The Polymorphic Pi-calculus: Theory and Implementation

David N. Turner

Ph. D.
University of Edinburgh
1995

Abstract

We investigate whether the π-calculus is able to serve as a good foundation for the design and implementation of a strongly-typed concurrent programming language. The first half of the dissertation examines whether the π-calculus supports a simple type system which is flexible enough to provide a suitable foundation for the type system of a concurrent programming language. The second half of the dissertation considers how to implement the π-calculus efficiently, starting with an abstract machine for π-calculus and finally presenting a compilation of π-calculus to C.

We start the dissertation by presenting a simple, structural type system for π-calculus, and then, after proving the soundness of our type system, show how to infer principal types for π-terms. This simple type system can be extended to include useful type-theoretic constructions such as recursive types and higher-order polymorphism. Higher-order polymorphism is important, since it gives us the ability to implement abstract datatypes in a type-safe manner, thereby providing a greater degree of modularity for π-calculus programs.

The functional computational paradigm plays an important part in many programming languages. It is well-known that the π-calculus can encode functional computation. We go further and show that the type structure of λ-terms is preserved by such encodings, in the sense that we can relate the type of a λ-term to the type of its encoding in the π-calculus. This means that a π-calculus programming language can genuinely support typed functional programming as a special case.

An efficient implementation of π-calculus is necessary if we wish to consider π-calculus as an operational foundation for concurrent programming. We first give a simple abstract machine for π-calculus and prove it correct. We then show how this abstract machine inspires a simple, but efficient, compilation of π-calculus to C (which now forms the basis of the Pict programming language implementation).

Acknowledgements

I would like to express my gratitude to Robin Milner, my supervisor, for his excellent guidance and encouragement. Many thanks also go to my friends in the Department of Computer Science in Edinburgh, who made my time studying in Edinburgh both profitable and enjoyable.

My joint work with Benjamin Pierce on the Pict programming language has been of great help in developing both the theoretical and the practical parts of this dissertation. I consider myself very fortunate to have been able to collaborate with Benjamin on such an enjoyable project and hope that this dissertation serves as a good foundation for both the Pict type system and the implementation of Pict.

This work was jointly supported by the U.K. Science and Engineering Research Council, and Harlequin Limited. I am indebted Phil Wadler for giving me time to finish this dissertation while working at the University of Glasgow.

Declaration

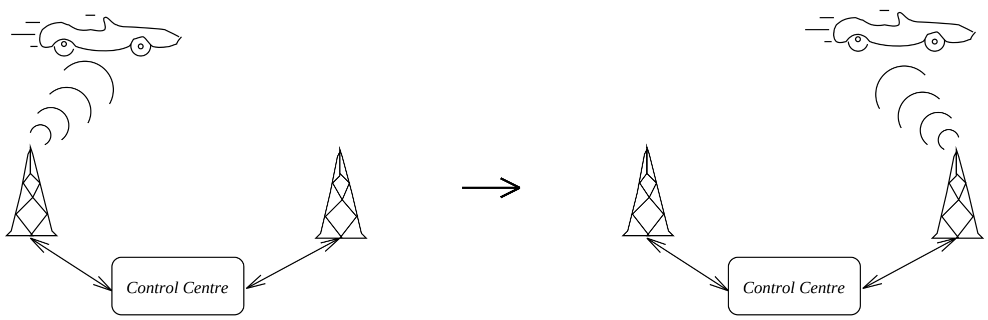
This dissertation has been composed by myself. The ideas and results contained in it, unless otherwise stated, are my own.

Contents

Chapter 1

Introduction

The π-calculus [MPW89a, MPW89b, MPW92] is a process calculus which is able to describe dynamically changing networks of concurrent processes. An example of such a process network is shown below. The network models a mobile telephone and two ground stations. To describe how the telephone can switch from using one ground station to another, we need to be able to change the communication topology of the network (unlinking the telephone from the first station and linking it to the second). The telephone must be able to accept messages (along its existing links) which tell it how to access other ground stations. Thus, as the car travels from region to region, details of other, closer, ground stations may be transmitted to the phone, enabling it to reroute its communications through the closest station.



The active agents of the π-calculus are *processes*, which exchange information over *channels*. A process of the form $c!v.P$ outputs the value v along the channel c and then continues as P. This communication is *synchronous*: P is prevented from executing until the communication on c has completed. Similarly, the process $c?x.Q$ waits to receive a value along c, continuing as Q with the value received substituted for the formal parameter x. (The π-calculus literature has many variations on the syntax used for input and output – we use a syntax which is most similar to that used in the Pict programming language [PT95b].)

Two processes may be run in parallel using the parallel composition operator $|$, thus enabling interactions between them. In the following example, since both processes wish to communicate on the channel c, an interaction is possible:

$$\begin{array}{rrcl} & c!v.P & | & c?x.Q \\ \rightarrow & P & | & \{v/x\}Q \end{array}$$

(We use the symbol $\rightarrow$ to denote process reduction, and $\{v/x\}Q$ to denote the substitution of v for x in Q).

This style of *synchronous rendezvous* is used in many process calculi, including CCS [Mil80, Mil89] and value-passing CCS [Mil80, Mil89]. However, unlike its predecessors, the π-calculus' channels not only provide the means of communication, but are also the values exchanged during communication.

This dissertation investigates whether the π-calculus is able to serve as a good foundation for the design and implementation of a strongly-typed concurrent programming language. The first half of the dissertation examines whether the π-calculus supports a simple type system which is flexible enough to provide a suitable foundation for the type system of a concurrent programming language. The second half of the dissertation considers how to implement the π-calculus efficiently, starting with an abstract machine for π-calculus and finally presenting a compilation of π-calculus to C.

The following sections summarise the contents of this dissertation.

1.1 The polyadic π-calculus

The input and output primitives of the π-calculus are *monadic*: exactly one channel is exchanged during each communication. The *polyadic* π-calculus [Mil91a] is a useful extension of the π-calculus which allows the atomic communication of tuples of channels. The additional structure introduced by polyadic communication is important, since it raises the possibility of runtime failure (the tuple sent along a channel may not have the same length as the tuple expected by the receiver). The monadic π-calculus has no corresponding operational notion of runtime failure, even though it can encode polyadic communication.

In Chapter 2 we give the syntax and semantics of the polyadic π-calculus, and then show why encoding the polyadic π-calculus in the monadic π-calculus destroys our operational notion of runtime failure. We also present some convenient derived forms and examples (which shall appear again in later chapters).

1.2 Process typing

The formal simplicity of the λ-calculus makes it an ideal foundation for the construction of type systems for sequential programming languages. We believe that the π-calculus can play a similar role in the construction of type systems for concurrent programming languages. In Chapter 3, we show that the polyadic π-calculus admits a simple typing discipline, which can easily be extended to include useful type-theoretic constructions such as recursive types and polymorphism. We show how our typing rules behave on the examples and derived forms of Chapter 2, and prove (by means of a subject-reduction theorem) that our type system guarantees freedom from runtime errors.

The simplicity of our π-calculus type system allows us to infer types automatically. The benefits of automatic type inference have been clearly demonstrated in languages such as Standard ML [MTH90] and Haskell [HJW$^+$92], where the programmer has to write only a minimum of explicit type information. We use similar, unification based, techniques to infer types for π-terms.

1.3 Recursive types

There are many useful programs which cannot be assigned a type in the simply-typed λ-calculus. Similarly, there are many useful π-calculus programs which cannot be assigned a type in our simple type system. An important deficiency is that we cannot support programming with recursive datatypes (for example, lists or trees). In Chapter 4, we present a simple solution to this problem: recursive types. In fact, as in the λ-calculus, recursive types make the typed π-calculus as expressive as the untyped monadic π-calculus, since we can assign a type to every monadic π-term.

1.4 Polymorphism

A common disadvantage of simple type systems is that, although they prevent common programming errors, they also disallow many useful and intuitively correct programs. Polymorphic type systems overcome much of this problem by allowing generic operations, that is, operations which can be safely applied to many different types of argument. List operations such as reversing and concatenation are good examples of generic operations, since they act completely independently of the types of the elements in the lists. The extra flexibility offered by a polymorphic type system seems to be enough to allow a more natural style of programming, where the type system is not perceived as 'getting in the way'.

In Chapter 5 we define an explicitly-typed polymorphic type system for π-calculus which arises as a natural extension of the simple type system we presented in Chapter 3. We illustrate the utility of polymorphic types in π-calculus programming using a number of examples, and then show how polymorphic channels can be used to model abstract datatypes in a type-safe manner. We prove our type system sound using techniques similar to those we used to prove the soundness of our monomorphic type system.

1.5 Relating typed λ-terms to typed π-terms

Our type system is constructed using type-theoretic techniques borrowed from the λ-calculus, so it is natural to ask if there is a precise relationship between well-typed λ-terms and well-typed π-terms. Milner [Mil90] has already shown that we can encode various λ-calculus reduction strategies in the π-calculus. In Chapter 6, we show that the type structure of a λ-term is often preserved by these encodings. In fact, in some cases, we can even prove that the principal type of a λ-term is directly related to its encoding's principal type in the π-calculus.

Perhaps the most interesting feature of these encodings is that (in the presence of polymorphism) they don't always work! For example, we find that the Damas-Milner type system [DM82] does not always agree with our π-calculus type system as to which types a λ-term may inhabit. This might not be surprising to those familiar with ML, since it is well-known that Damas-Milner polymorphism is unsafe in the presence of side-effects [Tof88]. The π-calculus is, by its very nature, a calculus containing side-effects, so it had better not allow the same kind of polymorphism as the Damas-Milner type system.

In fact, we find that the soundness of the Damas-Milner type system is closely connected to the precise evaluation order used (a result which was recently discovered by Leroy [Ler93], though not using encodings into the π-calculus). We find that the call-by-value encoding of λ-calculus does not preserve its Damas-Milner type structure, but the call-by-name encoding does.

1.6 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]. In Chapter 7, we 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 in Chapter 7.

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 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.

1.7 Compiling Pict to C

The primary motivation of the Pict [PT95b] project was to design and implement a high-level concurrent language purely in terms of π-calculus primitives. There have been many proposals for concurrent languages [Car86, Hol83, Rep92, Mat91, GMP89, etc.] which include communication primitives which are very similar to those of the π-calculus. However, to our knowledge, none have proposed using π-calculus primitives as the sole mechanism of computation.

The Pict language consists of two layers: a very simple core calculus (which is just π-calculus extended with built-in structured data), and a high-level language which is defined via translation into the core calculus. In Chapters 8 and 9, we describe an efficient compilation of core Pict to C. The compilation has been implemented and now forms part of the Pict programming language implementation. The compilation is (perhaps surprisingly) quite simple, and is designed so that it can exploit information provided by a number of program analyses.

Our compilation can be thought of as a more refined description of the abstract machine which we present in Section 7.11, where we are explicit about the exact representation of all runtime data and the implementation of operations such as environment lookup.

1.8 Useful information

Labelled items (such as definitions, lemmas or theorems) are labelled $c.n$, where c is the chapter in which the item occurs, and n indicates that the item is the n'th labelled item in that chapter.

Some familiarity with CCS and the π-calculus would be helpful for readers of this dissertation. Useful background reading can be found in [Mil89, MPW89a, MPW89b, MPW92, Mil90, Mil91a].

The implementation of the Pict programming language (referred to previously) is available electronically. The distribution includes a manual and tutorial, as well as a number of examples of X-Windows programs written in Pict. This dissertation is also available online.

Chapter 2

The polyadic π-calculus

The input and output primitives of the π-calculus are *monadic*: exactly one channel is exchanged during each communication. The *polyadic* π-calculus [Mil91a] is a useful extension of the π-calculus which allows the atomic communication of tuples of channels. The additional structure introduced by polyadic communication is important, since it raises the possibility of runtime failure (the tuple sent along a channel may not have the same length as the tuple expected by the receiver). The monadic π-calculus has no corresponding operational notion of runtime failure, even though it can encode polyadic communication.

We first give the syntax and semantics of the polyadic π-calculus, and then show why encoding the polyadic π-calculus in the monadic π-calculus destroys our operational notion of runtime failure.

2.1 Syntax

The syntax of the polyadic π-calculus is given in Definition 2.1. We require that all arguments to the summation operator are either input prefixes, output prefixes, or the nil process. This is commonly known as *guarded summation*, since every non-trivial term in a summation is guarded by an input or output prefix. We could allow full summation, but it adds very little useful power in exchange for the complexity it introduces in the formal semantics of our calculus.

The restriction operator $(\nu\, x)P$ binds the variable x in the process P. The input operator $x?[x_1, \ldots, x_n].P$ binds the variables $x_1, \ldots, x_n$ in P. We disallow duplicate bound variables in input prefixes. When the length of a sequence is clear from the context, or is unimportant, we let $\tilde{x}$ denote $x_1, \ldots, x_n$. We do not distinguish terms which are α-convertible.

Definition 2.1 (Process syntax)

$$
\begin{array}{rcll}
P, Q, R, S & ::= & P \mid P & \text{Parallel composition} \\
 & & (\nu\, x)P & \text{Restriction} \\
 & & P + P & \text{Summation} \\
 & & x?[x_1, \ldots, x_n].P & \text{Input} \\
 & & x![x_1, \ldots, x_n].P & \text{Output} \\
 & & *P & \text{Replication} \\
 & & \mathbf{0} & \text{Nil}
\end{array}
$$

It is very common for the continuation of an output to be the nil process, so we allow $x![x_1, \ldots, x_n]$ as an abbreviation for $x![x_1, \ldots, x_n].\mathbf{0}$.

The precedences of the operators are described below. For example, the term $(\nu\, x)x![a, b].P + Q$ denotes $((\nu\, x)x![a, b].P) + Q$ and $*P \mid Q$ denotes $(*P) \mid Q$. Note the precedence of the (meta-syntactic) substitution operator which, for example, implies that $\{y/x\}P \mid Q$ denotes $(\{y/x\}P) \mid Q$.

$$
\text{Parallel Composition} < \text{Summation} < \begin{cases} \text{Input, Output, Restriction,} \\ \text{Replication, Substitution.} \end{cases}
$$

Tuples have no interesting evaluation behaviour. We cannot, for example, embed communications inside tuples. Thus, all computation in the polyadic π-calculus is still based on processes communicating over channels, just as in the monadic π-calculus.

2.2 Semantics

We present the semantics of the polyadic π-calculus using a reduction relation (see [Mil91a] for more details). This style of semantics involves defining two relations on processes: a reduction relation, which formalises the actual communication behaviour of processes, and a structural congruence relation. The structural congruence relation allows us to rewrite a process so that any two active input or output prefixes can be syntactically juxtaposed. This simplifies the presentation of the reduction relation by reducing the number of cases we have to consider.

Definition 2.2 describes the reduction of π-terms. The first two rules state that we can reduce under both parallel composition and restriction. (The symmetric rule for parallel composition is redundant, because of the use of structural congruence.)

Definition 2.2 (Process reduction)

$$\frac{Q \to R}{P \mid Q \to P \mid R} \qquad \frac{P \to Q}{(\nu\, x)P \to (\nu\, x)Q}$$

$$(P + c?[x_1, \ldots, x_n].Q) \mid (c![y_1, \ldots, y_n].R + S) \to \{y_1, \ldots, y_n/x_1, \ldots, x_n\}Q \mid R$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

The communication rule takes two processes which are willing to communicate on the channel c, and simultaneously substitutes the free names $y_1, \ldots, y_n$ for the bound variables $x_1, \ldots, x_n$. (The simultaneous substitution of $y_1, \ldots, y_n$ for $x_1, \ldots, x_n$ is well-defined, since we disallow duplicate bound variables in input prefixes.) The remaining components of the summations (P and S) are discarded, since at most one component of a summation is allowed to execute. Note that the communication rule is the only rule which directly reduces a π-term.

The communication rule assumes that processes are in a particular format (for example, the inputting process must be on the left, and must be contained in a summation). The structural congruence rule allows us to rewrite processes so that they have the correct format for the communication rule. (Some rewriting may also be necessary before using the parallel composition rule, since it assumes that the next reduction will always occur in its right sub-component.)

The rule for communication is sufficient, since we are only considering guarded summation. If we allowed full summation, then we would not be able to assume that both participants in a communication are immediate sub-components of summations.

Definition 2.3 (Structural congruence) Let structural congruence, $\equiv$, be the smallest congruence relation which satisfies the axioms below.

$$*P \equiv P \mid *P$$

$$\begin{array}{rcl} P & \equiv & P \mid \mathbf{0} \\ P \mid Q & \equiv & Q \mid P \\ (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \end{array} \qquad \begin{array}{rcl} P & \equiv & P + \mathbf{0} \\ P + Q & \equiv & Q + P \\ (P + Q) + R & \equiv & P + (Q + R) \end{array}$$

$$(\nu\, x)P \mid Q \equiv (\nu\, x)(P \mid Q) \qquad x \notin fv(Q)$$

Definition 2.3 presents the structural congruence relation. Most of the rules simply assert the associativity and commutativity of the parallel composition and summation operators.

We now show some example reductions which illustrate simple uses of structural congruence. To infer that the following process can do a communication on c, we need to use the associativity and commutativity of parallel composition to bring the input and output prefixes together:

$$\begin{array}{ll} & c?[x,y].P \mid (R \mid c![a,b].Q) \\ \equiv & R \mid (c?[x,y].P \mid c![a,b].Q) \end{array}$$

The communication rule also expects both the input and output prefixes to be contained in summations, so we must use the identity and commutativity rules for summation:

$$\equiv \quad R \mid (\mathbf{0} + c?[x,y].P \mid c![a,b].Q + \mathbf{0})$$

We can now use the parallel composition and communication rules to infer the communication on c:

$$\rightarrow \quad R \mid (\{a,b/x,y\}P \mid Q)$$

The structural congruence rules also allow us to generate as many copies of a replicated process as we require. This allows us to ignore replication in the reduction rules. For example, we can use structural congruence to make a single copy of the replicated process $*c?[x, y].P$, which can then start communicating in the usual way.

$$
\begin{array}{ll}
 & *c?[x,y].P \mid (R \mid c![a,b].Q) \\
\equiv & (*c?[x,y].P \mid c?[x,y].P) \mid (R \mid c![a,b].Q) \\
\equiv & (*c?[x,y].P \mid R) \mid (\mathbf{0} + c?[x,y].P \mid c![a,b].Q + \mathbf{0}) \\
\rightarrow & (*c?[x,y].P \mid R) \mid (\{a,b/x,y\}P \mid Q)
\end{array}
$$

The benefits of a reduction-style semantics are most obvious when we consider the restriction operator: the reduction rules contain no mention of restriction, except for the rule which allows us to reduce underneath restriction. We have managed to separate the rules implementing communication from the rules which change the scope of restriction: in a labelled-transition semantics, the two operations are usually combined, resulting in more complicated rules.

In the case where a channel c is shared between two communicating processes, the reduction rule for restriction can be used directly:

$$
\begin{array}{ll}
 & (\nu\, c)(c?[x,y].P \mid c![a,b].Q) \\
\equiv & (\nu\, c)(\mathbf{0} + c?[x,y].P \mid c![a,b].Q + \mathbf{0}) \\
\rightarrow & (\nu\, c)(\{a,b/x,y\}P \mid Q)
\end{array}
$$

In the case where a private channel is being communicated to another process, we must first expand the scope of the private channel to encompass the recipient (using structural congruence), and then reduce the process:

$$
\begin{array}{ll}
 & c?[x,y].P \mid (\nu\, a)(c![a,b].Q) \\
\equiv & (\nu\, a)(c?[x,y].P \mid c![a,b].Q) \\
\equiv & (\nu\, a)(\mathbf{0} + c?[x,y].P \mid c![a,b].Q + \mathbf{0}) \\
\rightarrow & (\nu\, a)(\{a,b/x,y\}P \mid Q)
\end{array}
$$

Expanding the scope of a in the above example is only valid if a does not already occur in the process $c?[x, y].Q$. This condition is checked in the side-condition on the rule $(\nu\, x)P \mid Q \equiv (\nu\, x)(P \mid Q)$.

Suppose $x \in fv(Q)$ and we wish to apply the $(\nu\, x)P \mid Q \equiv (\nu\, x)(P \mid Q)$ rule. We achieve this by first α-converting the term $(\nu\, x)P$, renaming x so that it no longer occurs free in Q. Then we can apply the structural congruence rule. (We do not mention such α-conversions explicitly, since we do not distinguish α-convertible processes.)

2.3 Runtime failure

The additional structure introduced by the polyadic communication primitives is important, since it raises the possibility of runtime failure. For example, the process

$$c![v_1, v_2, v_3].P \mid c?[x_1, x_2].Q$$

is ill-formed, since it attempts to input a pair on c, when c is in fact being used to carry a triple.

Definition 2.4 (Runtime failure)

$$\frac{P\ \mathit{fails}}{P \mid Q\ \mathit{fails}} \qquad \frac{P\ \mathit{fails}}{(\nu\, x)P\ \mathit{fails}} \qquad \frac{P \equiv Q \qquad Q\ \mathit{fails}}{P\ \mathit{fails}}$$

$$\frac{m \neq n}{(P + c?[x_1, \ldots, x_m].Q) \mid (c![y_1, \ldots, y_n].R + S)\ \mathit{fails}}$$

Definition 2.4 formalises what we mean by runtime failure in the polyadic π-calculus (the rules mimic the reduction rules exactly, except for the communication rule, which actually detects the runtime failure).

2.4 Encoding polyadic communication

The ability to create private channels and communicate them to other processes allows us to encode polyadic communication in the monadic π-calculus [Mil91a]. We can encode polyadic output as below (we use the symbol $\doteq$ to denote definitional equality):

$$c![v_1, \ldots, v_n].P \quad \doteq \quad (\nu\, w)(c!w.w!v_1. \ldots w!v_n.P) \qquad w \notin \mathit{fv}(P, v_1, \ldots, v_n)$$

The encoding first creates a new channel w and sends it along c. It then transmits $v_1, \ldots, v_n$ sequentially along w and continues as P. The condition $w \notin fv(P, v_1, \ldots, v_n)$ ensures that the auxiliary variable w is not a free variable of either P or $v_1, \ldots, v_n$.

We can encode polyadic input using a similar composite communication; the components of the received tuple are bound to $x_1, \ldots, x_n$:

$$c?[x_1, \ldots, x_n].Q \ \doteq \ c?w.w?x_1. \ldots w?x_n.Q \qquad\qquad w \notin fv(Q, x_1, \ldots, x_n)$$

As the following example demonstrates, the communication of $[v_1, v_2]$ along c results, after a number of reduction steps, in the substitution of v_1 for x_1 and v_2 for x_2.

$$\begin{array}{crcr}
 & c![v_1, v_2].P & \mid & c?[x_1, x_2].Q \\
\doteq & (\nu\, w)(c!w.w!v_1.w!v_2.P) & \mid & c?w.w?x_1.w?x_2.Q \\
\rightarrow & (\nu\, w)(w!v_1.w!v_2.P & \mid & w?x_1.w?x_2.Q) \\
\rightarrow & (\nu\, w)(w!v_2.P & \mid & w?x_2.\{v_1/x_1\}Q) \\
\rightarrow & (\nu\, w)(P & \mid & \{v_2/x_2\}\{v_1/x_1\}Q)
\end{array}$$

After the exchange of w along c, w becomes a private channel shared between the inputting and outputting processes. The final result of the composite communication still mentions the private channel w, but this extra channel is harmless, since w is not mentioned in the scope of $(\nu\, w)$ (the side-conditions in the encoding ensure that $w \notin fv(P, Q, v_1, v_2, x_1, x_2)$).

2.5 Runtime failure in the monadic π-calculus

We now show why encoding the polyadic π-calculus in the monadic π-calculus destroys our operational notion of runtime failure. Consider the following process (which fails in the polyadic π-calculus):

$$c![v_1, v_2, v_3].P \mid c?[x_1, x_2].Q$$

If we examine the encodings of $c![v_1, v_2, v_3].P$ and $c?[x_1, x_2].Q$ into the monadic π-calculus, we find that the following reduction sequence is possible (we assume that $w \notin fv(P, Q, v_1, v_2, v_3, x_1, x_2)$):

$$
\begin{array}{rrcr}
 & c![v_1, v_2, v_3].P & | & c?[x_1, x_2].Q \\
\doteq & (\nu\, w)(c!w.w!v_1.w!v_2.w!v_3.P) & | & c?w.w?x_1.w?x_2.Q \\
\rightarrow & (\nu\, w)(w!v_1.w!v_2.w!v_3.P & | & w?x_1.w?x_2.Q) \\
\rightarrow & (\nu\, w)(w!v_2.w!v_3.P & | & w?x_2.\{v_1/x_1\}Q) \\
\rightarrow & (\nu\, w)(w!v_3.P & | & \{v_2/x_2\}\{v_1/x_1\}Q)
\end{array}
$$

It is now much harder to detect that our example has failed, since it can actually perform a number of reduction steps. Only after the communication of v_1 and v_2 along w do we encounter a problem: the process $w!v_3.P$ cannot proceed, since there are no processes which can cooperate with the output on w.

Thus, we find that a failure due to an ill-formed communication manifests itself as a deadlocked sub-process. It is not the case that the whole process deadlocks (the process $\{v_2/x_2\}\{v_1/x_1\}Q$ is free to proceed), so a precise definition of this failure in terms of deadlock is difficult. (Another reason why deadlock is not a suitable notion of failure is that there are many useful π-calculus programs which *expect* certain processes to become deadlocked. Often, it is assumed that deadlocked processes will be garbage collected so that, for example, a server which has no clients will be garbage collected without any need for explicit termination.)

These difficulties suggest that the monadic π-calculus does not have enough syntactic structure to support a simple type system: there is not enough information present in a π-term to guess whether the programmer expected it to deadlock or not. The polyadic π-calculus can be thought of as a minimal extension of the π-calculus which allows us to detect runtime failure purely syntactically.

2.6 Recursive process definitions

Most recent presentations of the π-calculus use a replication operator to enable processes to have infinite behaviour. The replication operator neatly replaces the much more complicated mechanism of recursively-defined processes used in earlier presentations of the π-calculus. However, it is still helpful to have high-level syntax for recursively-defined processes when writing π-calculus examples. We therefore provide a derived form for such definitions:

Definition 2.5 (Process definitions)

$$\begin{array}{rl} & \texttt{def}\ X_1[\tilde{x}_1] = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ X_n[\tilde{x}_n] = P_n\ \texttt{in}\ Q \\ \doteq & (\nu\, X_1)\ldots(\nu\, X_n)(*X_1?[\tilde{x}_1].P_1 \mid \ldots \mid *X_n?[\tilde{x}_n].P_n \mid Q) \end{array}$$

For each process definition $X_i[\tilde{x}_i] = P_i$ we create a new channel X_i and a replicated process $*X_i?[\tilde{x}_i].P_i$. The process $X_i?[\tilde{x}_i].P_i$ waits for a tuple of arguments to be sent on the channel X_i and then runs P_i with the arguments substituted for the formal parameters $\tilde{x}_i$. The replication operator enables X_i to be called arbitrarily often by providing an arbitrary number of copies of $X_i?[\tilde{x}_i].P_i$.

We can invoke the process definition X_i by simply sending the desired arguments along the channel X_i. For example, the following process repeatedly outputs b along y:

$$\texttt{def}\ X[x,a] = x![a].X![x,a]\ \texttt{in}\ X![y,b]$$

We can illustrate the behaviour of X by expanding the derived form:

$$\begin{array}{rl} & \texttt{def}\ X[x,a] = x![a].X![x,a]\ \texttt{in}\ X![y,b] \\ \doteq & (\nu\, X)(\,*X?[x,a].x![a].X![x,a] \mid \qquad X![y,b]\,) \\ \equiv & (\nu\, X)(\,*X?[x,a].x![a].X![x,a] \mid X?[x,a].x![a].X![x,a] \mid X![y,b]\,) \\ \rightarrow & (\nu\, X)(\,*X?[x,a].x![a].X![x,a] \mid \qquad y![b].X![y,b] \qquad) \\ \doteq & \texttt{def}\ X[x,a] = x![a].X![x,a]\ \texttt{in}\ y![b].X![y,b] \end{array}$$

A simple example of a mutually-recursive process definition is given below (the example repeatedly waits for a value on p and then retransmits it on q). We expand the derived form to illustrate how the scoping of the *Get* and *Put* channels allows mutually-recursive calls (both channels are in scope in the bodies of *Get* and *Put*, and in the process $Get![p,q]$).

$$\begin{array}{lcl} & & (\nu\, Get)(\nu\, Put)(\\ \texttt{def}\ Get[i,o] = i?[x].Put![i,o,x] & & \quad *Get?[i,o].i?[x].Put![i,o,x] \mid \\ \texttt{and}\ Put[i,o,x] = o![x].Get![i,o] & \doteq & \quad *Put?[i,o,x].o![x].Get![i,o] \mid \\ \texttt{in}\ Get![p,q] & & \quad Get![p,q] \\ & &) \end{array}$$

2.7 Returning results

It is very common for a process to behave in a 'functional' manner: accepting a number of arguments, doing some computation and then returning a result. In the π-calculus, it is necessary to return such results by means of an explicit communication, since processes do not have implicit results.

By convention, we write such 'functional' processes in the following form, using the last parameter as a result channel.

$$f?[\tilde{x}, r]. \ldots r![\mathit{results}] \ldots$$

Such processes are frequently replicated, to enable multiple calls to be executed, in which case we can write them as a process definition:

$$\texttt{def}\ f[\tilde{x}, r] = \ldots r![\mathit{results}] \ldots$$

This style of programming is very reminiscent of continuation-passing style, since r can be though of as the current continuation into which f inserts its result.

Although we don't need any special syntax to help define processes which return results, it is, however, convenient to have a derived form for getting results back from process definitions such as f:

Definition 2.6 (Getting results from processes)

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

The above definition calls f by creating a new channel r (a new channel is necessary to avoid interference), and sending it to f along with the arguments $\tilde{a}$. It then waits for the results to be sent back along r, continuing as P, with the results substituted for the bound variables $\tilde{x}$. (The fact that all communication is polyadic means that we can conveniently support calls which not only require multiple arguments, but which return multiple results.)

We omit the `in` and `let` keywords in nested `let` expressions, so that we need only write

$$\texttt{let}\ \tilde{x} = f(\ldots)\ \tilde{y} = g(\ldots)\ \texttt{in}\ P$$

instead of

$$\texttt{let}\ \tilde{x} = f(\ldots)\ \texttt{in}\ \texttt{let}\ \tilde{y} = g(\ldots)\ \texttt{in}\ P$$

Consider the process definition g below, whose last action is to get some result from f, and return it along r.

$$\texttt{def}\ g[\tilde{x}, r] = \texttt{let}\ \tilde{y} = f(\tilde{x})\ \texttt{in}\ r![\tilde{y}]$$

We often simplify processes such as g by making a *tail-call* to f:

$$\texttt{def}\ g[\tilde{x}, r] = f![\tilde{x}, r]$$

Now f returns its result along r directly, rather than returning it indirectly via g. We use tail-calls here to simplify our examples, but they do have a useful effect in practice, since they both save creating an extra result channel and avoid a communication in g.

2.8 Booleans

In [Mil91a] Milner demonstrated how data structures could be encoded in the π-calculus. For example, we can define the booleans *True* and *False* as below:

$$\begin{array}{l} \texttt{def}\ \mathit{True}[r] = (\nu\, b)(\, r![b] \;\mid\; *b?[t, f].t![]\,) \\ \texttt{def}\ \mathit{False}[r] = (\nu\, b)(\, r![b] \;\mid\; *b?[t, f].f![]\,) \end{array}$$

True and *False* do not take any parameters, other than a result channel r. They both create a new channel b, which serves as the location of the boolean value and return b along the result channel r. That is not all, however, since *True* and *False* both start a replicated process, whose purpose is to answer queries about the boolean b. The process must be replicated if we want to query a boolean more than once (omitting the replication would yield a *linear* boolean).

We can implement conditionals as below. Suppose that b is the location of a boolean value. If we send a pair of private channels $[t, f]$ along b, the boolean will reply using t, if it is true, or f, if it is false. Thus, P proceeds if b is true and Q proceeds if b is false.

Definition 2.7 (Conditionals)

$$\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q \doteq (\nu\, t)(\nu\, f)(b![t,f].(t?[].P + f?[].Q)) \qquad t,f \notin fv(P,Q)$$

It is now easy to implement the conjunction, disjunction and negation operations on booleans. Consider, for example, the behaviour of *And*, the conjunction operation. If b_1 is true it simply returns b_2 along the result channel r, otherwise it instructs *False* to create a boolean and return it along r.

$$\begin{array}{l}
\texttt{def}\ And[b_1,b_2,r] = \texttt{if}\ b_1\ \texttt{then}\ r![b_2]\ \texttt{else}\ False![r] \\
\texttt{def}\ Or[b_1,b_2,r] = \texttt{if}\ b_1\ \texttt{then}\ True![r]\ \texttt{else}\ r![b_2] \\
\texttt{def}\ Not[b,r] = \texttt{if}\ b\ \texttt{then}\ False![r]\ \texttt{else}\ True![r]
\end{array}$$

2.9 Lists

Lists have two constructors: *Nil* and *Cons*. *Nil* doesn't take any parameters, apart from a result channel. *Cons* takes the head and tail of the list as parameters, plus a result channel.

$$\begin{array}{l}
\texttt{def}\ Nil[r] = (\nu\, l)(\, r![l] \mid *l?[n,c].n![]\,) \\
\texttt{def}\ Cons[hd,tl,r] = (\nu\, l)(\, r![l] \mid *l?[n,c].c![hd,tl]\,)
\end{array}$$

The behaviour of *Nil* is similar to that of *True*. It creates a new channel l, the location of the list, and returns it along r. It then creates a replicated process which responds to requests on l by signalling on n, the first component of the request.

The behaviour of *Cons* is slightly more complicated, since it does not signal on c using the trivial value, but sends a pair $[hd, tl]$. This enables a process interrogating the cons cell to further interrogate the head and tail of the list (assuming, of course, that hd is the location of some data structure and tl is the location of another list).

We define a list pattern-matching derived form below. It operates similarly to the derived form for conditionals, except that it binds the variables hd and tl to the head and tail of l (if l is a cons cell).

Definition 2.8 (List pattern-matching)

$$\begin{array}{ll} & \texttt{match}\ l\ \texttt{with}\ \mathit{Nil}\ \texttt{=>}\ P\ \texttt{and}\ \mathit{Cons}[\mathit{hd},\mathit{tl}]\ \texttt{=>}\ Q \\ \doteq & (\nu\, n)(\nu\, c)(l![n,c].(n?[].P + c?[\mathit{hd},\mathit{tl}].Q)) \qquad n,c \notin \mathit{fv}(P,Q) \end{array}$$

We can now use list pattern-matching to write the list concatenation procedure. The *Concat* procedure take two lists, l_1 and l_2, as arguments, plus a result channel r. It responds by sending the concatenation of l_1 and l_2 along r.

$$\begin{array}{l} \texttt{def}\ \mathit{Concat}[l_1,l_2,r] = \\ \quad \texttt{match}\ l_1\ \texttt{with}\ \mathit{Nil}\ \texttt{=>} \\ \quad\quad r![l_2] \\ \quad \texttt{and}\ \mathit{Cons}[\mathit{hd},\mathit{tl}]\ \texttt{=>} \\ \quad\quad \texttt{let}\ \mathit{rest} = \mathit{Concat}(\mathit{tl},l_2)\ \texttt{in}\ \mathit{Cons}![\mathit{hd},\mathit{rest},r] \end{array}$$

If l_1 is nil then *Concat* immediately sends l_2 along r. Otherwise, we recursively calculate the concatenation of *tl* and l_2, naming the result *rest*. We then cons *hd* onto *rest*, instructing *Cons* to return the resulting list along r.

2.10 Process-based reference cells

We can easily encode updatable data structures using processes. For example, the process $\mathit{Cell}![x,\mathit{read},\mathit{update}]$ represents a reference cell whose current contents is x (the *read* and *update* channels can be used to read or modify the contents of the reference cell).

$$\begin{array}{l} \texttt{def}\ \mathit{Cell}[x,\mathit{read},\mathit{update}] = \\ \quad\quad \mathit{read}![x].\mathit{Cell}![x,\mathit{read},\mathit{update}] + \mathit{update}?[n].\mathit{Cell}![n,\mathit{read},\mathit{update}] \end{array}$$

The summation operator ensures that *read* and *update* requests cannot be executed concurrently. We can therefore guarantee that once an update request has been accepted, all subsequent read requests will be answered with the updated contents of the cell.

