Chapter 7

An abstract machine for π-calculus

If the π-calculus could be implemented efficiently, it would clearly serve as a flexible intermediate language for compilers of concurrent languages (in view of the diverse high-level constructs which have been shown to be encodable in the π-calculus). For example, the π-calculus can encode higher-order communication (the communication of processes along channels) [San93a, San93b], structured datatypes [Mil91a], mutable data, concurrent objects [Wal91], and even the λ-calculus [Mil90]. We now describe an abstract machine for the π-calculus which is simple and yet realistic. In fact, in Chapters 8 and 9 we present a compilation of π-calculus to C which is directly based on the abstract machine presented here.

We are primarily interested in an abstract machine which is suitable for implementation on a uniprocessor, where concurrent execution is simulated by interleaving the execution of processes. Distributed implementation poses further challenges, such as distributed garbage collection, which are outside the scope of this dissertation.

The reduction rules for our abstract machine are deterministic. At first sight this may seem surprising, since the π-calculus is a non-deterministic language. However, if we intend to use the π-calculus as a programming language, there is no need to simulate non-determinism, since such behaviour will naturally arise because of time-dependent interactions between the abstract machine and the operating system (for example, during input/output or interrupt handling). It is much more important for a π-calculus abstract machine to provide *fair* execution, guaranteeing that runnable processes will eventually be executed, and that processes waiting to communicate on a channel will eventually succeed (if sufficient communication partners become available).

Our first abstract machine for the π-calculus introduces the basic mechanisms for process creation, channel creation and communication. We prove that the reductions of our abstract machine correspond to valid π-calculus reductions (we would not expect the converse property to hold, since our abstract machine is deterministic, while π-calculus reduction is non-deterministic). We then make a number of refinements to both our abstract machine and our source language. In particular, we record variable bindings explicitly in environments, rather than using a substitution operation, so that the basic operations of our abstract machine are simple and efficient enough to be implemented directly.

7.1 Source language

We make two simplifications to the polyadic π-calculus before attempting to formulate an abstract machine for it. First, we restrict the replication operator $*P$ so that P can only be an input process. This restriction makes it significantly simpler to implement replication, since it becomes easy to detect when we need to create a new copy of the replicated process. We can easily illustrate the effect of this simplification on the π-calculus semantics from Section 2.2: we remove the structural congruence rule $*P \equiv P \mid *P$ and add the following new reduction rule (which uses a neater syntax for replicated input, replacing $*(c?[\tilde{x}].P)$ with $c?*[\tilde{x}].P$).

$$c?*[\tilde{x}].P \mid c![\tilde{y}].Q \rightarrow c?*[\tilde{x}].P \mid \{\tilde{y}/\tilde{x}\}P \mid Q$$

It is now clear that we only need to create a new copy of P at the instant at which the replicated input communicates with some other process on c. Compare this with the structural congruence rule $*P \equiv P \mid *P$, which gives us no hint as to when we should create new copies of P.

It is worth noting that by removing the replication rule from the structural congruence relation we significantly simplify the meta-theoretic properties of our

π-calculus semantics: it is easy to prove that the structural congruence relation is decidable since, for any process P, there are finitely many processes which are structurally congruent to P. It takes considerably more effort to prove that structural congruence is decidable if we retain the replication rule [EG95].

In practice, essentially all occurrences of replication appear in recursive process definitions (cf. Section 2.6) and encodings of data structures (cf. Sections 2.8 and 2.9). In both cases, the replicated input operator is all that is required. In theory, we can even encode full replication in terms of just replicated input:

$$*P \quad \doteq \quad (\nu\, c)(c?{*}[\,].(P \mid c![\,]) \mid c![\,]) \qquad c \notin fv(P)$$

though this encoding would not work well in a real implementation, since it would quickly fill up the heap (and run queue) with copies of P.

The second simplification we make is more surprising: we disallow the summation operator! By disallowing summation, we significantly simplify the implementation of communication. In fact, the mere presence of summation in our calculus can double the amount of storage required for a channel (see the next section for details). Experience with the Pict programming language [PT95a] suggests that essential uses of the summation operator are infrequent. Moreover, it is actually possible to implement (some versions of) the summation operator as a library module [PT95a]. By taking such an approach, we only pay the cost of summation when we use the summation library, rather than during every communication. In languages such as CML [Rep92] and Facile [GMP89] which mix functional and concurrent computation, communications are sufficiently infrequent that the additional cost of implementing summation may not be significant. In Pict, however, *all* computation is achieved via communication over channels, so the additional costs imposed by summation are unacceptable.

An additional advantage of implementing summation as a library module is that it encourages the programmer to use more specialised library modules in those contexts which do not require the full generality of the summation operator. For example, the following *Ref* process uses summation to choose between accepting messages on the *read* and *update* channels, but builds essentially the same summation at each iteration (modulo changes in the argument to *Ref*, which represents the current state).

$$\texttt{def}\ \mathit{Ref}[x] = \mathit{read}?[r].(r![x] \mid \mathit{Ref}![x]) + \mathit{update}?[n,r].(r![\,] \mid \mathit{Ref}![n])$$

We can therefore use an operator called the replicated choice operator [PT95a] to implement *Ref*. The replicated choice operator exploits the fact that *Ref* repeatedly waits for input on either *read* or *update*, and therefore manages to implement *Ref* using a small amount of work to set up the communications at each iteration. In addition, the replicated choice operator guarantees that concurrent *read* and *update* requests will be interleaved fairly (*read* and *update* requests are stored in a FIFO queue, which guarantees that all requests will be processed according to their order of arrival). It is very difficult, if not impossible, to implement a general summation operator which guarantees the same behaviour, since it cannot detect that the summation created during each iteration of the *Ref* process has anything to do with the summation created during the previous iteration. The easiest thing that it can do is to vary the order in which it checks for communications inside a summation, but this behaviour is insufficient to guarantee fair processing of requests if requests are being generated at different rates along the *read* and *update* channels.

7.2 Machine states

Channel queues form the key component of our abstract machine state. The elements suspended in a channel queue may be either readers, writers or replicated readers. We let *rs* range over queues of readers (including replicated readers), and *ws* range over queues of writers. We let $\bullet$ denote the empty queue, to clarify those positions where a queue is empty.

Definition 7.1 (Channel queues)

$$\begin{array}{rcll} C & ::= & S_1 :: \ldots :: S_n & \text{Channel queue} \\ \\ S & ::= & ?[\tilde{x}].P & \text{Reader} \\ & & ![\tilde{x}].P & \text{Writer} \\ & & ?{*}[\tilde{x}].P & \text{Replicated reader} \end{array}$$

It is never the case that a channel needs to contain both blocked readers and writers (since reader/writer pairs never delay communicating). Note that this is not the case in a calculus which allows mixed inputs and outputs inside the summation operator: in the expression $c![\tilde{a}].P + c?[\tilde{x}].Q$, the process $c![\tilde{a}].P$ is not allowed to communicate with $c?[\tilde{x}].Q$ and we therefore have to allow both $c![\tilde{a}].P$ and $c?[\tilde{x}].Q$ to block on the channel c.

