the scope of the forked threads.

```
main() {
  α, r1 := newLock 0
  r2 := malloc [int] guarded by α
  fork thread1[α]
  fork thread2[α]
}
```

Each thread tries to acquire lock α with a different strategy. The first thread (thread1) uses a technique called *spin lock*:

```
thread1 ∀[α](r1: ⟨lock(α)⟩^α, r2: ⟨?int⟩^α) {
  r3 := tslE r1 -- exclusive because we want to write
  if r3 = 0 jump criticalRegion1[α]
  jump thread1[α]
}
```

The code block loops actively, not releasing the processor, until it eventually grabs the lock.

```
criticalRegion1 ∀[α](r1: ⟨lock(α)⟩^α, r2: ⟨?int⟩^α)
    requires(α;;) {
  r2[0] := 1
```

```
  unlock r1
  yield
}
```

The second thread (thread2) uses a different technique called a *sleep lock*:

```
thread2 ∀[α](r1: ⟨lock(α)⟩^α, r2: ⟨?int⟩^α) {
  r3 := tslE r1 -- exclusive because we want to write
  if r3 = 0 jump criticalRegion2
  fork thread2[α]
}
```

This strategy features a cooperative approach. Instead of actively trying to grab the lock, it forks a copy of that thread and yields the processor to another thread in the pool.

```
criticalRegion2 ∀[α](r1: ⟨lock(α)⟩^α, r2: ⟨?int⟩^α)
    requires(α;;) {
  r2[0] := 2
  unlockE r1
  yield
}
```

These two techniques have advantages over each other. A spin lock is faster. It should be used when there is a reasonable expectation that the lock will be available in a short period of time. A short coming of the spin lock is demonstrated in this example:

```
main () {
  α, r1 := newLock -1
  fork release[α]
  jump spinLock[α]
}
```

The code block *main* creates a lock and forks a thread that unlocks it, thus taking the ownership of the lock when forked:

```
release ∀[α] (r1: ⟨lock(α)⟩^α) requires (α;;) {
  unlock r1
  yield
}
```

The code block spinLock uses the spin lock technique to acquire the lock's permission, as exemplified in the first example. The problem with this program is that it only works with machines with more than one processor.

Otherwise, because the spin locking thread does not relinquish the usage of the processor, the forked process that will unlock l will never be able to do so. The sleep lock technique, however, does context switching, which is an expensive operation (*i.e.*, degrades performance).

Libraries written in MIL use continuation passing style. In this model of programming the *user* passes a continuation (a label pointing to a code block) to the library's procedure. When computation is finished, the procedures runs the continuation label (either by forking or by jumping).

In continuation passing style, it is useful to pass *user data* to the continuation code. With existential types, it is possible to abstract the type of the user data. A data structure (a tuple) is created to keep the continuation label and the user data. Let $ContinuationType$ stands for

$$\forall[\alpha].((r1 : \langle ?\mathsf{int}\rangle^{\alpha}) \text{ requires } (;;\alpha))$$

Let $PackedUserData$ stands for

$$\exists X.\langle \forall[\alpha].((r1 : X) \text{ requires } (;;\alpha)), X\rangle^{\alpha}$$

A sketch of this usage is:

```
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]
}

library[α](r1: PackedUserData) {
  -- 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 $\langle ?\mathsf{int} \rangle^{\alpha}$ and places it 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 callback.

1.6 The π-Calculus Runtime

1.6.1 Architecture

The π-calculus runtime is a MIL library that is based in the virtual machine defined in [7]. The runtime implements the communication between processes. There are two procedures defined in this library: writeMessage and readMessage. A channel is defined by a data structure that keeps either callbacks to read a message or messages waiting to be consumed. The Figure 1.12 shows the abstract representation of the runtime. This runtime supports only monadic π-calculus, but since polyadic π-calculus can be encoded by monadic π-calculus, no expressivity is lost.

The writeMessage and the readMessage procedures work nearly the same. When a message is sent, by using the procedure writeMessage, we verify if there are input continuations so that the message is delivered to an input continuation. If there are no input continuations the message is enqueued. When the procedure readMessage is called, we supply it a callback (to handle the transmitted message) and verify if there are enqueued messages. If there are enqueued messages in the channel, the callback is invoked with an enqueued message as a parameter, otherwise the callback is enqueued until a message is written. So, using Java to describe the algorithm, this is a sketch of the implementation:

```
void readMessage(InputContinuation cont) {
  if (messages.size() > 0) {
    Message msg = messages.remove(0);
    cont.reduce(msg);
  } else {
    continuations.add(cont);
  }
}
```

The method *writeMessage*:

```
void writeMessage(Message msg) {
```

```
  if (continuations.size() > 0) {
    InputContinuation cont = continuations.remove(0);
    cont.reduce(msg);
    reduce(cont, msg);
  } else {
    messages.add(msg);
  }
}
```

1.6.2 Implementation

A channel needs to keep two attributes: a list of messages and a list of input continuations. MIL does not allow the use of values as indexes in load operations (only literals) and it does not permit the allocation of dynamic sizes of memory, hence we cannot have a tuple with dynamic length to implement the list of elements. Our implementation uses locks to impose the queueing of messages and of continuations. A Channel is the type:

$$\langle \mathsf{int}, Message, InputContinuation \rangle^{\alpha}$$

The first element of the tuple specifies the contents of the channel:

- 0 if it has no messages and no input continuations;
- 1 if it has at least one queued message;
- 2 if it has at least one queued input continuation;

The second element of the tuple is a queued message and the third element is a queued input continuation.

MIL has no notion of what classes are so we must adapt the code to the language. A Continuation class is simply a callback with user data attached to it (the implementation of the class). Using the continuation passing style explained in Section 1.5, the input callback is the data structure InputContinuation:

$$\begin{aligned} \exists X.\langle \forall [\alpha, Message].((&\mathsf{r}_1 : X, \\ &\mathsf{r}_2 : Channel, \\ &\mathsf{r}_3 : \langle \mathsf{lock}(\alpha) \rangle^{\alpha}) \ \mathsf{requires} \ (\alpha;;)), \\ X\rangle & \end{aligned}$$

Because messages can be of any type, we abstract them with the universal operator in both procedures. Let the signature of readMessage stands for:

$$\begin{aligned}\forall[\alpha, Message].((&\mathsf{r}_1 : InputContinuation,\\ &\mathsf{r}_2 : Channel,\\ &\mathsf{r}_3 : \langle \mathsf{lock}(\alpha)\rangle^{\alpha}) \text{ requires } (\alpha;;))\end{aligned}$$

Let the signature of writeMessage stands for

$$\begin{aligned}\forall[\alpha, Message].((&\mathsf{r}_1 : Message,\\ &\mathsf{r}_2 : Channel,\\ &\mathsf{r}_3 : \langle \mathsf{lock}(\alpha)\rangle^{\alpha}) \text{ requires } (\alpha;;))\end{aligned}$$

The initialisation of a channel is the responsibility of the client. Since no encapsulation is possible in an assembly language, the consistency of Channel is responsibility of the user of the library.

1.6.3 Usage Examples

To use any of the two procedures we first need to initialise the Channel data structure. To do so we need to create a dummy message and a dummy input continuation. Our implementation provides a dummy continuation (sink); the dummy message must be created by the user. This example shows the initialisation of a channel of type (int). Let the type variable IntContinuation stands for

$$\begin{aligned}\exists X.\langle\forall[\alpha](&\mathsf{r}_1 : X,\\ &\mathsf{r}_2 : int,\\ &\mathsf{r}_3 : \langle \mathsf{lock}(\alpha)\rangle^{\alpha}) \text{ requires } (\alpha;;),\\ &X\rangle^{\alpha}\end{aligned}$$

First we initialise a dummy InputContinuation:

```
r2 := malloc [∀[α](r1: int ,        -- the type of the user data
                   r2: int ,        -- the type of the message
                   r3: ⟨lock(α)⟩^α -- the lock of the channel
                  ) requires(α;;) , -- we hold the lock
              int] guarded by α    -- the user data (an int)
r2[0] := sink[int][int]
r2[1] := 0
```

The label sink has the type of the message abstracted and the type of the environment abstracted so it can be used to fill a dummy InputContinuation. The label sink just unlocks the lock α and yields the processor. The user data we use is an integer, because it is more convenient. Now we need to hide the user data with the existential operator:

```
r2 := pack r2, int as IntContinuation
```

Now, with the InputContinuation in register r_2 we are able to create the channel:

```
r1 := malloc [
        int ,              -- the status of the channel
        int ,              -- the type of the message
        IntContinuation -- the packed continuation
    ] guarded by α
r1[0] := 0  -- '0' marks an empty channel
r1[1] := 0  -- the dummy message
r1[2] := r2 -- the dummy input continuation
```

To send a message with the literal 10, emulating the process $\overline{x}\langle 10\rangle$, we use the channel initialised in r_1.

```
r2 := r1  -- move the channel to the second parameter
r1 := 10 -- move the message to the first parameter
jump writeMessage
```

If we wish to read a message through channel x, emulating the process $x(a).P$, we use the channel initialised in r_1. Let PType stands for the type:

$$\begin{aligned}\forall[\alpha](&r_1 : \langle\rangle^{\alpha},\\ &r_2 : int,\\ &r_3 : \langle \mathsf{lock}(\alpha)\rangle^{\alpha}) \text{ requires } (\alpha;;)\end{aligned}$$

Process P can be sketched below:

```
P PType {
  -- process P
}
```

To receive a message in the code block P we do:

```
r2 := r1 -- move channel 'x' to the second parameter
r1 := malloc [PType, ⟨⟩^α] guarded by α
r4 := malloc [] guarded by α
```