The process definition *Ref*, given an initial value x and a result channel r, creates a new reference cell (by creating two new *read* and *write* channels, and starting a *Cell* process).

$$\texttt{def}\ \mathit{Ref}[x,r] = (\nu\, \mathit{read})(\nu\, \mathit{update})(r![\mathit{read},\mathit{update}]\ \mid\ \mathit{Cell}![x,\mathit{read},\mathit{update}])$$

2.11 Channel-based reference cells

An alternative style of reference cell, which doesn't use the summation operator, is shown below. Each reference cell is represented using a single channel, rather than a process. Given an initial value x and a result channel r, the *ChanRef* process builds a new reference cell by creating a new channel *ref* and (asynchronously) writing x on it.

$$\texttt{def}\ \mathit{ChanRef}[x,r] = (\nu\, \mathit{ref})(r![\mathit{ref}] \ \mid\ \mathit{ref}![x])$$

The *Read* process, given a reference cell *ref* and a result channel r, reads a value x from *ref* (the current contents of the reference cell) then immediately puts it back, sending x back to the client using r.

$$\texttt{def}\ \mathit{Read}[\mathit{ref},r] = \mathit{ref}?[x].(\mathit{ref}![x] \ \mid\ r![x])$$

Similarly, given a reference cell *ref*, a new value v and a result channel r, the *Update* process reads the current contents of the reference cell from *ref* and replaces it with v, signalling completion on r by sending the empty tuple.

$$\texttt{def}\ \mathit{Update}[\mathit{ref},v,r] = \mathit{ref}?[x].(\mathit{ref}![v] \ \mid\ r![])$$

The *ChanRef*, *Read* and *Update* operations all preserve the invariant that there is at most one active writer on the *ref* channel. The first action of both the *Read* and *Update* processes is to read a value from the channel *ref*. Successfully reading a value from *ref* therefore has the effect of temporarily blocking all other *Read* and *Update* operations (since there is at most one active writer on the channel *ref*). Thus, we avoid any interference between concurrent *Read* and *Update* operations.

Chapter 3

Process typing

The most useful type systems for programming languages are those which can be typechecked automatically (usually during compilation). The ML type system [Mil77, DM82] is a particularly good example of such a type system, since the programmer does not even have to write any type information – it is all inferred automatically by the type checker.

If we wish type checking, or type inference, to be performed during compilation, rather than during evaluation, then we cannot hope to calculate the exact behaviour of a program, since this will in general depend on the input data, which is not available at compilation time. We must therefore make a conservative approximation of the behaviour of a program, assuming, for example, that both the `then` and `else` clauses of a conditional expression are executed, rather than attempting to calculate exactly which clauses are executed.

There are also good pragmatic reasons to avoid complicated calculations in a programming language's type system, since programmers need to understand any type errors reported by the type system.

Computation in the π-calculus is based on communication over *channels*. We simplify our type system by making two important decisions:

1. The type of a channel remains constant throughout its lifetime.

2. We do not specify temporal properties of channels.

These decisions avoid the need to consider causal relationships between communications, and significantly simplify our type system.

This chapter presents our monomorphic π-calculus type system. We first define the syntax of types and typing contexts, and then present the typing rules for processes, showing how the typing rules behave on the examples and derived forms of Chapter 2. We then prove (by means of a subject-reduction theorem) that our type system guarantees freedom from runtime errors.

3.1 Types and type contexts

Definition 3.1 gives the syntax of types. We have just one type constructor (the channel type constructor) since channels are the only data that we have in the polyadic π-calculus. We also allow type variables (which will enable us to do type inference later).

Definition 3.1 (Types)

$$\begin{array}{rcll} \delta & ::= & \uparrow[\delta_1, \ldots, \delta_n] & \text{Channel type} \\ & & \alpha & \text{Type variable} \end{array}$$

We record the types of free variables in a type context. Type contexts are (possibly empty) sequences of bindings of the form $x_1 : \delta_1, \ldots, x_n : \delta_n$, where $x_1, \ldots, x_n$ must be distinct variables. We sometimes let $\tilde{x} : \tilde{\delta}$ denote the context $x_1 : \delta_1, \ldots, x_n : \delta_n$.

Definition 3.2 (Type contexts) $\quad \Delta \; ::= \; x_1 : \delta_1, \ldots, x_n : \delta_n$

We leave the 'distinct variable' condition implicit in our typing rules. For example, if we mention $\Delta, x : \delta$ in a rule, we are implicitly assuming that x is not already mentioned in Δ. The expression $\Delta(x)$ denotes the type associated with x in Δ, and is defined as below:

Definition 3.3 (Context lookup)

$$\begin{array}{rcll} (\Delta, x : \delta)(x) & \doteq & \delta & \\ (\Delta, y : \delta)(x) & \doteq & \Delta(x) & \text{if } x \neq y \\ ()(x) & \doteq & \mathit{undefined} & \end{array}$$

3.2 Typechecking processes

In the π-calculus, processes have no explicit results. We can only interact with a process by communicating with it. Therefore, our typing judgements for processes take the form $\Delta \vdash P$, where Δ is a typing context which gives the types of the free variables of P. We can read $\Delta \vdash P$ as asserting that P uses its free variables consistently with the types given in Δ.

The simplest π-calculus process is the nil process, $\mathbf{0}$. It cannot communicate at all and hence is consistent with any context:

$$\Delta \vdash \mathbf{0} \quad \text{Nil}$$

The output operator $c![x_1, \ldots, x_n].P$ sends the tuple $[x_1, \ldots, x_n]$ along the channel c. P is the process which continues after the communication has completed. The Output typing rule

$$\frac{\Delta(c) = \uparrow[\Delta(x_1), \ldots, \Delta(x_n)] \qquad \Delta \vdash P}{\Delta \vdash c![x_1, \ldots, x_n].P} \ \text{Output}$$

states that if, in a context Δ, c is a channel carrying a tuple of length n, whose components match the types of the values $x_1, \ldots, x_n$ we are sending, and P is a well-formed process in the same context Δ, then $c![x_1, \ldots, x_n].P$ is a well-formed process. Note that the output operator is not a name-binding operator, so P is expected to be well-formed in the *same* context Δ. A simple example of a well-typed output is given below (A and B are arbitrary types).

$$a : A, b : B, c : \uparrow[A, B] \vdash c![a, b].\mathbf{0}$$

The input operator $c?[x_1, \ldots, x_n].P$ receives a tuple of length n along the channel c, binding the components of the received tuple to $x_1, \ldots, x_n$ in P. The Input typing rule

$$\frac{\Delta(c) = \uparrow[\delta_1, \ldots, \delta_n] \qquad \Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash P}{\Delta \vdash c?[x_1, \ldots, x_n].P} \ \text{Input}$$

checks that c is a channel carrying a tuple of length n, and that P is well-formed in the context Δ extended with the types of the bound variables. Note that the bound variables $x_1, \ldots, x_n$ must be distinct from the variables already bound in Δ (it is always possible to satisfy this condition by α-converting the bound variables).

A simple instance of the Input typing rule is given below (where $\Delta \doteq a : A, b : B, c : \uparrow[A, B]$). Note that P is typechecked in the context $\Delta, x : A, y : B$, since x and y are bound to the first and second components of the tuple sent along c.

$$\frac{\Delta(c) = \uparrow[A, B] \qquad \Delta, x : A, y : B \vdash P}{\Delta \vdash c?[x, y].P} \text{ Input}$$

The typing rule for $P \mid Q$ must ensure that P and Q use their free variables in a consistent manner. We therefore require that P and Q are well-typed in the *same* context. This ensures that any channels which are used in both P and Q must have the same type.

$$\frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P \mid Q} \text{ Prl}$$

We can now check that our previous two examples can be run in parallel:

$$\frac{\dfrac{\vdots}{\Delta \vdash c![a, b].\mathbf{0}} \text{ Output} \qquad \dfrac{\vdots}{\Delta \vdash c?[x, y].P} \text{ Input}}{\Delta \vdash c![a, b].\mathbf{0} \mid c?[x, y].P} \text{ Prl}$$

The Prl rule clearly disallows ill-formed examples such as $c![a, b].P \mid c?[x].Q$, since the left-hand process requires that c has type $\uparrow[A, B]$ while the right-hand process requires c to have type $\uparrow[X]$, for some X.

The restriction operator $(\nu\, x)P$ introduces a new channel x in the scope of P. The typing rule for restriction therefore extends the context Δ by adding a type binding for x. This rule is particularly simple because there is only one type constructor: the channel type. If we also had some basic types, such as integers, we would need an extra restriction on the Res rule, to ensure that x is given a channel type.

$$\frac{\Delta, x : \delta \vdash P}{\Delta \vdash (\nu\, x)P} \text{ Res}$$

We can now use restriction to localise the channels a, b and c in our previous example:

$$\vdash (\nu\, a)(\nu\, b)(\nu\, c)(c![a, b].\mathbf{0} \mid c?[x, y].P)$$

Note that the above process is consistent with the empty context. In fact, we can show that it is consistent with *any* context. This makes sense because a closed process cannot communicate with the outside world, and therefore, if it is internally consistent, is able to execute safely in any context.

The summation operator ensures that only one of its operands will ever execute, but we cannot (in general) statically determine which one it will be. Therefore, we require that both P and Q are consistent with the same context. This ensures that any possible execution of $P + Q$ will behave correctly:

$$\frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P + Q}\ \text{Smt}$$

Our types do not attempt to describe how often a channel is used. Thus, since the replication operator, $*P$, only serves to make an arbitrary number of copies of P available, the consistency of $*P$ only depends on the consistency of P:

$$\frac{\Delta \vdash P}{\Delta \vdash *P}\ \text{Repl}$$

The typing rules for processes are summarised in Definition 3.4.

Definition 3.4 (Process typing rules)

$$\frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P \mid Q} \quad \text{Prl}$$

$$\frac{\Delta, x : \delta \vdash P}{\Delta \vdash (\nu\, x)P} \quad \text{Res}$$

$$\frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P + Q} \quad \text{Smt}$$

$$\frac{\Delta(c) = \uparrow[\delta_1, \ldots, \delta_n] \qquad \Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash P}{\Delta \vdash c?[x_1, \ldots, x_n].P} \quad \text{Input}$$

$$\frac{\Delta(c) = \uparrow[\Delta(x_1), \ldots, \Delta(x_n)] \qquad \Delta \vdash P}{\Delta \vdash c![x_1, \ldots, x_n].P} \quad \text{Output}$$

$$\frac{\Delta \vdash P}{\Delta \vdash *P} \quad \text{Repl}$$

$$\Delta \vdash \mathbf{0} \quad \text{Nil}$$

3.3 Derived rules for process definitions

Our process typing rules give rise to the following admissible rule for process definitions:

Definition 3.5 (Typing of process definitions)

$$\frac{\begin{array}{c}\Delta, X_1 : \uparrow[\tilde{\delta}_1], \ldots, X_n : \uparrow[\tilde{\delta}_n] \vdash Q \\ \Delta, X_1 : \uparrow[\tilde{\delta}_1], \ldots, X_n : \uparrow[\tilde{\delta}_n], \tilde{x}_i : \tilde{\delta}_i \vdash P_i \qquad 1 \leq i \leq n\end{array}}{\Delta \vdash \texttt{def}\ X_1[\tilde{x}_1] = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ X_n[\tilde{x}_n] = P_n\ \texttt{in}\ Q} \quad \text{Def}$$

If we expand out the derived form (see Definition 2.5 on page 22) for any process definition, we find that we can construct a complete proof of its well-typedness using only the premises of the above rule. Suppose the premises of the above rule are true. For each definition X_i we have, using the Input rule and the second assumption, that

$$\Delta, X_1 : \uparrow[\tilde{\delta}_1], \ldots, X_n : \uparrow[\tilde{\delta}_n] \vdash X_i?[\tilde{x}_i].P_i$$

Therefore, using the Repl rule, we have

$$\Delta, X_1 : \uparrow[\tilde{\delta}_1], \ldots, X_n : \uparrow[\tilde{\delta}_n] \vdash *X_i?[\tilde{x}_i].P_i$$

Using the Prl rule and the first assumption we then find that

$$\Delta, X_1 : \uparrow[\tilde{\delta}_1], \ldots, X_n : \uparrow[\tilde{\delta}_n] \vdash *X_1?[\tilde{x}_1].P_1 \mid \ldots \mid *X_n?[\tilde{x}_n].P_n \mid Q$$

Finally, we can use the Res rule to prove that

$$\Delta \vdash (\nu\, X_1) \ldots (\nu\, X_n)(*X_1?[\tilde{x}_1].P_1 \mid \ldots \mid *X_n?[\tilde{x}_n].P_n \mid Q)$$

as required.

3.4 Processes which return results

The process typing rules also give rise to the following admissible rule for `let`. (We introduce the type abbreviation $\Rightarrow$, since it clarifies which types are the arguments, and which are the results.)

Definition 3.6 (Typing `let`)

$$[\delta_1, \ldots, \delta_m] \Rightarrow [\gamma_1, \ldots, \gamma_n] \;\doteq\; \uparrow[\delta_1, \ldots, \delta_m, \uparrow[\gamma_1, \ldots, \gamma_n]]$$

$$\frac{\Delta(f) = [\Delta(a_1), \ldots, \Delta(a_m)] \Rightarrow [\delta_1, \ldots, \delta_n] \qquad \Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash P}{\Delta \vdash \texttt{let}\ x_1, \ldots, x_n = f(a_1, \ldots, a_m)\ \texttt{in}\ P}\ \text{Let}$$

It is easy to check that the above rule is admissible by expanding out the derived form for `let` (Definition 2.6 on page 23). Suppose the premises of the above rule are true. If $\Delta' = \Delta, r : \uparrow[\delta_1, \ldots, \delta_n]$ then we can use the weakening lemma (Lemma 3.8, which we prove later) to prove that $\Delta', x_1 : \delta_1, \ldots, x_n : \delta_n \vdash P$, since the side-condition in our derived form ensures that $r \notin fv(P)$. We annotate the bound variables in the expansion of `let` $x_1, \ldots, x_n = f(a_1, \ldots, a_m)$ `in` P, to show that the conclusion of the above rule is valid:

$$\Delta \vdash (\nu\, r : \uparrow[\delta_1, \ldots, \delta_n])(f![a_1, \ldots, a_m, r] \mid r?[x_1 : \delta_1, \ldots, x_n : \delta_n].P)$$

3.5 Booleans

We can now typecheck our boolean examples. Let *Bool* be the type $\uparrow[\uparrow[\,], \uparrow[\,]]$. We reproduce the definitions of *True* and *False* below, indicating how they are typed by annotating bound variables with types.

$$\texttt{def}\ \mathit{True}[r : \uparrow[\mathit{Bool}]] = (\nu\, b : \mathit{Bool})(\, r![b] \ \mid\ *b?[t : \uparrow[\,], f : \uparrow[\,]].t![]\,)$$
$$\texttt{def}\ \mathit{False}[r : \uparrow[\mathit{Bool}]] = (\nu\, b : \mathit{Bool})(\, r![b] \ \mid\ *b?[t : \uparrow[\,], f : \uparrow[\,]].f![]\,)$$

True and *False* both have the type $\uparrow[\uparrow[\mathit{Bool}]]$, which can also be written as $[\,] \Rightarrow [\mathit{Bool}]$, making it clear that *True* and *False* both take no arguments and return a boolean.

Definition 3.7 (Typing of conditionals)

$$\frac{\Delta(b) = \mathit{Bool} \qquad \Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash \texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q}\ \text{If}$$

It is easy to check that above rule is admissible by expanding out the derived form for `if` (Definition 2.7 on page 25). Suppose the premises of the above rule are true. If $\Delta' = \Delta, t : \uparrow[\,], f : \uparrow[\,]$ then we can use the weakening lemma (Lemma 3.8, which we prove later) to prove that $\Delta' \vdash P$ and $\Delta' \vdash Q$, since the side-condition in our derived form ensures that $t, f \notin \mathit{fv}(P, Q)$. We annotate the bound variables in the expansion of $\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q$, to show that the conclusion of the above rule is valid:

$$\Delta \vdash (\nu\, t : \uparrow[\,])(\nu\, f : \uparrow[\,])(b![t, f].(t?[].P + f?[].Q))$$

It is now easy to verify the types of the conjunction, disjunction and negation operations on booleans.

$$\texttt{def}\ \mathit{And}[b_1 : \mathit{Bool}, b_2 : \mathit{Bool}, r : \uparrow[\mathit{Bool}]] = \texttt{if}\ b_1\ \texttt{then}\ r![b_2]\ \texttt{else}\ \mathit{False}![r]$$
$$\texttt{def}\ \mathit{Or}[b_1 : \mathit{Bool}, b_2 : \mathit{Bool}, r : \uparrow[\mathit{Bool}]] = \texttt{if}\ b_1\ \texttt{then}\ \mathit{True}![r]\ \texttt{else}\ r![b_2]$$
$$\texttt{def}\ \mathit{Not}[b : \mathit{Bool}, r : \uparrow[\mathit{Bool}]] = \texttt{if}\ b\ \texttt{then}\ \mathit{False}![r]\ \texttt{else}\ \mathit{True}![r]$$

3.6 Process-based reference cells

A process-based reference cell can be represented as a pair of channels: the first channel can be used to read the contents of the cell, and the second can be used to update the contents of the cell.

The type of *Cell* is $\uparrow[X, \uparrow[X], \uparrow[X]]$, for arbitrary but fixed X. Both the *read* and *update* channels have type $\uparrow[X]$.

$$\begin{array}{l}\texttt{def}\ \mathit{Cell}[x : X, \mathit{read} : \uparrow[X], \mathit{update} : \uparrow[X]] = \\ \qquad \mathit{read}![x].\mathit{Cell}![x, \mathit{read}, \mathit{update}] + \mathit{update}?[n : X].\mathit{Cell}![n, \mathit{read}, \mathit{update}]\end{array}$$

The process definition *Ref* takes an initial value x and creates a new reference cell. It has type $[X] \Rightarrow [\uparrow[X], \uparrow[X]]$.

$$\begin{array}{l}\texttt{def}\ \mathit{Ref}[x : X, r : \uparrow[\uparrow[X], \uparrow[X]]] = \\ \qquad (\nu\ \mathit{read} : \uparrow[X])(\nu\ \mathit{update} : \uparrow[X])(r![\mathit{read}, \mathit{update}]\ \mid\ \mathit{Cell}![x, \mathit{read}, \mathit{update}])\end{array}$$

This example highlights two weaknesses in our simple π-calculus type system. Firstly, we are forced to choose a single type X in the definitions of *Cell* and *Ref*, even though they clearly operate uniformly over *all* types X. We will address this problem in Chapter 5. Secondly, the user of a reference cell should never write on the *read* channel, and never read from the *write* channel, but this restriction cannot be enforced by our type system. Fortunately, Pierce and Sangiorgi [PS93] have shown that it is possible to refine the channel type constructor $\uparrow$ so that input and output capabilities can be manipulated separately. The Pict programming language [PT95b] adopts Pierce and Sangiorgi's refinement, enabling it to give *Ref* the type $[X] \Rightarrow [?X, !X]$ (the type $?X$ allows only read access, while the type $!X$ allows only write access).

3.7 Channel-based reference cells

Let *ChanRef* X be the type $\uparrow[X]$, the type of a reference cell represented as a channel. The *ChanRef* process has type $[X] \Rightarrow [\mathit{ChanRef}\ X]$.

$$\texttt{def}\ \mathit{ChanRef}[x : X, r : \uparrow[\mathit{ChanRef}\ X]] = (\nu\ \mathit{ref} : \uparrow[X])(r![\mathit{ref}]\ \mid\ \mathit{ref}![x])$$

The *Read* process has type $[\mathit{ChanRef}\ X] \Rightarrow [X]$.

$$\texttt{def}\ \mathit{Read}[\mathit{ref} : \mathit{ChanRef}\ X, r : \uparrow[X]] = \mathit{ref}?[x].(\mathit{ref}![x] \mid r![x])$$

The *Update* process has type $[\mathit{ChanRef}\ X, X] \Rightarrow [\,]$.

$$\texttt{def}\ \mathit{Update}[\mathit{ref} : \mathit{ChanRef}\ X, v : X, r : \uparrow[\,]] = \mathit{ref}?[x : X].(\mathit{ref}![v] \mid r![])$$

This example again highlights two weaknesses in our simple π-calculus type system. Firstly, just as with process-based reference cells, we are forced to choose a single type X in the definitions of *ChanRef*, *Read* and *Update*, even though they clearly operate uniformly over *all* types X. We will address this problem in Chapter 5. Secondly, the *ChanRef*, *Read* and *Update* operations all preserve the invariant that there is at most one active writer on the *ref* channel, but we have no way of ensuring that *ChanRef*, *Read* and *Update* are the only processes which manipulate the *ref* channel. This is precisely what *abstract datatypes* are useful for, and in Chapter 5 we show that the polymorphic extension of our π-calculus type system is able to provide just such a mechanism.

3.8 Properties of well-typed π-terms

If $x \notin \mathit{fv}(P)$ then we can add a new type binding for x without invalidating the typing of P:

Lemma 3.8 (Weakening)

If $\Delta \vdash P$ and $x \notin \mathit{fv}(P)$ then $\Delta, x : \delta \vdash P$.

Proof A simple induction on the structure of P. □

Similarly, if $x \notin \mathit{fv}(P)$ then we can remove x's type binding without invalidating the typing of P:

Lemma 3.9 (Strengthening)

If $\Delta, x : \delta \vdash P$ and $x \notin \mathit{fv}(P)$ then $\Delta \vdash P$.

Proof A simple induction on the structure of P. □

If each x_i and y_i have the same type in the context Δ then we can simultaneously substitute $y_1, \ldots, y_n$ for $x_1, \ldots, x_n$ while preserving the type of P:

Lemma 3.10 (Substitution)

If $\Delta \vdash P$ and $\Delta(x_i) = \Delta(y_i)$ for $1 \leq i \leq n$ then $\Delta \vdash \{y_1, \ldots, y_n/x_1, \ldots, x_n\}P$.

Proof A simple induction on the structure of P. □

3.9 Properties of structural congruence

In Definition 2.3 (on page 17), the structural congruence relation is defined as the least congruence relation which satisfies a given set of axioms. In fact, we can be more explicit and say that $\equiv$ is defined as the least relation satisfying the axioms given in Definition 2.3, plus the four rules below (defining $\equiv$ in this way allows us to use induction on the depth of the derivation of $P \equiv Q$ in proofs).

$$\frac{}{P \equiv P}\ \text{Refl} \qquad \frac{P \equiv Q}{Q \equiv P}\ \text{Sym}$$

$$\frac{P \equiv Q \qquad Q \equiv R}{P \equiv R}\ \text{Trans} \qquad \frac{P \equiv Q}{C[P] \equiv C[Q]}\ \text{Cong}$$

C denotes a process context (a process containing a 'hole'):

$$\begin{array}{rcl} C & ::= & [\,] \mid (\nu\, x)C \mid (C \mid P) \mid (P \mid C) \mid (C + P) \mid (P + C) \mid \\ & & c?[x_1, \ldots, x_n].C \mid c![x_1, \ldots, x_n].C \mid *C \end{array}$$

The structural congruence relation captures most of the runtime behaviour of the restriction operator. An important lemma, therefore, is that types are preserved under structural congruence. (Only part 1 of the lemma is actually necessary for our type soundness result, but part 2 is essential if we wish to use induction to prove that the Sym rule preserves the type of a process).

Lemma 3.11 (Types are preserved under structural congruence)

1) If $\Delta \vdash P$ and $P \equiv Q$ then $\Delta \vdash Q$.

2) If $\Delta \vdash Q$ and $P \equiv Q$ then $\Delta \vdash P$.

Proof We prove both parts simultaneously, using induction on the depth of the inference of $P \equiv Q$. We omit the cases involving the summation operator, since they are similar to the parallel composition cases.

case $P \equiv P \mid \mathbf{0}$

Part 1. We have, by assumption, that $\Delta \vdash P$. Therefore, using the Nil and Prl rules, we have $\Delta \vdash P \mid \mathbf{0}$ as required. Part 2 is easy.

case $P \mid Q \equiv Q \mid P$

Part 1. Clearly, if $\Delta \vdash P \mid Q$ then $\Delta \vdash P$ and $\Delta \vdash Q$ and the result follows using the Prl rule. Part 2 is similar.

case $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

Part 1. Clearly, if $\Delta \vdash (P \mid Q) \mid R$ then $\Delta \vdash P$, $\Delta \vdash Q$ and $\Delta \vdash R$. The result follows after two applications of the Prl rule. Part 2 is similar.

case $*P \equiv P \mid *P$

Part 1. If $\Delta \vdash *P$ then it must be that $\Delta \vdash P$. Therefore, using the Prl rule, we have $\Delta \vdash P \mid *P$ as required. Part 2 is easy.

case $(\nu\, x)P \mid Q \equiv (\nu\, x)(P \mid Q)$ where $x \notin fv(Q)$

Part 1. If $\Delta \vdash (\nu\, x)P \mid Q$ then it must be that $\Delta \vdash Q$ and $\Delta, x : \delta \vdash P$ for some δ. We can therefore use weakening (Lemma 3.8) to prove that $\Delta, x : \delta \vdash Q$, since $x \notin fv(Q)$. The result follows using the Prl and Res rules.

Part 2. If $\Delta \vdash (\nu\, x)(P \mid Q)$ then it must be that $\Delta, x : \delta \vdash Q$ and $\Delta, x : \delta \vdash P$ for some δ. We can therefore use strengthening (Lemma 3.9) to prove that $\Delta \vdash Q$, since $x \notin fv(Q)$. The result follows using the Prl and Res rules.

case $P \equiv P$

Immediate.

case $P \equiv Q$ where $Q \equiv P$

Part 1. We have by induction (Part 2) that $\Delta \vdash Q$ as required. Part 2 is similar.

case $P \equiv R$ where $P \equiv Q$ and $Q \equiv R$

Part 1. We have, by induction that $\Delta \vdash Q$ so, using induction again, we have that $\Delta \vdash R$ as required. Part 2 is similar.

case $C[P] \equiv C[Q]$ where $P \equiv Q$

A simple sub-induction on the structure of C proves the result. $\square$

Note that it is very important that none of the structural congruence rules delete or create any sub-terms. If we allowed such rules, types would not be preserved under structural congruence. For example, when read from left to right, the 'garbage collection' rule $(\nu\, x)(x![\tilde{a}].P) \equiv \mathbf{0}$ makes perfect sense (since the output $x![\tilde{a}].P$ can never succeed). However, when read from right to left, this rule allows us to 'magically' create an arbitrary term P which, in particular, may not be well-typed.

3.10 Type soundness

We are now able to prove one of our main soundness theorems: well-typed processes can never fail.

Theorem 3.12 (Well-typed processes never fail)

If $\Delta \vdash P$ then not (P *fails*).

Proof Suppose $\Delta \vdash P$ and P *fails*. We use induction on the depth of the inference of P *fails* to show a contradiction for all possible types of failure P can encounter:

case $P \mid Q$ *fails* since P *fails*

We have, by assumption, that $\Delta \vdash P$, so we can use induction to prove that not (P *fails*) and we have a contradiction, as required.

case $(\nu\, x)P$ *fails* since P *fails*

We have, by assumption, that $\Delta \vdash P$, so we can use induction to prove that not (P *fails*) and we have a contradiction, as required.

case P *fails* since $P \equiv Q$ and Q *fails*

We have, by assumption, that $\Delta \vdash P$, so we can use Lemma 3.11 to prove that $\Delta \vdash Q$. We can therefore use induction to prove that not (Q *fails*) and we have a contradiction, as required.

case $(P + c?[x_1, \ldots, x_m].Q) \mid (c![y_1, \ldots, y_n].R + S)$ *fails* since $m \neq n$

We have, by assumption, that $\Delta \vdash c?[x_1, \ldots, x_m].Q$ and $\Delta \vdash c![y_1, \ldots, y_n].R$. The type of c is clearly the same in both the inputting and outputting processes, so $m = n$ and we have a contradiction, as required. □

Since the definition of process failure only detects the immediate failure of a process, a subject-reduction theorem is required to prove that well-typed processes remains well-typed after a successful reduction step. A corollary of Theorems 3.12 and 3.13 is that a well-typed process cannot fail after any number of reduction steps.

Theorem 3.13 (Subject reduction)

If $\Delta \vdash P$ and $P \rightarrow Q$ then $\Delta \vdash Q$.

Proof We prove the result by induction on the depth of the inference of $P \rightarrow Q$.

case $P \mid Q \to P \mid R$ where $Q \to R$

It must be the case that $\Delta \vdash P$ and $\Delta \vdash Q$. We can therefore use induction to prove that $\Delta \vdash R$. The result follows using the Prl rule.

case $(\nu\, x)P \to (\nu\, x)Q$ where $P \to Q$

It must be the case that $\Delta, x : \delta \vdash P$. We can therefore use induction to prove that $\Delta, x : \delta \vdash Q$. The result follows using the Res rule.

case $(P{+}c?[x_1, \ldots, x_n].Q) \mid (c![y_1, \ldots, y_n].R{+}S) \to \{y_1, \ldots, y_n/x_1, \ldots, x_n\}Q \mid R$

We have, by assumption, that $\Delta \vdash c?[x_1, \ldots, x_n].Q$ and $\Delta \vdash c![y_1, \ldots, y_n].R$. Therefore, it must be the case that $\Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash Q$ and $\Delta(c) = \updownarrow[\delta_1, \ldots, \delta_n]$. However, we also have that $\Delta(c) = \updownarrow[\Delta(y_1), \ldots, \Delta(y_n)]$. We can therefore use our substitution lemma (Lemma 3.10) to prove that $\Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash \{y_1, \ldots, y_n/x_1, \ldots, x_n\}Q$. We have, using our strengthening lemma (Lemma 3.9), that $\Delta \vdash \{y_1, \ldots, y_n/x_1, \ldots, x_n\}Q$ (since $x_1, \ldots, x_n$ are not free in the substituted process). We already have, by assumption, that $\Delta \vdash R$, so the result follows using the Prl rule.

case $P \to Q$ where $P \equiv P'$, $P' \to Q'$ and $Q' \equiv Q$

We have, by assumption, that $\Delta \vdash P$. Therefore, using Lemma 3.11, we have that $\Delta \vdash P'$. Using induction we have that $\Delta \vdash Q'$ and the result follows by using Lemma 3.11 again. $\square$

3.11 Type inference

The benefits of automatic type inference have been clearly demonstrated in languages such as Standard ML [MTH90] and Haskell [HJW$^+$92], where the programmer has to write only a minimum of explicit type information. Using similar, unification based, techniques we now show that it is possible to automatically infer types for π-terms.

3.11.1 Substitutions

A substitution is a finite map from type variables to types. We let $\mathbf{dom}\,\sigma$ denote the domain of σ. A substitution naturally extends to an operation on both types and contexts, defined as below.

Definition 3.14 (Substitutions)

$$\sigma, \rho, \mu \quad ::= \quad \{\alpha_1 \mapsto \delta_1, \ldots, \alpha_n \mapsto \delta_n\}$$

$$\begin{array}{rcl} \sigma{\uparrow}[\delta_1, \ldots, \delta_n] & \doteq & {\uparrow}[\sigma\delta_1, \ldots, \sigma\delta_n] \\ \sigma\alpha & \doteq & \begin{cases} \sigma\alpha & \alpha \in \mathbf{dom}\,\sigma \\ \alpha & \text{otherwise} \end{cases} \\ \\ \sigma(x_1 : \delta_1, \ldots, x_n : \delta_n) & \doteq & x_1 : \sigma\delta_1, \ldots, x_n : \sigma\delta_n \end{array}$$

During type inference, it is necessary to compose substitutions. The composition of ρ and σ, written $\rho\sigma$, is defined below:

Definition 3.15 (Composition of substitutions)

$$\rho\sigma \quad \doteq \quad \{\alpha \mapsto \rho(\sigma\alpha) \mid \alpha \in (\mathbf{dom}\,\rho \cup \mathbf{dom}\,\sigma)\}$$

Lemma 3.16 (Simple properties of substitutions)

(1) $\{\}\sigma = \sigma\{\} = \sigma$ (2) $\rho(\sigma\mu) = (\rho\sigma)\mu$
(3) $\rho(\sigma\delta) = (\rho\sigma)\delta$ (4) $\rho(\sigma\Delta) = (\rho\sigma)\Delta$

Proof Straightforward from the definitions of substitution and composition. □

An important fact we prove about substitution is that typing judgements are closed under substitution. This fact is crucial for type inference, since our algorithm must be able to apply substitutions to the typing context without invalidating the types of π-terms it has already checked.

Lemma 3.17 (Preservation of process types under substitution)

If $\Delta \vdash P$ then $\sigma\Delta \vdash P$.

Proof A simple induction on the structure of P. □

3.11.2 Unification

Since π-calculus types are simple trees, we know from Robinson's work [Rob65] that there is a sound and complete unification algorithm for π-calculus types, which we refer to as *Unify*. Propositions 3.18 and 3.19 state the appropriate soundness and completeness properties of the algorithm.

Proposition 3.18 (Soundness of unification algorithm [Rob65])

If $\mathit{Unify}(\delta, \gamma) = \sigma$ then $\sigma\delta = \sigma\gamma$.

Proposition 3.19 (Completeness of unification algorithm [Rob65])

If $\sigma\delta = \sigma\gamma$ then $\mathit{Unify}(\delta, \gamma)$ succeeds, returning ρ, and there exists a μ such that $\sigma = \mu\rho$. Otherwise, *Unify* fails.

3.11.3 Inference algorithm

In Definition 3.20 we give an algorithm, X, which takes a type context Δ and a process P as arguments, and either fails (if no valid typing exists), or returns the minimal substitution ρ such that $\rho\Delta \vdash P$.

Definition 3.20 (Inference algorithm)

case $X(\Delta; P \mid Q)$

If $X(\Delta; P) = \rho$ and $X(\rho\Delta; Q) = \rho'$ then return $\rho'\rho$ else fail.

case $X(\Delta; P + Q)$

If $X(\Delta; P) = \rho$ and $X(\rho\Delta; Q) = \rho'$ then return $\rho'\rho$ else fail.

case $X(\Delta; (\nu\, x)P)$

If α is a fresh type variable and $X(\Delta, x : \alpha; P) = \rho$ then return ρ else fail.

case $X(\Delta; c?[x_1, \ldots, x_n].P)$

If $\alpha_1, \ldots, \alpha_n$ are fresh type variables, $\Delta(c) = \delta$, $\mathit{Unify}(\delta, \uparrow[\alpha_1, \ldots, \alpha_n]) = \rho$ and $X(\rho(\Delta, x_1 : \alpha_1, \ldots, x_n : \alpha_n); P) = \rho'$ then return $\rho'\rho$ else fail.

case $X(\Delta; c![x_1, \ldots, x_n].P)$

If $\Delta(c) = \delta$, $\Delta(x_i) = \delta_i$ for $1 \leq i \leq n$, $\mathit{Unify}(\delta, \uparrow[\delta_1, \ldots, \delta_n]) = \rho$ and $X(\rho\Delta; P) = \rho'$ then return $\rho'\rho$ else fail.

case $X(\Delta; *P)$

If $X(\Delta; P) = \rho$ then return ρ else fail.

case $X(\Delta; \mathbf{0})$

Return $\{\}$ (the empty substitution).

We do not formalise how the algorithm picks 'fresh' type variables. In the following proofs we will assume that whenever a type variable is declared to be ‘fresh’ it is distinct from any type variables mentioned either in the current context or in types which have already been computed. In practice this condition can easily be satisfied by using a global counter to number new type variables.

The soundness of our inference algorithm is demonstrated by the following theorem:

Theorem 3.21 (Soundness of inference algorithm)

If $X(\Delta; P) = \rho$ then $\rho\Delta \vdash P$.