A machine state is a pair of a heap and a run queue. The heap stores those channels which have been created so far, and any processes which are waiting to communicate on those channels. The run queue stores those processes which are currently runnable.

Definition 7.2 (Machine state components)

$$\begin{array}{rcll} H & ::= & x_1 \mapsto C_1, \ldots, x_n \mapsto C_n & \text{Heap} \\ R & ::= & P_1 :: \ldots :: P_n & \text{Run queue} \end{array}$$

The order in which bindings appear in the heap is irrelevant but the order in which processes appear in the run queue is important, since our abstract machine always executes the process at the head of the run queue. We therefore place newly created processes on the end of the run queue, to ensure that all runnable processes will eventually be executed. Similarly, the ordering of items in a channel queue is important, since we always wake up the process at the head of a channel queue whenever a communication becomes possible.

The expression $H\{x \mapsto C\}$ denotes the heap H, where the entry for x is updated to be C (if x does not already have an entry in H then $H\{x \mapsto C\}$ denotes the heap H extended with the binding $x \mapsto C$):

Definition 7.3 (Heap update)

$$\begin{array}{rcll} \bullet\{x \mapsto C\} & \doteq & x \mapsto C & \\ (H, x \mapsto C')\{x \mapsto C\} & \doteq & H, x \mapsto C & \\ (H, x' \mapsto C')\{x \mapsto C\} & \doteq & H\{x \mapsto C\}, x' \mapsto C' & x \neq x' \end{array}$$

7.3 Reduction rules

Our abstract machine is formulated as a set of reduction rules of the form $H, R \rightarrow H', R'$. Each rule takes the process at the head of the run queue R and executes one reduction step in that process. If the run-queue R is empty, then $H, R \not\rightarrow$ and execution has finished.

The nil process has no behaviour, so we simply remove it from the run queue, enabling the next process in the run queue to start executing.

$$\frac{}{H,\ \mathbf{0} :: R \rightarrow H,\ R}\ \text{Nil}$$

We interpret the parallel composition $P \mid Q$ in an asymmetric manner, placing Q at the end of the run queue (to be executed later) and continuing with the execution of P.

$$\frac{}{H,\ (P \mid Q) :: R \rightarrow H,\ P :: R :: Q}\ \text{Prl}$$

The restriction operator $(\nu\, x)P$ allocates a new channel c in the heap, substitutes c for the bound variable x, and continues executing P. The new channel is initially empty.

$$\frac{c \text{ fresh}}{H,\ (\nu\, x)P :: R \rightarrow H\{c \mapsto \bullet\},\ \{c/x\}P :: R}\ \text{Res}$$

If the channel c already has some blocked writers in its queue when we execute the input $c?[\tilde{x}].P$, we remove the first writer from the queue, substitute the supplied values $\tilde{a}$ for the bound variables $\tilde{x}$, place the unblocked process Q on the end of the run queue, and continue executing P.

$$\frac{H(c) = ![\tilde{a}].Q :: ws}{H,\ c?[\tilde{x}].P :: R \rightarrow H\{c \mapsto ws\},\ \{\tilde{a}/\tilde{x}\}P :: R :: Q}\ \text{Inp-W}$$

If the channel c already has some blocked readers in its queue when we execute the input $c?[\tilde{x}].P$ (rs ranges of queues of readers), we suspend the current process and put it at the end of the channel queue. Note that this rule also covers the case where the queue associated with c is empty.

$$\frac{H(c) = rs}{H,\ c?[\tilde{x}].P :: R \rightarrow H\{c \mapsto rs :: ?[\tilde{x}].P\},\ R}\ \text{Inp-R}$$

If the channel c already has some blocked readers in its queue when we execute the output $c![\tilde{a}].P$, we unblock the first reader in the queue, substituting $\tilde{a}$ for $\tilde{x}$ in the unblocked process. We place the unblocked process at the end of the run queue and continue executing P.

$$\frac{H(c) = ?[\tilde{x}].Q :: rs}{H,\ c![\tilde{a}].P :: R \rightarrow H\{c \mapsto rs\},\ P :: R :: \{\tilde{a}/\tilde{x}\}Q}\ \text{Out-R}$$

If the channel c already has some blocked writers in its queue when we execute the output $c![\tilde{a}].P$ (ws ranges over queues of writers), we suspend the current process and put it at the end of the channel queue. Note that this rule also covers the case where the queue associated with c is empty.

$$\frac{H(c) = ws}{H,\ c![\tilde{a}].P :: R \rightarrow H\{c \mapsto ws :: ![\tilde{a}].P\},\ R}\ \text{Out-W}$$

If the channel c contains only readers when we execute the replicated input $c?{*}[\tilde{x}].P$, we place the replicated input at the end of the channel queue.

$$\frac{H(c) = rs}{H,\ c?{*}[\tilde{x}].P :: R \rightarrow H\{c \mapsto rs :: ?{*}[\tilde{x}].P\},\ R}\ \text{Repl-R}$$

If the channel c already has some blocked writers in its queue when we execute the replicated input $c?{*}[\tilde{x}].P$, we fork a new copy of P (substituting the output values $\tilde{a}$ for the bound variables $\tilde{x}$) and unblock the writer, placing it at the end of the run queue. We do not remove the replicated input from the run queue, so that this rule has the effect of removing all writers from c, after which the previous rule will apply and the replicated input will be removed from the run queue.

$$\frac{H(c) = ![\tilde{a}].Q :: ws}{H,\ c?{*}[\tilde{x}].P :: R \rightarrow H\{c \mapsto ws\},\ c?{*}[\tilde{x}].P :: R :: \{\tilde{a}/\tilde{x}\}P :: Q}\ \text{Repl-W}$$

If the channel c contains a replicated input when we execute the output $c![\tilde{a}].P$, we place a new copy of the replicated process at the end of the run queue, and substitute $\tilde{a}$ for $\tilde{x}$ in the new process. We do not consume the replicated input, but we do put it back on the end of the channel queue, so that any other readers on c are able to proceed.

$$\frac{H(c) = ?{*}[\tilde{x}].Q :: rs}{H,\ c![\tilde{a}].P :: R \rightarrow H\{c \mapsto rs :: ?{*}[\tilde{x}].Q\},\ P :: R :: \{\tilde{a}/\tilde{x}\}Q}\ \text{Out-R*}$$

Note that in the case where c contains a single replicated input, the Out-R* rule has no effect on c's channel queue, since $?{*}[\tilde{x}].Q :: rs = rs :: ?{*}[\tilde{x}].Q$ when rs is the empty queue.

