Programming in the Pi-Calculus

A Tutorial Introduction to Pict

(Pict Version 3.8d)

Benjamin C. Pierce

Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
United Kingdom
benjamin.pierce@cl.cam.ac.uk

July 10, 1996

This version of the Tutorial corresponds to an interim release of Pict. It is not completely up to date with the language, and some sections are broken. Nevertheless, Chapters 1 to 7 are in fairly good shape.

Abstract

Pict is a programming language in the ML tradition, formed by adding high-level derived forms and a powerful static type system to a tiny core language. The core, Milner's pi-calculus, has been used as a theoretical foundation for a broad class of concurrent computations. The goal in Pict is to identify idioms that arise naturally when these primitives are used to build working programs — idioms such as basic data structures, protocols for returning results, higher-order programming, selective communication, and concurrent objects. The type system integrates a number of features found in recent work on theoretical foundations for typed object-oriented languages: higher-order polymorphism, simple recursive types, subtyping, and a useful partial type inference algorithm.

This is a tutorial introduction to Pict, with examples and exercises.

Consumer Safety Warning

Pict is an evolving language design and the current implementation is experimental software. You are welcome to use Pict in any way you like, but please keep in mind that future versions may differ substantially from what you find here.

Product Feedback Hotline

If you would like to be kept informed of new releases of Pict, please send your address to `benjamin.pierce@cl.cam.ac.uk`. Comments, suggestions, and bug reports are also welcome. (Please send these to both `benjamin.pierce@cl.cam.ac.uk` and `dnt@dcs.gla.ac.uk`.) Suggestions for improvement of the documentation are especially welcome.

Copying

Citations

The best bibliographic single citation for Pict is [PT96a]. Others that may be of interest are [PT95, PT96b, Tur96] and [Pie96] (this document).

Contents

Acknowledgements

The Pict language design and implementation are joint work with David N. Turner.

Robin Milner's past and present work on programming languages, concurrency, and the π-calculus in particular is strongly in the background of this project, and conversations with Robin have contributed many specific insights. The idea of basing a programming language design on the π-calculus was planted by Bob Harper and developed into a research project in the summer of 1992 in discussions of concurrent object-oriented programming languages with the Edinburgh ML Club. From Davide Sangiorgi, we learned about the higher-order π-calculus and the many ways of encoding λ-calculi in the π-calculus; we also did a lot of thinking together about static type systems for the π-calculus [PS93]. Didier Rémy helped build the original Pic compiler (on which the first version of the present Pict compiler was based [PRT93]) and joined in many discussions about the integration of processes and functions. Uwe Nestmann's research on proof techniques for compilations between concurrent calculi [NP96] sharpened our ideas about the formal foundations of Pict. Martin Steffen helped study the formal foundations of the core subtyping algorithm [**?**]. Dilip Sequeira contributed both code and ideas to the implementation of type inference and record type checking.

Conversations with Luca Cardelli, Georges Gonthier, Cliff Jones, Oscar Nierstrasz, and John Reppy have deepened our understanding of the π-calculus and concurrent programming languages.

This document began as notes for a series of lectures given at the Laboratory for Foundations of Computer Science, University of Edinburgh, during May and June, 1993. The language and notes were refined for another course at the *1993 Fränkische OOrientierungstage* sponsored by the University of Erlangen-Nürnberg, and again for a winter postgraduate course at the LFCS in 1994. Early drafts were written at INRIA-Roquencourt, with partial support from Esprit Basic Research Actions Types and Confer. Work has continued at the University of Edinburgh and, later, at the Computer Laboratory, University of Cambridge, with support from Confer and from the British Science and Engineering Research Council.

Part I

Basics

Chapter 1

Processes and Channels

The π-calculus of Milner, Parrow, and Walker [MPW92] can be compared to the λ-calculus developed by Church and his students in the 1920's and 30's [Chu41]. Though its origins predate computer science itself, the λ-calculus has come to be regarded as a *canonical* calculus capturing the notion of sequential computation in a clean, mathematically tractable way. Many of the fundamental issues of sequential programming languages can be studied by considering them in the more abstract setting of the λ-calculus. Conversely, the λ-calculus has influenced the design of numerous programming languages, notably Landin's ISWIM [Lan66] and McCarthy's LISP [McC78].

The π-calculus represents a synthesis and generalization of many years of work on process calculi such as CCS [Mil80, Mil89, etc.]. In the concurrency community, the π-calculus and similar "modern process calculi" are widely studied, and a substantial body of theoretical work has accrued. More important for our present purposes, though more difficult to quantify, is the observation that the π-calculus appears to be a much more *computationally complete* model of real-world concurrent programs than previous formal theories of concurrency. For example, in pure CCS there is no notion of "value": the entities passed along communication channels are just signals, carrying no additional information. This is fine for studying the basic concepts of concurrency, but as soon as we want to write a program, we find that we need various primitive structures such as integers and booleans that are not present in the pure calculus. These structures can be added, yielding a somewhat more complex system that nevertheless remains theoretically tractable [Mil89]. But value-passing CCS lacks another fundamental property: the ability to perform higher-order programming. For example the fundamental operation of constructing process networks by connecting processes and channels cannot be expressed in CCS, with or without values. Such considerations imply that, although there are several programming languages whose communication facilitites are based on CCS, we cannot design a complete language using *only* CCS as its formal foundation.