Proof We proceed by induction on the structure of P.

case $X(\Delta; P \mid Q) = \rho'\rho$

We have $X(\Delta; P) = \rho$ and so by induction $\rho\Delta \vdash P$. We also have that $X(\rho\Delta; Q) = \rho'$ and so by induction $\rho'\rho\Delta \vdash Q$. Now, by Lemma 3.17 we have that $\rho'\rho\Delta \vdash P$ and so using the Prl rule we have $\rho'\rho\Delta \vdash P \mid Q$ as required.

case $X(\Delta; P + Q) = \rho'\rho$

As above.

case $X(\Delta; (\nu\, x)P) = \rho$

We have $X(\Delta, x : \alpha; P) = \rho$ where α is fresh. Hence, by induction, $\rho\Delta, x : \rho\alpha \vdash P$ and we can conclude, using the Res rule that $\rho\Delta \vdash (\nu\, x)P$ as required.

case $X(\Delta; c?[x_1, \ldots, x_n].P) = \rho'\rho$

We have that $\Delta(c) = \delta$ and $\mathit{Unify}(\delta, \uparrow[\alpha_1, \ldots, \alpha_n]) = \rho$ where $\alpha_1, \ldots, \alpha_n$ are fresh. We also have that $X(\rho(\Delta, x_1 : \alpha_1, \ldots, x_n : \alpha_n); P) = \rho'$. By induction, we have that $\rho'\rho\Delta, x_1 : \rho'\rho\alpha_1, \ldots, x_n : \rho'\rho\alpha_n \vdash P$. Using Proposition 3.18 we find that $\rho\delta = \rho\uparrow[\alpha_1, \ldots, \alpha_n]$ which clearly implies that $\rho'\rho\delta = \uparrow[\rho'\rho\alpha_1, \ldots, \rho'\rho\alpha_n]$. We can therefore apply the Input rule to prove that $\rho'\rho\Delta \vdash c?[x_1, \ldots, x_n].P$ as required.

case $X(\Delta; c![x_1, \ldots, x_n].P) = \rho'\rho$

We have that $\Delta(c) = \delta$, $\Delta(x_i) = \delta_i$ for $1 \leq i \leq n$ and $\mathit{Unify}(\delta, \uparrow[\delta_1, \ldots, \delta_n]) = \rho$. We also have that $X(\rho\Delta; P) = \rho'$. By induction we have that $\rho'\rho\Delta \vdash P$. Using Proposition 3.18 we find that $\rho\delta = \rho\uparrow[\delta_1, \ldots, \delta_n]$ which clearly implies that $(\rho'\rho\Delta)(c) = \rho'\rho\delta = \rho'\rho\uparrow[\delta_1, \ldots, \delta_n] = \uparrow[(\rho'\rho\Delta)(x_1), \ldots, (\rho'\rho\Delta)(x_n)]$ and the result follows using the Output rule.

case $X(\Delta; *P) = \rho$

We have $X(\Delta)(P) = \rho$ and so by induction $\rho\Delta \vdash P$. Using the Repl rule $\rho\Delta \vdash *P$ as required.

case $X(\Delta; \mathbf{0}) = \{\}$

Immediate, since $\{\}\Delta \doteq \Delta$ and $\Delta \vdash \mathbf{0}$ using the Nil rule. $\square$

In the following proof, we sometimes need to take the union of two substitutions. But $\rho \cup \sigma$ is only well-defined when the domains of ρ and σ are disjoint, so we introduce an overwrite operation $\rho \oplus \sigma$ which combines arbitrary ρ and σ (the behaviour of σ takes precedence over the behaviour of ρ for any α which is in the domain of both ρ and σ):

Definition 3.22 (Union of substitutions)

$$\rho \oplus \sigma \quad \doteq \quad \begin{array}{l} \{\alpha \mapsto \rho(\alpha) \mid \alpha \in \mathbf{dom}\,\rho, \alpha \notin \mathbf{dom}\,\sigma\} \cup \\ \{\alpha \mapsto \sigma(\alpha) \mid \alpha \in \mathbf{dom}\,\sigma\} \end{array}$$

The following theorem demonstrates that our inference algorithm returns a principal substitution (if one exists).

Theorem 3.23 (Completeness of inference algorithm)
If $\sigma\Delta \vdash P$ then $X(\Delta; P)$ succeeds, returning ρ, and there exists a μ such that $\sigma\Delta = \mu\rho\Delta$.

Proof We proceed by induction on the structure of P.

case $\sigma\Delta \vdash P \mid Q$ where $\sigma\Delta \vdash P$ and $\sigma\Delta \vdash Q$

Using induction, we find that $X(\Delta; P)$ succeeds, returning ρ, and there exists a μ such that $\sigma\Delta = \mu\rho\Delta$. Hence, by induction, $X(\rho\Delta; Q)$ succeeds, returning ρ' and there exists a μ' such that $\mu\rho\Delta = \mu'\rho'\rho\Delta$. Therefore $X(\Delta; P \mid Q)$ succeeds, returning $\rho'\rho$, where $\sigma\Delta = \mu'(\rho'\rho)\Delta$ as required.

case $\sigma\Delta \vdash P + Q$ where $\sigma\Delta \vdash P$ and $\sigma\Delta \vdash Q$

As above.

case $\sigma\Delta \vdash (\nu\, x)P$ where $\sigma\Delta, x : \delta \vdash P$

Let α be a fresh type variable and $\sigma' = \sigma \oplus \{\alpha \mapsto \delta\}$. Since α is fresh we have that $\alpha \notin \Delta$ and $\sigma'\Delta = \sigma\Delta$. Therefore, $\sigma'(\Delta, x : \alpha) = \sigma\Delta, x : \delta$ so, by induction, $X(\Delta, x : \alpha; P)$ succeeds, returning ρ, and there exists a μ such that $\sigma'(\Delta, x : \alpha) = \mu\rho(\Delta, x : \alpha)$. This implies that $\sigma'\Delta = \mu\rho\Delta$. Therefore, $X(\Delta; (\nu\, x)P)$ succeeds, returning ρ and $\sigma\Delta = \sigma'\Delta = \mu\rho\Delta$ as required.

case $\sigma\Delta \vdash c?[x_1,\ldots,x_n].P$ where $(\sigma\Delta)(c) = \uparrow[\delta_1,\ldots,\delta_n]$ and $\sigma\Delta, x_1 : \delta_1,\ldots,x_n : \delta_n \vdash P$

Let $\alpha_1,\ldots,\alpha_n$ be fresh type variables and $\sigma' = \sigma \oplus \{\alpha_1 \mapsto \delta_1,\ldots,\alpha_n \mapsto \delta_n\}$. Since $\alpha_1,\ldots,\alpha_n$ are fresh we have that $\alpha_1,\ldots,\alpha_n \notin \Delta$ and $\sigma'\Delta = \sigma\Delta$. We also have that $\Delta(c) = \delta$, for some δ, where again $\alpha_1,\ldots,\alpha_n \notin \delta$. Thus, $\sigma'\delta = \sigma\delta = \uparrow[\delta_1,\ldots,\delta_n] = \sigma'\uparrow[\alpha_1,\ldots,\alpha_n]$. We can use Proposition 3.19 to show that *Unify*$(\delta, \uparrow[\alpha_1,\ldots,\alpha_n])$ succeeds, returning ρ, and there exists a μ such that $\sigma' = \mu\rho$. Using induction, we have that $X(\rho(\Delta, x_1 : \alpha_1,\ldots,x_n : \alpha_n); P)$ succeeds, returning ρ', and there exists a μ' such that $\mu\rho(\Delta, x_1 : \alpha_1,\ldots,x : n : \alpha_n) = \mu'\rho'\rho(\Delta, x_1 : \alpha_1,\ldots,x : n : \alpha_n)$. Therefore, $X(\Delta; c?[x_1,\ldots,x_n].P)$ succeeds, returning $\rho'\rho$ where $\sigma\Delta = \sigma'\Delta = \mu'(\rho'\rho)\Delta$ as required.

case $\sigma\Delta \vdash c![x_1,\ldots,x_n].P$ where $(\sigma\Delta)(c) = \uparrow[(\sigma\Delta)(a_1),\ldots,(\sigma\Delta)(a_n)]$ and $\sigma\Delta \vdash P$

It must be the case that $\Delta(c) = \delta$ and $\Delta(x_i) = \delta_i$ for $1 \leq i \leq n$ for some δ and $\delta_1,\ldots,\delta_n$ such that $\sigma(\Delta(c)) = \sigma(\uparrow[\delta_1,\ldots,\delta_n])$. We can therefore use Proposition 3.19 to show that *Unify*$(\delta, \uparrow[\delta_1,\ldots,\delta_n])$ succeeds, returning ρ, and there exists a μ such that $\sigma = \mu\rho$. Using induction, we have that $X(\rho\Delta; P)$ succeeds, returning ρ' and there exists a μ' such that $\mu\rho\Delta = \mu'\rho'\rho\Delta$. Therefore, $X(\Delta; c![x_1,\ldots,x_n].P)$ succeeds, returning $\rho'\rho$ where $\sigma\Delta = \mu'(\rho'\rho)\Delta$ as required.

case $\sigma\Delta \vdash *P$ where $\sigma\Delta \vdash P$

We have, by induction, that $X(\Delta; P)$ succeeds, returning ρ and there exists a μ such that $\sigma\Delta = \mu\rho\Delta$. The result is immediate since $X(\Delta; *P)$ succeeds, returning ρ.

case $\sigma\Delta \vdash \mathbf{0}$

We have that $X(\Delta; \mathbf{0})$ succeeds, returning $\{\}$ and the result follows by taking $\mu = \sigma$. □

Chapter 4

Recursive types

There are many useful programs which cannot be assigned a type in the simply-typed λ-calculus. Similarly, there are many useful π-calculus programs which cannot be assigned a type in our simple type system. An important deficiency is that we cannot support programming with recursive datatypes (for example, lists or trees). We now present a simple solution to this problem: recursive types. In fact, as in the λ-calculus, recursive types make the typed π-calculus as expressive as the untyped monadic π-calculus, since we can assign a type to every monadic π-term.

4.1 Type syntax

We extend our syntax of types with recursive types of the form $\mu\alpha.\delta$ (which bind the type variable α with scope δ).

Definition 4.1 (Recursive types)

δ	$::=$	α	Type variable
		$\uparrow[\delta_1, \ldots, \delta_n]$	Channel type
		$\mu\alpha.\delta$	Recursive type

There are now at least two possible ways to proceed: allow implicit folding and unfolding of recursive types (cf. Cardone and Coppo [CC91] or Amadio and Cardelli [AC91], for example), or require explicit annotations from the programmer (cf. MacQueen, Plotkin and Sethi [MPS86], for example). We choose the former, since it requires the minimum of changes to our typing rules and operational semantics.

The simplest way to allow implicit folding and unfolding of recursive types is to modify the definition of type equality so that it is insensitive to such operations. This is commonly achieved by considering the type $\mu\alpha.\delta$ as a finite specification of an infinite tree (obtained by repeatedly applying the rule $\mu\alpha.\delta = \{\mu\alpha.\delta/\alpha\}\delta$). With such an interpretation, we say that $\delta = \gamma$ whenever δ and γ denote the same infinite tree. We take a more direct approach here, defining equality by means of a bisimulation relation on types.

Our treatment of recursive types was inspired by Pierce and Sangiorgi's formalisation of subtyping for recursive types [PS93]. We refine their work by eliminating all uses of infinite trees: Pierce and Sangiorgi use a bisimulation relation defined over infinite trees, while we use a bisimulation relation defined directly over the syntax of types. We believe that by eliminating all uses of infinite trees from our presentation, we get a simpler treatment of recursive types, and a more direct proof of correctness for our type equality algorithm.

We now define what it means for two types to be bisimilar. Intuitively, two types are bisimilar if we cannot distinguish their type structure. The relation $\delta \Downarrow \gamma$ formalises the observations we can make of a type δ:

Definition 4.2 (Observation)

$$\alpha \Downarrow \alpha \qquad\qquad \uparrow[\delta_1, \ldots, \delta_n] \Downarrow \uparrow[\delta_1, \ldots, \delta_n] \qquad\qquad \frac{\{\mu\alpha.\delta/\alpha\}\delta \Downarrow \gamma}{\mu\alpha.\delta \Downarrow \gamma}$$

We allow type variables and channel types to be observed directly. In order to make bisimulation insensitive to the folding and unfolding of recursive types, we prevent any direct observation of recursive types. Instead, we unfold the recursion and observe the structure of the unfolded type. This means, for example, that the type $\mu\alpha.\delta$ and its unfolding $\{\mu\alpha.\delta/\alpha\}\delta$ have exactly the same observable type structure, and will not be distinguished by our bisimulation relation.

We require that all recursive types $\mu\alpha.\delta$ be *contractive* in α: all occurrences of α must be inside at least one channel type constructor. This disallows types such as $\mu\alpha.\alpha$ and $\mu\alpha.\mu\beta.\alpha$, which have no observable type structure, and guarantees that for every type δ there exists a (unique) γ such that $\delta \Downarrow \gamma$.

Definition 4.3 (Bisimulation)

Let R range over relations between types. The relation R is a bisimulation if $R \subseteq F(R)$ where F is the following function on relations: $(\delta, \gamma) \in F(R)$ if either

1. $\delta \Downarrow \alpha$ and $\gamma \Downarrow \alpha$.
2. $\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, $\gamma \Downarrow \uparrow[\gamma_1, \ldots, \gamma_n]$ and $(\delta_i, \gamma_i) \in R$ for $1 \leq i \leq n$.

The function F in Definition 4.3 is monotone, so we have by Tarski's fixpoint theorem [Tar55] that the greatest fixpoint of F exists and is equal to $\bigcup\{R \mid R \subseteq F(R)\}$. Let $\simeq$ be the greatest fixpoint of F. It follows from the definition of $\simeq$ that if $(\delta, \gamma) \in R$ for some bisimulation R then $\delta \simeq \gamma$. For example, if $X \doteq \mu\alpha.\uparrow[\alpha, \alpha]$ then the relation

$$\{(X, \uparrow[X, X]), (\uparrow[X, X], \uparrow[X, X]), (X, X)\}$$

is a bisimulation, and is sufficient to prove that $X \simeq \uparrow[X, X]$.

4.2 Typing rules

We can now reinterpret our process typing rules (Definition 3.4 on page 33), replacing syntactic type equality with $\simeq$ in both the Input and Output rules (the other typing rules remain unchanged):

Definition 4.4 (Typing rules using $\simeq$)

$$\frac{\Delta(c) \simeq \uparrow[\delta_1, \ldots, \delta_n] \qquad \Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash P}{\Delta \vdash c?[x_1, \ldots, x_n].P} \quad \text{Input}$$

$$\frac{\Delta(c) \simeq \uparrow[\Delta(x_1), \ldots, \Delta(x_n)] \qquad \Delta \vdash P}{\Delta \vdash c![x_1, \ldots, x_n].P} \quad \text{Output}$$

For example, we can now give a type to the process $x![x, x].\mathbf{0}$, which sends the pair $[x, x]$ along the channel x. If $X \doteq \mu\alpha.\uparrow[\alpha, \alpha]$ and $\Delta \doteq x : X$ then we can use the Output typing rule to show that $\Delta \vdash x![x, x].\mathbf{0}$ (we proved that $X \simeq \uparrow[X, X]$ in Section 4.1):

$$\frac{\Delta(x) = X \simeq \uparrow[X, X] = \uparrow[\Delta(x), \Delta(x)] \qquad \Delta \vdash \mathbf{0}}{\Delta \vdash x![x, x].\mathbf{0}}$$

More importantly, we can now typecheck our list processing examples from Section 2.9. Let *List* be the recursive type $\mu L.\uparrow[\uparrow[\,], \uparrow[\alpha, L]]$. It is easy to check that

$$\{(\mathit{List}, \uparrow[\uparrow[\,], \uparrow[\alpha, \mathit{List}]]),\ (\uparrow[\alpha, \mathit{List}], \uparrow[\alpha, \mathit{List}]),\ (\uparrow[\,], \uparrow[\,]),\ (\mathit{List}, \mathit{List}),\ (\alpha, \alpha)\}$$

is a bisimulation, which implies that $\mathit{List} \simeq \uparrow[\uparrow[\,], \uparrow[\alpha, \mathit{List}]]$. We reproduce the definitions of *Nil* and *Cons* below, indicating how they are typed by annotating bound variables with types.

$$\begin{array}{l}
\texttt{def}\ \mathit{Nil}[r : \uparrow[\mathit{List}]] = \\
\quad (\nu\, l : \mathit{List})(\, r![l] \ \mid\ *l?[n : \uparrow[\,], c : \uparrow[\alpha, \mathit{List}]].n![]\,)
\end{array}$$

$$\begin{array}{l}
\texttt{def}\ \mathit{Cons}[hd : \alpha, tl : \mathit{List}, r : \uparrow[\mathit{List}]] = \\
\quad (\nu\, l : \mathit{List})(\, r![l] \ \mid\ *l?[n : \uparrow[\,], c : \uparrow[\alpha, \mathit{List}]].c![hd, tl]\,)
\end{array}$$

The above annotations imply that *Nil* has type $[\,] \Rightarrow [\mathit{List}]$ and *Cons* has type $[\alpha, \mathit{List}] \Rightarrow [\mathit{List}]$.

Definition 4.5 (Typing of list pattern-matching)

$$\frac{\Delta(l) = \mathit{List} \qquad \Delta \vdash P \qquad \Delta, hd : \alpha, tl : \mathit{List} \vdash Q}{\Delta \vdash \texttt{match}\ l\ \texttt{with}\ \mathit{Nil}\ \texttt{=>}\ P\ \texttt{and}\ \mathit{Cons}[hd, tl]\ \texttt{=>}\ Q}\ \text{Match}$$

It is easy to check, by expanding out the derived form for `match` (Definition 2.8 on page 26), that the above rule is admissible. (We can use the same proof technique as we used in Section 3.5.)

We can now verify that *Concat* has type $[\mathit{List}, \mathit{List}] \Rightarrow [\mathit{List}]$:

$$\begin{array}{l}
\texttt{def}\ \mathit{Concat}[l_1 : \mathit{List}, l_2 : \mathit{List}, r : \uparrow[\mathit{List}]] = \\
\quad \texttt{match}\ l_1\ \texttt{with}\ \mathit{Nil}\ \texttt{=>} \\
\qquad r![l_2] \\
\quad \texttt{and}\ \mathit{Cons}[hd : \alpha, tl : \mathit{List}]\ \texttt{=>} \\
\qquad \texttt{let}\ \mathit{rest} : \mathit{List} = \mathit{Concat}(tl, l_2)\ \texttt{in}\ \mathit{Cons}![hd, \mathit{rest}, r]
\end{array}$$

4.3 Encoding the monadic π-calculus

The monadic π-calculus is clearly is special case of the polyadic π-calculus (where all tuples have arity one). Without the help of recursive types, there are many monadic π-terms which cannot be given a type in our simple type system (for example, the term $x![x].\mathbf{0}$). However, the following rules are admissible when we have recursive types:

Definition 4.6 (Typing monadic π-terms)

$$\frac{\Delta(x) = \mu\alpha.\uparrow[\alpha] \qquad \Delta, y : \mu\alpha.\uparrow[\alpha] \vdash P}{\Delta \vdash x?[y].P} \quad \text{Monadic Input}$$

$$\frac{\Delta(x) = \mu\alpha.\uparrow[\alpha] \qquad \Delta(y) = \mu\alpha.\uparrow[\alpha] \qquad \Delta \vdash P}{\Delta \vdash x![y].P} \quad \text{Monadic Output}$$

If every variable in Δ has type $\mu\alpha.\uparrow[\alpha]$, then $\Delta \vdash P$ for every monadic π-term P whose free variables are a subset of those bound in Δ, since the Monadic Input rule preserves the invariant that every variable has type $\mu\alpha.\uparrow[\alpha]$. We therefore regain the full power of the monadic π-calculus if we allow recursive types (though we do pay a price for that flexibility, since the type $\mu\alpha.\uparrow[\alpha]$ is rather uninformative).

4.4 Properties of $\simeq$

We now prove that $\simeq$ is a reasonable equality relation: it is a equivalence relation, it is preserved by substitution, and it is preserved by all type constructors.

Proposition 4.7 ($\simeq$ is an equivalence relation)

1) $\delta \simeq \delta$.

2) If $\delta \simeq \gamma$ then $\gamma \simeq \delta$.

3) If $\delta \simeq \delta'$ and $\delta' \simeq \delta''$ then $\delta \simeq \delta''$.

Proof Part 1 follows from the fact that the identity relation is a bisimulation. Part 2 follows from the fact that $\{(\gamma, \delta) \mid (\delta, \gamma) \in R\}$ is a bisimulation if R is a bisimulation. Part 3 follows from the fact that

$$\{(\delta, \delta'') \mid (\delta, \delta') \in R \text{ and } (\delta', \delta'') \in R'\}$$

is a bisimulation if R and R' are bisimulations. □

Before we can prove that $\simeq$ is preserved under substitution, we need to show how substitution affects the observations we can make of a type.

Lemma 4.8 (Observation and substitution)

1) If $\delta \Downarrow \alpha$ and $\gamma \Downarrow \gamma'$ then $\{\gamma/\alpha\}\delta \Downarrow \gamma'$.

2) If $\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$ then $\{\gamma/\alpha\}\delta \Downarrow \uparrow[\{\gamma/\alpha\}\delta_1, \ldots, \{\gamma/\alpha\}\delta_n]$.

Proof Both results can be proved by induction on the depth of the inference of the observation of δ. □

The following lemma proves that the observations we can make from the type $\{\gamma/\alpha\}\delta$ arise either from the original type δ, or from the substituted type γ. No essentially new observations arise when we substitute γ for α in δ.

Lemma 4.9 (Substitution and observation)

If $\{\gamma/\alpha\}\delta \Downarrow \gamma'$ then either

1. $\alpha \notin fv(\delta)$ and $\delta \Downarrow \gamma'$, or
2. $\delta \Downarrow \alpha$ and $\gamma \Downarrow \gamma'$, or
3. $\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$ and $\gamma' = \uparrow[\{\gamma/\alpha\}\delta_1, \ldots, \{\gamma/\alpha\}\delta_n]$.

Proof The result can be proved using induction on the depth of the inference of $\{\gamma/\alpha\}\delta \Downarrow \gamma'$. □

Our equality relation forces the free type variables of equal types to be the same:

Lemma 4.10 (Free type variables and $\simeq$)

If $\delta \simeq \gamma$ then $\alpha \in fv(\delta)$ iff $\alpha \in fv(\gamma)$.

Proof It is easy to see that if $\alpha \in fv(\delta)$ then we can eventually observe the type variable α. Now, since we can observe α, it must also be the case that we can observe an α in γ, since δ and γ are bisimilar, so $\alpha \in fv(\gamma)$ as required. We can use identical reasoning to prove that if $\alpha \in fv(\gamma)$ then $\alpha \in fv(\delta)$. □

We can now prove that $\simeq$ is preserved under substitution.

Proposition 4.11 ($\simeq$ is preserved under substitution)
If $\delta \simeq \delta'$ and $\gamma \simeq \gamma'$ then $\{\gamma/\alpha\}\delta \simeq \{\gamma'/\alpha\}\delta'$.

Proof The result follows if we can prove that

$$\{(\{\gamma/\alpha\}\delta, \{\gamma'/\alpha\}\delta') \mid (\delta, \delta') \in R\} \cup R \cup R'$$

is a bisimulation if R and R' are bisimulations containing the pairs (δ, δ') and (γ, γ') respectively. Suppose that $\{\gamma/\alpha\}\delta \Downarrow \beta$. We know from Lemma 4.9 that either

1. $\alpha \notin fv(\delta)$ and $\delta \Downarrow \beta$, in which case we have that $\delta' \Downarrow \beta$, since R is a bisimulation. Now, we know from Lemma 4.10 that $\alpha \notin fv(\delta')$ and the result follows, since $\{\gamma'/\alpha\}\delta' = \delta'$.

2. $\delta \Downarrow \alpha$ and $\gamma \Downarrow \beta$, in which case we know that $\delta' \Downarrow \alpha$ (since R is a bisimulation), and $\gamma' \Downarrow \beta$ (since R' is a bisimulation). Using Lemma 4.8, we have that $\{\gamma'/\alpha\}\delta' \Downarrow \beta$ as required.

Alternatively, if $\{\gamma/\alpha\}\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, we know from Lemma 4.9 that either

1. $\alpha \notin fv(\delta)$ and $\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, in which case we have, since R is a bisimulation, that $\delta' \Downarrow \uparrow[\delta'_1, \ldots, \delta'_n]$ where $(\delta_i, \delta'_i) \in R$. Now, we know from Lemma 4.10 that $\alpha \notin fv(\delta')$ and the result follows, since $\{\gamma'/\alpha\}\delta' = \delta'$.

2. $\delta \Downarrow \alpha$ and $\gamma \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, in which case we know that $\delta' \Downarrow \alpha$ (since R is a bisimulation), and $\gamma' \Downarrow \uparrow[\delta'_1, \ldots, \delta'_n]$ where $(\delta_i, \delta'_i) \in R'$ (since R' is a bisimulation). Using Lemma 4.8, we have that $\{\gamma'/\alpha\}\delta' \Downarrow \uparrow[\delta'_1, \ldots, \delta'_n]$ as required.

3. $\delta \Downarrow \uparrow[\tau_1, \ldots, \tau_n]$ and $\uparrow[\delta_1, \ldots, \delta_n] = \uparrow[\{\gamma/\alpha\}\tau_1, \ldots, \{\gamma/\alpha\}\tau_n]$, in which case we know that $\delta' \Downarrow \uparrow[\tau'_1, \ldots, \tau'_n]$ and $(\tau_i, \tau'_i) \in R$ (since R is a bisimulation). Using Lemma 4.8, we have that $\{\gamma'/\alpha\}\delta' \Downarrow \uparrow[\{\gamma'/\alpha\}\tau'_1, \ldots, \{\gamma'/\alpha\}\tau'_n]$ and the result follows, since for each $1 \leq i \leq n$, the pair $(\{\gamma/\alpha\}\tau_i, \{\gamma'/\alpha\}\tau'_i)$ is in our bisimulation relation.

We therefore have that $\{\gamma/\alpha\}\delta \simeq \{\gamma'/\alpha\}\delta'$, as required. $\square$

Before we prove that $\simeq$ is a congruence relation, we must prove the following lemmas about observation and recursive types.

Lemma 4.12 (Observation and recursive types)

1) If $\mu\alpha.\delta \Downarrow \beta$ then $\alpha \neq \beta$ and $\delta \Downarrow \beta$.

2) If $\mu\alpha.\delta \Downarrow \uparrow[\gamma_1, \ldots, \gamma_n]$ then there exist $\gamma'_1, \ldots, \gamma'_n$ such that $\delta \Downarrow \uparrow[\gamma'_1, \ldots, \gamma'_n]$ and $\{\mu\alpha.\delta/\alpha\}\gamma'_i = \gamma_i$ for $1 \leq i \leq n$.

3) If $\delta \Downarrow \gamma$ then $\mu\alpha.\delta \Downarrow \{\mu\alpha.\delta/\alpha\}\gamma$

Proof All three results can be proved using induction on the depth of the inference of the initial observation. All rely on the fact that occurrences of the recursively-bound variable α must be contractive. □

Proposition 4.13 ($\simeq$ is an congruence relation)

1) If $\delta_i \simeq \gamma_i$ for $1 \leq i \leq n$ then $\uparrow[\delta_1, \ldots, \delta_n] \simeq \uparrow[\gamma_1, \ldots, \gamma_n]$.

2) If $\delta \simeq \gamma$ then $\mu\alpha.\delta \simeq \mu\alpha.\gamma$.

Proof Part 1 follows from the fact that

$$\{(\uparrow[\delta_1, \ldots, \delta_n], \uparrow[\gamma_1, \ldots, \gamma_n])\} \cup R_1 \cup \ldots \cup R_n$$

is a bisimulation if each relation R_i is a bisimulation containing the pair (δ_i, γ_i). Part 2 follows from the fact that the following relation is a bisimulation if R is a bisimulation containing the pair (δ, γ).

$$\{(\mu\alpha.\delta, \mu\alpha.\gamma)\} \cup \{(\{\mu\alpha.\delta/\alpha\}\tau, \{\mu\alpha.\gamma/\alpha\}\tau') \mid (\tau, \tau') \in R\} \cup R$$

Suppose that $\{\mu\alpha.\delta/\alpha\}\tau \Downarrow \beta$. We know from Lemma 4.9 that either

1. $\alpha \notin fv(\tau)$ and $\tau \Downarrow \beta$, in which case we have that $\tau' \Downarrow \beta$, since R is a bisimulation. Now, we know from Lemma 4.10 that $\alpha \notin fv(\tau')$ and the result follows, since $\{\mu\alpha.\delta/\alpha\}\tau' = \tau'$.

2. $\tau \Downarrow \alpha$ and $\mu\alpha.\delta \Downarrow \beta$, in which case we know that $\tau' \Downarrow \alpha$, since R is a bisimulation. Now, using Lemma 4.12, we have that $\delta \Downarrow \beta$. Therefore, since R' is a bisimulation, we have that $\gamma \Downarrow \beta$. Using Lemma 4.12, we have that $\mu\alpha.\gamma \Downarrow \{\mu\alpha.\gamma/\alpha\}\beta = \beta$ as required.

Alternatively, if $\{\mu\alpha.\delta/\alpha\}\tau \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, we know from Lemma 4.9 that either

1. $\alpha \notin fv(\tau)$ and $\tau \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, in which case we have, since R is a bisimulation, that $\tau' \Downarrow \uparrow[\delta'_1, \ldots, \delta'_n]$ where $(\delta_i, \delta'_i) \in R$. Now, we know from Lemma 4.10 that $\alpha \notin fv(\tau')$ and the result follows, since $\{\mu\alpha.\gamma/\alpha\}\tau' = \tau'$.

2. $\tau \Downarrow \alpha$ and $\mu\alpha.\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, in which case we know that $\tau' \Downarrow \alpha$, since R is a bisimulation. Now, using Lemma 4.12, we have that there exist $\tau_1, \ldots, \tau_n$ such that $\delta \Downarrow \uparrow[\tau_1, \ldots, \tau_n]$ and $\{\mu\alpha.\delta/\alpha\}\tau_i = \delta_i$ for $1 \leq i \leq n$. Now, since R is a bisimulation, we have that there exist $\tau'_1, \ldots, \tau'_n$ such that $\gamma \Downarrow \uparrow[\tau'_1, \ldots, \tau'_n]$ and $(\tau_i, \tau'_i) \in R$ for $1 \leq i \leq n$. Using Lemma 4.12, we have that $\mu\alpha.\gamma \Downarrow \uparrow[\{\gamma/\alpha\}\tau'_1, \ldots, \{\gamma/\alpha\}\tau'_n]$. The result follows, since for each $1 \leq i \leq n$ the pair $(\{\mu\alpha.\delta/\alpha\}\tau_i, \{\mu\alpha.\gamma/\alpha\}\tau'_i)$ is in our bisimulation relation.

3. $\tau \Downarrow \uparrow[\tau_1, \ldots, \tau_n]$ and $\uparrow[\delta_1, \ldots, \delta_n] = \uparrow[\{\mu\alpha.\delta/\alpha\}\tau_1, \ldots, \{\mu\alpha.\delta/\alpha\}\tau_n]$, so we know that $\tau' \Downarrow \uparrow[\tau'_1, \ldots, \tau'_n]$ and $(\tau_i, \tau'_i) \in R$ (since R is a bisimulation). Lemma 4.8 proves that $\{\mu\alpha.\gamma/\alpha\}\tau' \Downarrow \uparrow[\{\mu\alpha.\gamma/\alpha\}\tau'_1, \ldots, \{\mu\alpha.\gamma/\alpha\}\tau'_n]$ and the result follows, since each pair $(\{\mu\alpha.\delta/\alpha\}\tau_i, \{\mu\alpha.\gamma/\alpha\}\tau'_i)$ is in our bisimulation relation.

The observations of $\mu\alpha.\delta$ and $\mu\alpha.\gamma$ are identical to those of $\{\mu\alpha.\delta/\alpha\}\delta$ and $\{\mu\alpha.\gamma/\alpha\}\gamma$, and we have already checked that such pairs are bisimilar. We therefore have that $\mu\alpha.\delta \simeq \mu\alpha.\gamma$, as required. □

4.5 Checking type equality

We now present an algorithm which, given a relation R and two types δ and γ, builds a bisimulation containing the pair (δ, γ). The algorithm fails if R, δ and γ do not match one of the cases below. The first clause takes precedence over the other clauses in the case where more than once clause matches R, δ and γ.

Definition 4.14 (Checking type equality)

case $Eq(R, \delta, \gamma)$ where $(\delta, \gamma) \in R$

Return R.

case $Eq(R, \delta, \gamma)$ where $\delta \Downarrow \alpha$ and $\gamma \Downarrow \alpha$

Return $R \cup \{(\delta, \gamma)\}$.

case $Eq(R, \delta, \gamma)$ where $\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$ and $\gamma \Downarrow \uparrow[\gamma_1, \ldots, \gamma_n]$

If $R_0 = R \cup \{(\delta, \gamma)\}$ and $R_i = Eq(R_{i-1}, \delta_i, \gamma_i)$ for $1 \leq i \leq n$ then return R_n.

The relation R contains pairs of types which have already been checked by the algorithm. We therefore simply return R if we encounter a pair we have already checked. Note that the fact that all recursive types are contractive guarantees that for every δ we can always effectively compute the (unique) type γ such that $\delta \Downarrow \gamma$.

The soundness of *Eq* depends crucially on the relation R which it is passed as an argument. For example, if $R = \{(\uparrow[\,], \uparrow[\alpha])\}$ then $Eq(R, \uparrow[\,], \uparrow[\alpha])$ succeeds, returning R, even though the types $\uparrow[\,]$ and $\uparrow[\alpha]$ are clearly not bisimilar. We must therefore prove that the result of *Eq* is sound assuming R is sound. The following definition formalises when a relation is a bisimulation relative to a second relation (which can be thought of as containing pairs of types which have already been checked for equality).

Definition 4.15 (Relative bisimulation) The relation R is a bisimulation relative to R' if $R \supseteq R'$ and $R \subseteq F(R) \cup R'$ where F is the function on relations given in Definition 4.3.

Suppose we wish to check that R is a bisimulation relative to R'. For each pair $(\delta, \gamma) \in R$ we need to check that either $(\delta, \gamma) \in R'$ (which we can interpret as meaning that δ and γ have been checked elsewhere), or $(\delta, \gamma) \in F(R)$ (which means we have to check the observable type structure of δ and γ in the normal way).

We can now state and prove the soundness of *Eq*. Note that in top-level calls the relation R passed as an argument to *Eq* will be $\{\}$, the empty relation. Thus, at the top-level we have that the result of *Eq* is a bisimulation relative to $\{\}$, which implies that R is a bisimulation.

Lemma 4.16 (Soundness of *Eq*)

If $Eq(R, \delta, \gamma) = R'$ then R' is a bisimulation relative to R and $(\delta, \gamma) \in R'$.

Proof We prove the result using induction on the depth on the inference of $Eq(R, \delta, \gamma) = R'$.

case $Eq(R, \delta, \gamma) = R$ where $(\delta, \gamma) \in R$

Immediate from the definition of relative bisimulation.

case $Eq(R, \delta, \gamma) = R \cup \{(\delta, \gamma)\}$ where $\delta \Downarrow \alpha$ and $\gamma \Downarrow \alpha$

Immediate, since δ and γ have identical observable type structure.

case $Eq(R, \delta, \gamma) = R_n$ where $\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, $\gamma \Downarrow \uparrow[\gamma_1, \ldots, \gamma_n]$, $R_0 = R \cup \{(\delta, \gamma)\}$ and $R_i = Eq(R_{i-1}, \delta_i, \gamma_i)$ for $1 \leq i \leq n$

We have, using induction, that R_i is a bisimulation relative to R_{i-1} for $1 \leq i \leq n$. Thus, we have that $R_i \supseteq R_{i-1}$ and $R_i \subseteq F(R_i) \cup R_{i-1}$. This clearly implies that $R_n \supseteq R$, since $R_0 = R \cup \{(\delta, \gamma)\}$. Now, since $R_n \subseteq F(R_n) \cup R_{n-1}$ and $R_{n-1} \subseteq F(R_{n-1}) \cup R_{n-2}$ we have that $R_n \subseteq F(R_n) \cup F(R_{n-1}) \cup R_{n-2}$. But $F(R_{n-1}) \subseteq F(R_n)$ since $R_{n-1} \subseteq R_n$ and F is monotone, so we therefore have that $R_n \subset F(R_n) \cup R_{n-2}$ i.e. R is bisimilar relative to R_{n-2}. Iterating this argument proves that R is bisimilar relative to R_0 i.e. $R_n \subseteq F(R_n) \cup R_0$. Now, since $R_0 = R \cup \{(\delta, \gamma)\}$, if we can prove that $(\delta, g) \in F(R_n)$ we can conclude that $R_n \subseteq F(R_n) \cup R$ as required. But it is easy to see that $(\delta, g) \in F(R_n)$, since $\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$, $\gamma \Downarrow \uparrow[\gamma_1, \ldots, \gamma_n]$ and $(\delta_i, \gamma_i) \in R_i \subseteq R_n$. Thus, R_n is a bisimulation relative to R, as required. $\square$

The following lemma proves the completeness of *Eq*: if δ and γ are bisimilar then *Eq* will succeed, returning a bisimulation containing (δ, γ).