7.4 Example reductions

The following example illustrates how the process $(\nu\, x)(x![\,].P \mid x?[\,].Q)$ performs an interaction along the channel x. We first create a fresh channel c, and substitute it for the bound variable x (we assume, for simplicity, that $x \notin fv(P, Q)$):

$$
\begin{array}{ll}
 & H,\ (\nu\, x)(x![\,].P \mid x?[\,].Q) \\
\rightarrow & H\{c \mapsto \bullet\},\ c![\,].P \mid c?[\,].Q
\end{array}
$$

Then we fork the process $c?[\,].Q$, leaving $c![\,].P$ at the head of the run queue:

$$\rightarrow \quad H\{c \mapsto \bullet\},\ c![\,].P :: c?[\,].Q$$

We can then execute the output $c![\,].P$, which has the effect of suspending $c![\,].P$ on c's queue:

$$\rightarrow \quad H\{c \mapsto ![\,].P\},\ c?[\,].Q$$

The next process on the run queue is $c?[\,].Q$, which unblocks the process P and continues executing Q:

$$\rightarrow \quad H\{c \mapsto \bullet\},\ Q :: P$$

Now both P and Q can proceed, since they are both on the run queue. Note that the channel c has reverted back to its empty state.

The following two examples illustrate the behaviour of our replication rules. The first example shows the reduction of the process $(\nu\, x)(x![\,].P \mid x?{*}[\,].Q)$, where we again assume that $x \notin fv(P, Q)$ for simplicity. The process $c![\,].P$ executes before the process $c?{*}[\,].Q$ and therefore blocks on the channel c until $c?{*}[\,].Q$ is executed (we then use the Repl-W rule to remove the blocked writer and the Repl-R rule to install the replicated input in the channel c).

$$
\begin{array}{lll}
 & H,\ (\nu\, x)(x![\,].P \mid x?{*}[\,].Q) & \\
\rightarrow & H\{c \mapsto \bullet\},\ c![\,].P \mid c?{*}[\,].Q & \text{Res} \\
\rightarrow & H\{c \mapsto \bullet\},\ c![\,].P :: c?{*}[\,].Q & \text{Prl} \\
\rightarrow & H\{c \mapsto ![\,].P\},\ c?{*}[\,].Q & \text{Out-W} \\
\rightarrow & H\{c \mapsto \bullet\},\ c?{*}[\,].Q :: P :: Q & \text{Repl-W} \\
\rightarrow & H\{c \mapsto ?{*}[\,].Q\},\ P :: Q & \text{Repl-R}
\end{array}
$$

In the following example, the process $c?{*}[\,].Q$ installs itself in the channel c before $c![\,].P$ gets executed. Thus, when we execute $c![\,].P$ we can use the Out-R* rule to fork a copy of Q immediately. There is no need to suspend the process P as in the previous example. Note that the status of c does not change after executing the output $c![\,].P$ since there are no other readers blocked on c.

$$
\begin{array}{lll}
 & H,\ (\nu\, x)(x?{*}[\,].Q \mid x![\,].P) & \\
\rightarrow & H\{c \mapsto \bullet\},\ c![\,].P \mid c?{*}[\,].Q & \text{Res} \\
\rightarrow & H\{c \mapsto \bullet\},\ c?{*}[\,].Q :: c![\,].P & \text{Prl} \\
\rightarrow & H\{c \mapsto ?{*}[\,].Q\},\ c![\,].P & \text{Repl-R} \\
\rightarrow & H\{c \mapsto ?{*}[\,].Q\},\ P :: Q & \text{Out-R*}
\end{array}
$$

7.5 Correctness of the abstract machine

We now prove that our abstract machine produces valid π-calculus reductions. First, we need to relate channel queues to ordinary π-terms. The expression $[\![C]\!]c$ denotes the π-calculus equivalent of the channel queue C located at c:

Definition 7.4 (Encoding channel queues as π-terms)

$$
\begin{array}{rcl}
[\![\bullet]\!]c & \doteq & \mathbf{0} \\
[\![![\tilde{a}].P :: C]\!]c & \doteq & c![\tilde{a}].P \mid [\![C]\!]c \\
[\![?[\tilde{x}].P :: C]\!]c & \doteq & c?[\tilde{x}].P \mid [\![C]\!]c \\
[\![?{*}[\tilde{x}].P :: C]\!]c & \doteq & c?{*}[\tilde{x}].P \mid [\![C]\!]c
\end{array}
$$

The run queue $P_1 :: \ldots :: P_n$ is equivalent to the parallel process $P_1 \mid \ldots \mid P_n$, so we can now easily relate abstract machine states to π-calculus processes:

Definition 7.5 (Encoding machine states as π-terms)

$$
\begin{array}{rl}
 & [\![c_1 \mapsto C_1, \ldots, c_n \mapsto C_n,\ P_1 :: \ldots :: P_n]\!] \\
\doteq & (\nu\, c_1) \ldots (\nu\, c_n)([\![C_1]\!]c_1 \mid \ldots \mid [\![C_n]\!]c_n \mid P_1 \mid \ldots \mid P_n)
\end{array}
$$

The order of bindings in the heap, or processes in the run queue, is irrelevant when we consider their encodings as π-terms, since the encoding of any permutation of $P_1 :: \ldots :: P_n$ is structurally congruent to the encoding of $P_1 :: \ldots :: P_n$ (we need only use the associativity and commutativity and parallel composition).

It is now easy to prove that our abstract machine produces valid π-calculus reductions. In fact, a single reduction in our abstract machine may correspond to either zero or one reductions in the π-calculus. For example, the Prl reduction rule, which executes a parallel process $P \mid Q$ by placing Q at the end of the run queue (leaving P at the head of the run queue), yields a new machine state which is structurally congruent to the original machine state. This is not surprising since the π-calculus reduction rules do not maintain a separate run queue, and therefore need no reduction steps to move a process onto the run queue. Similarly, the Res rule has no direct equivalent in the π-calculus reduction rules: it in fact corresponds to a combination of α-conversion and scope extrusion (one of the more tricky features of the π-calculus reduction rules). It is perhaps worth mentioning that α-conversion has real computation meaning in the π-calculus reduction rules, since it allows creation of fresh channels. This is quite unlike the λ-calculus, where α-conversion can be completely avoided during execution (of a closed program).

Theorem 7.6 (The abstract machine produces a valid execution)
If $H, R \rightarrow H', R'$ then $[\![H, R]\!] \rightarrow^{*}\equiv [\![H', R']\!]$.