```
r1[0] := P        -- the continuation
r1[1] := r4       -- empty user data
r1 := pack r1, ⟨⟩^α as IntContinuation
jump readMessage
```

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

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

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

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

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

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

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

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

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

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

Figure 1.4: Operational semantics (locks)

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

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

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

Figure 1.5: Operational semantics (memory)

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

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

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

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

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

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

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

Figure 1.6: Operational semantics (control flow)

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

Figure 1.7: Types

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

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

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

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

(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)

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

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

Figure 1.8: Typing rules for values $\boxed{\Psi \vdash v \colon \sigma}$ and for operands $\boxed{\Psi; \Gamma \vdash v \colon \sigma}$

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

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

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

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

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

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

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

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

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

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

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

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

Figure 1.9: Typing rules for instructions (thread pool and locks) $\boxed{\Psi; \Gamma; \Lambda \vdash I}$

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

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

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

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

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

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

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

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

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

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

Figure 1.10: Typing rules for instructions (memory and control flow) $\boxed{\Psi; \Gamma; \Lambda \vdash I}$

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

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

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

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

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

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

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

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

Figure 1.11: Typing rules for machine states

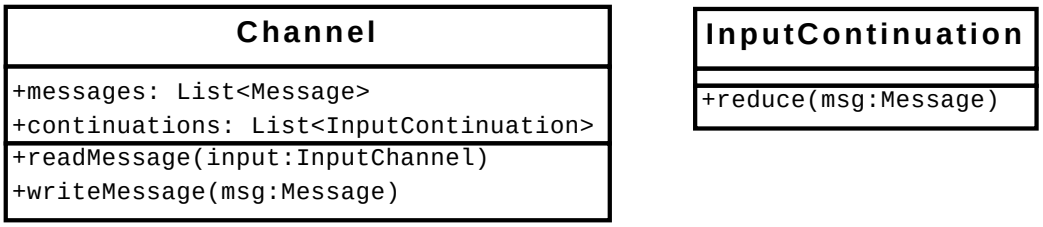

Figure 1.12: Class diagram of the π-calculus runtime

Chapter 2

Extending The Visitor Pattern

2.1 Intent

The Visitor [3] encapsulates the application of an operation over an object structure. This pattern facilitates the addition of new operations without modifying the classes on which they operate. Our extension separates three concerns: traversal, object structure, and operation.

2.2 Motivation

The object structure traversed by the visitor is usually defined by a class hierarchy, using inheritance and composition. Consider Figure 2.1, one object structure for this class hierarchy is: the tree is composed by instances of Addition, leaves are instances of Number.

This is an *ad hoc* object structure, thus it cannot be predicted, nor inferred automatically. The *convention*, is that the attributes of an object that share the same base class are the children of that object (like the Addition), but this is only true in a subset of problems. By using this convention, it is possible to create tools that generate visitors automatically. Conventions, however, have drawbacks, like the lack of flexibility and the absence of introspection. Object structures that are different from this convention cannot be target of code generation tools.

An alternative approach to specify the object structure is the implementation of a common interface that embodies the relation between objects, like the Composite pattern [3]. This approach provides introspection, gen-