Lemma 4.17 (Completeness of *Eq*)

If $(\delta, \gamma) \in R'$ for some bisimulation R' and $R \subseteq R'$ then $Eq(R, \delta, \gamma)$ succeeds, returning R'' where $(\delta, \gamma) \in R''$ and $R'' \subseteq R'$.

Proof We prove the result using induction on the number of pairs still to be checked by *Eq* (i.e. the size of the set $R' - R$). In the base case $R = R'$ and we therefore have that $(\delta, \gamma) \in R$ and $Eq(R, \delta, \gamma)$ succeeds, returning R as required. Otherwise, if the size of $R' - R$ is non-zero, we have two cases to consider: if $(\delta, \gamma) \in R$ then $Eq(R, \delta, \gamma) = R$ and the result follows. Otherwise, if $(\delta, \gamma) \notin R$ then it must be the case that either

1. $\delta \Downarrow \alpha$ and $\gamma \Downarrow \alpha$, in which case $Eq(R, \delta, \gamma)$ succeeds, returning $R \cup \{(\delta, \gamma)\}$, and the result follows.

2. $\delta \Downarrow \uparrow[\delta_1, \ldots, \delta_n]$ and $\gamma \Downarrow \uparrow[\gamma_1, \ldots, \gamma_n]$ where $(\delta_i, \gamma_i) \in R'$ for $1 \leq i \leq n$. Now, if $R_0 = R \cup \{(\delta, \gamma)\}$ we clearly have that $R_0 \subseteq R'$, so we can use induction to prove that $Eq(R_{i-1}, \delta_i, \gamma_i)$ succeeds, returning R_i where $(\delta_i, \gamma_i) \in R_i$ and $R_i \subseteq R'$, for $1 \leq i \leq n$. Thus, we have that $Eq(R, \delta, \gamma)$ succeeds, returning R_n, where $(\delta, \gamma) \in R_n$ and $R_n \subseteq R'$, as required.

Thus, in all cases $Eq(\delta, \gamma)$ succeeds, as required. $\square$

The following two propositions are simple corollaries of Lemmas 4.16 and 4.17, and give simplified statements of soundness and completeness.

Proposition 4.18 (Soundness of Eq)

If $Eq(\{\}, \delta, \gamma)$ succeeds then $\delta \simeq \gamma$.

Proof If $Eq(\{\}, \delta, \gamma) = R$ then from Lemma 4.16 we have that R is a bisimulation containing the pair (δ, γ). It follows from the definition of $\simeq$ that $\delta \simeq \gamma$. $\square$

Proposition 4.19 (Completeness of Eq)

If $\delta \simeq \gamma$ then $Eq(\{\}, \delta, \gamma)$ succeeds.

Proof If $\delta \simeq \gamma$ then there must exist a bisimulation R containing the pair (δ, γ). Now, since $\{\} \subseteq R$, we can use Lemma 4.17 to prove that $Eq(\{\}, \delta, \gamma)$ succeeds, as required. $\square$

4.6 Type soundness

It is easy to check that the weakening, strengthening and substitution lemmas from Section 3.8 still hold in the presence of recursive types (the proofs are identical to those in Section 3.8). We can then prove that types are preserved under structural congruence, using the same techniques as in Section 3.9.

We are now able to prove that our type system remains sound when it is extended with recursive types. The proofs are almost identical to those in Section 3.10.

Theorem 4.20 (Well-typed processes never fail)

If $\Delta \vdash P$ then not (P *fails*).

Theorem 4.21 (Subject reduction)

If $\Delta \vdash P$ and $P \rightarrow Q$ then $\Delta \vdash Q$.

It is interesting to note that the proof of Theorem 4.20 relies on the fact that if $\uparrow[\delta_1, \ldots, \delta_m] \simeq \uparrow[\gamma_1, \ldots, \gamma_n]$ then $m = n$ and $\delta_i \simeq \gamma_i$ for $1 \leq i \leq n$. This fact seems obvious, and indeed it follows immediately from the definition of $\simeq$, but it marks the dividing line between an 'acceptable' equality relation for types and an 'unacceptable' one. The proofs of weakening, strengthening, substitution, preservation of types under structural congruence and subject reduction remain valid for arbitrary congruence relations (for example, the universal relation).

Chapter 5

Polymorphism

A common disadvantage of simple type systems is that, although they prevent common programming errors, they also disallow many useful and intuitively correct programs. Polymorphic type systems overcome much of this problem by allowing generic operations, that is, operations which can be safely applied to many different types of argument. List operations such as reversing and concatenation are good examples of generic operations, since they act completely independently of the types of the elements in the lists. The extra flexibility offered by a polymorphic type system seems to be enough to allow a more natural style of programming, where the type system is not perceived as 'getting in the way'.

In this chapter we define an explicitly-typed polymorphic type system for π-calculus which arises as a natural extension of the simple type system presented earlier. We illustrate the utility of polymorphic types in π-calculus programming using a number of examples, and then show how polymorphic channels can be used to model abstract datatypes in a type-safe manner. We then prove (by means of a subject-reduction theorem) that our polymorphic type system guarantees freedom from runtime errors.

5.1 Typing rules

A simple example of a channel which can be used polymorphically is the channel f in the (explicitly-typed) process below:

$$f : \uparrow[\alpha, \uparrow[\alpha]] \vdash *f?[x : \alpha, r : \uparrow[\alpha]].r![x]$$

Intuitively, we should be able to send any pair of channels along f whose types are instances of α and $\uparrow[\alpha]$ respectively. In fact, we can think of f as having an additional type argument α, as shown below:

$$f : \uparrow[\alpha\,;\ \alpha, \uparrow[\alpha]] \vdash *f?[\alpha\,;\ x : \alpha, r : \uparrow[\alpha]].r![x]$$

The type of f is similarly extended to indicate that f requires an explicit type argument to be sent along with the pair of channels. For example, the following process can send the channels a and b along f, since the types of a and b match the types required by f (when we instantiate the type parameter α with the explicit type argument δ).

$$f : \uparrow[\alpha\,;\ \alpha, \uparrow[\alpha]], a : \delta, b : \uparrow[\delta] \vdash f![\delta\,;\ a, b]$$

Thus, from the server's pointer of view (i.e. the point of view of any process reading messages from f), the type $\uparrow[\alpha\,;\ \alpha, \uparrow[\alpha]]$ can be interpreted as a requirement that the server must behave correctly given any type α and a pair of channels of type α and $\uparrow[\alpha]$ respectively.

From the client's pointer of view (i.e. the point of view of any process writing messages on f), the type $\uparrow[\alpha\,;\ \alpha, \uparrow[\alpha]]$ can be interpreted as a guarantee that any process listening on f will behave correctly as long as the client supplies a type δ and two channels whose types match the types α and $\uparrow[\alpha]$ (after substituting the argument type δ for the type parameter α).

We now formally define our explicitly-typed polymorphic π-calculus. First, we give the syntax of explicitly-typed terms. We require an explicit type annotation on every bound variable, explicit type parameters in input expressions, and explicit type arguments in output expressions. The syntax for parallel composition, summation, replication and the nil process is unchanged.

Definition 5.1 (Explicitly-typed π-calculus)

P, Q, R, S	$::=$	$P \mid P$	Parallel composition
		$(\nu\, x : \delta)P$	Restriction
		$P + P$	Summation
		$x?[\alpha_1, \ldots, \alpha_m\ ;\ y_1 : \delta_1, \ldots, y_n : \delta_n].P$	Input
		$x![\delta_1, \ldots, \delta_m\ ;\ y_1, \ldots, y_n].P$	Output
		$*P$	Replication
		$\mathbf{0}$	Nil

Note that we allow polyadic type arguments as well as polyadic channel arguments. In the case where $m = 0$, we write just $x?[y_1 : \delta_1, \ldots, y_n : \delta_n].P$ instead of $x?[\ ;\ y_1 : \delta_1, \ldots, y_n : \delta_n].P$, and $x![y_1, \ldots, y_n].P$ instead of $x![\ ;\ y_1, \ldots, y_n].P$. The expression $x?[\alpha_1, \ldots, \alpha_m\ ;\ y_1 : \delta_1, \ldots, y_n : \delta_n].P$ binds the type variables $\alpha_1, \ldots, \alpha_m$ (which are in scope in both $\delta_1, \ldots, \delta_n$ and P). The type parameters $\alpha_1, \ldots, \alpha_m$ must always be pairwise-distinct.

We generalise our syntax for simple channel types so that channels may now contain type parameters. The types $\alpha_1, \ldots, \alpha_m$ are bound by the channel type constructor and have scope $\delta_1, \ldots, \delta_n$. The type parameters $\alpha_1, \ldots, \alpha_m$ must always be pairwise-distinct. In the case where $m = 0$, we write just $\uparrow[\delta_1, \ldots, \delta_n]$ instead of $\uparrow[\ ;\ \delta_1, \ldots, \delta_n]$.

Definition 5.2 (Polymorphic types)

δ	$::=$	$\uparrow[\alpha_1, \ldots, \alpha_m\ ;\ \delta_1, \ldots, \delta_n]$	Polymorphic channel type
		α	Type variable

The syntax of type contexts is just the same it was in our monomorphic type system (modulo the change in the syntax of types). The variables $x_1, \ldots, x_n$ must be pairwise-distinct. The expression $ftv(\Delta)$ denotes the free type variables of Δ and is defined to be the union of all the free type variables of those types contained in Δ.

Definition 5.3 (Type contexts) $\quad \Delta \ ::=\ x_1 : \delta_1, \ldots, x_n : \delta_n$

The typechecking rules for the nil process, parallel composition, summation and replication are the same as the typing rules we gave for our monomorphic type system:

$$\Delta \vdash \mathbf{0} \quad \text{Nil} \qquad \frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P \mid Q} \ \text{Prl}$$

$$\frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P + Q} \ \text{Smt} \qquad \frac{\Delta \vdash P}{\Delta \vdash *P} \ \text{Repl}$$

The typechecking rule for an explicitly-typed restriction is similar to our original rule for restriction, except that we now force the type assigned to x to be a channel type. This restriction on the type of x is not necessary to preserve the soundness of our type system, but simplifies reasoning about abstract datatypes encoded in the polymorphic π-calculus (see Section 5.7 for details).

$$\frac{\Delta, x : \uparrow[\tilde{\alpha}\,;\ \tilde{\delta}] \vdash P}{\Delta \vdash (\nu\, x : \uparrow[\tilde{\alpha}\,;\ \tilde{\delta}])P} \ \text{Res}$$

In the case of a polymorphic input we check that the body of the input requires no more type structure of $x_1, \ldots, x_n$ than is specified in the type of c. The condition $\alpha_1, \ldots, \alpha_m \notin \mathit{ftv}(\Delta)$ ensures that we do not capture any type variables which occur free in the context. This rule generalises the rule for input we gave in our monomorphic π-calculus type system of Chapter 3 (just set $m = 0$).

$$\frac{\begin{array}{c}\alpha_1, \ldots, \alpha_m \notin \mathit{ftv}(\Delta) \\ \Delta(c) = \uparrow[\alpha_1, \ldots, \alpha_m\,;\ \delta_1, \ldots, \delta_n] \qquad \Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash P\end{array}}{\Delta \vdash c?[\alpha_1, \ldots, \alpha_m\,;\ x_1 : \delta_1, \ldots, x_n : \delta_n].P} \ \text{Input}$$

A simple instance of the Input typing rule is given below (we let Δ be the context $f : \uparrow[\alpha\,;\ \alpha, \uparrow[\alpha]]$).

$$\frac{\Delta(f) = \uparrow[\alpha\,;\ \alpha, \uparrow[\alpha]] \qquad \alpha \notin \mathit{ftv}(\Delta) \qquad \Delta, x : \alpha, r : \uparrow[\alpha] \vdash P}{\Delta \vdash f?[\alpha\,;\ x : \alpha, r : \uparrow[\alpha]].P} \ \text{Input}$$

In the case of an output along a polymorphic channel c, we check that the channel values we are sending along c are substitution instances of the types specified in the type of c (the type arguments $\delta_1, \ldots, \delta_m$ make it explicit how we instantiate each abstracted variable $\alpha_1, \ldots, \alpha_m$). Again, this generalises the rule for output we gave in our monomorphic π-calculus type system of Chapter 3 (just set $m = 0$).

$$\frac{\Delta(c) = \uparrow[\alpha_1, \ldots, \alpha_m \,;\, \gamma_1, \ldots, \gamma_m] \quad \Delta(a_i) = \{\delta_1, \ldots, \delta_m / \alpha_1, \ldots, \alpha_m\}\gamma_i \qquad 1 \leq i \leq n \qquad \Delta \vdash P}{\Delta \vdash c![\delta_1, \ldots, \delta_m \,;\, a_1, \ldots, a_n].P} \text{ Output}$$

Using the Output rule, it is easy to check that $\Delta, x : \alpha, r : \uparrow[\alpha] \vdash r![x]$ (the type of r contains no type arguments, so the Output rule is just the same as our monomorphic Output rule in this case). Thus, we can use the Input and Repl rules to conclude that our original example of a replicated process reading from f is well-typed:

$$\dfrac{\dfrac{\cdots \qquad \dfrac{\vdots}{\Delta, x : \alpha, r : \uparrow[\alpha] \vdash r![x]} \text{ Output}}{\Delta \vdash f?[\alpha \,;\, x : \alpha, r : \uparrow[\alpha]].r![x]} \text{ Input}}{\Delta \vdash *f?[\alpha \,;\, x : \alpha, r : \uparrow[\alpha]].r![x]} \text{ Repl}$$

Furthermore, one can use the Output rule to check that the following output expression is well-typed, since the types of a and b are substitution instances of those specified in the type of f (we let Δ' denote the context $\Delta, a : \delta, b : \uparrow[\delta]$). The type δ in the output expression indicates that we instantiate the type argument α with the actual type δ.

$$\frac{\Delta'(f) = \uparrow[\alpha \,;\, \alpha, \uparrow[\alpha]] \quad \Delta'(a) = \delta = \{\delta/\alpha\}\alpha \qquad \Delta'(b) = \uparrow[\delta] = \{\delta/\alpha\}\uparrow[\alpha] \qquad \Delta' \vdash \mathbf{0}}{\Delta' \vdash f![\delta \,;\, a, b]} \text{ Output}$$

It is easy to check that our replicated input example is well-typed in the context Δ'. Thus, since both our input and output examples agree on the type of f, we can run them in parallel:

$$\dfrac{\dfrac{\vdots}{\Delta' \vdash f![\delta \,;\, a, b]} \text{ Output} \qquad \dfrac{\vdots}{\Delta' \vdash *f?[\alpha \,;\, x : \alpha, r : \uparrow[\alpha]].r![x]} \text{ Repl}}{\Delta' \vdash f![\delta \,;\, a, b] \mid *f?[\alpha \,;\, x : \alpha, r : \uparrow[\alpha]].r![x]} \text{ Prl}$$

It is worth noting that in general there may be any number of processes reading from the channel f. For example, there is no reason why we shouldn't have two copies of our replicated process serving requests along f:

$$f : \uparrow[\alpha \,;\, \alpha, \uparrow[\alpha]] \vdash (*f?[\alpha \,;\, x : \alpha, r : \uparrow[\alpha]].r![x]) \mid (*f?[\alpha \,;\, x : \alpha, r : \uparrow[\alpha]].r![x])$$

Our type system ensures that each process which inputs values from the channel f provides the same standard of service (i.e. makes the same requirements of its arguments). For example, we would certainly not expect the following process to be well-typed in the context $f : \uparrow[\alpha; \alpha, \uparrow[\alpha]]$, since the rightmost process makes more demands of its first argument x that is allowed by the type of f (it sends the empty tuple along x):

$$f : \uparrow[\alpha\,;\ \alpha, \uparrow[\alpha]] \not\vdash (*f?[\alpha\,;\ x : \alpha, r : \uparrow[\alpha]].r![x]) \mid f?[\alpha\,;\ x : \alpha, r : \uparrow[\alpha]].x![\,]$$

It is important to generalise types at input prefixes rather than at some later stage in the typing derivation, since otherwise we cannot guarantee that every process which uses a polymorphic channel will be sufficiently polymorphic. Suppose that we had the following typing rule, which allows one to generalise the type of x at any point in the type inference (as long the type variables $\tilde{\alpha}$ do not appear in the typing context):

$$\frac{\Delta, x : \uparrow[\tilde{\delta}] \vdash P}{\Delta, x : \uparrow[\tilde{\alpha}\,;\ \tilde{\delta}] \vdash P}\ \tilde{\alpha} \notin \Delta$$

If we had such a rule we could give a polymorphic type to the channel f in the following example:

$$f?[x, r].(r![x] \mid *f?[y, r].r![x])$$

The process reading on f behaves like an identity function the first time it is called, since it returns x along the result channel r. However, all subsequent calls also return x, the argument given to the first call of f. The principal type for the above example (in our monomorphic type system) gives f type $\uparrow[\alpha, \uparrow[\alpha]]$. So, at the top-level, it seems clear that f is polymorphic in the type α (and we could apply the above typing rule to generalise the type of f).

However, it would be unsound to let f be polymorphic. Suppose that we send f a pair of an integer x and a result channel r of type $\uparrow[Int]$. We will receive x back from f along r, as expected. But what if we now send f a boolean b and a result channel s of type $\uparrow[Bool]$ (we will certainly be able to do this if f is polymorphic). Now, instead of receiving b back along s, we receive x, a value of type Int, which is incompatible with the value of type $Bool$ that we expected to receive.

Our polymorphic typing rules deal with above example correctly, since we check that each separate input prefix is sufficiently polymorphic. If we tried to give f the type $\uparrow[\alpha; \alpha, \uparrow[\alpha]]$ then we would find that the second (replicated) input on f is ill-formed (since the type α occurs free in the type of x).

For ease of reference, we summarise the typing rules for explicitly-typed polymorphic π-terms in Definition 5.4.

Definition 5.4 (Polymorphic typing rules)

$$\Delta \vdash \mathbf{0} \qquad \text{Nil}$$

$$\frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P \mid Q} \qquad \text{Prl}$$

$$\frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P + Q} \qquad \text{Smt}$$

$$\frac{\Delta \vdash P}{\Delta \vdash *P} \qquad \text{Repl}$$

$$\frac{\Delta, x : \uparrow[\tilde{\alpha}\,;\, \tilde{\delta}] \vdash P}{\Delta \vdash (\nu\, x : \uparrow[\tilde{\alpha}\,;\, \tilde{\delta}])P} \qquad \text{Res}$$

$$\frac{\begin{array}{c}\alpha_1, \ldots, \alpha_m \notin \mathit{ftv}(\Delta) \\ \Delta(c) = \uparrow[\alpha_1, \ldots, \alpha_m\,;\, \delta_1, \ldots, \delta_n] \qquad \Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash P\end{array}}{\Delta \vdash c?[\alpha_1, \ldots, \alpha_m\,;\, x_1 : \delta_1, \ldots, x_n : \delta_n].P} \qquad \text{Input}$$

$$\frac{\begin{array}{c}\Delta(c) = \uparrow[\alpha_1, \ldots, \alpha_m\,;\, \gamma_1, \ldots, \gamma_m] \\ \Delta(a_i) = \{\delta_1, \ldots, \delta_m / \alpha_1, \ldots, \alpha_m\}\gamma_i \qquad 1 \leq i \leq n \qquad \Delta \vdash P\end{array}}{\Delta \vdash c![\delta_1, \ldots, \delta_m\,;\, a_1, \ldots, a_n].P} \qquad \text{Output}$$

One might be tempted to add a type restriction operator to the polymorphic π-calculus, to match the π-calculus's channel restriction operator. Intuitively, a type restriction operator should create a new type which is distinct from all other types, just as the channel restriction operator creates a new channel which is distinct from all other channels. However, such an operator is essentially useless, since there is no way to create values which inhabit the new type. What we really need is the ability to create a new type and some values of that type. This sounds rather like an abstract datatype, but we show in Section 5.7 that abstract datatypes can be encoded using just polymorphic channel types, so there seems to be no obvious use for a type restriction operator in the π-calculus.

5.2 Recursive process definitions

Our derived form for process definitions (Definition 2.5) used channels to model each recursively-defined process. We can therefore easily generalise our derived form to allow recursively-defined polymorphic process definitions. We allow explicit type parameters $\tilde{\alpha}_i$ in each definition and require explicit types for the channel parameters $\tilde{x}_i$, to match the type information required in our polymorphic input expressions.

For each process definition X_i we create a new channel X_i and a replicated process which does a polymorphic input on X_i. Note that we infer the appropriate explicit type for each channel X_i from the explicit types given in the process definition for X_i.

Definition 5.5 (Process definitions)

$$
\begin{array}{ll}
 & \texttt{def}\ X_1[\tilde{\alpha}_1\ ;\ \tilde{x}_1 : \tilde{\delta}_1] = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ X_n[\tilde{\alpha}_n\ ;\ \tilde{x}_n : \tilde{\delta}_n] = P_n\ \texttt{in}\ Q \\
\doteq & \\
 & (\nu\, X_1 : \uparrow[\tilde{\alpha}_1\ ;\ \tilde{\delta}_1]) \ldots (\nu\, X_n : \uparrow[\tilde{\alpha}_n\ ;\ \tilde{\delta}_n])(\\
 & \qquad *X_1?[\tilde{\alpha}_1\ ;\ \tilde{x}_1 : \tilde{\delta}_1].P_1 \mid \ldots \mid *X_n?[\tilde{\alpha}_n\ ;\ \tilde{x}_n : \tilde{\delta}_n].P_n \mid Q \\
 &)
\end{array}
$$

We provide the following high-level typing rule for process definitions, which can be proved admissible using the same techniques as we used in Section 3.3.

Definition 5.6 (Typing of polymorphic process definitions)

$$
\frac{\begin{array}{c}
\tilde{\alpha}_1, \ldots, \tilde{\alpha}_n \notin \mathit{ftv}(\Delta) \\
\Delta, X_1 : \uparrow[\tilde{\alpha}_1\ ;\ \tilde{\delta}_1], \ldots, X_n : \uparrow[\tilde{\alpha}_n\ ;\ \tilde{\delta}_n] \vdash Q \\
\Delta, X_1 : \uparrow[\tilde{\alpha}_1\ ;\ \tilde{\delta}_1], \ldots, X_n : \uparrow[\tilde{\alpha}_n\ ;\ \tilde{\delta}_n], \tilde{x}_i : \tilde{\delta}_i \vdash P_i \qquad 1 \leq i \leq n
\end{array}}{\Delta \vdash \texttt{def}\ X_1[\tilde{\alpha}_1\ ;\ \tilde{x}_1] = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ X_n[\tilde{\alpha}_n\ ;\ \tilde{x}_n] = P_n\ \texttt{in}\ Q}\ \text{Def}
$$

We can invoke the process definition X_i by simply sending the desired type and value arguments along the channel X_i. For example, the following process can repeatedly output b along y and c along z, even if b and c have different types:

$$\begin{array}{l} \texttt{def}\ X[\alpha\ ;\ x : \uparrow[\alpha], a : \alpha] = x![a].X![\alpha\ ;\ x, a] \\ \texttt{in}\ X![\delta\ ;\ y, b] \mid X![\gamma\ ;\ z, c] \end{array}$$

5.3 Processes which return results

It is possible to generalise our syntax for getting results from processes (Definition 2.6) so that we can get results from polymorphic processes:

Definition 5.7 (Getting results from polymorphic processes)

$$\begin{array}{ll} & \texttt{let}\ \tilde{x} : \tilde{\gamma} = f(\tilde{\delta}\ ;\ \tilde{a})\ \texttt{in}\ P \\ \doteq & (\nu\, r : \uparrow[\tilde{\gamma}])(f![\tilde{\delta}\ ;\ \tilde{a}, r] \mid r?[\tilde{x} : \tilde{\gamma}].P) \qquad r \notin fv(P, f, \tilde{a}) \end{array}$$

Our polymorphic typing rules also give rise to the following admissible rule for `let`. (We introduce the type abbreviation $\Rightarrow$, since it clarifies which types are the arguments, and which are the results.)

Definition 5.8 (Typing `let`)

$$\forall\tilde{\alpha}.[\tilde{\delta}] \Rightarrow [\tilde{\gamma}] \ \doteq\ \uparrow[\tilde{\alpha}\ ;\ \tilde{\delta}, \uparrow[\tilde{\gamma}]]$$

$$\frac{\begin{array}{c}\Delta(f) = \forall\tilde{\alpha}.[\tilde{\gamma}'] \Rightarrow [\tilde{\gamma}''] \\ \Delta(\tilde{a}) = \{\tilde{\delta}/\tilde{\alpha}\}\tilde{\gamma}' \qquad \tilde{\gamma} = \{\tilde{\delta}/\tilde{\alpha}\}\tilde{\gamma}'' \qquad \Delta, \tilde{x} : \tilde{\gamma} \vdash P\end{array}}{\Delta \vdash \texttt{let}\ \tilde{x} : \tilde{\gamma} = f(\tilde{\delta}\ ;\ \tilde{a})\ \texttt{in}\ P}\ \text{Let}$$

We can check that the above rule is admissible by expanding out the derived form for `let` (just as we did in Section 3.4).

5.4 Process-based reference cells

We can now give better types to our process-based reference cells (we gave monomorphic types for these examples in Section 3.6).

A process-based reference cell can be represented as a pair of channels: the first channel can be used to read the contents of the cell, and the second can be used to update the contents of the cell. The process definition *Cell* describes the behaviour of a reference cell whose current contents is x and which can be accessed via the channels *read* and *write*. Now that we have polymorphic channels, we can make *Cell* polymorphic in α, the type of x:

$$\texttt{def}\ \mathit{Cell}[\alpha\,;\ x:\alpha, \mathit{read}:\uparrow[\alpha], \mathit{update}:\uparrow[\alpha]] = \\ \quad \mathit{read}![x].\mathit{Cell}![\alpha\,;\ x, \mathit{read}, \mathit{update}] + \mathit{update}?[n:\alpha].\mathit{Cell}![\alpha\,;\ n, \mathit{read}, \mathit{update}]$$

The process definition *Ref* takes an initial value x of type α, for any α, and creates a new reference cell. Note that *Ref* creates a new instance of the *Cell* process, instantiated at the type α.

$$\texttt{def}\ \mathit{Ref}[\alpha\,;\ x:\alpha, r:\uparrow[\uparrow[\alpha],\uparrow[\alpha]]] = \\ \quad (\nu\, \mathit{read}:\uparrow[\alpha])(\nu\, \mathit{update}:\uparrow[\alpha])(r![\mathit{read}, \mathit{update}] \mid \mathit{Cell}![\alpha\,;\ x, \mathit{read}, \mathit{update}])$$

5.5 Channel-based reference cells

The types of our channel-based references (from Section 3.7) can similarly be generalised. Let *ChanRef* α be the type $\uparrow[\alpha]$, the type of a reference cell represented as a channel. Given a value x of type α, *ChanRef* returns a value of type *ChanRef* α.

$$\texttt{def}\ \mathit{ChanRef}[\alpha\,;\ x:\alpha, r:\uparrow[\mathit{ChanRef}\ \alpha]] = (\nu\, \mathit{ref}:\uparrow[\alpha])(r![\mathit{ref}] \mid \mathit{ref}![x])$$

The *Read* process is now parametric in the type α. It takes a reference cell and returns the current contents of that reference cell.

$$\texttt{def}\ \mathit{Read}[\alpha\,;\ \mathit{ref}:\mathit{ChanRef}\ \alpha, r:\uparrow[\alpha]] = \mathit{ref}?[x:\alpha].(\mathit{ref}![x] \mid r![x])$$

Similarly, the *Update* process now works correctly given any reference cell *ref* of type *ChanRef* α and value v of type α.

$$\texttt{def}\ \mathit{Update}[\alpha\,;\ \mathit{ref}:\mathit{ChanRef}\ \alpha, v:\alpha, r:\uparrow[\,]] = \mathit{ref}?[x:\alpha].(\mathit{ref}![v] \mid r![])$$

5.6 Lists

We could now give polymorphic types for our list examples from Section 4.2, but that would require us to add recursive types to our polymorphic type system. Thus, instead of further complicating our type system, we present an encoding of lists and polymorphic list operations which can be typechecked using only polymorphic types (our encoding is closely related to the Church-encoding of lists in polymorphic λ-calculus). In practice, it is probably better to add recursive types, instead of relying solely on polymorphic types, since some operations are more naturally described using the encoding of lists presented in Section 4.2 (in particular, finding the tail of a list is not a constant-time operation in our Church-encoded lists, while it is in the previous encoding).

We first recall the Church-encoding of lists in the polymorphic λ-calculus. The expression $\mathit{List}\ \alpha$ denotes the type $\forall\beta.(\beta \to (\alpha \to \beta \to \beta) \to \beta)$.

$$\begin{array}{rcl} \mathit{Nil} & \doteq & \Lambda\alpha.\Lambda\beta.\lambda(n:\beta).\lambda(c:\alpha\to\beta\to\beta).n \\ \mathit{Cons} & \doteq & \Lambda\alpha.\lambda(\mathit{hd}:\alpha).\lambda(\mathit{tl}:\mathit{List}\ \alpha). \\ & & \quad\Lambda\beta.\lambda(n:\beta).\lambda(c:\alpha\to\beta\to\beta).c\ \mathit{hd}\ (\mathit{tl}\ [\beta]\ n\ c) \end{array}$$

The type of Nil is $\forall\alpha.\mathit{List}\,\alpha$, and the type of Cons is $\forall\alpha.(\alpha \to \mathit{List}\,\alpha \to \mathit{List}\,\alpha)$. Each encoded list allows us to iterate a function over the elements of that list, accumulating a result of type β. Thus, for instance, the expression

$$l\ [\mathit{Int}]\ 0\ (\lambda(e:\alpha).\lambda(x:\mathit{Int}).x+1)$$

computes the size of the list l, since l applies the function $\lambda(e:\alpha).\lambda(x:\mathit{Int}).x+1$ to each of its elements, using the value 0 as the initial value.

Another example of a function which uses this encoding of lists is the *Concat* function, shown below. *Concat* has the effect of concatenating the lists l_1 and l_2, and has type $\forall\alpha.(\mathit{List}\ \alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha)$:

$$\mathit{Concat} \ \doteq\ \Lambda\alpha.\lambda(l_1:\mathit{List}\ \alpha).\lambda(l_2:\mathit{List}\ \alpha).l_1\ [\mathit{List}\ \alpha]\ l_2\ (\mathit{Cons}\ [\alpha])$$

We now give the π-calculus version of the above list encoding, where the expression $\mathit{List}\ \alpha$ now denotes the type $\uparrow[\beta\ ;\ \beta, [\alpha,\beta] \Rightarrow [\beta], \uparrow[\beta]]$. The process definition *Nil* accepts a type α and a result channel r as arguments, and returns the location of a process definition implementing the empty list.

$$\begin{array}{l}
\texttt{def}\ \mathit{Nil}[\alpha\,;\ r : \uparrow[\mathit{List}\ \alpha]] = \\
\quad \texttt{def}\ \mathit{nil}[\beta\,;\ n : \beta, c : [\alpha, \beta] \Rightarrow [\beta], r : \uparrow[\beta]] = r![n] \\
\quad \texttt{in}\ r![\mathit{nil}]
\end{array}$$

The process definition *Cons* accepts a type α, the head of the list *hd*, the tail of the list *tl* and a result channel r as arguments, and returns the location of a process definition implementing a cons cell.

$$\begin{array}{l}
\texttt{def}\ \mathit{Cons}[\alpha\,;\ \mathit{hd} : \alpha, \mathit{tl} : \mathit{List}\ \alpha, r : \uparrow[\mathit{List}\ \alpha]] = \\
\quad \texttt{def}\ \mathit{cons}[\beta\,;\ n : \beta, c : [\alpha, \beta] \Rightarrow [\beta], r : \uparrow[\beta]] = \\
\quad\quad \texttt{let}\ \ x : \beta = \mathit{tl}(\beta\,;\ n, c)\ \ \texttt{in}\ \ c![\mathit{hd}, x, r] \\
\quad \texttt{in}\ \ r![\mathit{cons}]
\end{array}$$

In the definition of *cons*, we use our derived syntax for `let` to get the result of accumulating c over the tail of the list. This yields a result x of type β which we then pass on to c along with the head list element *hd*.

The π-calculus version of the concatenate function is shown below. It is a little more verbose than the λ-calculus version, since we have to explicitly construct the partial application of *Cons* to α (using the local process definition *cons*).

$$\begin{array}{l}
\texttt{def}\ \mathit{Concat}[\alpha\,;\ l_1 : \mathit{List}\ \alpha, l_2 : \mathit{List}\ \alpha, r : \uparrow[\mathit{List}\ \alpha]] = \\
\quad \texttt{def}\ \mathit{cons}[\mathit{hd} : \alpha, \mathit{tl} : \mathit{List}\ \alpha, r : \uparrow[\mathit{List}\ \alpha]] = \mathit{Cons}![\alpha\,;\ \mathit{hd}, \mathit{tl}, r] \\
\quad \texttt{in}\ \ l_1![\mathit{List}\ \alpha\,;\ l_2, \mathit{cons}, r]
\end{array}$$

5.7 Abstract datatypes

Abstract datatypes are a well-known and important program structuring technique. In [MP88], Mitchell and Plotkin showed that the typing behaviour of an abstract datatype is correctly modeled by an existential type. In fact, it also turns out that it is possible to encode existential types in the polymorphic λ-calculus [Rey83]. A similar technique is applicable in the polymorphic π-calculus, enabling us to provide support for programming with abstract datatypes in the π-calculus.

The following example illustrates how we can package up our booleans and boolean operations (from Section 2.8) in an abstract datatype. We have already show in Section 3.5 that *True*, *False*, *And*, *Or* and *Not* have simple, monomorphic, types. We have annotated the bound variables of each process definition to indicate these types (where *Rep* denotes $\uparrow[\uparrow[\,], \uparrow[\,]]$, the representation type of booleans).

$$
\begin{array}{l}
(\nu\, bool : BoolPackage)(\\
\quad \texttt{def}\ True[r : \uparrow[Rep]] = (\nu\, b)(\, r![b] \ \mid\ *b?[t,f].t![]\,) \\
\quad \texttt{def}\ False[r : \uparrow[Rep]] = (\nu\, b)(\, r![b] \ \mid\ *b?[t,f].f![]\,) \\
\quad \texttt{def}\ And[b_1 : Rep, b_2 : Rep, r : \uparrow[Rep]] = \texttt{if}\ b_1\ \texttt{then}\ r![b_2]\ \texttt{else}\ False![r] \\
\quad \texttt{def}\ Or[b_1 : Rep, b_2 : Rep, r : \uparrow[Rep]] = \texttt{if}\ b_1\ \texttt{then}\ True![r]\ \texttt{else}\ r![b_2] \\
\quad \texttt{def}\ Not[b : Rep, r : \uparrow[Rep]] = \texttt{if}\ b\ \texttt{then}\ False![r]\ \texttt{else}\ True![r] \\
\quad \texttt{in}\ bool![Rep\ ;\ True, False, And, Or, Not] \\
\mid \\
\quad bool?[\\
\quad\ \ Bool\ ; \\
\quad\ \ True : [\,] \Rightarrow [Bool], \\
\quad\ \ False : [\,] \Rightarrow [Bool], \\
\quad\ \ And : [Bool, Bool] \Rightarrow [Bool], \\
\quad\ \ Or : [Bool, Bool] \Rightarrow [Bool], \\
\quad\ \ Not : [Bool] \Rightarrow [Bool] \\
\quad].P \\
)
\end{array}
$$

The channel *bool* is polymorphic: it expects to be sent some representation type *Bool*, and a collection of operations on the type *Bool*. The type of the channel *bool* is given below:

$$
\begin{array}{lll}
BoolPackage \doteq \uparrow[& & \\
\quad Bool\ ; & & \text{The representation of booleans} \\
\quad [\,] \Rightarrow [Bool], & & \text{Implementation of } True \\
\quad [\,] \Rightarrow [Bool], & & \text{Implementation of } False \\
\quad [Bool, Bool] \Rightarrow [Bool], & & \text{Implementation of } And \\
\quad [Bool, Bool] \Rightarrow [Bool], & & \text{Implementation of } Or \\
\quad [Bool] \Rightarrow [Bool] & & \text{Implementation of } Not \\
] & &
\end{array}
$$

Our aim is to hide the representation type *Rep* inside an abstract datatype, thereby ensuring that all uses of boolean values outside the abstract datatype are independent of the actual representation of booleans. Thus, instead of using our boolean process definitions directly, we define them outside the scope of P, and send them to P all together along the channel *bool* (we assume that *bool* is a fresh channel, not used in P). Operationally, the above process is equivalent to the process

$$
\begin{array}{l}
\texttt{def}\ \mathit{True}[r : \uparrow[\mathit{Rep}]] = (\nu\, b)(\, r![b] \ \mid\ *b?[t,f].t![]\,) \\
\texttt{def}\ \mathit{False}[r : \uparrow[\mathit{Rep}]] = (\nu\, b)(\, r![b] \ \mid\ *b?[t,f].f![]\,) \\
\texttt{def}\ \mathit{And}[b_1 : \mathit{Rep}, b_2 : \mathit{Rep}, r : \uparrow[\mathit{Rep}]] = \texttt{if}\ b_1\ \texttt{then}\ r![b_2]\ \texttt{else}\ \mathit{False}![r] \\
\texttt{def}\ \mathit{Or}[b_1 : \mathit{Rep}, b_2 : \mathit{Rep}, r : \uparrow[\mathit{Rep}]] = \texttt{if}\ b_1\ \texttt{then}\ \mathit{True}![r]\ \texttt{else}\ r![b_2] \\
\texttt{def}\ \mathit{Not}[b : \mathit{Rep}, r : \uparrow[\mathit{Rep}]] = \texttt{if}\ b\ \texttt{then}\ \mathit{False}![r]\ \texttt{else}\ \mathit{True}![r] \\
\texttt{in}\ \{\mathit{Rep}/\mathit{Bool}\}P
\end{array}
$$

However, the typing behaviour of the two processes is very different. In the latter process, the boolean representation *Rep* is visible in the process P. In the former, the fact that the channel *bool* is polymorphic in the type *Bool* forces P to behave independently of the actual representation of booleans.