The π-calculus, on the other hand, does support both higher-order programming and natural encodings of primitive datatypes. The ability to pass channels themselves as values between processes — its defining characteristic — turns out to yield sufficient power to construct dynamically evolving communication topologies and to express a broad range of higher-level constructs. Basic algebraic datatypes like numbers, queues, and trees can be encoded as processes, using techniques reminiscent of Church's encodings in the λ-calculus. Indeed, the λ-calculus itself can be encoded fairly straightforwardly by considering β-reduction as a kind of communication [Mil90]. Thus, the step from the π-calculus to a high-level notation suitable for general-purpose concurrent programming should be of the same order of magnitude as the step from λ-calculus to early dialects of LISP.

The π-calculus in its present form is not the final word on foundational calculi for concurrency. Milner himself is now considering much more refined systems [Mil92, Mil95], and heated discussion continues in the concurrency community as to what should constitute a general theory of concurrency. Nevertheless, it seems we've reached a good point to begin experimenting. If Lisp (or, if you prefer, ML, or Scheme, or Haskell) is a language based directly on the λ-calculus, then what could a language based directly on the π-calculus look like? The programming language Pict represents our attempt to find out.

This chapter offers an informal introduction to a fragment of the Pict language closely corresponding to the pure π-calculus.[1] Chapter 2 develops a more precise treatment of the operational semantics of an even smaller fragment, called the *core language*. Chapter 3 introduces some refinements in the type system sketched in Chapter 1, principally the idea of subtyping. Chapter 4 reintroduces some convenient syntactic forms that were dropped between Chapters 1 and 2 and shows how they can be understood via simple translations into the core. Chapter 5 defines the first-order part of the type system of the core language precisely. Chapter 6 adds some more complex translation rules yielding convenient high-level syntactic constructs such as function application. The remaining chapters develop examples, illustrate programming styles, and cover more advanced aspects of the type system.

1.1 Simple Processes

The π-calculus is a notation for describing concurrent computations as systems of communicating agents. The basic unit of computation is a *process*.

The simplest process, written `()`, has no observable behavior. To make this process expression into a complete Pict program — albeit not a very useful one — we prefix it with the keyword `run`:

```
    run ()
```

Two or more processes may be executed in parallel by separating them with bars and enclosing them in parentheses:

```
    run (() | () | ())
```

This program, which is not much more useful than the first, consists of three subprocesses in parallel; each of these, when it is given the chance to begin execution, immediately finishes and disappears. The order in which the three processes are executed is arbitrary.

We may be interested in watching which order is chosen for these three to begin their short lives. This can be accomplished with a process of the form `print!"abc"`, which causes the string `abc` to be printed on the standard output stream. For example:

```
    run ( print!"peering"
        | print!"absorbing"
        | print!"translating")

peering
absorbing
translating
```

In this document, lines of output from the running program are left-justified to distinguish them from program text.

[1]Readers familiar with the theoretical literature will notice that the language presented here is not precisely the original formulation of the π-calculus. The primary differences are: (1) like the systems of Honda and Tokoro [HT91] and Boudol [Bou92], output in this fragment is asynchronous: the sender cannot tell when it has actually occurred; (2) channels are typed; (3) the polyadic π-calculus is slightly generalized to allow the communication not only of tuples of channels, but of tuples of tuples, etc; and (4) for technical convenience, booleans are included in the core language. There are also many differences in concrete syntax.

1.2 Channels and Communication

Besides processes, the other significant entities in the π-calculus are *channels* (also called *names* in the π-calculus literature). A channel is a port over which one process may send messages to another.

Suppose `x` is a channel. Then the expression `x![]` denotes a *sender* process that transmits the *signal* `[]` along the channel `x`. This transmission cannot occur until another process is ready to accept a value along the same channel `x`; such a *receiver* process has the form `x?[] = e`, where `e` is a process expression indicating what to do after the signal has been received. When the two are placed in parallel, the communication can take place:[2]

```
    run (x?[] = print!"Got it!" | x![])
```

```
Got it!
```

Note that the nearly identical program

```
    run x?[] = (print!"Got it!" | x![])
```

prints nothing, since the output `x![]` cannot take place until *after* the input `x?[]` has succeeded. In general, the body `e` of an input expression `x?y = e` remains completely inert until after a communication along `x` has occurred.

Sending a signal from one process to another is a useful way of achieving synchronization between concurrent threads of activity. For example, a signal sent from process `e` to process `f` might carry the information that `e` has finished modifying some data structure that it shares with `f`, or that it has completed some action that `f` requested. This sort of synchronization is a ubiquitous feature of Pict programs.