Proof We proceed by case analysis on the abstract machine rules. We have omitted most of the output and replication cases, since they are similar to the input cases:

case Nil: $H,\ \mathbf{0} :: R \rightarrow H,\ R$

It is easy to see that $[\![H,\ \mathbf{0} :: R]\!] \equiv [\![H,\ R]\!]$, since $\mathbf{0} \mid P \equiv P$.

case Prl: $H,\ (P \mid Q) :: R \rightarrow H,\ P :: R :: Q$

Here, $[\![H,\ (P \mid Q) :: R]\!] \equiv [\![H,\ P :: R :: Q]\!]$ using the associativity and commutativity of parallel composition.

case Res: $H,\ (\nu\, x)P :: R \rightarrow H\{c \mapsto \bullet\},\ \{c/x\}P :: R$ where c is fresh

We can prove that $[\![H,\ (\nu\, x)P :: R]\!] \equiv [\![H,\ (\nu\, c)\{c/x\}P :: R]\!]$ using α-conversion (there will be no name clash problems since c is fresh). Now, using scope extrusion and the fact that $[\![\bullet]\!]c \doteq \mathbf{0}$ we have that $[\![H,\ (\nu\, c)\{c/x\}P :: R]\!] \equiv [\![H\{c \mapsto \bullet\},\ \{c/x\}P :: R]\!]$ as required.

case Inp-W: $H,\ c?[\tilde{x}].P :: R \rightarrow H\{c \mapsto ws\},\ \{\tilde{a}/\tilde{x}\}P :: R :: Q$ where $H(c) = ![\tilde{a}].Q :: ws$

In this case we have that $[\![H(c)]\!]c \doteq c![\tilde{a}].Q \mid [\![ws]\!]c$. We can therefore use the reduction rule for communication to prove that $c![\tilde{a}].Q \mid c?[\tilde{x}].P \to Q \mid \{\tilde{a}/\tilde{x}\}P$ and the result follows.

case Inp-R: $H,\ c?[\tilde{x}].P :: R \to H\{c \mapsto rs :: ?[\tilde{x}].P\},\ R$ where $H(c) = rs$

Here, $[\![H,\ c?[\tilde{x}].P :: R]\!] \equiv [\![H\{c \mapsto rs :: ?[\tilde{x}].P\},\ R]\!]$ since $[\![rs :: ?[\tilde{x}].P]\!]c \equiv [\![rs]\!]c \mid c?[\tilde{x}].P$.

case Out-R*: $H,\ c![\tilde{a}].P :: R \to H\{c \mapsto rs :: ?{*}[\tilde{x}].Q\},\ P :: R :: \{\tilde{a}/\tilde{x}\}Q$ where $H(c) = ?{*}[\tilde{x}].Q :: rs$

In this case we have that $[\![H(c)]\!]c \doteq c?{*}[\tilde{x}].Q \mid [\![rs]\!]c$. We can therefore use the reduction rule for replicated input (cf. Section 7.1) to prove that $c?{*}[\tilde{x}].Q \mid c![\tilde{a}].P \to c?{*}[\tilde{x}].Q \mid P \mid \{\tilde{a}/\tilde{x}\}Q$ and the result follows. $\square$

Note that the converse of the above result is *not* true, nor would be expect it to be, since π-calculus reduction is non-deterministic and our abstract machine is deterministic.

However, we can prove that if our abstract machine deadlocks then there are no possible π-calculus reductions from that machine state. Strictly speaking, we can only prove such a result if we make some restrictions on machine states. For example, our abstract machine will deadlock on the following machine state because the arity of the input expression $c?[y,z].Q$ is incorrect

$$\{c \mapsto ![x].P\}, c?[y,z].Q :: c?[w].R$$

but the π-calculus reduction rules can ignore the erroneous term $c?[y,z].Q$ and infer the following reduction:

$$\begin{array}{rl} & [\![\{c \mapsto ![x].P\}, c?[y,z].Q :: c?[w].R]\!] \\ \doteq & (\nu\, c)(c![x].P \mid c?[y,z].Q \mid c?[w].R) \\ \to & (\nu\, c)(P \mid c?[y,z].Q \mid \{x/w\}R) \end{array}$$

A slightly different problem occurs if channel queues are allowed to contain mixtures of readers and writers. For example, our abstract machine deadlocks on the following machine state

$$\{c \mapsto ![x].P :: ?[y].Q\}, \bullet$$

but the π-calculus reduction rules can reduce the input and output expressions contained in c's channel queue:

$$\begin{array}{rl} & [\![\{c \mapsto ![x].P :: ?[y].Q\}, \bullet]\!] \\ \doteq & (\nu\, c)(c![x].P \mid c?[y].Q) \\ \rightarrow & (\nu\, c)(P \mid \{x/y\}Q) \end{array}$$

We say that a machine state H, R is *well-formed* if $\vdash [\![H, R]\!]$ and no channel queue in H contains a mixture of input and output terms. The fact that $[\![H, R]\!]$ is well-typed guarantees that $[\![H, R]\!]$ is free from runtime errors (and also ensures that $[\![H, R]\!]$ is a closed expression). It is easy to check that, starting from an initial configuration $\{\}, P$, our abstract machine rules preserve the invariant that no channel queue in contains a mixture of input and output terms.

Theorem 7.7 (Deadlocks)
If H, R is well-formed and $H, R \not\rightarrow$ then $[\![H, R]\!] \not\rightarrow$.

Proof If we inspect the reduction rules for our abstract machine, we find that every well-formed machine state H, R is reducible if R is non-empty. In the case where R is empty it must be the case that $[\![H, R]\!] \not\rightarrow$ since our invariant on channel queues ensures that there are no input and output terms active on the same channel. $\square$

An important result we might hope to prove about our abstract machine is that it implements a *fair* reduction strategy. We do not attempt to prove such as result here, however, since the formal definition of fairness for π-calculus is a topic of current research [Pie95]. However, we do conjecture that our abstract machine guarantees what Pierce calls *process fairness*: any individual process that is infinitely often able to communicate (i.e., some communication partner is simultaneously available infinitely often) must eventually do so. The fact that we use FIFO queues for both channel queues and the run queue ensures that if a process is blocked on a channel queue then it will eventually become unblocked (assuming sufficient communication partners become available).

7.6 Simplifying replicated input

It is useful to make a further restriction on π-terms which guarantees that a channel never contains a mixture of ordinary and replicated inputs, and that a replicated input never encounters waiting writers when it executes. For example, if replicated inputs only appear in contexts of the following form (where Q never uses $c_1, \ldots, c_n$ for input and $c_1, \ldots, c_n$ are distinct variables)

$$(\nu\, c_1) \ldots (\nu\, c_n)(c_1?{*}[x_1].P_1 \mid \ldots \mid c_n?{*}[x_n].P_n \mid Q)$$

then we can guarantee that whenever a replicated input $c?{*}[\tilde{x}].P$ is executed, the channel c will always be empty. We therefore no longer need the Repl-W rule (which dealt with the case where c already contained some writers) and need not check the status of c before executing a replicated input in the Repl rule:

$$\frac{}{H,\ c?{*}[\tilde{x}].P :: R \rightarrow H\{c \mapsto ?{*}[\tilde{x}].P\},\ R} \qquad \text{Repl}$$

Our restriction on replication also guarantees that there will never be any other readers on a channel containing a replicated reader. Thus, whenever we output a value on such a channel there is no need to change the state of c, as can be seen in the new Out-R* rule:

$$\frac{H(c) = ?{*}[\tilde{x}].Q}{H,\ c![\tilde{a}].P :: R \rightarrow H,\ P :: R :: \{\tilde{a}/\tilde{x}\}Q} \qquad \text{Out-R*}$$

The above restriction on replication is not problematic in practice, since most uses of replication (for example, in recursively-defined processes) have exactly this format. In fact, we can translate any replicated input into an equivalent replicated input of the above form:

$$c?{*}[\tilde{x}].P \quad \doteq \quad (\nu\, d)(\, d?{*}[\,].c?[\tilde{x}].(P \mid d![\,]) \ \mid\ d![\,]\,)$$

In the Pict implementation, we enforce this restriction on replicated input using a combination of syntactic restrictions and special typing rules (Pict implements the I/O channel types proposed by Pierce and Sangiorgi [PS93], and can therefore easily check that a channel is never used for input in a particular context).