When we first presented the encoding of booleans in π-calculus, we said that booleans are represented using channels which, when sent a pair of channels $[t, f]$, will always respond on exactly one of t and f. Until now, nothing in our type system enforced such a constraint on values of type *Bool*, since *Bool* was simply an abbreviation for the type $\uparrow[\uparrow[\,], \uparrow[\,]]$, which makes no constraints on what a process reading from such a channel does with the values it receives.

However, now that we have packaged up all our boolean operations in an abstract datatype, we can be sure that all occurrences of values of type *Bool* in P must have been constructed via some number of applications of *True*, *False*, *And*, *Or* and *Not*. It is easy to see that *True* and *False* produce processes which satifying our protocol for booleans. Similarly, assuming that their boolean arguments satisfy our protocol, *And*, *Or* and *Not* all produce well-behaved booleans. Thus, we have proved that all values of type *Bool* in P satisfy our protocol for booleans. The above reasoning is quite informal, but even so we believe that it is still very useful in practice. We leave the issue of how to formalise the above reasoning as an interesting open problem.

Note that the above reasoning relies crucially on the fact that the only way of constructing values of type *Bool* is to use the operations provided by the abstract datatype. We now see why we changed our typing rule for restriction to force the type given to a restricted name to be a channel type. If we had not made this restriction, we could easily write expressions such as $(\nu\, x : \mathit{Bool})P$, thereby breaking our invariant that every value of type *Bool* is created by an operation within the abstract datatype implementation.

Like our encoding of booleans, our channel-based reference cells from Section 5.5 use channels in a very controlled manner (the *Read* and *Update* operations preserve the invariant that at most one value is ever stored in the channel implementing the ref cell, see Section 2.11 for details). Thus, our channel-based reference cells provide another good example where representation hiding would be useful. It is certainly possible to hide the representation of *ChanRef* δ, for any given δ, using the same technique as we used for booleans. However, a much better solution would be to make the type constructor *ChanRef* abstract, and then provide polymorphic operations which work for any reference cells of type *ChanRef* α.

In order to make *ChanRef* abstract, we need to be able to send it along a channel (just as we sent *Rep* along the channel *bool*). Unfortunately, the type system presented does not allow the communication of type constructors along channels (since we do not implement high-order polymorphism). But there is no reason why we cannot add such a feature. In the Pict language [PT95b] the author, in collaboration with Benjamin Pierce, has developed a higher-order polymorphic π-calculus calculus, which enables one to communicate type constructors along channels, and thereby implement abstract datatypes for type constructors such as *ChanRef* and *List*.

5.8 Type soundness

We need to modify the π-calculus reduction semantics we gave in Section 2.2 to take account of the fact that we now communicate both type and channel arguments along channels. We need only modify the communication rule, as shown below. The rest of the reduction rules remain unchanged (the behaviour of structural congruence is also unchanged, modulo the fact that the restriction operator now contains an explicit type annotation).

Definition 5.9 (Polymorphic communication)

$$
\begin{array}{rl}
& (P + c?[\alpha_1, \ldots, \alpha_m \,;\, x_1 : \delta_1, \ldots, x_n : \delta_n].Q) \\
& \mid (c![\gamma_1, \ldots, \gamma_m \,;\, y_1, \ldots, y_n].R + S) \\
\rightarrow & \{y_1, \ldots, y_n / x_1, \ldots, x_n\}\{\gamma_1, \ldots, \gamma_m / \alpha_1, \ldots, \alpha_m\}Q \mid R
\end{array}
$$

An alternative way of specifying the behaviour of polymorphic π-terms would be to say that a polymorphic π-term P has exactly the same behaviour as its type erasure, $\mathit{erase}(P)$, defined as below.

Definition 5.10 (Type erasure)

$$
\begin{array}{rcl}
\mathit{erase}(P \mid Q) & \doteq & \mathit{erase}(P) \mid \mathit{erase}(Q) \\
\mathit{erase}((\nu\, x : \delta)P) & \doteq & (\nu\, x)\mathit{erase}(P) \\
\mathit{erase}(P + Q) & \doteq & \mathit{erase}(P) + \mathit{erase}(Q) \\
\mathit{erase}(x?[\alpha_1, \ldots, \alpha_m \,;\, y_1 : \delta_1, \ldots, y_n : \delta_n].P) & \doteq & x?[y_1, \ldots, y_n].\mathit{erase}(P) \\
\mathit{erase}(x![\delta_1, \ldots, \delta_m \,;\, y_1, \ldots, y_n].P) & \doteq & x![y_1, \ldots, y_n].\mathit{erase}(P) \\
\mathit{erase}(*P) & \doteq & *\mathit{erase}(P) \\
\mathit{erase}(\mathbf{0}) & \doteq & \mathbf{0}
\end{array}
$$

In fact, it is easy to check that the two definitions are equivalent (for well-typed terms):

Proposition 5.11 (Type erasure)

1) If $P \rightarrow Q$ then $\mathit{erase}(P) \rightarrow \mathit{erase}(Q)$.

2) If $\Delta \vdash P$ and $\mathit{erase}(P) \rightarrow R$ then there exists a Q such that $P \rightarrow Q$ and $\mathit{erase}(Q) = R$.

Proof A simple induction on the structure of P. $\square$

The above property is useful from the point of view of implementation, since it means that we need not maintain explicit type information at runtime. It is worth noting that the corresponding property is not always true of functional languages (we will have more to say about this in Chapter 6).

We also modify our definition of runtime failure, to take account of possible type argument mismatches:

Definition 5.12 (Polymorphic runtime failure)

$$\frac{m \neq m' \text{ or } n \neq n'}{\begin{array}{l}(P + c?[\alpha_1, \ldots, \alpha_m\ ;\ x_1 : \delta_1, \ldots, x_n : \delta_n].Q) \\ \mid (c![\gamma_1, \ldots, \gamma_{m'}\ ;\ y_1, \ldots, y_{n'}].R + S)\end{array}} \ \textit{fails}$$

The properties we can prove about well-typed polymorphic π-terms are essentially the same as those we proved in Section 3.8 for our monomorphic type system. For example, if $x \notin \mathit{fv}(P)$ then we can add a new type binding for x without invalidating the typing of P:

Lemma 5.13 (Weakening)

If $\Delta \vdash P$ and $x \notin \mathit{fv}(P)$ then $\Delta, x : \delta \vdash P$.

Proof A simple induction on the structure of P. □

Similarly, if $x \notin \mathit{fv}(P)$ then we can remove x's type binding without invalidating the typing of P:

Lemma 5.14 (Strengthening)

If $\Delta, x : \delta \vdash P$ and $x \notin \mathit{fv}(P)$ then $\Delta \vdash P$.

Proof A simple induction on the structure of P. □

If each x_i and y_i have the same type in the context Δ then we can simultaneously substitute $y_1, \ldots, y_n$ for $x_1, \ldots, x_n$ while preserving the type of P:

Lemma 5.15 (Substitution)

If $\Delta \vdash P$ and $\Delta(x_i) = \Delta(y_i)$ for $1 \leq i \leq n$ then $\Delta \vdash \{y_1, \ldots, y_n/x_1, \ldots, x_n\}P$.

Proof A simple induction on the structure of P. □

If P is well-typed in the context Δ, then whenever we simultaneously substitute $\tilde{\delta}$ for $\tilde{\alpha}$ in Δ and P we get a well-typed term.

Lemma 5.16 (Type substitution)

If $\Delta \vdash P$ then $\{\delta_1, \ldots, \delta_n/\alpha_1, \ldots, \alpha_n\}\Delta \vdash \{\delta_1, \ldots, \delta_n/\alpha_1, \ldots, \alpha_n\}P$.

Proof A simple induction on the structure of P. □

It is now easy to prove, using the above lemmas, that types are preserved under structural congruence.

Lemma 5.17 (Types are preserved under structural congruence)

1) If $\Delta \vdash P$ and $P \equiv Q$ then $\Delta \vdash Q$.

2) If $\Delta \vdash Q$ and $P \equiv Q$ then $\Delta \vdash P$.

Proof Similar to the proof of Lemma 3.11 □

We prove type soundness in exactly the same way as we proved type soundness for our monomorphic type system. Since only a few rules have been changed, the proofs of these theorems are very similar to those for our monomorphic type system.

Theorem 5.18 (Well-typed processes never fail)

If $\Delta \vdash P$ then not $(P\ \mathit{fails})$.

Proof Similar to the proof of Theorem 3.12 □

Theorem 5.19 (Subject reduction)

If $\Delta \vdash P$ and $P \rightarrow Q$ then $\Delta \vdash Q$.

Proof Similar to the proof of Theorem 3.13 □

Chapter 6

Relating typed λ-terms to typed π-terms

Our type system is constructed using type-theoretic techniques borrowed from the λ-calculus, so it is natural to ask if there is a precise relationship between well-typed λ-terms and well-typed π-terms. Milner [Mil90] has already shown that we can encode various λ-calculus reduction strategies in the π-calculus. We now show that the type structure of a λ-term is often preserved by these encodings. In fact, in some cases, we can even prove that the principal type of a λ-term is directly related to its encoding's principal type in the π-calculus.

Perhaps the most interesting feature of these encodings is that (in the presence of polymorphism) they don't always work! For example, we find that the Damas-Milner type system [DM82] does not always agree with our π-calculus type system as to which types a λ-term may inhabit. This might not be surprising to those familiar with ML, since it is well-known that Damas-Milner polymorphism is unsafe in the presence of side-effects [Tof88]. The π-calculus is, by its very nature, a calculus containing side-effects, so it had better not allow the same kind of polymorphism as the Damas-Milner type system.

In fact, we find that the soundness of the Damas-Milner type system is closely connected to the precise evaluation order used (a result which was recently discovered by Leroy [Ler93], though he did not use encodings into the π-calculus). We find that the call-by-value encoding of λ-calculus does not preserve its Damas-Milner type structure, but the call-by-name encoding does.

6.1 Encoding λ-terms

The syntax for λ-terms is given below. We let the expression $fv(e)$ denote the free variables of e (it is defined in the usual way).

Definition 6.1 (λ-calculus syntax)

$$
\begin{array}{rcll}
e & ::= & x & \text{Variable} \\
 & & \lambda x.e & \text{Abstraction} \\
 & & e\ e & \text{Application}
\end{array}
$$

6.1.1 Call-by-value reduction

Definition 6.2 presents Milner's encoding of the call-by-value λ-calculus reduction strategy in the polyadic π-calculus. We assume that the set of λ-calculus variables is a subset of the set of π-calculus variables (this avoids having to rename λ-calculus variables when translating λ-terms).

Definition 6.2 (Call-by-value λ-calculus encoding)

$$
\begin{array}{rcl}
[\![x]\!]a & \doteq & a![x] \\
[\![\lambda x.e]\!]a & \doteq & (\nu\, f)(a![f] \mid *f?[x,b].[\![e]\!]b) \\
[\![e\ e']\!]a & \doteq & (\nu\, b)(\nu\, c)([\![e]\!]b \mid b?[f].([\![e']\!]c \mid c?[x].f![x,a]))
\end{array}
$$

The translation of λ-terms is parameterised on an auxiliary channel a. This channel is the location where the encoded λ-term returns its result. The encoding introduces auxiliary variables (ranged over by $a, b, \ldots$) which we assume are always distinct from λ-calculus variables. The encoding has the property that $fv([\![e]\!]a) = fv(e) \cup \{a\}$ for all e.

If e is just a variable, then we just return that variable along a immediately. If e is a λ-abstraction, we first create a new channel f, which we can think of as the location of the closure $\lambda x.e$. We immediately send f along a and start the replicated process $*f?[x,b].[\![e]\!]b$. This process acts as a compute server: if we send along f a pair of an argument x and a result channel b, the server will respond by computing the value of e and returning it on b.

We evaluate an application node $e\,e'$ left-to-right: we start e running, wait for the result, f, to be sent along b, then start e' running and wait for the result, x, to be sent along c. We now have two values: a function f and the its argument x. We apply f to x by sending the pair $[x, a]$ to f. The function f will send its result along a once it is finished (recall that the result of the whole term is supposed to be sent along a).

In the following example, the function $\lambda x.x$ is already a value, so it immediately sends the channel f along b (as well as creating the replicated process $*f?[x, b].[\![x]\!]b$, which implements the function $\lambda x.x$):

$$\begin{array}{ll} & [\![(\lambda x.x)y]\!]a \\ \doteq & (\nu\, b)(\nu\, c)([\![\lambda x.x]\!]b \mid b?[f].([\![y]\!]c \mid c?[x].f![x, a])) \\ \doteq & (\nu\, b)(\nu\, c)((\nu\, f)(b![f] \mid *f?[x, b].[\![x]\!]b) \mid b?[f].([\![y]\!]c \mid c?[x].f![x, a])) \\ \rightarrow & (\nu\, b)(\nu\, c)(\nu\, f)(*f?[x, b].[\![x]\!]b \mid [\![y]\!]c \mid c?[x].f![x, a]) \end{array}$$

The process implementing the application node, now that it has received the function f along b, evaluates the function argument y. Again, y is already a value and therefore signals on its result channel c immediately:

$$\begin{array}{ll} \doteq & (\nu\, b)(\nu\, c)(\nu\, f)(*f?[x, b].[\![x]\!]b \mid c![y] \mid c?[x].f![x, a]) \\ \rightarrow & (\nu\, b)(\nu\, c)(\nu\, f)(*f?[x, b].[\![x]\!]b \mid f![y, a]) \end{array}$$

The application node now has two values: f, a channel representing the function $\lambda x.x$, and y, the function argument. It therefore sends the pair $[y, a]$ along f, causing the replicated process on f to compute the value of $\lambda x.x$ applied to y.

$$\rightarrow (\nu\, b)(\nu\, c)(\nu\, f)(*f?[x, b].[\![x]\!]b \mid [\![y]\!]a)$$

The final result is structurally congruent to the following process

$$\equiv [\![y]\!]a \mid (\nu\, b)(\nu\, c)(\nu\, f)(*f?[x, b].[\![x]\!]b)$$

and it therefore becomes clear that the replicated input on f can execute no further communications (since no other process has access to the channel f). Thus, the final result of executing $[\![(\lambda x.x)y]\!]a$ is equivalent to $[\![y]\!]a$, as expected.

6.1.2 Encoding let-expressions

The Damas-Milner typing rules [DM82] rely on explicit 'let' expressions to indicate where type generalisation is allowable. The expression 'let $x = e_1$ in e_2' is intended to have the same behaviour as $(\lambda x.e_2)\, e_1$, but this indirect interpretation of 'let' unfortunately yields a rather complex encoding of 'let' in the π-calculus. We therefore use a direct encoding (which corresponds to the optimisation of 'let' that is usually made in compilers for functional languages):

Definition 6.3 (Call-by-value let-expressions)

$$[\![\text{let } x = e \text{ in } e']\!]a \quad \doteq \quad (\nu\, b)([\![e]\!]b \mid b?[x].[\![e']\!]a)$$

We now hint how to prove that the direct encoding of 'let' is equivalent to the indirect one. We let $\sim$ and $\approx$ denote the strong and weak congruence respectively (see [San93c] for definitions of $\sim$ and $\approx$).

First, we expand out the definition of $[\![(\lambda x.e')e]\!]$:

$$\begin{array}{ll} & [\![(\lambda x.e')e]\!] \\ \doteq & (\nu\, b)(\nu\, c)([\![\lambda x.e']\!]b \mid b?[f].([\![e]\!]c \mid c?[x].f![x,a])) \\ \doteq & (\nu\, b)(\nu\, c)((\nu\, f)(b![f] \mid *f?[x,b].[\![e']\!]b) \mid b?[f].([\![e]\!]c \mid c?[x].f![x,a])) \end{array}$$

We can then execute the communication on the local channel b, yielding the following process:

$$\approx \quad (\nu\, b)(\nu\, c)(\nu\, f)(*f?[x,b].[\![e']\!]b \mid [\![e]\!]c \mid c?[x].f![x,a])$$

which can be rewritten, using structural congruence and the fact that $(\nu\, b)P \sim P$ if $b \notin fv(P)$, as follows:

$$\sim \quad (\nu\, c)([\![e]\!]c \mid (\nu\, f)(*f?[x,b].[\![e']\!]b \mid c?[x].f![x,a]))$$

The channel f and the replicated input on f can be moved inside the input prefix $c?[x].f![x,a]$, since they cannot interact with anything until the input on c has completed.

$$\sim \quad (\nu\, c)([\![e]\!]c \mid c?[x].(\nu\, f)(*f?[x,b].[\![e']\!]b \mid f![x,a]))$$

We can then execute the communication on f:

$$\approx \quad (\nu\, c)([\![e]\!]c \mid c?[x].([\![e']\!]a \mid (\nu\, f)(*f?[x,b].[\![e']\!]b))$$

yielding a process which is equivalent to $[\![\text{let } x = e \text{ in } e']\!]a$, since the channel f cannot appear in $[\![e']\!]a$.

$$\begin{array}{ll} \sim & (\nu\, c)([\![e]\!]c \mid c?[x].[\![e']\!]a) \\ \doteq & [\![\text{let } x = e \text{ in } e']\!]a \end{array}$$

6.1.3 Call-by-name reduction

Definition 6.4 presents Ostheimer and Davie's [OD93] encoding of the call-by-name λ-calculus. We use Ostheimer and Davie's encoding, rather than Milner's, since it shares much of the structure of the call-by-value encoding we have already presented and can easily be modified to implement call-by-need evaluation (where the evaluation of function arguments is shared).

We use the notation $\langle\!\langle e\rangle\!\rangle a$ to denote the call-by-name encoding of e. Just as in the call-by-value encoding, the auxiliary channel a is used to communicate the result of evaluating e. The encoding of $\lambda x.e$ is therefore just as before: we create a new channel f to represent the function $\lambda x.e$ and immediately send f along the result channel a.

Definition 6.4 (Call-by-name λ-calculus encoding)

$$\begin{array}{rcl} \langle\!\langle x\rangle\!\rangle a & \doteq & x![a] \\ \langle\!\langle \lambda x.e\rangle\!\rangle a & \doteq & (\nu\, f)(a![f] \mid *f?[x,b].\langle\!\langle e\rangle\!\rangle b) \\ \langle\!\langle e\; e'\rangle\!\rangle a & \doteq & (\nu\, b)(\nu\, x)(\langle\!\langle e\rangle\!\rangle b \mid b?[f].(f![x,a] \mid *x?[c].\langle\!\langle e'\rangle\!\rangle c)) \end{array}$$

The behaviour of an encoded application $\langle\!\langle e\; e'\rangle\!\rangle$ is as follows: We start $\langle\!\langle e\rangle\!\rangle b$ executing and then wait for it to return a function f along b. Now, instead of forcing the evaluation of the argument e', as in the call-by-value encoding, we start a new replicated process on the channel x and apply f to the argument x and result channel a. If f wishes to get the value associated with its argument x it must communicate with the replicated process on x. Whenever we send some result channel c along x, the replicated process $*x?[c].\langle\!\langle e'\rangle\!\rangle c$ will respond by starting a new copy of $\langle\!\langle e'\rangle\!\rangle c$ running (which will return its result along c).

The following example illustrates how the encoding of $(\lambda x.y)e$ is able to reduce to y without evaluating e: The function $\lambda x.y$ is already a value, so it immediately sends the channel f along b (as well as creating the replicated process $*f?[x,b].[\![y]\!]b$, which implements the function $\lambda x.y$):

$$
\begin{aligned}
&\langle\!\langle(\lambda x.y)e\rangle\!\rangle a\\
\doteq\ &(\nu\, b)(\nu\, x)(\langle\!\langle\lambda x.y\rangle\!\rangle b \mid b?[f].(f![x,a] \mid *x?[c].\langle\!\langle e\rangle\!\rangle c))\\
\doteq\ &(\nu\, b)(\nu\, x)((\nu\, f)(b![f] \mid *f?[x,b].\langle\!\langle y\rangle\!\rangle b) \mid b?[f].(f![x,a] \mid *x?[c].\langle\!\langle e\rangle\!\rangle c))\\
\rightarrow\ &(\nu\, b)(\nu\, x)(\nu\, f)(*f?[x,b].\langle\!\langle y\rangle\!\rangle b \mid f![x,a] \mid *x?[c].\langle\!\langle e\rangle\!\rangle c)
\end{aligned}
$$

The process implementing the application node, now that it has received the function f along b, starts a replicated process on the channel x and sends the pair $[x,a]$ to f:

$$\rightarrow (\nu\, b)(\nu\, x)(\nu\, f)(*f?[x,b].\langle\!\langle y\rangle\!\rangle b \mid \langle\!\langle y\rangle\!\rangle a \mid *x?[c].\langle\!\langle e\rangle\!\rangle c)$$

This has the effect of starting the process $\langle\!\langle y\rangle\!\rangle a$ executing, as required. Note that b, c and f cannot be equal to y or a (since we assume that all auxiliary variables are distinct), so the previous process is structurally congruent to

$$\equiv \langle\!\langle y\rangle\!\rangle a \mid (\nu\, b)(\nu\, x)(\nu\, f)(*f?[x,b].[\![y]\!]b \mid *x?[c].\langle\!\langle e\rangle\!\rangle c)$$

and it therefore becomes clear that neither the replicated input on f nor the replicated input on x can participate in further reductions (since no other process has access to the channels f and x). The above process is therefore equivalent to the term $\langle\!\langle y\rangle\!\rangle a$, as required.

We provide an optimised encoding of let-expressions for the call-by-name encoding, just as we did for the call-by-value encoding:

Definition 6.5 (Call-by-name let-expressions)

$$\langle\!\langle \text{let } x = e \text{ in } e'\rangle\!\rangle \quad \doteq \quad (\nu\, x)(\langle\!\langle e'\rangle\!\rangle a \mid *x?[b].\langle\!\langle e\rangle\!\rangle b)$$

6.1.4 Call-by-need reduction

It is easy to refine the previous call-by-name encoding so that it shares the evaluation of function arguments, and therefore implements call-by-need reduction (the correctness of this encoding is proved by Brock and Ostheimer [BO95]). We just replace the encoding of application in Definition 6.4 with the one below:

Definition 6.6 (Call-by-need application)

$$\begin{array}{rcl}\langle\!\langle e\,e' \rangle\!\rangle a & \doteq & (\nu\, b)(\nu\, x)(\langle\!\langle e \rangle\!\rangle b \mid b?[f].(f![x,a] \mid \langle\!\langle x := e' \rangle\!\rangle)) \\ \langle\!\langle x := e \rangle\!\rangle & \doteq & x?[c].(\nu\, d)(\langle\!\langle e \rangle\!\rangle d \mid d?[r].(c![r] \mid *x?[c].c![r]))\end{array}$$

The above encoding differs in that, rather than starting the replicated process $*x?[c].\langle\!\langle e' \rangle\!\rangle c$ on the channel x, we start a single input on the channel x. Thus, the first time we receive a signal on x (i.e. the first time the value of e' is demanded) we go ahead and evaluate e'. Once we have received r, the result of evaluating e', we return r along c. However, we also start the replicated process $*x?[c].c![r]$ so that any subsequent requests for the value of e' will be answered directly (by returning the value r, rather than re-evaluating e').

Since the encoding of application has changed, the intended semantics of 'let' has also changed. It is easy to modify the previous call-by-name encoding of 'let' so that it implements call-by-need reduction (the definition of $\langle\!\langle x := e \rangle\!\rangle$ remains the same as in Definition 6.6):

Definition 6.7 (Call-by-need let-expressions)

$$\langle\!\langle \text{let } x = e \text{ in } e' \rangle\!\rangle \quad \doteq \quad (\nu\, x)(\langle\!\langle e' \rangle\!\rangle a \mid \langle\!\langle x := e \rangle\!\rangle)$$

6.2 Encoding λ-calculus types

We use the usual notation for λ-calculus types. We assume that the set of λ-calculus type variables coincides with the set of π-calculus type variables (this avoids renaming λ-calculus type variables when translating λ-calculus types).

Definition 6.8 (λ-calculus types)

$$\begin{array}{rcll}\tau & ::= & \alpha & \text{Type variable} \\ & & \tau \to \tau & \text{Function type}\end{array}$$

A λ-calculus typing context is a (possibly empty) sequence of bindings of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$, where $x_1, \ldots, x_n$ must be distinct variables.

Definition 6.9 (λ-calculus type contexts) $\Gamma \ ::= \ x_1 : \tau_1, \ldots, x_n : \tau_n$

For reference, Definition 6.10 gives the usual typing rules for the simply-typed λ-calculus. At this stage is matters little whether we use a Church-style or a Curry-style presentation of the simply-typed λ-calculus. However, we will soon see that there is a significant difference when we come to consider polymorphic type systems.

Definition 6.10 (λ-calculus typing rules)

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau}$$

6.2.1 Call-by-value encoding

In the following definition we give an encoding of λ-calculus types as π-calculus types, written $[\![\tau]\!]$. The encoding of function types matches our representation of call-by-value functions in the π-calculus: a function is represented as a channel along which we send a pair of a value of the argument type τ and a channel capable of carrying a result of type τ'.

Definition 6.11 (Type encoding)

$$\begin{array}{rcl} [\![\alpha]\!] & \doteq & \alpha \\ [\![\tau \to \tau']\!] & \doteq & \uparrow[[\![\tau]\!], \uparrow[\![\tau']\!]] \end{array}$$

(Our encoding introduces a lot of unary channel types, so we allow types of the form $\uparrow[\delta]$ to be written as $\uparrow\delta$.) We extend our encoding to contexts, applying our encoding of types in a point-wise fashion.

Definition 6.12 (Context encoding)

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] \ \doteq \ x_1 : [\![\tau_1]\!], \ldots, x_n : [\![\tau_n]\!]$$

The following proposition proves that the type structure of the λ-calculus is preserved by the call-by-value encoding into the π-calculus. Note that the type of the auxiliary channel a is $\uparrow[\![\tau]\!]$, since a is used to return the result of e, which we know has type τ.

Proposition 6.13 (Preservation of λ-calculus type structure)
If $\Gamma \vdash e : \tau$ then $[\![\Gamma]\!], a : \uparrow[\![\tau]\!] \vdash [\![e]\!]a$.

Proof We use induction on the structure of e.

case $\Gamma \vdash x : \tau$ where $\Gamma(x) = \tau$

We have that $[\![\Gamma]\!](x) = [\![\tau]\!]$ and therefore $[\![\Gamma]\!], a : \uparrow[\![\tau]\!] \vdash a![x]$ as required.

case $\Gamma \vdash \lambda x.e : \tau' \to \tau$ where $\Gamma, x : \tau' \vdash e : \tau$

We have, using induction, that $[\![\Gamma, x : \tau']\!], b : \uparrow[\![\tau]\!] \vdash [\![e]\!]b$. Therefore, using weakening (Lemma 3.8) we have that $[\![\Gamma]\!], f : \uparrow[[\![\tau']\!], \uparrow[\![\tau]\!]], x : [\![\tau']\!], b : \uparrow[\![\tau]\!] \vdash [\![e]\!]b$. Now, using the Input and Repl rules we have that $[\![\Gamma]\!], f : \uparrow[[\![\tau']\!], \uparrow[\![\tau]\!]] \vdash *f?[x, b].[\![e]\!]b$. Using the Output rule we have that $[\![\Gamma]\!], a : \uparrow[\![\tau' \to \tau]\!], f : \uparrow[[\![\tau']\!], \uparrow[\![\tau]\!]] \vdash a![f]$, since $[\![\tau' \to \tau]\!] \doteq \uparrow[[\![\tau']\!], \uparrow[\![\tau]\!]]$. Thus, using weakening and the Prl and Res rules we have that $[\![\Gamma]\!], a : \uparrow[\![\tau' \to \tau]\!] \vdash (\nu\, f)(a![f] \mid *f?[x, b].[\![e]\!]b)$ as required.

case $\Gamma \vdash e\, e' : \tau$ where $\Gamma \vdash e : \tau' \to \tau$ and $\Gamma \vdash e' : \tau'$

We have, using induction, that $[\![\Gamma]\!], b : \uparrow[\![\tau' \to \tau]\!] \vdash [\![e]\!]b$ and $[\![\Gamma]\!], d : \uparrow[\![\tau']\!] \vdash [\![e']\!]d$. Clearly, the inputs on b and d will give f type $[\![\tau' \to \tau]\!] \doteq \uparrow[[\![\tau']\!], \uparrow[\![\tau]\!]]$ and x type $[\![\tau']\!]$. Thus, the output $f![x, a]$ is well-formed if a has type $\uparrow[\![\tau']\!]$, as required. □

The obvious question we can now ask is whether our π-calculus typing discipline admits any types for $[\![e]\!]$, other than those allowed by the λ-calculus type discipline. The following result proves that, although the translation of a λ-term $[\![e]\!]a$ may be assigned a type which does not correspond to a valid typing of e, any type assigned to $[\![e]\!]a$ contains at least as much structure as some valid λ-calculus typing for e.

To see why not all typings for encoded λ-terms are the image of some valid λ-calculus typing consider the λ-term $\lambda x.x$. There is no type τ such that $[\![\tau]\!] = \uparrow[\uparrow[\,], \uparrow(\uparrow[\,])]$, but it is easy to check that $a : \uparrow(\uparrow[\uparrow[\,], \uparrow(\uparrow[\,])]) \vdash [\![\lambda x.x]\!]a$. Thus, there are certainly π-calculus types for $[\![\lambda x.x]\!]a$ which do not correspond to the encoding of a λ-calculus type for $\lambda x.x$. However, it is the case that the aforementioned π-calculus type for $[\![\lambda x.x]\!]a$ is a substitution instance of the encoding of $\alpha \to \alpha$, which certainly is a type for $\lambda x.x$.

Thus, rather than trying to prove that every π-calculus typing for $[\![e]\!]a$ is the image of some valid λ-calculus typing for e, we prove that every π-calculus typing for $[\![e]\!]a$ is a substitution instance of a valid λ-calculus typing for e:

Proposition 6.14 (π-calculus typings reflect λ-calculus type structure)

If $\Delta, a : \delta \vdash [\![e]\!]a$ then there exist Γ, τ and σ such that $\Gamma \vdash e : \tau$, $\sigma[\![\Gamma]\!] = \Delta$ and $\sigma{\uparrow}[\![\tau]\!] = \delta$.

Proof We use induction on the structure of e.

case $\Delta, a : \delta \vdash [\![x]\!]a \doteq a![x]$

It must be the case that $\Delta = y_1 : \delta_1, \ldots, x : \delta_x, \ldots, y_n : \delta_n$ and $\delta = {\uparrow}\delta_x$. The context $\Gamma = y_1 : \alpha_1, \ldots, x : \alpha_x, \ldots, y_n : \alpha_n$, type $\tau = \alpha_x$ and substitution $\sigma = \{\alpha_1 \mapsto \delta_1, \ldots, \alpha_x \mapsto \delta_x, \ldots, \alpha_n \mapsto \delta_n\}$ give the required result, since $\Gamma \vdash x : \alpha_x$, $\sigma\Gamma = \Delta$ and $\sigma{\uparrow}[\![\alpha_x]\!] = {\uparrow}\delta_x = \delta$ as required.

case $\Delta, a : \delta \vdash [\![\lambda x.e]\!]a \doteq (\nu\, f)(a![f] \mid *f?[x, b].[\![e]\!]b)$

It must be the case that $\delta = {\uparrow}({\uparrow}[\delta_x, \delta_b])$ where $\Delta, a : \delta, f : {\uparrow}[\delta_x, \delta_b], x : \delta_x, b : \delta_b \vdash [\![e]\!]b$. However, using strengthening (Lemma 3.9), we have that $\Delta, x : \delta_x, b : \delta_b \vdash [\![e]\!]b$ and we can use induction to show that there exist Γ, τ_x, τ and σ such that $\Gamma, x : \tau_x \vdash e : \tau$, $\sigma[\![\Gamma, x : \tau_x]\!] = \Delta, x : \delta_x$ and $\sigma{\uparrow}[\![\tau]\!] = \delta_b$. Therefore, using the rule for abstraction we have that $\Gamma \vdash \lambda x.e : \tau_x \to \tau$ where $\sigma[\![\Gamma]\!] = \Delta$ and $\sigma{\uparrow}[\![\tau_x \to \tau]\!] \doteq \sigma{\uparrow}({\uparrow}[[\![\tau_x]\!], {\uparrow}[\![\tau]\!]]) = {\uparrow}({\uparrow}[\delta_x, \delta_b]) = \delta$ as required.

case $\Delta, a : \delta \vdash [\![e_1\, e_2]\!]a \doteq (\nu\, b)(\nu\, c)([\![e_1]\!]b \mid b?[f].([\![e_2]\!]c \mid c?[x].f![x, a]))$

Using strengthening (Lemma 3.9) to eliminate any unnecessary auxiliary variables, we have that $\Delta, b : \delta_b \vdash [\![e_1]\!]b$ and $\Delta, c : \delta_c \vdash [\![e_2]\!]c$ for some δ_b and δ_c. Therefore, using induction, we have that there exist Γ_1, τ_1 and σ_1 such that $\Gamma_1 \vdash e_1 : \tau_1$, $\sigma_1[\![\Gamma_1]\!] = \Delta$ and $\sigma_1{\uparrow}[\![\tau_1]\!] = \delta_b$, and there exist Γ_2, τ_2 and σ_2 such that $\Gamma_2 \vdash e_2 : \tau_2$, $\sigma_2[\![\Gamma_2]\!] = \Delta$ and $\sigma_2{\uparrow}[\![\tau_2]\!] = \delta_c$. However, it must also be the case that $\delta_b = {\uparrow}({\uparrow}[\sigma_2[\![\tau_2]\!], \delta]) = {\uparrow}\sigma_1[\![\tau_1]\!]$, because of the communications along the auxiliary channels b, c and f. It is easy to check that, since ${\uparrow}[\sigma_2[\![\tau_2]\!], \delta] = \sigma_1[\![\tau_1]\!]$ and $\sigma_1[\![\Gamma_1]\!] = \Delta = \sigma_2[\![\Gamma_2]\!]$, there exist ρ and σ such that $\rho(\tau_2 \to \alpha) = \rho\tau_1$, $\sigma[\![\rho\tau_1]\!] = \sigma_1[\![\tau_1]\!]$, $\sigma[\![\rho(\tau_2 \to \alpha)]\!] = {\uparrow}[\sigma_2[\![\tau_2]\!], \delta]$, $\rho\Gamma_1 = \rho\Gamma_2$ and $\sigma[\![\rho\Gamma_1]\!] = \Delta = \sigma[\![\rho'\Gamma_2]\!]$. Thus, since typings are preserved under substitution in the λ-calculus, we have that $\rho\Gamma_1 \vdash e_1 : \rho\tau_1$ and $\rho\Gamma_2 \vdash e_2 : \rho\tau_2$, where $\rho\tau_1 = \rho\tau_2 \to \tau$ for some τ. We therefore have that $\rho\Gamma_1 \vdash e_1\, e_2 : \tau$, $\sigma[\![\rho\Gamma_1]\!] = \Delta$ and $\sigma{\uparrow}[\![\tau]\!] = \delta$ as required. $\Box$

The previous two propositions are enough to prove that the principal type of e in the simply-typed λ-calculus coincides with the principal type of $[\![e]\!]$ in the simply-typed π-calculus:

Theorem 6.15 (Relating principal types)

1) If $\Gamma \vdash e : \tau$ is a principal typing for e, then $[\![\Gamma]\!], a : {\uparrow}[\![\tau]\!] \vdash [\![e]\!]a$ is a principal typing for $[\![e]\!]a$.