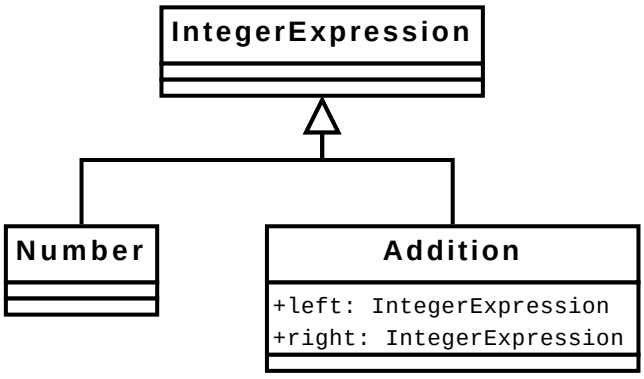

Figure 2.1: An example of a class hierarchy.

eralising the retrieval of the children of a class of objects, but imposes the implementation of an interface on every object that needs to be navigated, that may not possible.

Two common techniques of making the Visitor reusable is by using the Decorator pattern, where the decorator implements the traversal of the tree and the decorated implements the code logic, and the Template Method, where inheritance is used to separate the traversal from the code logic. Both techniques, however, are hindered by a hard-coded object structure, making the generic implementations brittle to change. The concept of traversal and the code logic are mixed at the interface level, to workaround this the separation of responsibilities is performed using the Decorator pattern or using inheritance, but the interface for both concerns is still the same (*i.e.*, the *Visitor*).

The Guided Visitor [1] proposes the separation of navigation from computation code. With this kind of visitor, the object structure is obtained in the class where the traversal algorithm is performed, in the Guide. By decoupling these two concepts it is possible to reuse traversal strategies and apply them to different object structures.

We propose breaking the visitor into three concerns, each represented by an interface, solving an isolated problem: object structure, traversal, and computation code.

2.3 Applicability

Use the extended visitor when:

- the object structure is dynamic. By having an object structure that is nor hardcoded into a class hierarchy, it is possible to make it dynamic.
- you are dealing with complex object structures. Object structures are an object, hence they can be composed or extended, as any other class.
- your requirements change. The extended visitor is adaptable. The object structure can be altered without affecting the class hierarchy, or the visitor itself. The same thing goes to the traversal strategy.

2.4 Structure

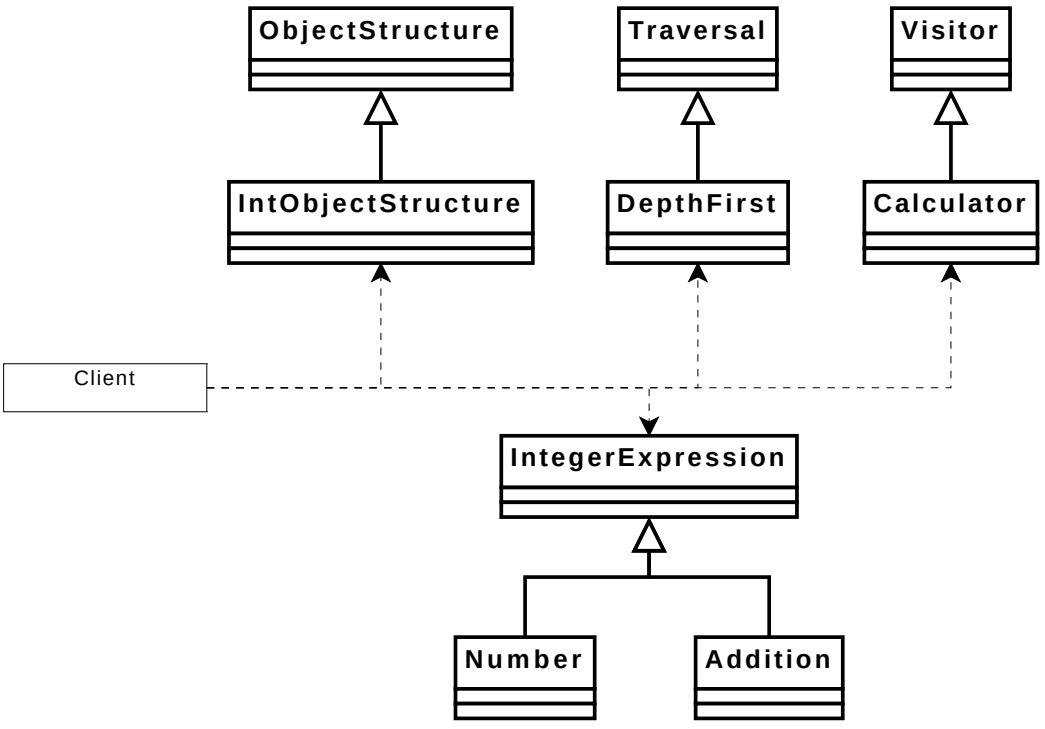

Figure 2.2: The structure of the extended visitor.

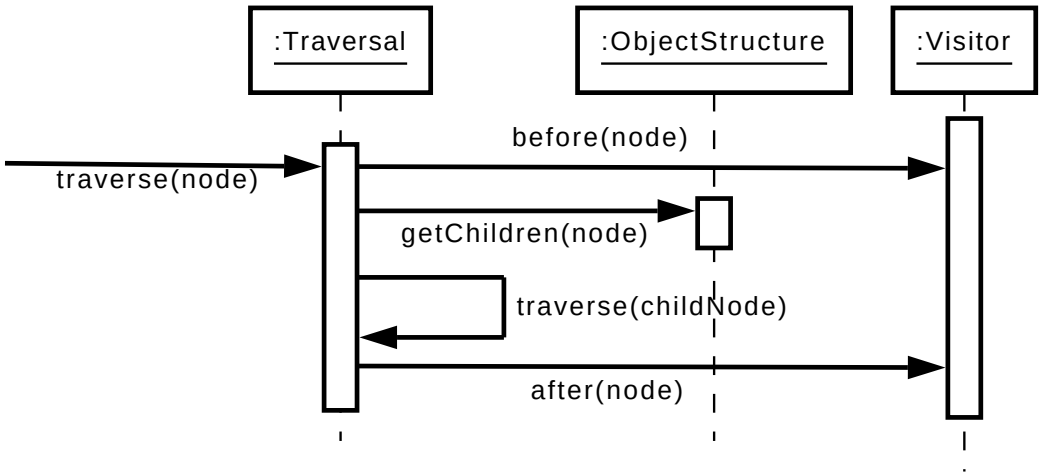

Figure 2.3: Sequence diagram of a visitor.

2.5 Participants

The *Visitor* has two methods that represent two events: before traversing the children of the node and afterwards traversal. The visitor defines an abstract interface where the operation that is applied to the object structure is implemented. The *ConcreteVisitor* (Calculator) is the actual implementation of the *Visitor* interface. A *Node* is any element that may be traversed. It can be of any type. The *ObjectStructure* is an interface for the retrieval the children of a node, if possible. A *ConcreteObjectStructure* (IntObjectStructure) is an implementation of the previous interface. The *Traversal* is the interface for the controller of the application of the operation (the visitor). A *ConcreteTraversal* (DepthFirst).

2.6 Collaborations

A client that uses the Extended Visitor pattern must create three objects, an object structure, a traversal strategy, and the visitor that applies the operation. When an element is visited two methods, corresponding to two events are called, one before its children are visited and one after its children are visited. Figure 2.3 illustrates the traversal using a depth visitor.

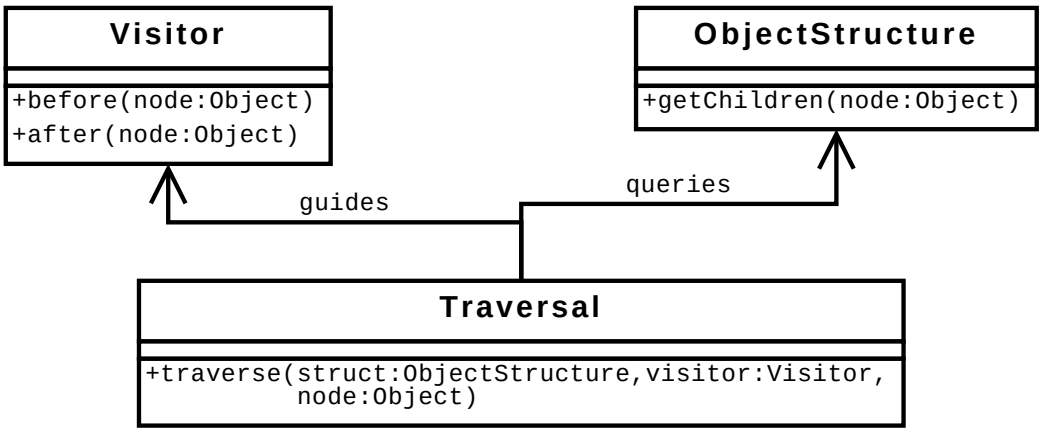

Figure 2.4: Class diagram of the implementation.

2.7 Consequences

Adding new *Node*s is easy. Contrary to the usual Visitor pattern, adding a new *Node* to the object structure is just a matter of updating the *ObjectStructure*'s implementation.

Visiting a class is more verbose. In the classic Visitor, because all the concerns are mixed within the same class, a single parameter is needed, the *Node* to visit. The Extended Visitor needs the user to explicitly create a traversal object, an object structure object, and a visitor object.

Leverages encapsulation. The classic Visitor demands a close coupling between the *Visitor* and the *Node*s, since the *Node* calls methods of the *Visitor* and vice-versa. With the Extended Visitor, *Node*s are not aware, even at interface level, of the visitor. The visitor may, or may not, be aware of the interface of the *Node* too, since it receives instances casted to Object.

Code reuse. Because of the separation of concerns and cohesion of classes, it is possible to reuse parts of the Extended Visitor in various situations, with different classes of *Node*s.

2.8 Implementation

We define three classes that embody separation of concerns. The class diagram presented in Figure 2.4. The Traversal is the controller class responsible for guiding the Visitor through the tree generated by the ObjectStructure. The

implementation of the ObjectStructure describes the children of a group of objects, that may share the same base class (or may not). The implementation of the Visitor depends on the operation that needs to be applied over a structure of objects.

2.9 Sample Code

We now show the implementation of the Extended Visitor, to implement a calculator of integer expressions, using the class hierarchy depicted in Figure 2.1. First we start by defining the IntObjectStructure:

```
Iterable getChildren(Object node) {
  if (node instanceof Addition) {
    Addition add = (Addition) node;
    return Arrays.asList(add.left, add.right);
  }
  throw new UnsupportedNodeException(node);
}
```

The only node that has children are the ones of type Addition.

Next we implement a generic depth first traversal strategy:

```
public void traverse(ObjectStructure struct, Visitor
   visitor,
                     Object node) {
  visitor.before(node);
  try {
    for (Object child: struct.getChildren(node)) {
      traverse(struct, visitor, child);
    }
  } catch (UnsupportedNodeException e) {
    // no children, skip it
  }
  visitor.after(node);
}
```

Finally the visitor Calculator, that calculates an integer expression, is implemented with this method:

```
int total = 0;
public void before(Object node) {
  if (node instanceof Number) {
```

```
    total += ((Node)obj).value;
  }
}
```

2.10 Known Uses

Compilers use the visitor pattern extensively. Having a powerful tool to create more expressive algorithms and that allows code reuse, lessens the burned of its implementation.

All previous choices of visitors dismiss a feature we believe gives our extension the edge: metadata. With metadata it is possible to empower a visitor with knowledge. For example, we may be able to know the parent/child relationship of a node, or the name of the attribute that references the node. This opens up new possibilities, like rule based visitors, where there is a rule base dispatching algorithm in the visitor for a client class, leveraging the expressive power of the client code. By decorating the traversed elements with metadata it is possible to take advantage of these benefits.

When you need different dispatching algorithms for the same traversal and object structure. With metadata, it is possible to create visitors that dispatch method calls to a specialised client class.

Chapter 3

The Backend

3.1 Framing

3.1.1 Introduction

A frame is a data structure that holds variables available in a certain scope. Frames are used to specify parameters, local variables, and global variables defined in a function. An frame of a function is allocated when the function is called and deallocated when the function exits.

The parameters and local variables are instantiated when a function is called. Local variables, parameters, and global variables should only exist whilst they are needed. If a local variable is only used for the computation of a temporary value, the memory associated with it should be freed when the computation is over, or, at least, when the function is finished. When a function (the nested) is defined inside another function (the nesting), it may reference variables defined in the nesting function. These variables, however, must not be freed when the nesting function exits, their values must be stored inside the nested function's frame.

In the π-calculus there are no functions or local variables. Input processes are analogous to functions, in what framing is concerned. Frames define the names known in a certain scope. Names used by processes must be present in a certain scope, defined as a global variable, or defined as a parameter of an input channel. Frames specify the free names available to a group of processes.

The rules for defining a variable in a frame is the following:

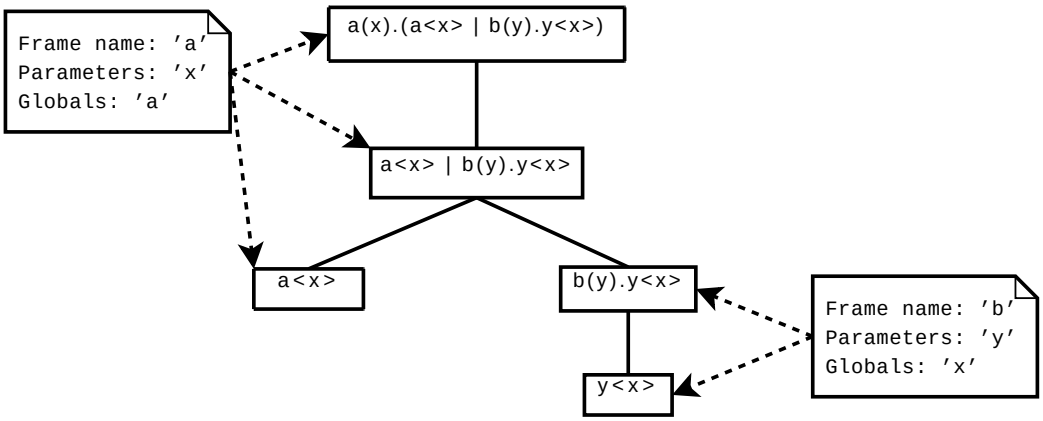

Figure 3.1: How frames annotate an AST

(F1) the names used in the arguments of an output are variables;

(F2) the name of the input channel used in a replication is a variable;

(F3) the global variables defined in a frame are variables in the parent frame;

A variable is a global variable of a frame if it is not a parameter (of that frame).

Figure 3.1 illustrates an example of framing is applied to a simple AST. We explain the process from the leaves to the root process.

The output process $\overline{y}\langle x\rangle$ uses a name as a parameter, with rule F1 it becomes a variable in the enclosing frame. Moving to the parent process $b(y).\overline{y}\langle x\rangle$, because it is an input process, a new frame is defined. The variable x is global in that frame, because it is not a parameter of the input process. The output process $\overline{a}\langle x\rangle$, similarly, uses a name (x) as a paremeter, then x becomes a variable in the related frame. Both processes, $\overline{a}\langle x\rangle$ and $b(y).\overline{y}\langle x\rangle$, declared the variable x as global, hence x is a variable in the frame defined by the input process $a(x).\overline{a}\langle x\rangle \mid b(y).\overline{y}\langle x\rangle$. Since the input process has a parameter named x, the variable x is not global.

3.1.2 Implementation

Our compiler creates a representation of frames in the same step it performs semantic checking. A new frame is mapped to each input process. Frames

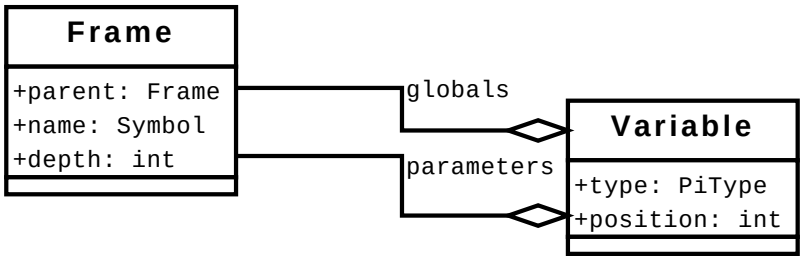

Figure 3.2: The class diagram of Frame

are named after the name of the channel. Figure 3.2 illustrates the class diagram of frames.

Stacks are used in visitors to allow the communication in nodes of different depth. The visitor navigating the AST holds a stack for defining the current frame. Each time it visits an input process a new frame is created and the names of the arguments of the channel are defined as parameters of that frame. This frame is pushed into a stack of frames. The top of the stack is the current frame enclosing a group of processes. When the visitor leaves the input process the current frame is removed from the top of the stack of frames.

When the visitor enters an output process it adds all the instances of type *NameValue*, present in the arguments, as variables to the current frame.

There is a stack in the visitor to hold Replication objects, allowing the visitor to know wether an InputPrefix instance is contained in a Replication instance. When a visitor enters an InputPrefix that is child of a Replication instance it adds a variable (with the name of the channel) to the current frame.

3.2 Translation

After semantic analysis we translate the AST to an abstract assembly language. The translation can be performed in the same traversal where semantic analysis is done, but this usually makes the code more complex. Translating directly to a concrete assembly language reduces the portability of the compiler, since it bounds one compiler to one platform.

Abstract assembly languages generalise concepts that exist in various

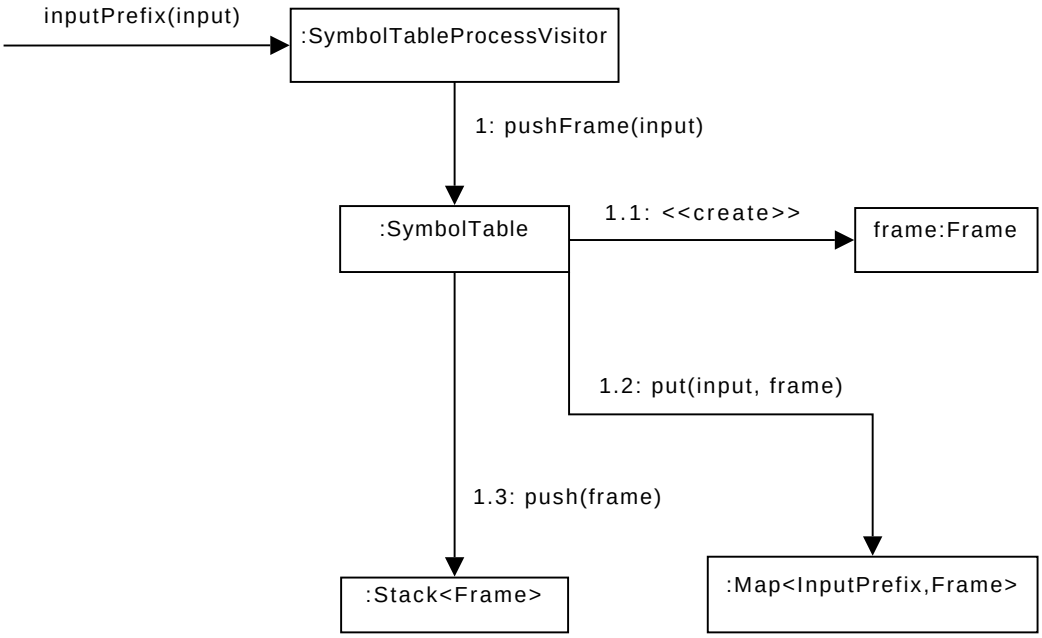

Figure 3.3: Sequence diagram of how frames are created.

platforms. Various compilers may target the same abstract assembly language, that will enable code generation for various architectures.

We show an overview of the implementation of the translation step, followed by an in-depth analysis of each component that is part of the backend.

3.2.1 Overview

Our compiler targets MIL, an abstract typed assembly language that has the concept of threads. In our compiler, code generation is performed after semantic checking. The code generation traverses the AST twice: first to associate a MIL *Label* to each process and then to translate the nodes. The generated code uses the runtime library defined in Chapter 1.6.

By using composite visitors [9], through the decorator pattern [3], we apply various operations separately, *i.e.*, each in its own class, in the same traversal. For example, the class FramePusherVisitor makes the decorated visitor access a *current frame* while traversing the AST. This visitor is used both in the process labeling traversal and in the code generation traversal. The ScopeVisitor makes the symbol table be coherent to the decorated visitor. In

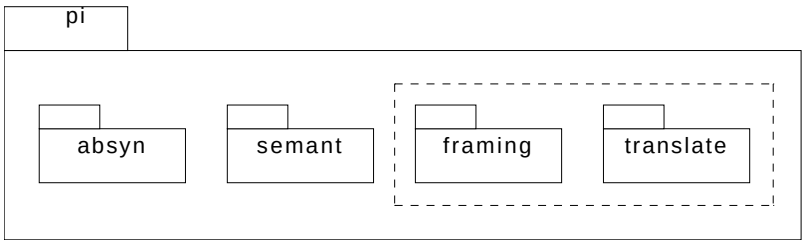

Figure 3.4: The package diagram of the compiler

this case, the class is used only once, but since the code for handling the symbol table is separated from the code generation, it is motivation enough to create a new visitor.

The basic idea of the translation is: sequential operations are sequentially performed in a single thread and processes being run in parallel are each run in separate threads.

So, considering that each process has a label attached to it, the process $P \mid Q$ roughly translates to:

```
fork P[α]
fork Q[α]
```

The code generation for the nil process, **0**, is pretty straight forward:

```
yield
```

The process $\overline{x}\langle 10\rangle$ has this translation sketch:

```
r1 := 10
r2 := x
jump writeMessage[α]
```

The input process $x(a).P$ is a bit trickier, since we need to pack the user data (which we will address further), but we can sketch it as:

```
r1 := malloc[ContinuationType, UserDataType] guarded by α
r1[0] := continuation
r1[1] := userdata
r1 := pack UserDataType, r1 as PackedContinuation
r2 := x
jump readMessage[α]
```

where the continuation is something like:

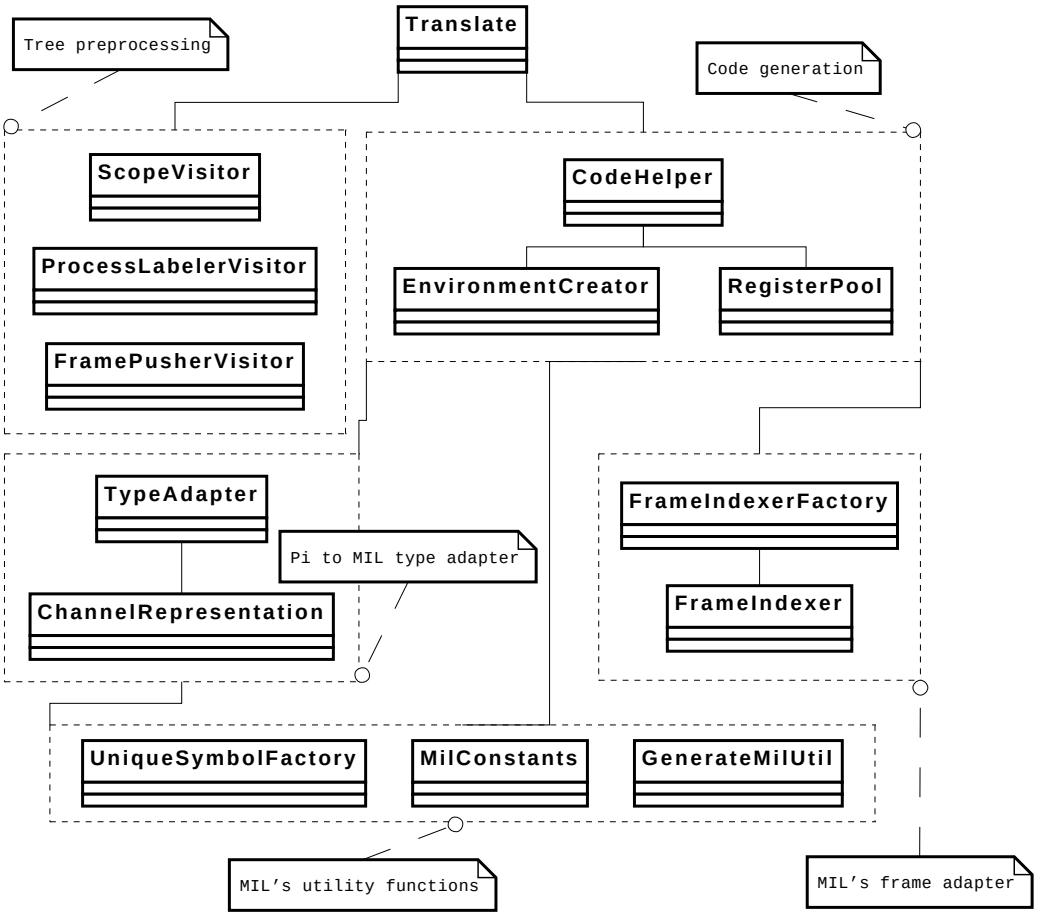

Figure 3.5: The class diagram of the package *translate*.