In fact, we can even do a better job of compiling outputs on the channels $c_1, \ldots, c_n$ in processes of the form

$$(\nu\, c_1) \ldots (\nu\, c_n)(c_1?{*}[x_1].P_1 \mid \ldots \mid c_n?{*}[x_n].P_n \mid Q)$$

since we know that once Q starts executing, we must have already executed all the replicated inputs on the channels $c_1, \ldots, c_n$. Thus, whenever we execute an output on the channel c_i we need not test the status of c_i (since we know that c_i must contain a single replicated input, and that outputs on c_i do not change its status). Of course, if we use c_i in a higher-order manner (by sending c_i along a channel to some other process), then the process which receives c_i will still have to test the status of c_i, as before.

The above refinements to replicated input and outputs along channels containing replicated inputs are necessary to get reasonable performance from functions which are encoded as processes. In a (strict) functional language we do not need to test the status of a function before calling it. In the π-calculus, we represent functions as processes which communicate on some distinguished channel. For example, the identity function might be represented as the following process (r is the channel along which id returns its result, and the process P represents the rest of the program)

$$(\nu\, id)(\; id?{*}[x, r].r![x] \;\mid\; P\,)$$

It would be very disappointing if every use of id in P (i.e. every output on the channel id) required us to test the status of id. Fortunately, the above process fits our criteria for optimising the Out-R* rule, since id is statically known to contain only a replicated input when the process P starts executing.

In fact, now that we have disallowed mixtures of inputs and replicated inputs on a channel, we can test the status of a channel more efficiently. A channel can now be in just one of four states: empty, containing blocked writers, containing blocked readers, or containing a single replicated reader. We can represent this information using a single status value which is stored in the channel. With this representation we can test a channel's status using a single multi-way conditional expression. If we allowed mixtures of readers and replicated readers in a channel queue, then we would need two conditional expressions to implement an output: one to test whether the channel contained any readers, and in the case where there is a reader in the channel, a second test to determine whether the reader is replicated or not.

7.7 Asynchronous communication

In the Pict programming language [PT95b], the implementation of communication is further simplified by the use of *asynchronous* communication. The asynchronous π-calculus is a simple sub-calculus of the π-calculus where we restrict the continuation P in every output process $c![\tilde{a}].P$ to be $\mathbf{0}$. In fact, asynchronous outputs are so common that we have already introduced some derived syntax for them, writing $c![\tilde{a}]$ instead of $c![\tilde{a}].\mathbf{0}$.

It is well known that synchronous communication can be simulated using explicit acknowledgments in an asynchronous calculus. For example, we can simulate the following synchronous communication

$$c![\tilde{a}].P \mid c?[\tilde{x}].Q \rightarrow P \mid \{\tilde{a}/\tilde{x}\}Q$$

using the asynchronous communication sequence below:

$$\begin{array}{ll} & (\nu\, k)(c![\tilde{a}, k] \mid k?[\,].P \mid c?[\tilde{x}, k].(k![\,] \mid Q)) \\ \rightarrow & (\nu\, k)(k?[\,].P \mid k![\,] \mid \{\tilde{a}/\tilde{x}\}Q) \\ \rightarrow & (\nu\, k)(P \mid \{\tilde{a}/\tilde{x}\}Q) \end{array}$$

The local channel k serves as an acknowledgment channel: Q signals on k when it has received the data, allowing P to continue.

Of course, the above encoding of synchronous communication in terms of asynchronous communication is much less efficient than implementing synchronous communication directly, but experience with Pict suggests that synchronous communication is in fact very rare. This is largely due to the fact that functions are encoded as processes (using essentially the same result-passing convention as we introduced in Section 2.7). This means there are many processes of the form $f?{*}[\tilde{x}, r]. \ldots r![\mathit{results}]$ which accept some arguments $\tilde{x}$ and a result channel r, compute something, and return some results along r. The communication which returns the results along r is asynchronous. Moreover, the standard calling convention for such processes also uses asynchronous communication, as can be seen in our derived form for getting results (also from Section 2.7):

$$\begin{array}{lll} & \texttt{let}\ x_1, \ldots, x_n = f(a_1, \ldots, a_m)\ \texttt{in}\ P & \\ \doteq & (\nu\, r)(f![a_1, \ldots, a_m, r] \mid r?[x_1, \ldots, x_n].P) & \quad r \notin \mathit{fv}(P, f, a_1, \ldots, a_m) \end{array}$$

There is no point in using a synchronous output to communicate with f, since we are not interested in detecting when the f starts executing, but when it finishes (and returns its results on r).

We now show how we can simplify our abstract machine if we only have to implement asynchronous communication. Firstly, the syntax of channel queues can be simplified, since we no longer need to put suspended output processes on a channel queue (we also retain the simplifications to replicated input proposed in Section 7.6). This is a useful simplification, since it is much cheaper to store a tuple of values in a channel queue, rather than store both a tuple of values and a suspended process. We will have more to say about the cost of suspending processes in Section 7.9.

Definition 7.8 (Asynchronous channel queues)

$$\begin{array}{rcll} C & ::= & ?[\tilde{x}_1].P_1 :: \ldots :: ?[\tilde{x}_n].P_n & \text{Queue of readers} \\ & & ![\tilde{x}_1] :: \ldots :: ![\tilde{x}_n] & \text{Queue of writers} \\ & & ?{*}[\tilde{x}].P & \text{Replicated reader} \end{array}$$

We can now simplify our communication rules to take account of the fact that all communication is asynchronous. In the case where we read a value from a channel which already contains a blocked writer, we need only extract the written values from the channel. There is no longer any need to put any writer processes back on the run queue:

$$\frac{H(c) = ![\tilde{a}] :: ws}{H,\ c?[\tilde{x}].P :: R \rightarrow H\{c \mapsto ws\},\ \{\tilde{a}/\tilde{x}\}P :: R}\ \text{AInp-W}$$

The reduction rules for asynchronous output expressions can also be simplified, since there is no other work to do once we have executed our output expression:

$$\frac{H(c) = ?[\tilde{x}].P :: rs}{H,\ c![\tilde{a}] :: R \rightarrow H\{c \mapsto rs\},\ R :: \{\tilde{a}/\tilde{x}\}P}\quad \text{AOut-R}$$

$$\frac{H(c) = ws}{H,\ c![\tilde{a}] :: R \rightarrow H\{c \mapsto ws :: ![\tilde{a}]\},\ R}\quad \text{AOut-W}$$

$$\frac{H(c) = ?{*}[\tilde{x}].P}{H,\ c![\tilde{a}] :: R \rightarrow H,\ R :: \{\tilde{a}/\tilde{x}\}P}\quad \text{AOut-R*}$$