2) If $\Delta, a : \delta \vdash [\![e]\!]a$ is a principal typing for $[\![e]\!]$ then there exists a principal typing $\Gamma \vdash e : \tau$ for e such that $[\![\Gamma]\!] = \Delta$ and ${\uparrow}[\![\tau]\!] = \delta$.

Proof Part 1. Suppose that $\Gamma \vdash e : \tau$ is a principal typing for e. We have, using Proposition 6.13, that $[\![\Gamma]\!], a : {\uparrow}[\![\tau]\!] \vdash [\![e]\!]a$. This typing must be principal for $[\![e]\!]a$, since Proposition 6.14 tells us that every π-calculus typing for $[\![e]\!]a$ is a substitution instance of some λ-calculus typing for e (and we know any λ-calculus typing for e must be a substitution instance of $\Gamma \vdash e : \tau$).

Part 2. Suppose that $\Delta, a : \delta \vdash [\![e]\!]a$ is a principal typing for $[\![e]\!]a$. We have, using Proposition 6.14, that there exist Γ, τ and σ such that $\Gamma \vdash e : \tau$, $\sigma[\![\Gamma]\!] = \Delta$ and $\sigma{\uparrow}[\![\tau]\!] = \delta$. This typing must be principal for e, since Proposition 6.13 tells us that every λ-calculus typing correspond directly to a π-calculus typing for $[\![e]\!]a$ (and we know that any π-calculus typing for $[\![e]\!]a$ must be a substitution instance of $\Delta, a : \delta \vdash [\![e]\!]a$). $\Box$

6.2.2 Call-by-name encoding

We now prove that Ostheimer and Davie's encoding of the call-by-name λ-calculus preserves the type structure of λ-terms. The encoding of types is given in Definition 6.16. The encoding of function types reflects the fact that function arguments are not values, but are channels which we can use to trigger the evaluation of the given argument.

Definition 6.16 (Type encoding)

$$\begin{array}{rcl} \langle\!\langle \alpha \rangle\!\rangle & \doteq & \alpha \\ \langle\!\langle \tau \to \tau' \rangle\!\rangle & \doteq & {\uparrow}[{\uparrow\uparrow}\langle\!\langle \tau \rangle\!\rangle, {\uparrow}\langle\!\langle \tau' \rangle\!\rangle] \end{array}$$

As before, we extend our encoding to contexts. However, unlike the call-by-value encoding of contexts, we do not apply our encoding of types in a point-wise fashion, since each free variable x_i in an encoded term no longer ranges over values, but over channels which we can use to trigger the evaluation of the expression bound to x_i.

Definition 6.17 (Context encoding)

$$\langle\!\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle\!\rangle \quad \doteq \quad x_1 : \uparrow\uparrow\langle\!\langle \tau_1 \rangle\!\rangle, \ldots, x_n : \uparrow\uparrow\langle\!\langle \tau_n \rangle\!\rangle$$

The following proposition proves that the type structure of the λ-calculus is preserved by the call-by-name encoding into the π-calculus. (The same proposition holds if we replace the call-by-name encoding of application with the call-by-need encoding.)

Proposition 6.18 (Preservation of λ-calculus type structure)
If $\Gamma \vdash e : \tau$ then $\langle\!\langle \Gamma \rangle\!\rangle, a : \uparrow\langle\!\langle \tau \rangle\!\rangle \vdash \langle\!\langle e \rangle\!\rangle a$.

Proof Similar to proof of Proposition 6.13. □

As before, it is not the case that every π-calculus typing of $\langle\!\langle e \rangle\!\rangle a$ is equal to the encoding of some λ-calculus typing for e. In Proposition 6.14 we got around this problem by proving that every π-calculus typing is a substitution instance of some encoded λ-calculus typing. Unfortunately, this trick does not work for the call-by-name encoding. For example, since the variable x is not a free variable of the expression $\lambda y.y$, it may be assigned any type by our π-calculus type system:

$$x : \delta, a : \uparrow\langle\!\langle \beta \to \beta \rangle\!\rangle \vdash \langle\!\langle \lambda y.y \rangle\!\rangle a$$

In the above example, the type δ is unconstrained, but our encoding of contexts requires that every variable bound in the context has a type of the form $\uparrow\uparrow\langle\!\langle \tau \rangle\!\rangle$. In fact, the type δ may be more general that $\uparrow\uparrow\langle\!\langle \tau \rangle\!\rangle$, and we have a counter-example to the call-by-name equivalent of Theorem 6.15.

The above problem with unused variables may in turn cause λ-abstractions to have types which are too general. In the following example, the type δ, which corresponds to the type of the bound variable y, is unconstrained by our π-calculus type system (and may therefore cause the type of a to fail to be an instance of a λ-calculus type):

$$x : \uparrow\uparrow\alpha, a : \uparrow(\uparrow[\delta, \uparrow\alpha]) \vdash \langle\langle \lambda y.x \rangle\rangle a$$

We therefore prove a slightly weaker result about π-calculus typings of call-by-name encoded terms. We prove that if all encoded λ-calculus variables are constrained to have a type which is the encoding of some λ-calculus type then the π-calculus typing for $\langle\langle e \rangle\rangle a$ is equal to some λ-calculus typing for e. Note that the previous condition on variables is a global one, it is not sufficient to constrain just the top-level free variables of an encoded term.

Proposition 6.19 (π-calculus typings reflect λ-calculus type structure)

If all λ-calculus variables in $\langle\langle e \rangle\rangle a$ are assigned a type of the form $\uparrow\uparrow\langle\langle \tau \rangle\rangle$, for some τ, then $\langle\langle \Gamma \rangle\rangle, a : \delta \vdash \langle\langle e \rangle\rangle a$ implies there exists a τ such that $\Gamma \vdash e : \tau$ and $\uparrow\langle\langle \tau \rangle\rangle = \delta$.

Proof A simple induction on the structure of e. □

Note that Propositions 6.18 and 6.19 remain true if we replace the call-by-name encoding of application with the call-by-need encoding. Thus, in the case of simply-typed λ-calculus, there is no distinguishable difference between the π-calculus typing of the call-by-name and call-by-need encodings.

6.3 Encoding recursive types

We can extend both our call-by-value and call-by-name encodings of λ-calculus types to encompass recursive types. We conjecture that results similar to those in the previous section can be proved when we have recursive types in both the λ-calculus and the π-calculus.

Definition 6.20 (Encoding recursive types)

$$[\![\mu\alpha.\tau]\!] \doteq \mu\alpha.[\![\tau]\!] \qquad\qquad \langle\langle \mu\alpha.\tau \rangle\rangle \doteq \mu\alpha.\langle\langle \tau \rangle\rangle$$

It is well-known that every λ-term inhabits the type $\mu\alpha.(\alpha \to \alpha)$, but it is interesting to note that $[\![\mu\alpha.(\alpha \to \alpha)]\!] \doteq \mu\alpha.\uparrow[\alpha, \uparrow\alpha]$, since in [Mil91a] Milner shows that every call-by-value λ-term inhabits the type $\mu\alpha.\uparrow[\alpha, \uparrow\alpha]$ (we have rewritten

Milner's type in our notation). In fact, $\mu\alpha.(\alpha \to \alpha)$ is the least informative recursive type we can give to a λ-term: the previous propositions prove that the type structure of an encoded λ-term in fact contains a much more accurate reflection of its original λ-calculus type structure.

6.4 Encoding polymorphic λ-terms

We first consider how to encode the explicitly-typed polymorphic λ-calculus of Girard and Reynolds [Gir72, Rey74], since having explicit term syntax for type abstraction and type application clarifies some of the semantic issues we encounter when encoding polymorphic λ-terms in the π-calculus. The syntax for terms is as follows:

Definition 6.21 (Polymorphic λ-terms)

e	$::=$	x	Variable
		$\lambda x{:}\tau.e$	Abstraction
		$e\ e$	Application
		$\Lambda\alpha.e$	Type abstraction
		$e\ [\tau]$	Type application

The syntax for polymorphic types is given below. Unlike the Damas-Milner type system, there are no restrictions on the positions where a polymorphic type may occur.

Definition 6.22 (Polymorphic types)

τ	$::=$	α	Type variable
		$\tau \to \tau$	Function type
		$\forall\alpha.\tau$	Polymorphic type

The polymorphic typing rules for variables and application are just the same as in the simply-typed λ-calculus. The typing rule for λ-abstraction is slightly different, since we now how an explicit type annotation on the bound variable x. We also have additional rules for type abstraction and type application:

Definition 6.23 (Polymorphic typing rules)

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\,e' : \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \alpha \notin \Gamma \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau'}{\Gamma \vdash e\,[\tau] : \{\tau/\alpha\}\tau'}$$

The call-by-value and call-by-name encodings of variables, λ-abstraction and application remain as before, except that we now translate the explicit type annotations on λ-bound variables to explicit π-calculus type annotations (the call-by-need encoding can be similarly modified):

Definition 6.24 (Encoding explicitly-typed λ-terms)

$$\begin{array}{rcl}
[\![x]\!]a & \doteq & a![x] \\
[\![\lambda x{:}\tau.e]\!]a & \doteq & (\nu\, f)(a![f] \mid *f?[x : [\![\tau]\!], b].[\![e]\!]b) \\
[\![e\,e']\!]a & \doteq & (\nu\, b)(\nu\, c)([\![e]\!]b \mid b?[f].([\![e']\!]c \mid c?[x].f![x, a])) \\
\\
\langle\!\langle x \rangle\!\rangle a & \doteq & x![a] \\
\langle\!\langle \lambda x{:}\tau.e \rangle\!\rangle a & \doteq & (\nu\, f)(a![f] \mid *f?[x : \uparrow\uparrow\langle\!\langle \tau \rangle\!\rangle, b].\langle\!\langle e \rangle\!\rangle b) \\
\langle\!\langle e\,e' \rangle\!\rangle a & \doteq & (\nu\, b)(\nu\, x)(\langle\!\langle e \rangle\!\rangle b \mid b?[f].(f![x, a] \mid *x?[c].\langle\!\langle e' \rangle\!\rangle c))
\end{array}$$

The reduction behaviour of type applications is the same for call-by-value, call-by-name and call-by-need evaluation (since the type argument in a type application never needs to be evaluated). The following encodings of type abstraction and type application ensure that $(\Lambda\alpha.e)\,[\tau]$ reduces to $\{\tau/\alpha\}e$, as required. Note that the call-by-value and call-by-name encodings only differ in the way they encode the explicit type argument τ (the call-by-need encoding is identical to the call-by-name encoding).

Definition 6.25 (Encoding type abstraction and application)

$$\begin{array}{rcl}
[\![\Lambda\alpha.e]\!]a & \doteq & (\nu\, f)(a![f] \mid *f?[\alpha; b].[\![e]\!]b) \\
\langle\!\langle \Lambda\alpha.e \rangle\!\rangle a & \doteq & (\nu\, f)(a![f] \mid *f?[\alpha; b].\langle\!\langle e \rangle\!\rangle b) \\
\\
[\![e\,[\tau]]\!]a & \doteq & (\nu\, b)([\![e]\!]b \mid b?[f].f![[\![\tau]\!]; a]) \\
\langle\!\langle e\,[\tau] \rangle\!\rangle a & \doteq & (\nu\, b)(\langle\!\langle e \rangle\!\rangle b \mid b?[f].f![\langle\!\langle \tau \rangle\!\rangle; a])
\end{array}$$

The encoding of type abstraction is similar to the encoding of λ-abstraction, except that the abstracted variable is now a type variable. Thus, when we send a pair of a type τ and a result channel b to f, the encoding returns along b the value of e, instantiated at the type τ. The encoding of type application is similar to the encoding of application in that we evaluate the expression e, wait for it to return some type function f, and then apply f to the type argument τ. There is no need to evaluate the type argument τ itself. For example,

$$\begin{array}{ll} & [\![(\Lambda\alpha.e)\ [\tau]]\!] \\ \doteq & (\nu\, b)([\![\Lambda\alpha.e]\!]b \mid b?[f].f![[\![\tau]\!];a]) \\ \doteq & (\nu\, b)((\nu\, f)(b![f] \mid *f?[\alpha;c].[\![e]\!]c) \mid b?[f].f![[\![\tau]\!];a]) \\ \rightarrow & (\nu\, b)(\nu\, f)(*f?[\alpha;c].[\![e]\!]c \mid f![[\![\tau]\!];a]) \\ \rightarrow & (\nu\, b)(\nu\, f)(*f?[\alpha;c].[\![e]\!]c \mid \{[\![\tau]\!]/\alpha\}[\![e]\!]a) \end{array}$$

The above process is equivalent to $\{[\![\tau]\!]/\alpha\}[\![e]\!]a$, since the channels f and b are unused in $\{[\![\tau]\!]/\alpha\}[\![e]\!]a$. It is then easy to check that $\{[\![\tau]\!]/\alpha\}[\![e]\!]a = [\![\{\tau/\alpha\}e]\!]a$, as required.

Now that we have seen the encodings of type abstraction and type application, we can see how we should encode the type $\forall\alpha.\tau$ in the π-calculus, since in all encodings a value of type $\forall\alpha.\tau$ is represented as a channel along which we can send a pair of a type and a result channel (the polymorphic term will then respond by returning an appropriately instantiated value along the result channel). (The encodings of function types and type variables are unchanged, but we reproduce them here for ease of reference.)

Definition 6.26 (Encoding polymorphic types)

$$\begin{array}{rcl@{\qquad\qquad}rcl} [\![\alpha]\!] & \doteq & \alpha & \langle\!\langle\alpha\rangle\!\rangle & \doteq & \alpha \\ [\![\tau \rightarrow \tau']\!] & \doteq & \uparrow[[\![\tau]\!], \uparrow[\![\tau']\!]] & \langle\!\langle\tau \rightarrow \tau'\rangle\!\rangle & \doteq & \uparrow[\uparrow\uparrow\langle\!\langle\tau\rangle\!\rangle, \uparrow\langle\!\langle\tau'\rangle\!\rangle] \\ [\![\forall\alpha.\tau]\!] & \doteq & \uparrow[\alpha; \uparrow[\![\tau]\!]] & \langle\!\langle\forall\alpha.\tau\rangle\!\rangle & \doteq & \uparrow[\alpha; \uparrow\langle\!\langle\tau\rangle\!\rangle] \end{array}$$

Strictly speaking, the above encoding translates an explicitly-typed λ-term into a partially-typed π-term (since, for example, we do not give explicit types for f and b in the encoding of λ-abstraction). However, it turns out that all the missing type information is uniquely determined by the explicit type information already present in the encoded term. This is not surprising, since the missing

π-calculus type information is completely determined by the result type of the λ-term, and it is well-known that the λ-terms we are encoding have unique result types (if they are typable).

We write $P \sqsupseteq Q$ whenever the type erasures of P and Q are equal and P contains more explicit type information than Q. The following proposition proves that if e is typable then there exists a fully-typed version of $[\![e]\!]a$ which is typable in the π-calculus:

Proposition 6.27 (Preservation of λ-calculus type structure)

If $\Gamma \vdash e : \tau$ then there exists a P such that $[\![\Gamma]\!], a : \uparrow[\![\tau]\!] \vdash P$ and $P \sqsupseteq [\![e]\!]a$.

Proof We use induction on the structure of e. Most cases are similar to those in the proof of Proposition 6.13, so we only show the new cases.

case $\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau$ where $\Gamma \vdash e : \tau$ and $\alpha \notin \Gamma$

We have, by induction, that there exists a $P \sqsupseteq [\![e]\!]b$ such that $[\![\Gamma]\!], b : \uparrow[\![\tau]\!] \vdash P$. Thus, using the Input rule $[\![\Gamma]\!], f : \uparrow[\alpha; \uparrow[\![\tau]\!]] \vdash f?[\alpha; b : \uparrow[\![\tau]\!]].P$ since $\alpha \notin fv([\![\Gamma]\!])$. It is now easy to prove that $[\![\Gamma]\!], a : \uparrow[\![\forall\alpha.\tau]\!] \vdash (\nu\, f : [\![\forall\alpha.\tau]\!])(a![f] \mid *f?[\alpha; b : \uparrow[\![\tau]\!]].P)$ as required.

case $\Gamma \vdash e\,[\tau] : \{\tau/\alpha\}\tau'$ where $\Gamma \vdash e : \forall\alpha.\tau'$

We have, by induction, that there exists a $P \sqsupseteq [\![e]\!]b$ where $[\![\Gamma]\!], b : \uparrow[\![\forall\alpha.\tau']\!] \vdash P$. Now it is easy to see that $[\![\Gamma]\!], b : \uparrow(\uparrow[\alpha; \uparrow[\![\tau']\!]]), a : \uparrow(\{[\![\tau]\!]/\alpha\}[\![\tau']\!]) \vdash b?[f : [\![\forall\alpha.\tau']\!]].f![[\![\tau]\!]; a]$ and the result follows since $[\![\forall\alpha.\tau']\!] \doteq \uparrow[\alpha; \uparrow[\![\tau']\!]]$ and $[\![\{\tau/\alpha\}\tau']\!] = \{[\![\tau]\!]/\alpha\}[\![\tau']\!]$. □

The next proposition proves that if any fully-typed version of $[\![e]\!]a$ is typable in the π-calculus then e is typable in the λ-calculus. Note that, unlike Proposition 6.19, we need not make a global restriction on the types assigned to λ-calculus variables, since the explicit type annotations present in an encoded term already make the same restriction.

Proposition 6.28 (π-calculus typings reflect λ-calculus type structure)

If there exists a P such that $P \sqsupseteq [\![e]\!]a$ and $[\![\Gamma]\!], a : \delta \vdash P$ then there exists a τ such that $\delta = \uparrow[\![\tau]\!]$ and $\Gamma \vdash e : \tau$.

Proof We use induction on the structure of e. We omit the cases for variables and application they are straightforward.

case $[\![\Gamma]\!], a : \delta \vdash [\![\lambda x{:}\tau.e]\!]a \doteq (\nu\, f)(a![f] \mid *f?[x : [\![\tau]\!]; b].[\![e]\!]b)$

It must be the case that $[\![\Gamma]\!], x : [\![\tau]\!], b : \delta' \vdash P$ for some $P \sqsupseteq [\![e]\!]b$ and δ' (we can ignore the bindings for a and f since they do not occur in $[\![e]\!]b$). Hence, using induction, we have that $\Gamma, x : \tau \vdash e : \tau'$ for some τ' such that $\delta' = \uparrow[\![\tau']\!]$. We therefore have that the type of f is $[\![\tau \to \tau']\!]$ and the result follows, since $\delta = \uparrow[\![\tau \to \tau']\!]$.

case $[\![\Gamma]\!], a : \delta \vdash [\![\Lambda\alpha.e]\!]a \doteq (\nu\, f)(a![f] \mid *f?[\alpha; b].[\![e]\!]b)$

It must be the case that $[\![\Gamma]\!], b : \delta' \vdash P$ for some $P \sqsupseteq [\![e]\!]b$ and δ' (we can ignore the bindings for a and f since they do not occur in $[\![e]\!]b$). Hence, using induction, we have that $\Gamma \vdash e : \tau'$ for some τ' such that $\delta' = \uparrow[\![\tau']\!]$. It must be the case that $\alpha \notin \mathit{fv}([\![\Gamma]\!])$ and hence also $\alpha \notin \mathit{fv}(\Gamma)$. Thus, we have $\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau'$ and $\delta = \uparrow\uparrow[\alpha; \uparrow[\![\tau']\!]] = \uparrow[\![\forall\alpha.\tau']\!]$ as required.

case $[\![\Gamma]\!], a : \delta \vdash [\![e\,[\tau]]\!]a \doteq (\nu\, b)([\![e]\!]b \mid b?[f].f![[\![\tau]\!]; a])$

It must be the case $[\![\Gamma]\!], b : \delta' \vdash P$ for some $P \sqsupseteq [\![e]\!]b$ so, using induction, there exists a τ' such that $\Gamma \vdash e : \tau'$ and $\delta' = \uparrow[\![\tau']\!]$. Now, it must be the case that $\delta' = \uparrow\uparrow[\alpha; \delta'']$ for some δ'' such that $\{[\![\tau]\!]/\alpha\}\delta'' = \delta$. Hence it must be the case that $\tau' = \forall\alpha.\tau''$. Thus, we have that $\Gamma \vdash e\,[\tau] : \{\tau/\alpha\}\tau''$ and $\delta = \{[\![\tau]\!]/\alpha\}\uparrow[\![\tau'']\!] = \uparrow[\![\{\tau/\alpha\}\tau'']\!]$ as required. □

Similar results hold for the call-by-name and call-by-need encodings of polymorphic λ-terms since the encoding of type abstraction and application is essentially the same as above:

Proposition 6.29 (Preservation of λ-calculus type structure)
If $\Gamma \vdash e : \tau$ then there exists a P such that $\langle\!\langle\Gamma\rangle\!\rangle, a : \uparrow\langle\!\langle\tau\rangle\!\rangle \vdash P$ and $P \sqsupseteq \langle\!\langle e\rangle\!\rangle a$.

Proof Similar to the proof of Proposition 6.27. □

Proposition 6.30 (π-calculus typings reflect λ-calculus type structure)

If there exists a P such that $P \sqsupseteq \langle\!\langle e\rangle\!\rangle a$ and $\langle\!\langle\Gamma\rangle\!\rangle, a : \delta \vdash P$ then there exists a τ such that $\delta = \uparrow\langle\!\langle\tau\rangle\!\rangle$ and $\Gamma \vdash e : \tau$.

Proof Similar to the proof of Proposition 6.28. □

Note that the interpretation of λ-terms given here depends crucially on the fact that type abstraction and type application have real computational content.

Each instance of a polymorphic value $\Lambda\alpha.e$ is completely separate from all other instances (since each time we instantiate α with some type τ our encoding recomputes the value of e). We will see in the next section that whenever our encoding fails to have this property we run into trouble encoding polymorphic types.

6.5 Damas-Milner polymorphism

The Damas-Milner type system [DM82] relies on let-expressions to indicate where type generalisation is allowable, rather than using explicit type abstraction and type application constructs.

The types used in the Damas-Milner type system have the same syntax as those used in simply-typed λ-calculus, but typing contexts are generalised so that variables may be bound to type schemes of the form $\forall\tilde{\alpha}.\tau$. (We write $x : \tau$ whenever the variable x is bound to a type scheme which has no type quantifiers.)

Definition 6.31 (Damas-Milner type contexts)

$$\Gamma \quad ::= \quad x_1 : \forall\tilde{\alpha}_1.\tau_1, \ldots, x_n : \forall\tilde{\alpha}_n.\tau_n \qquad \text{Type context}$$

The following typing rules implement the Damas-Milner type system (the rules for abstraction and application are the same as for the simply-typed λ-calculus).

Definition 6.32 (Damas-Milner typing rules)

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau}$$

$$\frac{\Gamma(x) = \forall\tilde{\alpha}.\tau}{\Gamma \vdash x : \{\tilde{\tau}/\tilde{\alpha}\}\tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \forall\tilde{\alpha}.\tau \vdash e' : \tau'}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau'}\ \tilde{\alpha} \notin \Gamma$$

6.5.1 Call-by-name evaluation

The call-by-name encoding of let-expressions recomputes e every time the value of x is requested (the process expression $*x?[b].\langle\!\langle e\rangle\!\rangle b$ responds to each request for the value of x by starting a new copy of $[\![e]\!]b$ running, which will eventually send its result along b):

$$\langle\!\langle \text{let } x = e \text{ in } e' \rangle\!\rangle \quad \doteq \quad (\nu\, x)(\langle\!\langle e' \rangle\!\rangle a \mid *x?[b].\langle\!\langle e \rangle\!\rangle b)$$

In the polymorphic π-calculus we can extend this interaction so that, instead of just waiting for a result channel b to be sent along x, we wait for a tuple of type arguments $\tilde{\alpha}$ to be sent along x. We can then respond by recomputing the value of e, instantiated at the given types:

$$\langle\!\langle \text{let } x = e \text{ in } e' \rangle\!\rangle \quad \doteq \quad (\nu\, x)(\langle\!\langle e' \rangle\!\rangle a \mid *x?[\tilde{\alpha}\,;\, b].\langle\!\langle e \rangle\!\rangle b)$$

This means that each use of x (i.e. each occurrence of the variable x in the expression e') can now specify that x should be instantiated with some given types (this neatly matches the Damas-Milner typing rule for variables). Thus, just as in the previous section, we find that type instantiation is closely coupled to recomputation, since each time we instantiate x we start a new copy of e executing.

We now show formally how one can encode the Damas-Milner type system in the polymorphic π-calculus (assuming a call-by-name reduction strategy). The encoding of types remains as before (cf. Definition 6.16), but we generalise our encoding of contexts to take account of the fact that variables may now be bound to type schemes:

Definition 6.33 (Context encoding)

$$\langle\!\langle x_1 : \forall\tilde{\alpha}_1.\tau_1, \ldots, x_n : \forall\tilde{\alpha}_n.\tau_n \rangle\!\rangle \quad \doteq \quad x_1 : \uparrow[\tilde{\alpha}_1; \uparrow\langle\!\langle \tau_1 \rangle\!\rangle], \ldots, x_n : \uparrow[\tilde{\alpha}_n; \uparrow\langle\!\langle \tau_n \rangle\!\rangle]$$

The encoding of $\forall\tilde{\alpha}_i.\tau_i$ as $\uparrow[\tilde{\alpha}_i; \uparrow\langle\!\langle \tau_i \rangle\!\rangle]$ captures the fact that we access each variable x_i by sending a tuple of types and a result channel along x_i. The process implementing x_i responds by instantiating the expression bound to x_i with the given types, evaluating the expression, and returning the final result along the given result channel. Note that in the case where x_i is monomorphic (i.e. $\tilde{\alpha}_i$ is the empty sequence) we get exactly the same encoding as we had before (cf. Definition 6.17).

The following proposition proves that λ-terms which are typable in the Damas-Milner type system can be encoded as well-typed polymorphic π-terms.

Proposition 6.34 (Preservation of λ-calculus type structure)
If $\Gamma \vdash e : \tau$ then there exists a P such that $\langle\!\langle \Gamma \rangle\!\rangle, a : \uparrow\langle\!\langle \tau \rangle\!\rangle \vdash P$ and $P \sqsupseteq \langle\!\langle e \rangle\!\rangle a$.

Proof We use induction on the structure of e. We omit the cases for abstraction and application, since they are similar to those in the proof of Proposition 6.13.

case $\Gamma \vdash x : \{\tilde{\tau}/\tilde{\alpha}\}$ where $\Gamma(x) = \forall\tilde{\alpha}.\tau$

We have that $\langle\!\langle \Gamma \rangle\!\rangle(x) = \uparrow[\tilde{\alpha}; \uparrow\langle\!\langle \tau \rangle\!\rangle]$. Thus, since $\{\langle\!\langle \tilde{\tau} \rangle\!\rangle/\tilde{\alpha}\}\langle\!\langle \tau \rangle\!\rangle = \langle\!\langle \{\tilde{\tau}/\tilde{\alpha}\}\tau \rangle\!\rangle$ we have that $\langle\!\langle \Gamma \rangle\!\rangle, a : \uparrow\langle\!\langle \{\tilde{\tau}/\tilde{\alpha}\}\tau \rangle\!\rangle \vdash x![\langle\!\langle \tilde{\tau} \rangle\!\rangle \; ; \; a]$ as required.

case $\Gamma \vdash \text{let } x = e \text{ in } e' : \tau'$ where $\Gamma \vdash e : \tau$, $\Gamma, x : \forall\tilde{\alpha}.\tau \vdash e' : \tau'$ and $\tilde{\alpha} \notin \Gamma$

Using induction twice we have that there exist P and Q such that $\langle\!\langle \Gamma \rangle\!\rangle, b : \uparrow\langle\!\langle \tau \rangle\!\rangle \vdash P$, $P \sqsupseteq \langle\!\langle e \rangle\!\rangle b$, $\langle\!\langle \Gamma, x : \forall\tilde{\alpha}.\tau \rangle\!\rangle, a : \uparrow\langle\!\langle \tau' \rangle\!\rangle \vdash Q$ and $Q \sqsupseteq \langle\!\langle e' \rangle\!\rangle a$. It is then easy to see that the input $x?[\tilde{\alpha} \; ; \; b : \uparrow\langle\!\langle \tau \rangle\!\rangle].P$ is well-formed, and the result follows. $\square$

Leroy [Ler93] has already shown that by taking a call-by-name semantics, the Damas-Milner type system can be proved sound for a language containing imperative features such as reference cells or exceptions (it is well known that the Damas-Milner type system is unsound if such a language has a call-by-value reduction semantics [Tof88]). (Harper and Lillibridge [HL92] consider similar issues in their study of the typing properties of CPS conversion for an extension of F_ω with control operators.) The π-calculus can encode stateful computation, so the fact that call-by-name terms are well-typed in the π-calculus confirms Leroy's observation. Note that since only let-bound expressions may be polymorphic, the above proposition remains true even if we evaluate function arguments strictly.

Much as before, we can prove that if all λ-calculus variables in $\langle\!\langle e \rangle\!\rangle a$ are assigned a type of the form $\uparrow[\tilde{\alpha}; \uparrow\langle\!\langle \tau \rangle\!\rangle]$, for some $\tilde{\alpha}$ and τ, then every π-calculus typing of $\langle\!\langle e \rangle\!\rangle a$ is equal to the encoding of some Damas-Milner typing for e.

Proposition 6.35 (π-calculus typings reflect λ-calculus type structure)

If $P \sqsupseteq \langle\!\langle e \rangle\!\rangle a$ and all λ-calculus variables in P are assigned a type of the form $\uparrow[\tilde{\alpha}; \uparrow\langle\!\langle \tau \rangle\!\rangle]$, for some $\tilde{\alpha}$ and τ, then $\langle\!\langle \Gamma \rangle\!\rangle, a : \delta \vdash P$ implies there exists a τ such that $\Gamma \vdash e : \tau$ and $\uparrow\langle\!\langle \tau \rangle\!\rangle = \delta$.

Proof A simple induction on the structure of e. $\square$

6.5.2 Call-by-value evaluation

If we choose to evaluate let-expressions using a call-by-value semantics, we run into trouble encoding some terms which are typable in the Damas-Milner type system. The problem stems from the fact that the encoding of the expression e in the call-by-value encoding of 'let' (reproduced below) is evaluated exactly once, rather than every time the value x is used, as is the case in the call-by-name encoding.

$$[\![\text{let } x = e \text{ in } e']\!]a \quad \doteq \quad (\nu\, b)([\![e]\!]b \mid b?[x].[\![e']\!]a)$$

The Damas-Milner typing rule for 'let' allows the type of x to be generalised (assuming that the types we are generalising do not occur free in the type environment). However, our π-calculus typing rules do not allow the type of x (in the encoded let-expression) to be generalised (for reasons we explained in Chapter 5).

The Damas-Milner type system is sound for a call-by-value λ-calculus, so why don't our polymorphic π-calculus typing rules allow x to be polymorphic? The problem stems from the fact that π-calculus processes may interact with each other in more ways that pure functions may interact with each other in the call-by-value λ-calculus. The typing rules for π-calculus must therefore be more conservative about where types may be generalised. There is nothing in our type system which identifies 'better behaved' processes (such as encoded λ-terms), so there is no easy way to allow the type of x to be generalised.

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:

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

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)

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

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.

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

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

case Inp-R: $H,\ c?[\tilde{x}].P :: R \rightarrow H\{c \mapsto rs :: ?[\tilde{x}].P\},\ R$ 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*: $H,\ c![\tilde{a}].P :: R \rightarrow H\{c \mapsto rs :: ?{*}[\tilde{x}].Q\},\ P :: R :: \{\tilde{a}/\tilde{x}\}Q$ 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 $c?{*}[\tilde{x}].Q \mid c![\tilde{a}].P \rightarrow c?{*}[\tilde{x}].Q \mid P \mid \{\tilde{a}/\tilde{x}\}Q$ and the result follows. □

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:

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

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. $\Box$

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):

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

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:

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

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

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

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)

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

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)

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

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.

Chapter 8

Compiling Pict to C: Design

The primary motivation of the Pict [PT95b] project was to design and implement a high-level concurrent language purely in terms of π-calculus primitives. There have been many proposals for concurrent languages [Car86, Hol83, Rep92, Mat91, GMP89, etc.] which include communication primitives which are very similar to those of the π-calculus. However, to our knowledge, none have proposed using π-calculus primitives as the sole mechanism of computation.

The Pict language consists of two layers: a very simple core calculus (which is just asynchronous π-calculus extended with built-in structured data), and a high-level language which is defined via translation into the core calculus. This yields a very compact formal definition (the core language type system can be presented in four pages, the operational semantics in one page, the derived forms in three pages and the derived typing rules in two pages). Moreover, this means that Pict programs can be compiled in the same way as they are formally specified (first translate the high-level Pict program into the core calculus, and then compile the core calculus).

The efficiency of Pict therefore relies exclusively on the efficient compilation of channel-based communication. Compiling such a language poses a number of challenges to the implementor:

Process creation: Very large numbers of processes are created during execution (processes are created at least as frequently as functions are called in a functional language), so process creation must be very fast, and must consume very little memory.

Process scheduling: Whenever a process becomes blocked, it is necessary to preserve the current state of the process, and then remove it from the run-queue. Since Pict processes tend to run for a very short period of time before blocking, it is important to ensure that such context switches can be executed very quickly. Moreover, since there will be many blocked processes in the system, some of which may remain blocked for a long time, we must ensure that a blocked process is represented efficiently in memory.

Channel-based communication: The protocol required to implement channel-based communication is rather expensive, both in terms of code size and execution time. These costs arise because every channel may be in one of three possible states: empty, containing blocked readers, or containing blocked writers. The code for each input or output operation in a process must be able to deal with all of these possible channel states. Fortunately, the status of a channel is, in many cases, known at compile-time, enabling us to specialise the code for communication and sometimes even avoiding testing the status of a channel at all.

Channel representation: A channel may, in general, contain an arbitrary number of blocked readers or writers. However, it turns out that a large percentage of channels only ever contain at most one reader or writer. In fact, if the compiler has access to linear type information [KPT96], we can even *guarantee* that certain channels will contain at most one reader or writer. We therefore optimise our channel representation for this case (enabling a more compact channel representation and a faster implementation of communication). We pay a small additional cost in space and time in the case where a channel must hold more than one reader or writer.

We now describe our compilation of core Pict into C. We compile to C, instead of native code, since it allows us to generate efficient code without sacrificing portability (though we do incur slightly increased compilation times and some loss of efficiency). An additional benefit of this approach is that we can easily allow C code to be embedded inside Pict programs, enabling one to make use of the extensive operating system and library functions already available in C. For example, the author, in collaboration with Benjamin Pierce, has used this feature to develop an X-windows interface which can be controlled by Pict processes.

This chapter describes the Pict core language and the decisions we made when choosing representations for Pict data. (We don't describe some of Pict's built-in datatypes, such as records and tuples, since their implementation is standard.) Chapter 9 describes the actual compilation of Pict to C.

8.1 Source language

In Chapter 7 we described and motivated a number of simplifications to the π-calculus (such as omitting summation, using asynchronous communication and disallowing the general replication operator) which significantly simplify its implementation. Pict's core language incorporates those same simplifications, but also makes two extensions to the source language proposed in Chapter 7.

Firstly, we include integers, booleans and conditional expressions as primitives, since it is not feasible to use encodings to implement such important datatypes. Moreover, by compiling integer and boolean operations into the corresponding operations provided by C, we give the C compiler a reasonable chance of optimising them, and avoid having to reimplement all the 'standard' optimisations of arithmetic and boolean operations in our Pict compiler.

Secondly, we allow C code to be embedded in π-terms. This allows easy access to the operations and libraries available in C. In fact, in the Pict programming language this feature is available to the programmer (not just the compiler writer).