```
a := r2 -- receive the value of the parameter
jump P[α]
```

Finally, because we only allow replicated input processes, the translation of the replication is similar to the input, but with a different continuation. Consider the continuation of the process $!x(a).P$:

```
a := r1
fork P
jump grabLock[α]
```

Where the generated output of the code fragment grabLock is something like:

```
r5 := tslE r1
if r5 = 0 jump tryAgain
fork grabLock[α] -- sleep spin lock
yield
```

The last block (grabBlock) tries to read the message again:

```
r1 := malloc[ContinuationType, UserDataType] guarded by α
r1[0] := continuation
r1[1] := userdata
r1 := pack UserDataType, r1 as PackedContinuation
r2 := x
jump readMessage[α]
```

3.2.2 Frame Indexing

The frame indexing is where we map an abstract frame to a concrete data structure, in MIL. Concrete frames are tuples that arrange the global variables and then the parameters sequentially, as illustrated by Figure 3.6. We also implement operations to retrieve the concrete location of the data structure for a certain variable.

A frame indexer is implemented in the class FrameIndexer, depicted in Figure 3.7. We are able to obtain the π-calculus type for a certain index. We can also retrieve the name of the variable in a certain index. Finally, it is possible find the index associated with a certain variable.

The FrameIndexerFactory is a Flyweight factory [3]. We use this pattern to optimize the creation speed and the memory usage of instances of FrameIndexer .

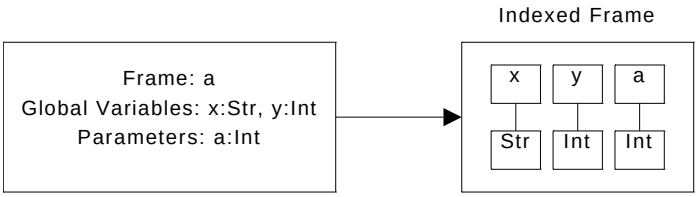

Figure 3.6: Frame indexing example.

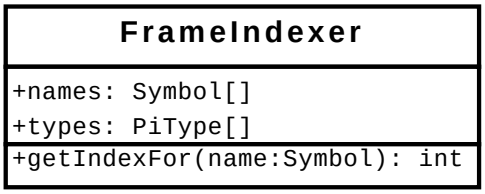

Figure 3.7: The class diagram of the FrameIndexer

3.2.3 Type Adapting

Throughout all code generation, adapting from π-calculus types to MIL types is a recurring task. The class TypeAdapter implements this operation. The normal adapting is straight forward in case of primitive types, an integer type in π-calculus maps to an integer type in MIL; the analogous is true to the string type. Link types are mapped to the Channel type specified in the π-calculus runtime, see Chapter 1.6.

To generate the appropriate Channel for a certain link type we use the class ChannelRepresentation. The TypeAdapter is also a Flyweight factory of instances of ChannelRepresentation, since these are used very frequently in code generation.

The class ChannelRepresentation generates a MIL Channel according to the argument (the Message) of the link type, remember we are dealing with monadic π-calculus. This is a recursive operation, since the type of the parameter must also be adapted from π-calculus to MIL. Each channel representation creates a TypeFragment, associating a type variable to the adapted type. This way, code generation is more comprehensible to a human and less code is generated, since a type variable is considerably smaller than the Channel type.

3.2.4 Code Generation Helper

To aid the translation we use the class CodeHelper, a Façade [3] to type adapting, to frame indexing, to register pooling, and to environment creation. Register pooling allows the reuse of registers. Environment creation is the initialisation of a frame in MIL.

Register pooling is performed in the class RegisterPool. This class has two methods, one to allocate a register and another to free a register. When a register is allocated we return it if there is none in the pool. When a register is freed, it is placed in the pool. If we allocate a register and there is a register in the pool, that one is returned, not needing to increase the register use. It is possible to know how many registers are being used at the same time.

Environment creation is implemented in the class EnvironmentCreator. It generates a MIL type for a given frame. First, it uses the FrameIndexerFactory to index the variables. Afterwards, the EnvironmentCreator adapts each π-calculus type to a MIL type. Finally, a tuple is created and bound to a type variable. This tuple comprises the adapted types (from the indexed frame). The environment is passed between processes and contains the values of the existing variables. When communicating with the runtime, it is the user data sent to the input continuation.

The CodeHelper contains a group of methods to generate blocks of code (code fragments). There is a concept of a *current code block* where instructions are appended to. There is a method to close the current code block, where we free the registers not freed explicitly and generate the code signature. The closed code block is then added as a new code fragment to the generated tree.

There are two methods in the CodeHelper that are used to generate code for a process. These were not placed in the Translate class because unit testing was easier, and because the idea is that most code generation is handled by the CodeHelper, not the Translate, which should be used more like a mediator between the traversal and the code generation.

Code generation for the output process is straightforward. If the parameter is a literal, its valued is converted to a MIL value. If the parameter is a name, then we use the frame indexing to know where the name is located in the environment variable. The value is loaded from the environment to the register that passes the message. Afterwards the a jump to the writeMessage is done.

When the input process is run, a new frame is created, this corresponds

to the allocation and initialisation of a new frame object. The initialisation is the copy the global variables present in the new frame from the old environment. After *frame switching*, we prepare the call for the readMessage, defined in the runtime.

Chapter 4

Conclusion

In our work we describe MIL. We implement a π-calculus runtime library written in that language. Afterwards, we introduce an extension of the Visitor design pattern, outlining its uses and advantages over the classical implementation. By separating the traversal, the object structure, and the visitor, we increase the reuse and maintenance of the code. Finally, we show the backend of the compiler. Our backend translates the π-calculus to MIL.

Chapter 5

Appendix

This is an example of a π-calculus program:

```
(new a:(int))(!a(x).a<x> | a<1>)
```

This is the generated MIL code:

```
version error 0.0

registers 7

packed: exists Env. <[l](r1: Env, r2: int, r3: <lock(l)>^l) requires (l;;), Env>^l

channel: <int, int, packed>^l

mainEnv: <channel>^l

main []() requires (;;) {
  l, r3 := newLock -1 -- Create the closure lock.
  r5 := malloc [channel] guarded by l
  r6 := malloc [[l](r1: int, r2: int, r3: <lock(l)>^l) requires (l;;), int] guarde
-- allocate channel: (int)
  r6[0] := sink[int][int] -- initialize the input channel continuation with a dumm
  r6[1] := 0 -- store the dummy environment into the input channel
  r6 := pack int, r6 as packed -- pack the input channel
  r7 := malloc [int, int, packed] guarded by l -- allocate data for channel: (int)
  r7[0] := 0 -- the initial status is 0 (empty)
```