In this way, we avoid placing many redundant processes on the run queue during asynchronous communications. Consider, for example, the reduction of the asynchronous process $c![\,] \mid c?[\,].P$ using our new reduction rules (where we assume that the channel c has already been created, and $H(c) = \bullet$)

$$
\begin{array}{lll}
 & H, c![\,] \mid c?[\,].P & \\
\rightarrow & H, c![\,] :: c?[\,].P & \text{Prl} \\
\rightarrow & H\{c \mapsto ![\,]\}, c?[\,].P & \text{AOut-W} \\
\rightarrow & H\{c \mapsto \bullet\}, P & \text{AInp-W}
\end{array}
$$

It is wasteful to use a synchronous reduction strategy to evaluate the above process, since we must store the nil process in c's channel queue, and then unblock the same nil process after interacting on c (both operations are a waste of time and space, since the nil process has no behaviour):

$$
\begin{array}{lll}
 & H, c![\,].\mathbf{0} \mid c?[\,].P & \\
\rightarrow & H, c![\,].\mathbf{0} :: c?[\,].P & \text{Prl} \\
\rightarrow & H\{c \mapsto ![\,].\mathbf{0}\}, c?[\,].P & \text{Out-W} \\
\rightarrow & H\{c \mapsto \bullet\}, P :: \mathbf{0} & \text{Inp-W}
\end{array}
$$

Note that the initial expressions in both examples are identical, since $c![\,]$ is just a shorthand for $c![\,].\mathbf{0}$.

7.8 Creating fewer processes

One of the key problems we encounter when executing π-calculus programs is that processes are very short-lived. Consider, for example, the process $(\nu\, r)(f![\tilde{a}, r] \mid r?[\tilde{x}].P)$, which creates a result channel r, sends the arguments $\tilde{a}$ and the result channel r to f, and waits for a reply on r. Our abstract machine executes the expressions $f![\tilde{a}, r]$ and $r?[\tilde{x}].P$ as separate processes. This is rather wasteful, since both expressions do relatively little work. It would be much better if we could execute both $f![\tilde{a}, r]$ and $r?[\tilde{x}].P$ within the same thread of control (i.e. without having to place either process on the run queue).

We therefore modify our abstract machine so that it is able to execute a number of actions within the same thread of control. The relation $H, P, R \Downarrow H', R'$ formalises how we execute a process within a single thread of control.

It takes a heap H, a process P and a run queue R and executes a number of operations in P. This yields an updated heap H' and run queue R'. The process P will either execute completely, or become blocked on some channel queue in the heap, so we never need to return a process as the result of evaluating P.

The relation $H, R \to H', R'$ now formalises just our process scheduling policy (it picks the first process out of the run queue and executes it):

$$\frac{H, P, R \Downarrow H', R'}{H, P :: R \to H', R'}\ \text{Sched}$$

The following reduction rules now do all the work of implementing communication. As we will see in Section 7.9, there are significant advantages to executing as many operations as possible within a single thread of control, since, in practice, suspending a process means preserving the current process context in the heap, and executing a new process requires us to load a new process context back out of the heap.

The reduction rule for restriction is very similar to the reduction rule we gave previously. We simply allocate a fresh channel c and continue executing the process P:

$$\frac{c \text{ fresh} \qquad H\{c \mapsto \bullet\},\ \{c/x\}P,\ R \Downarrow H',\ R'}{H,\ (\nu\, x)P,\ R \Downarrow H',\ R'}\ \text{Res}$$

However, our rule for parallel composition is significantly different to what we have seen before. Instead of putting Q on the end of the run queue (to be executed later), we evaluate *both* P and Q within the same thread of control:

$$\frac{H,\ P,\ R \Downarrow H',\ R' \qquad H',\ Q,\ R' \Downarrow H'',\ R''}{H,\ P \mid Q,\ R \Downarrow H'',\ R''}\ \text{Prl}$$

The evaluation rule for the nil process returns the heap and run queue unchanged:

$$\frac{}{H,\ \mathbf{0},\ R \Downarrow H,\ R}\ \text{Nil}$$

Executing a replicated input expression is an atomic operation, since we are guaranteed to find the channel c empty:

$$\frac{}{H,\ c?{*}[\tilde{x}].P,\ R \Downarrow H\{c \mapsto ?{*}[\tilde{x}].P\},\ R}\ \text{Repl}$$

If we execute an input on a channel which already contains a writer, then we continue executing P within the current thread of control:

$$\frac{H(c) = ![\tilde{a}] :: ws \qquad H\{c \mapsto ws\},\ \{\tilde{a}/\tilde{x}\}P,\ R \Downarrow H',\ R'}{H,\ c?[\tilde{x}].P,\ R \Downarrow H',\ R'}\ \text{AInp-W}$$

If we execute an output on a channel which does not contain any readers, we store the output values in the channel's queue:

$$\frac{H(c) = ws}{H,\ c![\tilde{a}],\ R \Downarrow H\{c \mapsto ws :: ![\tilde{a}]\},\ R}\ \text{AOut-W}$$

All infinite behaviour in π-terms arises as a result of interactions between processes and replicated input expressions. The following rule, which implements such interactions, places the process $\{\tilde{a}/\tilde{x}\}P$ on the end of the run queue, and therefore ensures that any evaluation $H, P, R \Downarrow H', R'$ is always finite.

$$\frac{H(c) = ?{*}[\tilde{x}].P}{H,\ c![\tilde{a}],\ R \Downarrow H,\ R :: \{\tilde{a}/\tilde{x}\}P}\ \text{AOut-R*}$$

For reasons which will become clear in a moment, in the AOut-R rule we do not execute the unblocked process P within the same thread of control as the current process, even though we could do so without breaking the property that every evaluation $H, P, R \Downarrow H', R'$ is always finite:

$$\frac{H(c) = ?[\tilde{x}].P :: rs}{H,\ c![\tilde{a}],\ R \Downarrow H\{c \mapsto rs\},\ R :: \{\tilde{a}/\tilde{x}\}P}\ \text{AOut-R}$$

With the above set of rules, the maximum amount of work required to execute a process P can always be determined from the structure of P. If we replaced the AOut-R rule with the following rule, which executes the unblocked process P within the same thread of control, this would no longer be the case (since we do not know, in general, how big the unblocked process P will be).

$$\frac{H(c) = ?[\tilde{x}].P :: rs \qquad H\{c \mapsto rs\},\ \{\tilde{a}/\tilde{x}\}P,\ R \Downarrow H',\ R'}{H,\ c![\tilde{a}],\ R \Downarrow H',\ R'}$$

It is important to know that a single thread will not execute for too long, since there may be other processes waiting to execute on the run queue. In the case of a user interface, for example, it may be important for those waiting processes to execute soon, so that they can provide quick responses to user input.