We let a range over atomic values: variables, integers and booleans. Atomic values are a generalisation of channel values (which are the only kind of atomic value we have in the pure polyadic π-calculus).

Definition 8.1 (Atomic values)

$$
\begin{array}{lcll}
a & ::= & x & \text{Variable} \\
  &   & 0, 1, 2, \ldots & \text{Integer constant} \\
  &   & \mathrm{true}, \mathrm{false} & \text{Boolean constant}
\end{array}
$$

The syntax for processes is given below. Note that we may now send arbitrary atoms along channels, rather than just channel names. (There is no need for the subject of a communication to be an arbitrary atom, since integer and boolean values may not be used for communication.)

Definition 8.2 (Process syntax)

P	$::=$	$x![a_1, \ldots, a_n]$	Asynchronous output
		$x?[y_1, \ldots, y_n].P$	Input
		$x?{*}[y_1, \ldots, y_n].P$	Replicated input
		$(\nu\, x)P$	Channel creation
		$P \mid P$	Parallel composition
		$\mathbf{0}$	Null process
		if a then P else P	Conditional
		let $x =$ `"C code"` in P	Inlined C code

The null process, $\mathbf{0}$, is actually definable in the above calculus (using the deadlocked process $(\nu\, x)x![\,]$, for example), but we retain it here, since $\mathbf{0}$ can be implemented much more efficiently than $(\nu\, x)x![\,]$.

We make the same restrictions on where replicated input can occur as we did in Chapter 7, Section 7.6: replicated inputs may 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)

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

Arbitrary C expressions may be included inside π-terms using the expression form 'let $x =$ `"C code"` in P'. The inlined C code is treated as a string by the Pict compiler, but is allowed to refer to any Pict variable which is in scope. For example, if y and z are integer variables then the expression 'let $x =$ `"y + z"` in P' has the effect of binding x to the value computed by the C expression `y + z`.

8.2 Variable binding

One of the benefits of compiling to a (reasonably) high-level language such as C is that we can reuse its built-in variable-binding constructs. With such a compilation, a Pict variable is most simply represented by a C variable of the same name. In this way we avoid having to consider many of the low-level details about implementing variable-binding efficiently (such as register allocation).

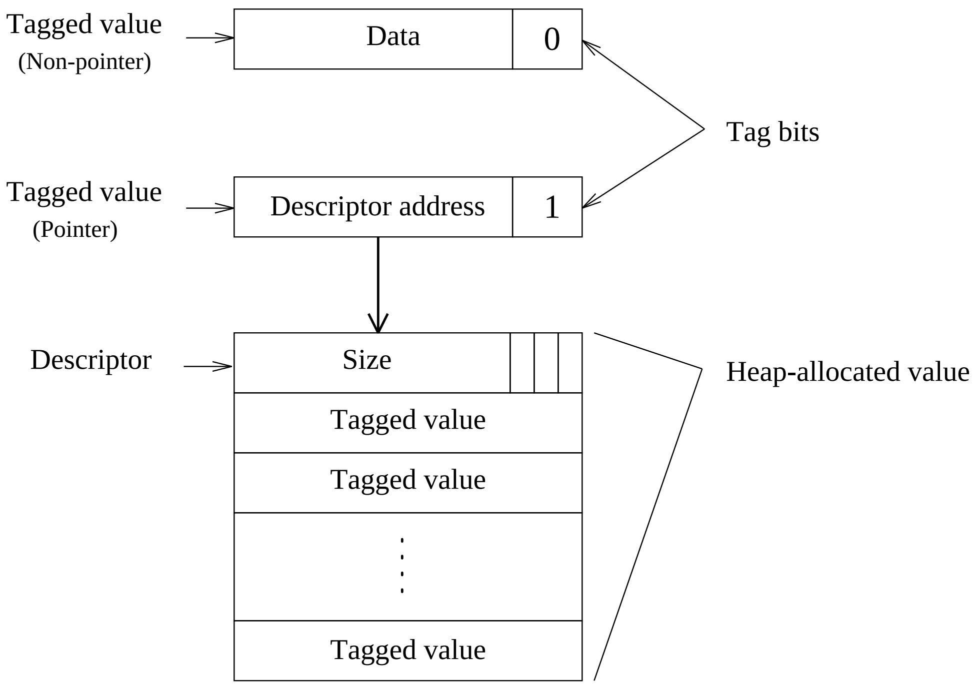
The problem, of course, is that C has a very restricted notion of variable binding (since it has no higher-order functions). For example, whenever we wish to suspend a Pict process and store it on a channel queue, we must capture the current variable bindings and store them in the channel, along with the code for that process. C has no built-in mechanism for implementing this operation, so at such points in the computation we must explicitly preserve the current variable-bindings in a closure.

Fortunately, this does not mean we are back in the situation where we have to implement variable-bindings without any help from the C compiler since, in practice, many variables are consumed before we ever need to create a closure. The C compiler is free to implement such variable bindings as it chooses (for example, in registers).

8.3 Data representations

All runtime data is accessed via a single machine word. We use the least-significant bit of each word as a tag (to inform the garbage collector whether that word is a pointer into the heap or not).

Definition 8.3 (Runtime data)



The above picture describes the general format of our runtime data. A zero tag bit indicates that a value is not a pointer into the heap. A tag bit of one signifies that a value is a pointer to a heap-allocated object. Subtracting one from such a value yields the address of a *descriptor* in the heap, which gives further details about the type of the object.

The three least-significant bits of a descriptor indicate what type of value is present. The remaining bits give the size of the object (in words, including the descriptor word). All heap-allocated objects must be an integral number of words long. The following definition presents all the possible descriptor tag values as C macro definitions, since it is convenient to refer to the tags by name in the C code which follows.

Definition 8.4 (Tag values)

`#define Empty`	`0`	Empty channel
`#define OneReader`	`1`	Channel with one reader
`#define OneWriter`	`2`	Channel with one writer
`#define ManyReaders`	`3`	Channel with many readers
`#define ManyWriters`	`4`	Channel with many writers
`#define Replicated`	`5`	Channel with replicated reader
`#define Tuple`	`6`	Tuple of values

The tag values `0` to `5` all indicate both that the object following the descriptor is a channel and the status of the channel (it would be wasteful to have a separate channel status word, in addition to the heap descriptor).

Tuples are used to store many different kinds of high-level data, such as closures and FIFO queues. The garbage collector does not need to distinguish between such kinds of high-level data, and therefore considers everything as a simple tuple of tagged values.

Our garbage collector ignores pointers which point to addresses outside the heap. This means that we can store pointers to C data structures inside Pict data structures (a necessary feature if we wish to allow the easy transfer of data between Pict programs and C code). We also allow Pict pointers to be stored in C data structures (i.e. outside the Pict heap), but we do not describe that mechanism here.

8.3.1 Integers and booleans

Since we use the least-significant bit of every data value as a tag bit (to tell the garbage collector whether that value is a pointer or not), we must represent the Pict integer i using the C integer $2 * i$:

Definition 8.5 (Integer representation)

Integer i	i	0

C's built-in boolean operators interpret any non-zero integer as 'true' and zero as 'false'. However, it is easier to implement conjunction, disjunction and negation if we represent 'true' using the integer `1`, and 'false' using the integer `0`. C's built-in comparison operators return values of this form anyway, so we do not incur any additional cost when implementing Pict comparison operators in terms of C comparison operators.

Definition 8.6 (Boolean representation)

true	0	1
false	0	0

The above representation for 'true' clashes with our tagging scheme for Pict data, but does not cause the garbage collector any problems, since the garbage collector ignores pointers which point to addresses outside the heap (and the address 0 will certainly be outside the heap).

8.3.2 Closures

Processes are represented by heap-allocated closures. A closure stores a code pointer (the address of a C function) and all the free variables of the process:

Definition 8.7 (Closures)

n+2	1	1	0	Code pointer	Value 1	- - - - -	Value n

The garbage collector treats each closure just like any other tuple of tagged values (it always ignores pointers which point to addresses outside the heap, so it never gets confused by the presence of a code pointer in a tuple). The function address stored in a closure will always point to a function of the form described in Section 9.2.

8.3.3 Channels

It turns out, in practice, that the majority of input operations find that the channel queue is empty, and the majority of output operations find that the channel queue contains exactly one reader (or a replicated reader). This is largely due to the fact that we encode functions as processes. For example, if we examine how the encoding of function application (reproduced below) behaves under our compilation, we find that the input on r always finds the channel queue for r empty (since r is a fresh channel).

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

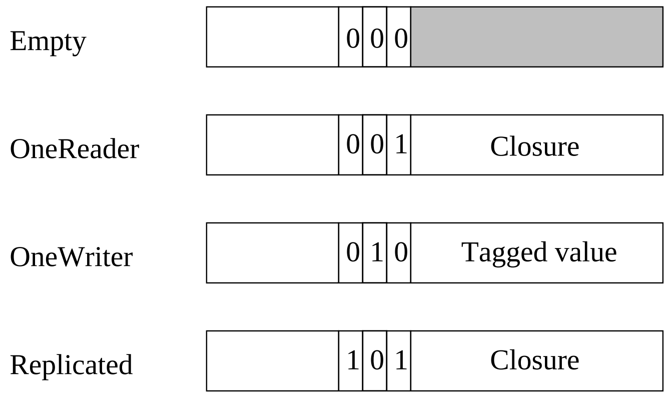
Moreover, when f does eventually return its result along r, it will find exactly one reader already in the channel queue (the process $r?[x_1,\ldots,x_n].P$). Furthermore, it is usually the case that the function f is represented using a replicated input on f, so the output $f![a_1,\ldots,a_m]$ will usually find that f contains a single replicated process.

We therefore optimise our channel representation for the case where a channel contains at most one (possible replicated) reader. In the case where we need to store more than one reader, or more than one written value, we must allocate extra storage during communication.

The representation of channels used here is based on experience the author has gained from implementing the Pict compiler, and tries to do a good job for the most common types of channel usage in Pict programs. However, it would be much better to leave the decision about channel representation until compilation time: if we had a program analysis which could tell us about the (approximate) usage of each channel we could choose the representation of each channel according to its expected usage. We will have more to say about such analyses in Section 9.18.

A channel's descriptor indicates both the size of the channel (two words) and the current channel status. The subsequent word contains a tagged data value. We only use the three least-significant bits of each channel descriptor, since the garbage collector knows that all channels are two words long.

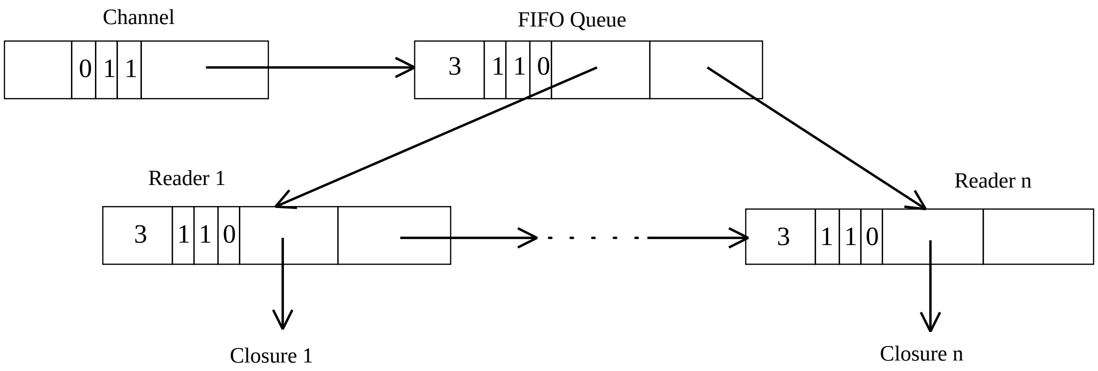
Definition 8.8 (Simple channel values)



If a channel's status is `Empty` then the content of its value field is irrelevant (the garbage collector never examines the value field in this case). If a channel's status is either `OneReader` or `OneWriter` then its value field contains a pointer to the closure for the reader, or the written value respectively. A channel containing a replicated reader has the same format as a channel containing an ordinary reader (though it will be treated very differently during communication, of course). Since we only have one word available to store any written data, we must allocate a separate piece of storage in the heap for the written data if the arity of a channel is greater than one.

If we need to store more than one blocked reader or writer, then we must allocate a separate queue structure in the heap. For instance, if a channel's status is `ManyReaders` then its value field points to a FIFO queue which is used to store blocked readers, as shown below. (The representation of a channel whose status is `ManyWriters` is just like that shown below, except that we store values in the queue, rather than closures.)

Definition 8.9 (Complex channel values)

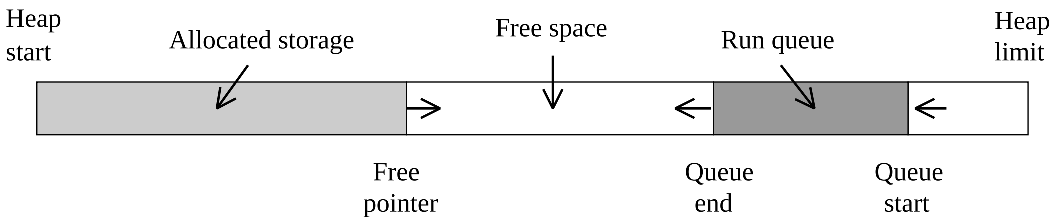


Channels use FIFO queues to ensure fairness. FIFO queues are implemented by keeping pointers to both the start and end of the queue. This enables fast insertion at the end of the queue (when we get a new reader or writer), and fast removal from the front of the queue (when we unblock a waiting process).

8.3.4 Run queue

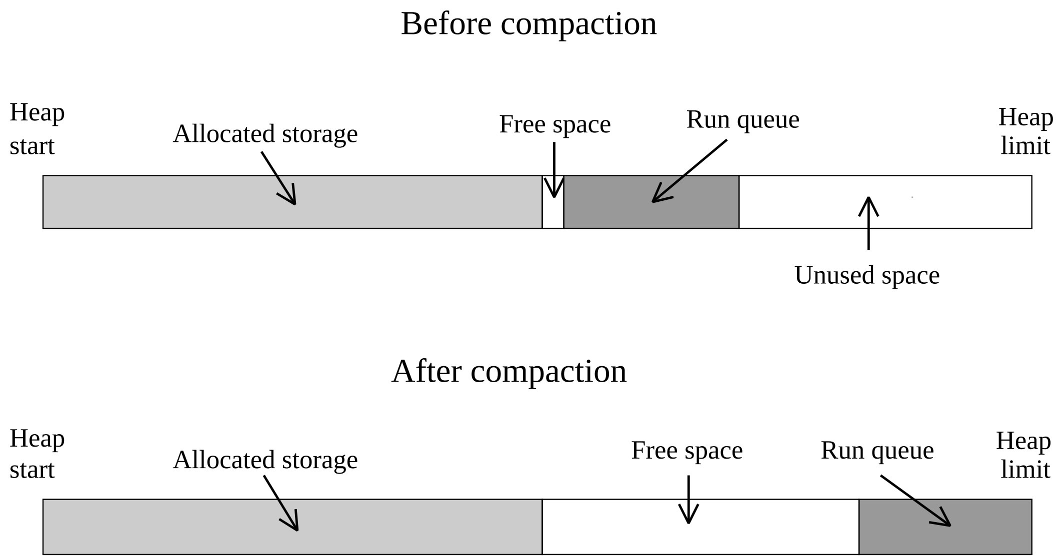
Since we add and remove processes from the run queue very frequently, it is important to implement the run queue in such a way that run queue entries do not consume permanent storage. We therefore allocate run queue entries at the opposite end of the heap from where we allocate ordinary storage:

Definition 8.10 (Heap storage)



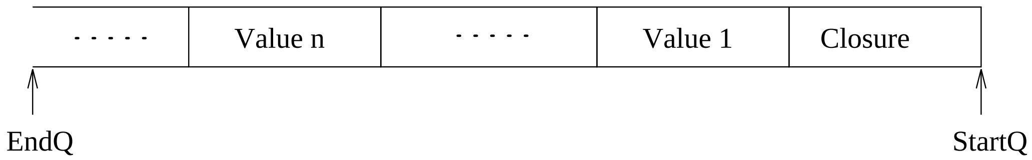
As we add entries to the end of the run queue, it grows towards the middle of the heap. Similarly, as we allocate memory in the ordinary part of the heap, the free pointer moves towards the centre of the heap. This memory model has the benefit that we need only perform a single test to determine whether we need to do a garbage collection (we just test whether the difference between the end of the run queue and the free pointer is sufficiently large for the allocation we wish to do).

But what about when we remove entries from the start of the run queue? We end up with a gap at the end of the heap. Fortunately, this gap can easily be reused: whenever we need to do a garbage collection we first check if there is enough space in the gap between the start of the run queue and the end of the heap. If there is, we just shift the run queue back up to the end of the heap instead of doing a real garbage collection. Now, it turns out that the run queue is usually very short (containing only one or two processes), so copying the run queue back to the end of the heap is significantly cheaper than doing a garbage collection.



The data contained within the run queue has a very simple format:

Definition 8.11 (Run queue)



Each closure is followed by some number of values (which correspond to the arguments given to that process). For instance, if we output two values v_1 and v_2

along some channel which already contains a reader (represented by the closure c), then we add the closure c, followed by the values v_1 and v_2, to the end of the run queue. This avoids having to allocate any permanent storage for v_1 and v_2 (note that it is not always possible to execute the unblocked process immediately, so we will in general need to store v_1 and v_2 somewhere until the unblocked process can consume them).

Chapter 9

Compiling Pict to C: Implementation

We present our compilation of Pict to C in three parts: The expression $[\![a]\!]$ denotes the C expression which implements the Pict atom a, the expression $[\![P]\!]$ denotes the C statement sequence which implements the process P, and the expression $[\![?[\tilde{y}].P]\!]$ denotes the C statement sequence which builds a closure for the process abstraction $?[\tilde{y}].P$.

9.1 Basic definitions

The following macro definitions are used throughout the C code which follows:

Definition 9.1 (Basic macro definitions)

```
#define OFFSET(x,i)  ((Val *)(x-1))[i]
#define STATUS(x)    OFFSET(x,0)
#define VAL(x)       OFFSET(x,1)
#define TUPLE(x)     (Tuple+(x<<3))
#define TAG(x)       ((Val)(x)+1)
```

The `OFFSET` macro takes as arguments a tagged word `x` and an integer `i` and extracts the `i`'th word of the heap object pointed to by `x` (note that since `x` is tagged we must subtract `1` from `x` to get address of the start of the object in the heap). We assume that the type identifier `Val` is already defined to be the type of integers of the same size as a machine word.

The `STATUS` macro extracts the first word pointed to by `x` (which always contains a descriptor). The `VAL` macro extracts the second word pointed to by `x` which, in the case where `x` is a channel, contains the value stored in the channel. The `TUPLE` macro constructs a valid descriptor for a tuple of size `x`. The `TAG` macro takes a pointer value and tags it (so that it can be stored in the heap).

All processes refer to a small number of global variables which hold pointers to the start and end of the run queue and the next free allocation space in the heap:

Definition 9.2 (Global variables)

`Val *EndQ;`	Pointer to the end of the run queue
`Val *StartQ;`	Pointer to the start of the run queue
`Val *Free;`	Pointer to the next free space in the heap

9.2 Encoding processes

As mentioned in Chapter 8, processes are represented using heap-allocated closures. Each closure contains the address of a C function of the form described in the following definition.

The function f represents the process abstraction $?[y_1, \ldots, y_m].P$ whose free variables are $x_1, \ldots, x_n$ (note that $\{x_1, \ldots, x_n\} = fv(P) - \{y_1, \ldots, y_m\}$). We declare local variables in f for both the free variables $x_1, \ldots, x_n$ and the abstracted variables $y_1, \ldots, y_m$.

The first thing f does is to check that there is enough free space in the heap. We allocate ordinary storage at the end of the allocation region (pointed to by `Free`), and allocate run queue entries the opposite end of the heap (pointed to by `EndQ`). Thus, if `Free +` $heap(P)$ `> EndQ` then there is not enough space in the heap to allocate $heap(P)$ words and we call the garbage collector.

Definition 9.3 (Process template)

```
void f (void) {
   Val closure,y1,...,ym,x1,...,xn;
   if (Free + heap(P) > EndQ) Gc(heap(P));
   /* Get closure pointer */
   closure = StartQ[0];
   /* Get arguments */
   y1 = StartQ[-1];
   ...
   ym = StartQ[-m];
   StartQ -= m+1;
   /* Bind free variables */
   x1 = OFFSET(closure,2);
   ...
   xn = OFFSET(closure,n+2);
   /* Execute process */
   [[P]]
}
```

The expression $heap(P)$ denotes the total number of words we wish to allocate in both the ordinary storage area and the run queue. We pass $heap(P)$ as an argument to the garbage collector, which guarantees that if it returns control there will be at least $heap(P)$ words free in the heap (we can always determine the maximum amount of storage required by a process, see Definition 9.29 in Section 9.17 for details).

We load the variable `closure` with the first value in the run queue (which is always a pointer to a closure for the current process). After that, we initialise the argument variables $y_1, \ldots, y_m$, loading their values from the run queue. The free variables $x_1, \ldots, x_n$ are then loaded with the values stored in `closure`. (In the case where P has no free variables, we can can omit the code which loads `closure`, since it is never used in the body of f.) We assume that the free variables $x_1, \ldots, x_n$ are given in some canonical order, so that we know where they are stored in the closure.

The expression $[\![P]\!]$ denotes the C code which implements the process P (the actual translation of processes into C is described later). The code implementing

P can now refer to the variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ in the same way as we would in a normal C program. This also means that the C compiler is free to optimise the storage for both sets of variables in the same way as it would for any other C program.

9.3 The scheduler

Now that we have described the representations of the run queue and the processes that inhabit it, we can explain how we execute processes. The following top-level C function is responsible for initializing the heap space and run queue, placing the initial process on the run queue, and executing it (and any other processes which are subsequently placed on the run queue). Once there are no more processes to run, the top-level program returns, and the whole program finishes. (This code implements the behaviour described in the Sched rule of Section 7.11.)

Definition 9.4 (Main program and scheduler loop)

```
void main (int argc, char **argv)
{
   /* Initialise heap and run queue */
   Free = ...; StartQ = ...; EndQ = ...;
   /* Put initial process on run queue */
   *StartQ-- = initialProcess;
   /* Scheduler loop */
   while (StartQ != EndQ) {
      ((void(*)(void))(OFFSET(*StartQ,1)))();
   }
}
```

Recall that `StartQ` and `EndQ` are pointers to the start and end of the run queue respectively. The expression `*StartQ` therefore denotes the first value in the run queue, which must be a pointer to a closure. The expression `OFFSET(*StartQ,1)` extracts the first data value in the closure, which is always a pointer to a C function of the form described in the previous section. We therefore cast the type of `OFFSET(*StartQ,1)` to `void(*)(void)` (the type of pointers to functions which take no arguments and return no result), enabling us to apply the resulting value to the empty argument list `()`. This has the effect of running the first process on the run queue. When that process terminates, it returns control to the scheduler loop, which checks if there are any more processes in the run queue, repeating the above procedure if the run queue is non-empty.

The above scheduler is so simple that there is no reason why it should really be a separate function: each process, once it has finished executing its own code could just call the next process in the run queue itself. However, such a compilation scheme runs into a well-known problem with compiling to C (see [Jon92], for example): Suppose that the processes $P_1, P_2, \ldots$ are on the run queue. We start P_1 running by calling the function representing P_1. Once it has finished doing its own work, P_1 calls the function representing P_2, and so on. This behaviour will eventually cause C's stack to overflow, since we only return from the functions representing $P_1, P_2, \ldots$ when there are no processes left to execute (i.e. when the whole Pict program has finished executing).

Of course, a clever C compiler might notice that in each function P_i the call to P_{i+1} is the last action of P_i, and therefore remove the stack frame for P_i *before* calling P_{i+1}. Unfortunately, we are not aware of any C compiler which does this optimisation. Instead, the Pict compiler makes a virtue out of necessity and uses the scheduler to do various useful checks (for example, checking if there have been any interrupts or if any input/output data is available).

9.4 Atoms

The compilation of integer and boolean atoms into C expressions is straightforward (given the representations of integers and booleans from Section 8.3.1). The compilation of Pict variables is trivial, since we maintain the convention that every Pict variable is represented by a C variable of the same name:

Definition 9.5 (Compiling atoms)

$$\begin{array}{rcl@{\qquad\qquad}rcl} [\![x]\!] & \doteq & x & [\![\mathrm{true}]\!] & \doteq & \mathtt{1} \\ [\![i]\!] & \doteq & 2 * i & [\![\mathrm{false}]\!] & \doteq & \mathtt{0} \end{array}$$

9.5 Process abstractions

Processes are represented using heap-allocated closures, where each closure contains the address of a C function whose general format is described in Section 9.2. The expression $[\![?[y_1, \ldots, y_m].P]\!]$ yields a C statement which builds a closure for the process abstraction $?[y_1, \ldots, y_m].P$ (allocating the closure at the next free space in the heap).

Definition 9.6 (Process abstractions) $[\![?[y_1, \ldots, y_m].P]\!] \doteq$

```
Free[0] = TUPLE(n + 2);
Free[1] = (Val)&f;
Free[2] = x₁;
...
Free[n + 1] = xₙ;
Free += n + 2;
```

We first write an appropriate descriptor word (which says that this object is a tuple of size $n+2$). Then we write the address of the function f (which contains the code implementing $?[y_1, \ldots, y_m].P$) and the values $x_1, \ldots, x_n$ into the closure. Finally, we increment `Free` by $n + 2$ words (the size of the closure we have just created).

We assume that the variables $x_1, \ldots, x_n$ are given in some canonical order (so that when we come to implement the code for $?[y_1, \ldots, y_m].P$ we know which part of the closure each free variable is stored in). We assume that the name of the function f is fresh (and that the function f has itself already been defined, using the template given in Section 9.2 and the encoding of processes which follows).

9.6 The null process

The null process has no behaviour and is implemented as the empty instruction sequence. This has the effect of returning control to the scheduler if there is no other work to be done in the current process. (This code implements the behaviour described in the Nil rule of Section 7.11.)

Definition 9.7 (The null process) $[\![0]\!] \doteq$ `/* Nothing */`

9.7 Channel creation

To create a new channel, we assign the current value of the free space pointer to the local variable x (after tagging it, to indicate that x it is a pointer into the heap). We then set the first word of x (the descriptor word) to `Empty` and increment the free pointer by the size of the channel (two words). We need not initialise the value field of the channel, since the garbage collector knows that if a channel's status is `Empty`, it should not scan the value field. The process P is compiled in the scope of the local variable x. (This code implements the behaviour described in the Res rule of Section 7.11.)

Definition 9.8 (Channel creation) $[\![(\nu\, x)P]\!] \doteq$

```
{
   Val x = TAG(Free);
   Free[0] = Empty;
   Free += 2;
   [[P]]
}
```

9.8 Conditional expressions

The boolean values 'true' and 'false' are represented by the integers `1` and `0` respectively. This representation is consistent with that used by C's conditional statement, so we can just interpret Pict conditionals using C conditionals.

Definition 9.9 (Conditionals) $[\![$`if` a `then` P `else` $Q]\!] \doteq$

```
if ([[a]]) { [[P]] } else { [[Q]] }
```

9.9 Parallel composition

The compilation of parallel composition is short and sweet: we compile parallel composition using C's sequential composition operator! The π-calculus is a concurrent, non-deterministic, language, but C is both sequential and deterministic, so it should not be surprising if we find that the compilation to C has the effect of sequentialising the execution of π-terms. (This code implements the behaviour described in the Prl rule of Section 7.11.)

Definition 9.10 (Parallel composition) $[\![P \mid Q]\!] \doteq [\![P]\!] \,;\, [\![Q]\!]$

The above compilation of $P \mid Q$ clearly determines that P will always execute before Q. Such deterministic behaviour is, not surprisingly, actually very helpful when tracking down programming errors. However, it is possible to simulate some form of non-determinism, if required: for instance, we could compile $P \mid Q$ so that we test a random number at runtime and then decide which process to execute on the basis of that test.

It is worth noting that the author has *never* felt the need to actually do this in the Pict compiler since, for non-trivial programs, plenty of non-deterministic behaviour arises due to time-dependent interactions between Pict code and the operating system (for example, during input/output or interrupt handling).

9.10 Inline C code

Since all Pict variables are represented using C variables, it is easy to insert user-defined C code into the code produced by our compiler:

Definition 9.11 (Interfacing with C) $[\![\texttt{let } x = \texttt{"C code" in } P]\!] \doteq$

```
{
   Val x = C code;
   [[P]]
}
```

The value resulting from the C expression containing in the string `"C code"` is bound to the variable x. For example, the usual integer operations can be defined using inline C code as follows:

Definition 9.12 (Integer operations)

Addition	:	`let` $x =$ `"x + y" in` P
Subtraction	:	`let` $x =$ `"x - y" in` P
Multiplication	:	`let` $x =$ `"x * (y >> 1)" in` P
Division	:	`let` $x =$ `"x / (y >> 1)" in` P
Comparison	:	`let` $x =$ `"x == y" in` P

Recall that a Pict integer i is represented using the C integer $2 * i$. Thus, to implement $i + j$ it is sufficient to just add the C representations of i and j, since $(2 * i) + (2 * j) = 2 * (i + j)$. Subtraction can be implemented in the same way, since $(2*i)-(2*j) = 2*(i-j)$. Comparison operations are also unaffected by our representation of integers since, for example, $2 * i = 2 * j$ iff $i = j$. We therefore pay no additional cost for doing simple arithmetic and comparison operations on tagged integers.

However, our representation of integers does cause some extra work when we multiply integers, since $(2 * i) * (2 * j) \neq 2 * (i * j)$: we must divide one of the C operands by two before multiplying the C representations of i and j (it is necessary to divide by two *before* multiplying the integers, otherwise we would lose precision). In fact, we use a right-shift operation to divide j by two, since that is usually faster than doing a real division). Note that, in practice, it is often the case that one of the operands in a multiplication is a constant, in which case we can compute the right-shift operation at compile-time and we avoid any additional cost for multiplying tagged integers. The implementation of division behaves similarly.

Boolean conjunction, disjunction and negation can be implemented using C's bitwise operators (there is no need to use C's more general logical operators, which are designed to allow any non-zero integer to be interpreted as 'true').

Definition 9.13 (Boolean operations)

$$
\begin{array}{rcl}
\text{Conjunction} & : & \text{let } a = \texttt{"b \& c"} \text{ in } P \\
\text{Disjunction} & : & \text{let } a = \texttt{"b | c"} \text{ in } P \\
\text{Negation} & : & \text{let } a = \texttt{"b \^{} 1"} \text{ in } P
\end{array}
$$

Our style of code generation gives the C optimiser the chance to place intermediate values in registers. For example, the Pict code

let $x =$ `"y - z"` in
  let $b =$ `"x == 0"` in
    if b then P else Q

is translated into the following C code, which we would expect any reasonable optimising C compiler to implement using registers to hold the values of x and b:

```
{
  Val x = y - z;
  {
    Val b = x == 0;
    if (b) { ⟦P⟧ } else { ⟦Q⟧ }
  }
}
```

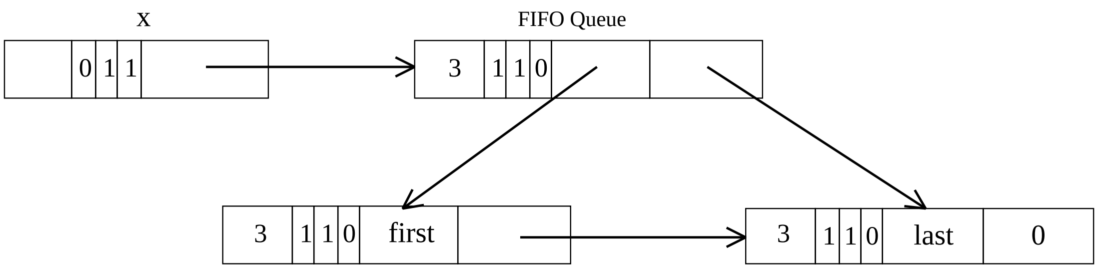
9.11 FIFO queue creation

The following function allocates a new FIFO queue containing the two queue elements `first` and `last`.

Definition 9.14 (FIFO queue creation)

```
Val CreateQueue (Val first, Val last) {
   Val fifo = TAG(Free);
   /* Allocate FIFO */
   Free[0] = TUPLE(3);
   Free[1] = TAG(Free+3);
   Free[2] = TAG(Free+6);
   /* Allocate first queue element */
   Free[3] = TUPLE(3);
   Free[4] = first;
   Free[5] = TAG(Free+6);
   /* Allocate second queue element */
   Free[6] = TUPLE(3);
   Free[7] = last;
   Free[8] = 0;
   Free += 9;
   return fifo;
}
```

The resulting data structure has the following structure. The FIFO contains two pointers (to the first and last queue elements). The first queue element contains `first`, and is linked to the last queue element, which contains `last`.



9.12 FIFO queue insertion

The following function inserts the value `val` at the end of `fifo`. The local variable `last` is a pointer to the last element in the queue, so we make the second field of `last` point to `TAG(Free)`, the location of the new queue element, and then update `fifo` so that it also points to the new element. We then build the new queue element and increment `Free` by the number of words we have allocated.

Definition 9.15 (FIFO queue insertion)

```
void InsertLast (Val fifo, Val val) {
   Val last = OFFSET(fifo,2);
   OFFSET(last,2) = TAG(Free);
   OFFSET(fifo,2) = TAG(Free);
   Free[0] = TUPLE(3);
   Free[1] = val;
   Free[2] = 0;
   Free += 3;
}
```

9.13 FIFO queue removal

The function `RemoveFirst` returns the first element in the channel `x`'s queue (`x` must always contain a fifo queue). If, after removing the first element, we find that the queue has length one, we remove the whole queue structure from `x` and change `x`'s status to `status`.

Definition 9.16 (FIFO queue removal)

```
void RemoveFirst (Val x, int status) {
   Val fifo = VAL(x);
   Val first = OFFSET(fifo,1);
   Val next = OFFSET(first,2);
   if (next == OFFSET(fifo,2)) {
      STATUS(ch) = status;
      VAL(ch) = OFFSET(next,1);
   } else {
      OFFSET(fifo,1) = next;
   }
   return OFFSET(first,1);
}
```

The value `first` is a pointer to the first element in `fifo`. The value `next` is a pointer to the next element in the queue. Thus, if `next == OFFSET(fifo,2)` then `next` is the last element in the queue, and we reset `x`'s status and put the contents of `next` in `x`'s value field.

9.14 Output expressions

The behaviour of an asynchronous output expression is dependent on the status of the channel we are outputting on. We therefore use a `switch` statement to select the appropriate thing to do when outputting on a channel. The following code presents just the outer structure of the `switch` statement, since we present the actual code for each case separately.

Definition 9.17 (Output) $[\![x![a_1,\ldots,a_n]]\!] \doteq$

```
switch (STATUS(x)) {
   ...
}
```

If x already contains one reader, then we reset the status of x to `Empty` and place the closure for the reader (which is stored in the value field of x) on the run queue. We then place the translations of the output atoms $a_1, \ldots, a_n$ after the closure on the run queue, ready to be consumed by the reader when the scheduler restarts it. (This code implements the behaviour described in the AOut-R rule of Section 7.11, but is specialised for the case where there is only one reader.)

Definition 9.18 (Output: `OneReader`)

```
case OneReader:
   STATUS(x) = Empty;
   EndQ[0] = VAL(x);
   EndQ[-1] = ⟦a1⟧;
   ...
   EndQ[-n] = ⟦an⟧;
   EndQ -= n+1;
   break;
```

If x contains a replicated reader, we do exactly the same thing, except that we do not reset the status of x to `Empty` (since communicating with a replicated reader does not consume that reader). It is worth noting that in both the `OneReader` and the `Replicated` cases we need not allocate any permanent storage (storage allocated in the run queue can be reclaimed without doing a garbage collection, as explained in Section 8.3.4). (This code implements the behaviour described in the AOut-R* rule of Section 7.11.)

Definition 9.19 (Output: `Replicated`)

```
case Replicated:
   EndQ[0] = VAL(x);
   EndQ[-1] = ⟦a1⟧;
   ...
   EndQ[-n] = ⟦an⟧;
   EndQ -= n+1;
   break;
```

If the status of x is `Empty`, we set the status of x to `OneWriter`, allocate a tuple to store the output atoms $a_1, \ldots, a_n$, and place a (tagged) pointer to that tuple in the value field of x. If $n = 1$ we need not allocate a tuple at all, since we already have enough space to store the output atom in the value field. Similarly, if $n = 0$, we need not allocate a tuple, we need only write a dummy value (say zero) in the value field of x (it is necessary to write something in the value field of x, since the garbage collector will scan the value field of any non-empty channel, though we could get around this problem by having a special descriptor tag for this case, just as we have a special case already for empty channels). (This code implements the behaviour described in the AOut-W rule of Section 7.11, but is specialised for the case where the channel is empty.)