```
  r7[2] := r6 -- move the dummy input channel
  r7[1] := 0 -- move the dummy output message
  r5[0] := r7 -- initialize (int)
  r1 := r5 -- Move the environemnt to the first register
  jump main_new_a[l]
}

main_new_a [l](r1: mainEnv, r3: <lock(l)>^l) requires (l;;) {
  jump main_parallel[l]
}

main_parallel [l](r1: mainEnv, r3: <lock(l)>^l) requires (l;;) {
  unlockE r3 -- Unlock the closure lock.
  fork main_parallel_left[l] -- Fork left
  fork main_parallel_right[l] -- Fork right
  yield
}

main_parallel_left [l](r1: mainEnv, r3: <lock(l)>^l) requires (;;) {
  r5 := tslE r3 -- Try to grab the lock.
  if r5 = 0 jump main_replication[l] -- Got the lock. Run the left process.
  fork main_parallel_left[l] -- Try to grab the lock later.
  yield
}

main_parallel_right [l](r1: mainEnv, r3: <lock(l)>^l) requires (;;) {
  r5 := tslE r3 -- Try to grab the lock.
  if r5 = 0 jump main_out_a[l] -- Got the lock. Run the right process.
  fork main_parallel_right[l] -- Try to grab the lock later.
  yield
}

main_replication [l](r1: mainEnv, r3: <lock(l)>^l) requires (l;;) {
  jump a_in_a[l]
}

aEnv: <channel, int>^l
```

```
a_in_a [l](r1: mainEnv, r3: <lock(l)>^l) requires (l;;) {
  r5 := malloc [channel, int] guarded by l -- alloc space for the new env 'a'
  r7 := r1[0]
  r5[0] := r7
  r5[1] := 0 -- initialize 'x'
  r2 := r1[0] -- move the channel a as 2nd arg
  r7 := malloc [[l](r1: aEnv, r2: int, r3: <lock(l)>^l) requires (l;;), aEnv] guar
  r7[0] := read_replicate_a
  r7[1] := r5
  r1 := pack aEnv, r7 as exists Env. <[l](r1: Env, r2: int, r3: <lock(l)>^l) requi
  jump inputMessage[l][int]
}

read_replicate_a [l](r1: aEnv, r2: int, r3: <lock(l)>^l) requires (l;;) {
  r1[1] := r2
  fork a_out_a[l]
  jump try_reading_again_a[l]
}

try_reading_again_a [l](r1: aEnv, r3: <lock(l)>^l) requires (;;) {
  r7 := tslE r3
  if r7 = 0 jump read_again_a[l]
  fork try_reading_again_a[l]
  yield
}

read_again_a [l](r1: aEnv, r3: <lock(l)>^l) requires (l;;) {
  r2 := r1[0]
  r7 := malloc [[l](r1: aEnv, r2: int, r3: <lock(l)>^l) requires (l;;), aEnv] guar
  r7[0] := read_replicate_a
  r7[1] := r1
  r1 := pack aEnv, r7 as exists Env. <[l](r1: Env, r2: int, r3: <lock(l)>^l) requi
  jump inputMessage[l][int]
}

a_out_a [l](r1: aEnv, r3: <lock(l)>^l) requires (l;;) {
  r2 := r1[0] -- the channel 'a'
  r1 := r1[1] -- the output message is the variable 'x'
```

```
  jump outputMessage[l][int]
}

main_out_a [l](r1: mainEnv, r3: <lock(l)>^l) requires (l;;) {
  r2 := r1[0] -- the channel 'a'
  r1 := 1 -- the output message is an integer literal
  jump outputMessage[l][int]
}

InputContinuation: exists Env. <[l](r1: Env, r2: Message, r3: <lock(l)>^l) require

Channel: <int, Message, InputContinuation>^l

ReduceCode: [l,Message] (r2: Channel,
                  r3: <lock(l)>^l) requires (l;;)

OutputCode: [l,Message] (r1: Message,
                  r2: Channel,
                  r3: <lock(l)>^l) requires (l;;)

OutputUnlockedCode: [l,Message] (r1: Message,
                          r2: Channel,
                          r3: <lock(l)>^l) requires (;;)

InputCode: [l,Message] (r1: InputContinuation,
                  r2: Channel,
                  r3: <lock(l)>^l) requires (l;;)

InputUnlockedCode: [l,Message] (r1: InputContinuation,
                         r2: Channel,
                         r3: <lock(l)>^l) requires (;;)

sink [Env,Message,l](r1: Env, r2: Message, r3: <lock(l)>^l) requires (l;;) {
  unlockE r3
  yield
}

outputMessage OutputCode {
```

```
  r4 := r2[0] -- Grab the status of the channel.
  if r4 = 0   -- Empty. Place our message into the channel.
    jump outputFill[l][Message]
  r4 := r4 - 1 -- if r4 = 1
  if r4 = 0   -- Has a message already. Try again.
    jump outputScheduleMessage[l][Message]
  r4 := r4 - 1 -- if r4 = 2
  if r4 = 0   -- Has an input channel. Redux.
    jump outputReduce[l][Message]

  -- should never reach this code
  jump outputMessage[l][Message]
}

outputFill OutputCode {
  r2[0] := 1
  r2[1] := r1
  unlockE r3
  yield
}

outputScheduleMessage OutputCode {
  unlockE r3
  jump outputGrabLock[l][Message]
}

outputGrabLock OutputUnlockedCode {
  r4 := tslE r3
  if r4 = 0 -- we grab the lock, back to the begining
     jump outputMessage[l][Message]

  -- try again:
  fork outputGrabLock[l][Message]
  yield
}

outputReduce OutputCode {
  r2[1] := r1
```

```
  jump reduce[l][Message]
}

reduce ReduceCode {
  r2[0] := 0 -- flag it as empty
  r4 := r2[2] -- the input
  r2 := r2[1] -- the message

  x, r4 := unpack r4 -- unpack the tuple
  r1 := r4[1] -- the env
  r4 := r4[0] -- the continuation
  jump r4[l]
}

inputMessage InputCode {
  r4 := r2[0]
  if r4 = 0
    jump inputFill[l][Message]

  r4 := r4 - 1 -- if r4 = 1
  if r4 = 0
    jump inputReduce[l][Message]

  r4 := r4 - 1 -- if r4 = 2
  if r4 = 0
    jump inputSchedule[l][Message]
  -- should never reach this code
  jump inputMessage[l][Message]
}

inputFill InputCode {
  r2[0] := 2  -- has an input
  r2[2] := r1 -- put the input into the pool
  unlockE r3  -- we are done; yield
  yield
}

inputReduce InputCode {
```

```
  r2[2] := r1
  jump reduce[l][Message]
}

inputSchedule InputCode {
  unlockE r3
  jump inputGrabLock[l][Message]
}

inputGrabLock InputUnlockedCode {
  r4 := tslE r3
  if r4 = 0 jump inputMessage[l][Message]
  fork inputGrabLock[l][Message]
  yield
}
```

Bibliography

[1] Martin Bravenboer and Eelco Visser. Guiding visitors: Separating navigation from computation, November 29 2001.

[2] Cormac Flanagan and Martn Abadi. Types for safe locking, February 02 1999.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] 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.

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

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

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

[8] Vasco T. Vasconcelos and Francisco Martins. A multithreaded typed assembly language. In *Proceedings of TV06 - Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, 2006.

[9] J. M. W. Visser and Joost Visser. Visitor combination and traversal control. Technical report, July 11 2001.