7.9 Machines states with environments

It is unrealistic to use a substitution operation to record variable binding information, so we now present a refinement of our previous abstract machine which uses explicit environments to record the bindings of variables to channels. This avoids any use of substitution and means that process terms are never modified during execution. Moreover, by using explicit environments, it becomes clearer where some of the real costs are in executing π-calculus programs.

An environment E is simply a finite mapping from variables to channels. The expression $E\{x \mapsto c\}$ denotes the finite map E extended with the mapping $x \mapsto c$ (we assume that x is always distinct from any other variables bound in E). The expression $E(x)$ denotes the channel associated with x in E and is undefined if x in not bound in E. We often write $E(\tilde{a})$ as an abbreviation for $E(a_1), \ldots, E(a_n)$.

Definition 7.9 (Environments)

$$E \quad ::= \quad x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$$

We need to store a process' environment whenever we suspend a process on a channel. We therefore modify the elements of a channel queue to store this information (we retain all the proposed simplifications to our abstract machine proposed in previous sections). Note in particular that we do not need to use any process environments to implement a queue of writers. This would not be the case if we allowed synchronous output.

Definition 7.10 (Channel queues)

C	$::=$	$(E_1, ?[\tilde{x}_1].P_1) :: \ldots :: (E_n, ?[\tilde{x}_n].P_n)$	Queue of readers
		$![\tilde{x}_1] :: \ldots :: ![\tilde{x}_n]$	Queue of writers
		$(E, ?{*}[\tilde{x}].P)$	Replicated reader

As before, a machine state is a pair of a heap and a run queue (which are defined as below). The run queue now also stores an environment for each process which is currently runnable.

Definition 7.11 (Machine state components)

$$\begin{array}{lcll} H & ::= & x_1 \mapsto C_1, \ldots, x_n \mapsto C_n & \text{Heap} \\ R & ::= & (E_1, P_1) :: \ldots :: (E_n, P_n) & \text{Run queue} \end{array}$$

7.10 Reduction rules with environments

Our process scheduling rule now removes both the first process P and its environment E from the run queue and executes P:

$$\frac{H, E, P, R \Downarrow H', R'}{H, (E, P) :: R \rightarrow H', R'} \text{ Sched}$$

The reduction rule for the nil process returns the heap and run queue unchanged:

$$\frac{}{H,\ E,\ \mathbf{0},\ R \rightarrow H,\ R} \text{ Nil}$$

The rule for parallel composition executes both P and Q in the same environment E:

$$\frac{H,\ E,\ P,\ R \Downarrow H',\ R' \qquad H',\ E,\ Q,\ R' \Downarrow H'',\ R''}{H,\ E,\ P \mid Q,\ R \Downarrow H'',\ R''} \text{ Prl}$$

The restriction operator $(\nu\, x)P$ allocates a new channel in the heap and continues executing P, recording the binding of x to c in the environment E.

$$\frac{c \text{ fresh} \qquad H\{c \mapsto \bullet\},\ E\{x \mapsto c\},\ P,\ R \Downarrow H',\ R'}{H,\ E,\ (\nu\, x)P,\ R \Downarrow H',\ R'} \text{ Res}$$

If there already is a writer available when we execute the input $x?[\tilde{y}].P$, we extract the stored data, record it in the environment E and continue executing P.

$$\frac{\begin{array}{c} E(x) = c \qquad H(c) = ![\tilde{a}] :: ws \\ H\{c \mapsto ws\},\ E\{\tilde{y} \mapsto \tilde{a}\},\ P,\ R \Downarrow H',\ R' \end{array}}{H,\ x?[\tilde{y}].P,\ R \Downarrow H',\ R'} \text{ AInp-W}$$

If there are no writers available when we execute the input $x?[\tilde{y}].P$ then we must store the current process and its environment in the channel's queue.

$$\frac{E(x) = c \qquad H(c) = rs}{H,\ E,\ x?[\tilde{y}].P,\ R \Downarrow H\{c \mapsto rs :: (E, ?[\tilde{y}].P)\},\ R}\ \text{Inp-R}$$

If the channel c already has some blocked readers in its queue when we execute the output $x![\tilde{a}]$, we unblock the first reader in the queue and record the bindings $\tilde{y} \mapsto E(\tilde{a})$ in the environment of the unblocked process.

$$\frac{E(x) = c \qquad H(c) = (F, ?[\tilde{y}].P) :: rs}{H,\ E,\ x![\tilde{a}],\ R \Downarrow H\{c \mapsto rs\},\ R :: (F\{\tilde{y} \mapsto E(\tilde{a})\}, P)}\ \text{AOut-R}$$

If the channel c already has some blocked writers in its queue when we execute the output $x![\tilde{a}]$, we store the output values $E(\tilde{a})$ in the channel queue.

$$\frac{E(x) = c \qquad H(c) = ws}{H,\ E,\ x![\tilde{a}],\ R \Downarrow H\{c \mapsto ws :: ![E(\tilde{a})]\},\ R}\ \text{AOut-W}$$

Our restrictions on replicated input guarantee that whenever $x?*[\tilde{y}].P$ is executed, the channel c will always be empty. We therefore need only suspend $x?*[\tilde{y}].P$ and place it in the channel c.

$$\frac{}{H,\ E,\ x?*[\tilde{y}].P,\ R \Downarrow H\{E(x) \mapsto (E, ?*[\tilde{y}].P)\},\ R}\ \text{Repl}$$

If the channel associated with x contains a replicated input when we execute the output $x![\tilde{a}]$, we place a new copy of the replicated process at the end of the run queue, recording the bindings $\tilde{y} \mapsto E(\tilde{a})$ in the environment of the new process.

$$\frac{H(E(x)) = (F, ?*[\tilde{y}].Q)}{H,\ E,\ x![\tilde{a}],\ R \Downarrow H,\ R :: (F\{\tilde{y} \mapsto E(\tilde{a})\}, Q)}\ \text{AOut-R*}$$

7.11 Using environments more efficiently

In the π-calculus, processes are created very frequently, and tend run for a very short amount of time before become blocked (or terminating). This stands in sharp contrast with concurrent languages such as CML [Rep92] or Facile [GMP89], where each process may do a significant amount of (sequential) work before terminating, or communicating on a channel. It is therefore important to allow processes to be added and removed from the run queue without allocating any permanent storage in the heap. Similarly, since processes are very short-lived, it makes sense to try and store environment bindings in some kind of temporary area, rather than allocating environment entries in the heap. Fortunately, as we will see in Chapter 8, once we have a mechanism in place for storing run queue entries in a temporary area, it is possible to use the same mechanism to store many of the environment bindings that are generated during execution.

Whenever the evaluation rules from the previous section place a process on the run queue, they always need to store a number of argument bindings at the same time. For example, the AOut-R rule needs to store the bindings for $\tilde{y}$ somewhere (until the process P is able to consume them):

$$\frac{E(x) = c \qquad H(c) = (F, ?[\tilde{y}].P) :: rs}{H,\ E,\ x![\tilde{a}],\ R \Downarrow H\{c \mapsto rs\},\ R :: (F\{\tilde{y} \mapsto E(\tilde{a})\}, P)}\ \text{AOut-R}$$

Fortunately, since we need only store the bindings for $\tilde{y}$ in memory until P starts executing, we can store them in the run queue itself (rather than adding them to the environment F, as in the rule above).