Definition 9.20 (Output: `Empty`)

```
case Empty:
   STATUS(x) = OneWriter;
   VAL(x) = TAG(Free);
   Free[0] = TUPLE(n+1);
   Free[1] = [[a_1]];
   ...
   Free[n] = [[a_n]];
   Free += n+1;
   break;
```

If x already contains one writer, then we must allocate a FIFO data structure of the form described in Section 8.3.3. We change the status of x to `ManyWriters`, allocate a FIFO data structure containing two queue elements (one for the value which was already stored in x, and the other for the tuple of values $a_1, \ldots, a_n$). We arrange the queue so that the value which was already stored in x is placed first in the queue. (This code implements the behaviour described in the AOut-W rule of Section 7.11, but is specialised for the case where there is just one writer.)

Definition 9.21 (Output: `OneWriter`)

```
case OneWriter:
   {
      tuple = TAG(Free);
      Free[0] = TUPLE(n+1);
      Free[1] = ⟦a1⟧;
      ...
      Free[n] = ⟦an⟧;
      Free += n+1;
      VAL(x) = CreateQueue(VAL(x),tuple);
   }
   break;
```

If there are already many writers in x, the situation is slightly simpler, since the FIFO data structure has already been allocated, and we need only allocate an extra queue element (and a tuple for $a_1, \ldots, a_n$). We insert the new queue element, which is at address `Free`, at the end of the FIFO. (This code implements the behaviour described in the AOut-W rule of Section 7.11, but is specialised for the case where the channel contains more than one writer.)

Definition 9.22 (Output: `ManyWriters`)

```
case ManyWriters:
   {
      Val tuple = TAG(Free);
      Free[0] = TUPLE(n+1);
      Free[1] = ⟦a1⟧;
      ...
      Free[n] = ⟦an⟧;
      Free += n+1;
      InsertLast(VAL(x),tuple);
   }
   break;
```

In the case where x holds many readers we must implement the inverse of the previous operation and remove the first reader from the FIFO queue of readers.

If, after removing one reader from the FIFO queue, there is only one reader left, the `RemoveFirst` function sets the status of x back to `OneReader`. (This code implements the behaviour described in the AOut-R rule of Section 7.11, but is specialised for the case where there is more than one reader.)

Definition 9.23 (Output: `ManyReaders`)

```
case ManyReaders:
   EndQ[0] = RemoveFirst(x,OneReader);
   EndQ[-1] = ⟦a_1⟧;
   ...
   EndQ[-n] = ⟦a_n⟧;
   EndQ -= n+1;
   break;
```

Clearly, the code for implementing an output expression is very large in comparison to the constructs we have presented earlier! What is more, output expressions are a fundamental part of the computation mechanism used by Pict, and are therefore very common in programs. If we actually tried to compile every asynchronous output as indicated above, we would generate huge amounts of C code for anything other than toy programs.

We could trade off code size against execution time by inlining the code for the commonly executed cases and putting all the other cases in a library function (in the hope that the library function will be called relative infrequently). In Section 9.18 we describe a more flexible technique for optimising communication based on finding approximations of a channel's status at compile time.

9.15 Replicated input expressions

Because of the restrictions we made on the occurrences of replicated input (see Section 7.6 for details), we can be sure that the status of x is always `Empty` when we execute the expression $x?{*}[y_1, \ldots, y_n].P$. We therefore set the status of x to `Replicated` and place a (tagged) pointer to the closure for $x?{*}[y_1, \ldots, y_n].P$ in the value field of x (recall that the expression $[\![?[y_1, \ldots, y_n].P]\!]$ returns a sequence of C statements which builds a closure for the given process abstraction at the next free space in the heap). (This code implements the behaviour described in the Repl rule of Section 7.11.)

Definition 9.24 (Replicated input) $[\![x?{*}[y_1,\ldots,y_n].P]\!] \doteq$

```
{
  STATUS(x) = Replicated;
  VAL(x) = TAG(Free);
  [[?[y1,...,yn].P]]
}
```

9.16 Input expressions

The behaviour of input expressions is in most cases similar to that of output expressions. The main difference is that before we check the status of x, we allocate a closure for the process abstraction $?[y_1,\ldots,y_n].P$ at the next free space in the heap (the local variable `closure` is bound to the tagged address of this closure). In the case where there already is an output value in the channel, this is rather wasteful, since we build a closure for $?[y_1,\ldots,y_n].P$ and then immediately put it on the run queue, when we could have just executed P directly. Fortunately, this case is sufficiently rare that we can make do with this simpler, more uniform, treatment of input.

Definition 9.25 (Input) $[\![x?[y_1,\ldots,y_n].P]\!] \doteq$

```
{
  Val closure = TAG(Free);
  [[?[y1,...,yn].P]]
  switch (STATUS(x)) {
    ...
  }
}
```

Is the case where there is one writer in the channel, we put our closure on the run queue, followed by the elements of the stored tuple of values. (This code implements the behaviour described in the Inp-W rule of Section 7.11, but is specialised for the case where the channel contains just one writer.)

Definition 9.26 (Input: `OneWriter`)

```
case OneWriter:
  {
    Val tuple = VAL(x)
    STATUS(x) = Empty;
    EndQ[0] = closure;
    EndQ[-1] = OFFSET(tuple,1);
    ...
    EndQ[-n] = OFFSET(tuple,n);
    EndQ -= n+1;
  }
  break;
```

It is rather a shame that we have to copy the individual values onto the run queue instead of just copying a pointer to the tuple. However, if we wish to avoid this cost we need multiple entry points to each process abstraction, since in the case where a closure is put on the run queue by an output operation, it will find its arguments on the run queue, while in the case where a closure is put on the run queue by an input operation it will find its arguments in a tuple.

In the case where the channel is empty, we store our closure in the channel and set the channel's status to `OneReader`. (This code implements the behaviour described in the Inp-R rule of Section 7.11, but is specialised for the case where the channel is empty.)

Definition 9.27 (Input: `Empty`)

```
case Empty:
  STATUS(x) = OneReader;
  VAL(x) = closure;
  break;
```

The remaining cases use the FIFO queue manipulation functions in a similar way to those used in the compilation of output expressions. (This code implements the behaviour described in the Inp-R and Inp-W rules of Section 7.11, but is specialised for the case where the channel contains just one reader, more than one reader, or more than one writer.)

Definition 9.28 (Input: `OneReader`,`ManyReaders`,`ManyWriters`**)**

```
case OneReader:
  STATUS(x) = ManyReaders;
  VAL(x) = CreateQueue(VAL(x),closure);
  break;
case ManyReaders:
  InsertLast(VAL(x),closure);
  break;
case ManyWriters:
  {
    Val tuple = RemoveFirst(x,OneWriter);
    EndQ[0] = closure;
    EndQ[-1] = OFFSET(tuple,1);
    ...
    EndQ[-n] = OFFSET(tuple,n);
    EndQ -= n+1;
  }
```

9.17 Heap usage

Now that we have seen all of the compilation of processes, it is easy to determine the maximum number of words that a process may need to allocate. This information is used in the process template given in Definition 9.3, since the first action of every process is to check whether there is enough free space for all of the allocation it might do.

Definition 9.29 (Heap usage)

$$
\begin{array}{rcl}
heap(\mathbf{0}) & \doteq & 0 \\
heap((\nu\, x)P) & \doteq & 2 + heap(P) \\
heap(P \mid Q) & \doteq & heap(P) + heap(Q) \\
heap(\text{if } b \text{ then } P \text{ else } Q) & \doteq & max(heap(P), heap(Q)) \\
heap(\text{let } x = \texttt{"C code"} \text{ in } P) & \doteq & heap(\texttt{"C code"}) + heap(P) \\
heap(x![a_1, \ldots, a_n]) & \doteq & 10 + n \\
heap(x?{*}[y_1, \ldots, y_n].P) & \doteq & closureSize(?[y_1, \ldots, y_n].P) \\
heap(x?[y_1, \ldots, y_n].P) & \doteq & 9 + closureSize(?[y_1, \ldots, y_n].P)
\end{array}
$$

The expression $heap(\texttt{"C code"})$ denotes the number of words allocated by the inline C code (in Pict, this number is provided explicitly by the programmer). We calculate the number of words required to hold the closure for the process abstraction $?[y_1, \ldots, y_n].P$ by calculating the size of the set of free variables of $?[y_1, \ldots, y_n].P$, and adding two (one word for the descriptor and one word for the code pointer).

Definition 9.30 (Closure size)

$$closureSize(?[y_1, \ldots, y_n].P) \doteq |fv(P) - \{y_1, \ldots, y_n\}| + 2$$

9.18 Optimising communication

The code required to implement input and output expressions is very large (considering the fact that they are the fundamental operations of Pict, and are therefore pervasive throughout Pict programs). Fortunately, there is plenty of scope for optimisation: any program analysis which is able to determine the state of a channel at compile time (or at least, a set of possible states) is potentially very useful. For example, if the channel x is known to contain a replicated process (a very common situation, since functions are implemented as replicated processes), then we can avoid testing the status of x and execute the appropriate code directly:

Definition 9.31 (Optimised output) $[\![x![a_1, \ldots, a_n]]\!] \doteq$

```
EndQ[0] = VAL(x);
EndQ[-1] = [[a1]];
...
EndQ[-n] = [[an]];
EndQ -= n+1;
```

Specialising the code for input and output expressions has two benefits: the code becomes small enough that we can inline it, and we avoid testing the status of x. The Pict compiler uses a local program analysis (i.e. an analysis which does not attempt to track higher-order uses of channels) to determine when channels are guaranteed to contain replicated readers and when channels are guaranteed to be empty (two of the most common cases).

A number of other researchers have proposed more sophisticated analyses which produce similar information (usually about the maximum size of a channel queue) [KNY95, NN94]. The author, in collaboration with Kobayashi and Pierce, has recently developed a linear type system for Pict which can determine when a channel is used by exactly one reader/writer pair [KPT96]. If x is has a linear type then the code presented in Definition 9.32 is sufficient to implement an output on x (the case for linear input expressions is similar).

The linear type system ensures that if x has linear type then it is used by exactly one reader/writer pair. Thus, since the expression above is using x's write capability, the rest of the program must only have the capability to do a single read from x. This means that x can either be empty (because the rest of the program has not yet used its read capability), or x can contain a single reader (if the rest of the program has already done a read on x).

Definition 9.32 (Linear output) $[\![x![a_1, \ldots, a_n]]\!] \doteq$

```
if (STATUS(x) == Empty) {
   STATUS(x) = OneWriter;
   VAL(x) = TAG(Free);
   Free[0] = TUPLE(n);
   Free[1] = [[a1]];
   ...
   Free[n] = [[an]];
   Free += n+1;
} else {
   EndQ[0] = VAL(ch);
   EndQ[-1] = [[a1]];
   ...
   EndQ[-n] = [[an]];
   EndQ -= n+1;
}
```

Note that in the case where x already contains a reader we do not need to reset the status of x to `Empty`, since the linearity of x ensures that no other processes will use it in the future (both the input and output capabilities have been consumed).

9.19 Performance

The purpose of this chapter was to investigate whether the π-calculus can be implemented efficiently enough for it to be considered as a reasonable operational foundation for concurrent programming. To get a rough idea of the performance of the code generated by our Pict compiler we wrote three solutions to the "nqueens" problem. We first wrote an ML program to solve the problem, and then translated that into Pict (using Pict's high-level derived forms for functions). The table below indicates that the Pict version of the program is approximately five times slower than that produced by the New Jersey ML compiler (version 0.93). Considering the simplicity of our compilation (Pict's code generator is implemented using just 900 lines of ML), the performance of Pict was quite a surprise!

	Time	Code size
New Jersey ML	2.3s	380Kb
Pict	12s	25Kb
CML	46s	545Kb

To get an idea how fast our implementation of communication is, we converted our ML program into a CML [Rep92] program (CML is a concurrent extension of ML which implements channel-based communication on top of New Jersey ML). We converted all functions so that, instead of returning their results directly, they returned their results along an explicit result channel (this matches the way we return results from functions encoded in Pict). We did not use channels to implement the functions themselves, since CML has no built-in support for replicated inputs. As can be seen above, CML ran our example almost four times slower than Pict.

The reader should not attach too much importance to the exact figure quoted here, since there are a number of differences which are difficult to quantify (such as differences in garbage collection strategy). However, it is reassuring to find that the performance of Pict code, which uses just channel-based communication, is in the same ball-park as that of functional code. It is especially worth bearing in mind that Pict loses some performance just because we compile to C rather than to native code. Tarditi, Acharya and Lee [TAL90] found that compiling to C rather than to native code costs almost a factor of two for New Jersey ML code. If we are paying a similar price by compiling Pict to C, we might reasonably expect a native code generator to improve the performance of Pict to within a factor of two or three from New Jersey ML.

Our Pict compiler produces much smaller programs than the New Jersey compiler. This is not because we are compiling π-calculus, but because Pict has a tiny runtime system (just 800 lines of C). Much of the code which would normally live in the runtime system appears as inline C code in Pict programs. This gives our Pict compiler the ability to discard any C code which is unused in the program being compiled, and helps contribute to the very small code size of Pict programs.

Chapter 10

Related work

In this chapter we review related work on type systems and implementation techniques for concurrent calculi. We have already shown in Chapter 6 that there is a very strong relation between π-calculus types and λ-calculus types, so we will not discuss λ-calculus type systems here.

10.1 Type systems

We first compare our type system with Milner's original sort system [Mil91a, Mil91b] for the polyadic π-calculus (which inspired our π-calculus type system). Milner's sort system partitions the channels in a π-term using *subject sorts*. Each subject sort X is associated with an *object sort* $[X_1, \ldots, X_n]$ (this association is written as $X \mapsto [X_1, \ldots, X_n]$). An object sort describes how members of a particular subject sort may be used for communication. For example, in the following process we say x has sort X and y has sort Y. Channels of sort X carry pairs of channels of sort Y, and channels of sort Y carry the empty tuple.

$$x : X \mapsto [Y, Y] \qquad y : Y \mapsto [\,] \qquad x![y, y].y![]$$

The above example is well-sorted, since x is only ever used to communicate the pair $[y, y]$, which has the expected sort $[Y, Y]$. Similarly, y is only ever used to communicate the empty tuple.

The sort Y is equivalent to the type $\uparrow[\,]$ in our type system, and the sort X is equivalent to the type $\uparrow[\uparrow[\,], \uparrow[\,]]$. Our typing for the previous example is:

$$x : \uparrow[\uparrow[\,], \uparrow[\,]], y : \uparrow[\,] \vdash x![y, y].y![]$$

Milner also allows recursive sorts. For example, the sort of integer lists might be *List*, where

$$\mathit{List} \mapsto [\mathit{Nil}, \mathit{Cons}] \quad \mathit{Nil} \mapsto [\,] \quad \mathit{Cons} \mapsto [\mathit{Int}, \mathit{List}]$$

Compare this with the corresponding recursive type in our type system:

$$\mu \mathit{List}.\uparrow[\uparrow[\,], \uparrow[\mathit{Int}, \mathit{List}]]$$

We conjecture that the well-typed π-terms of our type system coincide with Milner's well-sorted π-terms. Note, however, that Milner's sorts allow us to partition channels more carefully than we can with our types. For example, the sorts P and V below are *not* considered to be equivalent.

$$P \mapsto [\,] \qquad V \mapsto [\,]$$

This matching of sorts 'by name' rather than 'by structure' is quite attractive. Suppose p and v have different intended uses and we wish to avoid p being accidentally used in place of v (as might well be the case if p and v are part of a semaphore). If we set $p : P$ and $v : V$ then Milner's sorting system will detect any confusion of p and v, but note that this technique is only useful in an explicitly-typed calculus.

As previously mentioned, our type system evolved from work on Milner's sort system. We decided to use more traditional type-theoretic techniques for three reasons:

1. Milner's sort system is very simple, but all sort information is global. For example, a closed π-term does not have a trivial type – all the sort information required inside the π-term is visible at the top-level. This seems rather unsatisfactory, and also causes a number of technical problems in proving subject-reduction.

2. Although it is possible to formulate polymorphic sorts, they are much more complicated than polymorphic types in our system.

3. Once one has a sufficiently powerful polymorphic type system, it is possible to distinguish values of isomorphic type using the abstract datatype encodings presented in Section 5.7.

Vasconcelos and Honda [VH93] have independently proposed a monomorphic type system for the polyadic π-calculus which is very similar to ours (except that they treat recursive types in a more traditional way, interpreting recursive types as regular trees). They prove that their type system is sound and has the principal type property using essentially identical techniques to those proposed here.

Pierce and Sangiorgi [PS93] have developed a π-calculus type system which uses the idea of I/O tags to capture additional information about how a channel is used, controlling whether a channel may be written to, read from, or both. Their type system can be thought of as a refinement of the type system presented here (though it was developed independently). We give the essence of the idea below, using our own notation. We introduce two new channel types: $?\delta$ and $!\delta$, the types of input-only and output-only channels respectively. Clearly, an ordinary channel of type $\uparrow\delta$, which allows either input or output, can be used in place of an input-only or an output-only channel. In terms of the subtyping relation we have:

$$\uparrow\delta \leq\, ?\delta \qquad\qquad \uparrow\delta \leq\, !\delta$$

As is usual in subtyping systems, we introduce a rule of subsumption for values:

$$\frac{\Delta \vdash v : \delta' \qquad \delta' \leq \delta}{\Delta \vdash v : \delta}$$

We then refine the typechecking rules for the input and output operators so that they use the new channel types:

$$\frac{\Delta \vdash c : !\delta \qquad \Delta \vdash v : \delta \qquad \Delta \vdash P}{\Delta \vdash c!v.P}$$

$$\frac{\Delta \vdash c : ?\delta \qquad \vdash p : \delta; \Delta' \qquad \Delta, \Delta' \vdash P}{\Delta \vdash c?p.P}$$

Clearly, we can derive our original rules for input and output from the above rules since, if $\Delta \vdash c : \uparrow\delta$, we can use subsumption to show that either $\Delta \vdash c : !\delta$ or $\Delta \vdash c : ?\delta$ as required.

Suppose, for the purposes of explanation, we introduce record patterns and values, with the following typing rules:

$$\frac{\Delta \vdash v_1 : \delta_1 \qquad \ldots \qquad \Delta \vdash v_n : \delta_n}{\Delta \vdash \{l_1 = v_1, \ldots, l_n = v_n\} : \{l_1 : \delta_1, \ldots, l_n : \delta_n\}}$$

$$\frac{\vdash p_1 : \delta_1; \Delta_1 \qquad \ldots \qquad \vdash p_n : \delta_n; \Delta_n}{\vdash \{l_1 = p_1, \ldots, l_n = p_n\} : \{l_1 : \delta_1, \ldots, l_n : \delta_n\}; \Delta_1, \ldots, \Delta_n}$$

We also allow subtyping on records in the usual way. Pierce and Sangiorgi also give subtyping rules for each new channel types.

$$\frac{\delta_1 \leq \gamma_1 \qquad \ldots \qquad \delta_n \leq \gamma_n}{\{l_1 : \delta_1, \ldots, l_n : \delta_n, \ldots\} \leq \{l_1 : \gamma_1, \ldots, l_n : \gamma_n\}}$$

$$\frac{\delta \leq \delta'}{?\delta \leq ?\delta'} \qquad \frac{\delta' \leq \delta}{!\delta \leq !\delta'} \qquad \frac{\delta \leq \delta' \qquad \delta' \leq \delta}{\uparrow\delta \leq \uparrow\delta'}$$

The subtyping rule for input-only channels says that we can forget information about the value we receive from input-only channels. For example, we can forget the field $age : Int$ in the record being sent along c below:

$$\frac{\Delta, n : String \vdash P}{c : ?\{name : String, age : Int\} \vdash c?\{name = n\}.P}$$

The subtyping rule for output-only channels is the dual of the above, since it says that we can send extra information along output-only channels. For example, we can send a record with an extra field $male = true$ along c below:

$$\frac{c : !\{name : String, age : Int\} \vdash P}{c : !\{name : String, age : Int\} \vdash c!\{name = \text{Dave}, age = 25, male = true\}.P}$$

The rule for ordinary channel types states that they are invariant in the subtyping relation: subtyping is only safe on input-only or output-only channels. Suppose we allow covariant subtyping on input/output channels (a similar example can be constructed if allow contravariant subtyping). We can now typecheck the following example by using the subsumption and (incorrect) channel subtyping rule to promote the type of c to be $\uparrow\{name : String\}$. Clearly, this example can cause a runtime error, since it has sent a record along c which is lacking the *age* field.

$$\frac{\delta \leq \delta'}{\uparrow\delta \leq \uparrow\delta'}$$

$$\frac{c : \uparrow\{name : String, age : Int\} \vdash P}{c : \uparrow\{name : String, age : Int\} \vdash c!\{name = \text{Dave}\}.P}$$

The Pict type system [PT95b] uses Pierce and Sangiorgi's channel subtyping, as well as subtyping for record values and patterns.

10.2 Type inference

Vasconcelos and Honda [VH93] have independently developed a type inference algorithm very similar to the one presented here. Their algorithm, takes only a process P as argument, and constructs the principal context in which P is well-formed. We can easily derive a similar algorithm from our type inference algorithm X:

Definition 10.1 (New type inference algorithm)
If $\Delta = x_1 : \alpha_1, \ldots, x_n : \alpha_n$ and $X(\Delta)(P) = \sigma$ where $x_1, \ldots, x_n$ are the free variables of P and $\alpha_1, \ldots, \alpha_n$ are distinct type variables then return $\sigma\Delta$.

We conjecture that the above algorithm is equivalent to that proposed by Vasconcelos and Honda (modulo the fact that we do not do type inference for recursive types). Note that in the case of a closed π-term P our algorithms already coincide, since Vasconcelos and Honda's algorithm returns a trivial type for P.

Gay [Gay93] has developed an algorithm which infers principle sorts for π-terms in Milner's sort system. The algorithm is based on a unification algorithm for sorts. For example, we can unify the sorts A and D below

$$\begin{array}{lll} A \mapsto [B, C] & B \mapsto [A] & C \mapsto [\,] \\ D \mapsto [E, F] & E \mapsto [D] & F \mapsto [\,] \end{array}$$

yielding a substitution which identifies A and D, B and E and C and F. Sort unification is similar to unification for regular trees [Cou83] (the algorithm used by Vasconcelos and Honda in their type inference algorithm).

The use of subject sorts is particularly convenient when unifying recursive sorts. Many unification algorithms for recursive trees use tags to record nodes which have already been visited, thereby ensuring that the unification algorithm always terminates. This is unnecessary in sort unification, since each node (object sort) is already labelled with a subject sort. We need only keep a record of which subject sorts have already been visited.

10.3 Polymorphic types

Vasconcelos [Vas94] has independently proposed a polymorphic type system for π-calculus which is a special case of the polymorphic type system presented here. It relies on explicit let-expressions to indicate where type generalisation may occur (just like the Damas-Milner type system does). The form of let-expression used in Vasconcelos' type system is essentially the same as our derived form for process definitions:

$$\texttt{def}\ X_1[\tilde{x}_1] = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ X_n[\tilde{x}_n] = P_n\ \texttt{in}\ Q$$

His type system allows the process definitions $X_1, \ldots, X_n$ to be given given polymorphic types, while forcing ordinary channels to be used monomorphically. In Section 5.2 we showed how our typing rules for polymorphic channels give rise to a derived typing rule for polymorphic process definitions. We conjecture, therefore, that Vasconcelos' polymorphic calculus is a strict sub-calculus of our polymorphic π-calculus. Note, however, that one benefit of restricting type generalisation to process definitions is that type inference becomes much simpler (in fact, just like the Damas-Milner type system, Vasconcelos' type system has principal types which can be computed automatically).

Languages such as PFL [Hol83], Poly/ML [Mat91], CML [Rep92] and Facile [GMP89] which are concurrent extensions of Standard ML [MTH90] all allow a limited form of channel polymorphism, since they retain Standard ML's polymorphic type system. It is well-known that the Damas-Milner type system is unsound in the presence of side-effecting computation, so some care has to be taken with the typing of any channel creation operator. For example, in CML [Rep92] the `channel` function creates new channels, and has the type `unit -> '_a channel`. The weakly polymorphic type `'_a` in the type of `channel` is necessary to avoid giving types to unsound programs such as the one below:

```
let
  val ch = channel()
in
  ... send(ch,33) ... (if accept(ch) then x else y) ...
end
```

The expression `channel()` has type `'_a channel`. The fact that `'_a` is a weak type variable means that it is unsafe to generalise that type. If we allowed the type of `ch` to be polymorphic in `'_a` then we could instantiate the type of `ch` to be `int channel` in the expression `send(ch,33)`, and `bool channel` in the expression `if accept(ch) then x else y`. This is clearly unsound since one process is sending an integer along `ch` while the other is expecting a boolean.

A side-effect of the above treatment of channel creation is that channels do not have the same status as functions and other datatypes when it comes to polymorphic typing. For instance, the following `server` function is supposed to wait for a pair `(x,y)` to be sent along `id` and then reply by sending `x` along `r`. As explained in Chapter 5, there is no reason why the channel `id` cannot be polymorphic (the `server` example is essentially a transliteration of the first example in Chapter 5). Unfortunately, the type of `id` can never be polymorphic, because the type of `channel` is weakly polymorphic.

```
val id = channel()
fun server() = let
  val (x,r) = receive(ch)
in
  send(r,x); server()
end
```

Thomsen [Tho93] illustrates how one can improve upon the behaviour of polymorphic channels using an effect system (in the context of the Facile language). However, even though his type system is quite complicated he is still not able to allow the type of `id` be polymorphic.

10.4 Implementation

There have been a number of proposals for abstract machines for concurrent calculi [Ama94, Car85, GMP89, etc.]. At a high-level, our implementation of communication is almost identical to that described by Cardelli [Car85]. However, our abstract machine is unique in that it implements nothing but communicating processes. For example, the abstract machines proposed by both Amadio [Ama94] and Giacalone et. al. [GMP89] use a separate SECD machine to implement each process (since the execution of a Facile processes may involve both λ-calculus reduction and communication).

In practice, the fact that we rely on communication as our sole computational mechanism means that we have to take more care when representing channels and processes. For instance, we are unaware of any implementation of channel-based communication which implements a replicated input construct (most concurrent languages rely on the functional part of the language to express infinite behaviour, or prove built-in recursive process definitions)

We are not aware of any compiler which implements a language whose sole computational mechanism is channel-based communication. However, the compilation of π-calculus to C presented in this dissertation is quite closely related to the SML to C compiler described in [TAL90]. The stackless representation used here for π-calculus processes is very reminiscent of the continuation-passing style of code generation used in the New Jersey ML compiler [AM87] (upon which the SML to C compiler is built).

Chapter 11

Conclusions and further work

Throughout this dissertation we have looked at the π-calculus from the perspective of concurrent programming. We have shown that the π-calculus admits a simple type system which can be extended to include many of the more advanced type-theoretic features familiar from the λ-calculus. In fact, in the Pict programming language [PT95b] the author, in collaboration with Benjamin Pierce, has developed a higher-order polymorphic π-calculus, which also includes subtyping, higher-order subtyping and extensible records. Experience to date suggests that refinements one can make to λ-calculus type systems are also applicable to the typed π-calculus.

The fact that communication protocols can be enforced using abstract datatypes (see Section 5.7 for details) gives us real reason to believe that our type system will be able to catch a significant number of the most common errors in π-calculus programs. For this reason, we have avoided complicating our channels types with complex protocol-like information (which often make type checking and type inference much more difficult). In Pict, the abstract datatypes provided by our polymorphic type system are used extensively throughout most library code [PT95c], as well as being used to ensure that the internal representations of built-in datatypes such as integers and booleans can only be manipulated using the functions supplied by the compiler.

Of course, that it not to say that there are no useful refinements one can make to channel types. The Pict type system includes one very useful refinement, proposed by Pierce and Sangiorgi [PS93], which distinguishes input and output capabilities on channels (see Chapter 10 for details). The linear type system developed by the author, in collaboration with Kobayashi and Pierce [KPT96], presents another useful refinement to channel types. However, both of these refinements add only a modest amount of information to channel types, and therefore retain simple type inference and type checking algorithms.

The problem of how to do type inference for the polymorphic π-calculus is an interesting problem where further work is required. The expressiveness of our polymorphic type system makes it very unlikely that type inference will be decidable in general. The Pict compiler currently gets around this problem by using a partial type inference algorithm. This allows most, but not all, explicit type information to be omitted. Experience with Pict suggests that a partial type inference algorithm is quite usable in practice, since one often writes explicit type information in programs anyway (as a simple form of program documentation). However, if any sort of type inference is to appear in the formal definition of Pict, a more abstract description of partial type inference is required.

The compilation of π-calculus to C presented in this dissertation now forms the basis of the Pict language implementation. The compilation implemented in the Pict compiler really does match what we have presented here (modulo a few simple refinements which avoid incrementing the free space pointer too often, and which cache the values of global variables as local variables to enable the C compiler to do a better job of optimising Pict programs). It is nice to find that the π-calculus has such a simple and concise compilation which, despite its simplicity, is able to provide reasonable performance. Having an efficient implementation of π-calculus makes it possible to do real programming in Pict. The largest Pict programs developed so far (which comprise approximately four thousand lines of Pict code) implement a graphical user interface toolkit, the performance of which is perfectly acceptable, though some form of incremental garbage collection would be helpful.

Our compilation of π-calculus to C is designed so that it can easily exploit information about a channel's status at compilation time. An interesting area for further work is the development of appropriate program analyses for π-calculus. The linear type system proposed by Kobayashi, Pierce and the author [KPT96] is one example of such an analysis, but there are many further potential refinements.

The examples presented in Chapter 2 hint at a problem one encounters when programming in the π-calculus: the π-calculus is quite a low-level language (its most irritating feature is the need to always deal with result channels explicitly). We solved this problem here by defining a number of derived forms. The Pict language takes a similar approach: it starts with a core language (which is just asynchronous π-calculus extended with built-in structured data), and then defines a high-level language via a series of derived forms. This style of formal language definition is very concise (the type system can be presented in four pages, the operational semantics in one page, the derived forms in three pages and the derived typing rules in two pages). It is not necessary to extend the Pict type system when we add new derived forms, since we can simply derive high-level typing rules from our basic π-calculus typing rules (as we did in Sections 3.3 and 3.4, for example). This has the additional benefit that the soundness of the high-level Pict type system depends only on the soundness of Pict's core language type system.

In conclusion, we believe that the π-calculus type system presented here enables one to use the π-calculus as a simple type-theoretic foundation for concurrent programming. Moreover, our compilation of π-calculus to C is efficient enough to allow one also to use the π-calculus as a basis for compiling high-level concurrent programming languages.

Bibliography

[AC91] Robert M. Amadio and Luca Cardelli. Subtyping recursive types. In *ACM Symposium on Principles of Programming Languages*, January 1991.

[AM87] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987.

[Ama94] Roberto M. Amadio. Translating core Facile. Technical Report ECRC–TR–3–94, European Computer-Industry Research Centre, 1994.

[BO95] Simon Brock and Gerald Ostheimer. Process semantics of graph reduction. Technical Report CS/95/2, Computer Science Division, University of St Andrews, 1995.

[Car85] Luca Cardelli. An implementation model of redezvous communication. In *Seminar on Concurrency*, number 197 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

[Car86] Luca Cardelli. The Amber machine. In *Combinators and Functional Programming Languages*, number 242 in Lecture Notes in Computer Science. Springer-Verlag, 1986.

[CC91] Felice Cardone and Mario Coppo. Type inference with recursive types. Syntax and semantics. *Information and Computation*, 92(1):48–80, 1991.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, January 1982.

[EG95] Joost Engelfriet and Tjalling Gelsema. Multisets and structural congruence of the pi-calculus with replication. Technical Report TR95–02, Department of Computer Science, Leiden University, Leiden, The Netherlands, 1995.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *ACM Symposium on Principles of Programming Languages*, 1993.

[Gir72] J.Y. Girard. *Interprétation Fonctionelle et Elimination des Coupres dans l' Arithmétique d'Ordre Supérieur*. PhD thesis, Paris, 1972.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. In *International Journal of Parallel Programming*, volume 18, April 1989.

[HJW$^+$92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. Technical report, Version 1.2, March 1992.

[HL92] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. Technical Report CS–CMU–92–210, Department of Computer Science, Carnegie Mellon University, October 1992.

[Hol83] S. Holmström. PFL: A functional language for parallel programming. In *Declarative Programming Workshop, Programming Methodology Group, Chalmers University of Technology, University of Goteborg, Sweden*, 1983.

[Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), April 1992.

[KNY95] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis on communication for asynchronous concurrent programming languages. Technical Report 95–04, Department of Information Science, University of Tokyo, April 1995.

[KPT96] Naoki Kobayashi, Benjamin Pierce, and David N. Turner. Linearity and the pi-calculus. In *ACM Symposium on Principles of Programming Languages*, 1996.

[Ler93] Xavier Leroy. Polymorphism by name for references and continuations. In *ACM Symposium on Principles of Programming Languages*, pages 220–231, 1993.

[Mat91] David C. J. Matthews. A distributed concurrent implementation of Standard ML. Technical Report ECS–LFCS–91–174, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1991.

[Mil77] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1977.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.

[Mil90] Robin Milner. Functions as processes. Technical Report 1154, Institut National de Recherche en Informatique et en Automatique, Sophia-Antipolis, February 1990. Final version in *Journal of Mathematical Structures in Computer Science* 2(2):119–141, 1992.

[Mil91a] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, University of Edinburgh, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

[Mil91b] Robin Milner. Sorts in the π-calculus. Presented at the Third Workshop on Concurrency and Compositionality, Goslar, Germany, 1991.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. In *ACM Transactions on Programming Languages and Systems*, pages 470–502, July 1988.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[MPW89a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part 1. Technical Report ECS–LFCS–89–85, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1989.

[MPW89b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part 2. Technical Report ECS–LFCS–89–86, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1989.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NN94] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1994.

[OD93] Gerald K. Ostheimer and Antony J. T. Davie. π-calculus characterisations of some practical λ-calculus reduction strategies. Technical Report CS/93/14, Department of Mathematical and Computing Sciences, University of St Andrews, October 1993.

[Pie95] Benjamin C. Pierce. Abstract machines and fairness for Pict. Personal Communication, 1995.

[PS93] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *IEEE Symposium on Logic in Computer Science*, June 1993.

[PT95a] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, April 1995.

[PT95b] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus, 1995.

[PT95c] Benjamin C. Pierce and David N. Turner. *Pict Standard Libraries*. Department of Computer Science, University of Cambridge and University of Glasgow, 1995.

[Rep92] John H. Reppy. Concurrent ML: Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, number 693 in LNCS, pages pp. 165–198. Springer-Verlag, 1992.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Lecture Notes in Computer Science 19, Springer-Verlag*, 1974.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In *IFIP Congress, Paris*, September 1983.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1), January 1965.

[San93a] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, 1993. Also published as technical report ECS–LFCS–93–266.

[San93b] D. Sangiorgi. From π-calculus to Higher-Order π-calculus — and back. In *Proceedings TAPSOFT, LNCS 668.* Springer-Verlag, 1993.

[San93c] Davide Sangiorgi. A theory of bisimulation for the π-calculus. Technical Report ECS–LFCS–93–270, Laboratory for Foundations of Computer Science, University of Edinburgh, 1993.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical Report CMU–CS–90–187, Department of Computer Science, Carnegie Mellon University, November 1990.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tho93] Bent Thomsen. Polymorphic sorts and types for concurrent functional programs. Technical Report ECRC–93–10, European Computer-Industry Research Centre, 1993.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Department of Computer Science, University of Edinburgh, May 1988.

[Vas94] Vasco T. Vasconcelos. Predicative polymorphism in the π-calculus. In *Proceedings of 5th Conference on Parallel Architectures and Languages*, Lecture Notes in Computer Science. Springer-Verlag, July 1994.

[VH93] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic pi-calculus. In *Proceedings of CONCUR '93*, July 1993. Also available as Keio University Report CS-92-004.

[Wal91] David J. Walker. π-calculus semantics of object-oriented programming languages. In *Conference on Theoretical Aspects of Computer Software, Tohoku University, Japan*, 1991.