We therefore add a new component to the elements of our run queue: a local environment L. Local environments have exactly the same abstract description as the environments E introduced in the previous section, but we expect them to be implemented differently. A local environment L is intended to be a very short-lived entity and cannot be shared amongst more than one process (otherwise it would be difficult to avoid allocating it in the heap). In fact, in Chapter 8 we will see that the environments $L_1, \ldots, L_n$ can actually be stored within the run queue (which itself lives in a special temporary storage area). A global environment E may potentially be very long-lived, and is allocated in the heap. Many processes may share the same global environment E.

Definition 7.12 (Run queue)

$$R \quad ::= \quad (E_1, L_1, P_1) :: \ldots :: (E_n, L_n, P_n)$$

The efficient implementation of run queues mentioned earlier relies on the fact that we allocate and deallocate storage in the run queue in a very regular manner: we add new processes at the tail of the run queue and remove processes from the head of the run queue, but we never attempt to insert or remove a process from the middle of the run-queue, for instance. Moreover, there is only one run queue. Storage allocated for channel queues, on the other hand, has a much less well-defined lifetime: when a process blocks on a channel we generally have no idea when that process will be unblocked. Thus, whenever a process becomes blocked on a channel, we must build a new environment in the heap. We ensure that the environment we build is minimal (i.e. contains only those bindings which might actually be used when the process is unblocked) so that in the case where a process remains blocked on a channel queue for a significant amount of time we do not retain pointers to values which could actually be garbage collected.

Definition 7.13 presents the reduction rules for our final abstract machine. This abstract machine is the one upon which our compilation of π-calculus to C is based. Most of the reduction rules are similar to rules we have already seen, so we just explain a few of the more important features.

For those environment entries which are created within the same thread of control it is often possible to avoid storing such entries in memory at all. In the compilation of π-calculus to C, presented in Chapters 8 and 9, we use C's built-in variable-binding mechanism to store variable bindings which are created within the same thread of control. We therefore need not do any explicit allocation for such bindings, and it is reasonable to expect an optimising C compiler to store most of them in registers. To capture the fact that we can implement environment entries within the same thread of control differently from those stored in the run queue or heap, we add a new local environment L to our evaluation relation. Evaluations now take the form $H, E, L, P, R \Downarrow H', R'$, where E contains those variable bindings which are stored in the heap and L contains those bindings created during the current thread of execution.

The binding for a variable x may now be stored in either the global environment E or the local environment L. The expression $(E \cup L)(x) \doteq L(x)$ if x is bound in L and $(E \cup L)(x) \doteq E(x)$ otherwise. Note that it is always possible to statically determine whether a variable is bound locally or not, so the expression $(E \cup L)(x)$ does not require any runtime tests.

Definition 7.13 (Reduction rules using local environments)

$$\frac{H,\ E,\ L,\ P,\ R \Downarrow H',\ R'}{H,\ (E,\ L,\ P) :: R \rightarrow H,\ R} \quad \text{Sched}$$

$$\frac{}{H,\ E,\ L,\ \mathbf{0},\ R \Downarrow H,\ R} \quad \text{Nil}$$

$$\frac{H,\ E,\ L,\ P,\ R \Downarrow H',\ R' \qquad H',\ E,\ L,\ Q,\ R' \Downarrow H'',\ R''}{H,\ E,\ L,\ (P \mid Q),\ R \Downarrow H'',\ R''} \quad \text{Prl}$$

$$\frac{c \text{ fresh} \qquad H\{c \mapsto \bullet\},\ E,\ L\{x \mapsto c\},\ P \Downarrow H',\ R'}{H,\ E,\ L,\ (\nu\, x)P,\ R \Downarrow H',\ R'} \quad \text{Res}$$

$$\frac{\begin{array}{c}(E \cup L)(x) = c \qquad H(c) = ![\tilde{a}] :: ws \\ H\{c \mapsto ws\},\ E,\ L\{\tilde{y} \mapsto \tilde{a}\},\ P,\ R \Downarrow H',\ R'\end{array}}{H,\ (E, L, x?[\tilde{y}].P) :: R \Downarrow H',\ R'} \quad \text{Inp-W}$$

$$\frac{(E \cup L)(x) = c \qquad H(c) = rs \qquad F = (E \cup L)\lfloor fv(P)}{H,\ E,\ L,\ x?[\tilde{y}].P,\ R \Downarrow H\{c \mapsto rs :: (F, ?[\tilde{y}].P)\},\ R} \quad \text{Inp-R}$$

$$\frac{(E \cup L)(x) = c \qquad H(c) = (F, ?[\tilde{y}].P) :: rs \qquad (E \cup L)(\tilde{a}) = \tilde{b}}{H,\ E,\ L,\ x![\tilde{a}],\ R \Downarrow H\{c \mapsto rs\},\ R :: (F, \{\tilde{y} \mapsto \tilde{b}\}, P)} \quad \text{AOut-R}$$

$$\frac{(E \cup L)(x) = c \qquad H(c) = ws \qquad (E \cup L)(\tilde{a}) = \tilde{b}}{H,\ E,\ L,\ x![\tilde{a}],\ E,\ R \Downarrow H\{c \mapsto ws :: ![\tilde{b}]\},\ R} \quad \text{AOut-W}$$

$$\frac{(E \cup L)(x) = c}{H,\ E,\ L,\ x?*[\tilde{y}].P,\ R \Downarrow H\{c \mapsto ((E \cup L)\lfloor fv(P), ?*[\tilde{y}].P)\},\ R} \quad \text{Repl}$$

$$\frac{(E \cup L)(x) = c \qquad H(c) = (F, ?*[\tilde{y}].P) \qquad (E \cup L)(\tilde{a}) = \tilde{b}}{H,\ E,\ L,\ x![\tilde{a}],\ R \rightarrow H,\ R :: (F, \{\tilde{y} \mapsto \tilde{b}\}, P)} \quad \text{AOut-R*}$$

The Res rule now stores the new binding $x \mapsto c$ in the local environment of the current thread. Note that the evaluation rule for parallel composition does not require any storage allocation, since both P and Q are executed within the current thread of control (and we therefore need not preserve any environment entries in the heap).

Since in the Inp-R and Repl rules we need to suspend the current process and store it and its environment entries on a channel queue, we build a new, minimal, environment $(E \cup L)\lfloor fv(P)$ in the heap (the expression $\lfloor fv(P)$ restricts the domain of $E \cup L$ to the set of free variables of P).

The AOut-R and AOut-R* rules now store the argument bindings $\tilde{y} \mapsto \tilde{b}$ in the run queue itself, ready to be consumed by P once it starts executing. This does not require any permanent storage to be allocated in the heap, which is particularly important in the case of the AOut-R* rule, since all infinite or recursive behaviour arises as a result of the AOut-R* rule.