It is often useful for two processes to exchange some value when they synchronize. In particular, the π-calculus allows a *channel* to be passed from the sender to the receiver as part of the act of communication. As we shall see, this ability is a primary source of the π-calculus's surprising expressive power.

```
    run (x?z = print!"Got it!" | x!y)
```

```
Got it!
```

The name `x` plays the same role in the expressions `x!y` and `x?z = e`: it is the "location" where the two processes meet and exchange information. But the roles of `y` and `z` are different: `y` can be thought of as an *argument* of the message, whereas `z` is a *bound name* that, at the moment of communication, is replaced by the received value `y`.

Of course, the receiving process may do other things with channels it obtains by communication. In particular, it may use them for communication, either as the channel on which to send or receive a value...

```
    run ( x!y
        | x?z = z!u
        | y?w = print!"Got it!")
```

```
Got it!
```

... or as the value that it sends along some other channel:

[2]This is not yet a complete program: if you try to run it, the compiler will complain about `x` being an unbound name.

```
    run ( x!y
        | x?z = a!z
        | a?w = print!"Got it!")

Got it!
```

1.3 Replication

Concurrent programs typically involve infinite behavior. For example, they often contain *server processes* that repeatedly perform some actions in response to requests from other processes. Such a server is never "finished": when a request has been handled, it simply waits for another, deals with it, waits again, and so on.

Infinite behaviors are provided in the π-calculus by *replicated input* expressions, like

```
    x?*[] = print!"Got it!"
```

(with a `*` after the `?`). A replicated input expression behaves just like an ordinary input expression, except that it is not used up during communication: when it is placed in parallel with a sender and a communication occurs, the result is a running copy of the body of the receiver *and* a copy of the replicated receiver itself. For example, putting a replicated receiver in parallel with three senders

```
    run ( x?*[] = print!"Got it!"
        | x![]
        | x![]
        | x![] )

Got it!
Got it!
Got it!
```

results in three copies of the receiver's body (and a copy of the receiver itself), whereas the same program with an unreplicated receiver results in just one copy of the receiver's body (plus two frustrated sender processes).

```
    run ( x?[] = print!"Got it!"
        | x![]
        | x![]
        | x![] )

Got it!
```

1.4 Values and Patterns

In general, each communication step involves the atomic transmission of a single *value* from sender to receiver. In the fragment of Pict we consider in this chapter, values may be constructed in just two ways:

1. a channel is a value;
2. if $\mathtt{v}_1$ through $\mathtt{v}_n$ are values, then the tuple `[`$\mathtt{v}_1$ `...` $\mathtt{v}_n$`]` is a value. Note that we write tuples with just whitespace between adjacent elements.

For example, if `x` and `y` are channels, then `x`, `[x y]`, and `[x [[y x]] y [] x]` are values. The signal `[]` is just the empty tuple of values. Character strings like `"Got it"` are also values, but for purposes of this chapter they may only appear as arguments to `print`.

The general form of a sender process is `x!v`, where `x` is a channel and `v` is a value. Symmetrically, a receiver process has the form `x?p = e`, where `p` is a *pattern* built according to the following rules:

1. a variable is a pattern;

2. if $\mathtt{p}_1$ through $\mathtt{p}_n$ are patterns binding distinct sets of variables, then $[\mathtt{p}_1\ \ldots\ \mathtt{p}_n]$ is a pattern.

For example, `[]`, `[x y z]`, and `[[] x y [[]]]` are patterns.

When a sender `x!v` communicates with a receiver `x?p = e`, the value `v` is matched against the pattern `p` to yield a set of bindings for the variables in `p`. For example, matching the value `[a b []]` against the pattern `[m n o]` yields the bindings $\{\mathtt{m} \mapsto \mathtt{a},\ \mathtt{n} \mapsto \mathtt{b},\ \mathtt{o} \mapsto \mathtt{[]}\}$. More precisely:

1. any value `v` matches a variable pattern `x`, yielding the singleton binding $\{\mathtt{x} \mapsto \mathtt{v}\}$;

2. if `p` has the form $[\mathtt{p}_1\ \ldots\ \mathtt{p}_n]$ and `v` has the form $[\mathtt{v}_1\ \ldots\ \mathtt{v}_n]$ and, for each i, the value $\mathtt{v}_i$ matches the subpattern $\mathtt{p}_i$ yielding the binding Δ_i, then `v` matches the whole pattern `p`, yielding the set of bindings $\Delta_1 \cup \cdots \cup \Delta_n$.

1.5 Types

Given an arbitrary pattern `p` and value `v`, it is perfectly possible that `p` does *not* match `v`. As in the λ-calculus, it is convenient to impose a *type discipline* on the use of channels that permits such situations to be detected statically. Like ML, Pict is *statically typed*, in the sense that a well-typed program cannot encounter a situation where the value provided by a sender does not match the pattern of a possible receiver.

The type system of Pict is quite rich, incorporating features such as higher-order polymorphism, subtyping, records, recursive types, and a powerful type-inference mechanism. But there is no need to discuss all of these at once; they will be introduced one by one in later chapters. For now, two rules suffice:

1. If the values $\mathtt{v}_1$ through $\mathtt{v}_n$ have the types $\mathtt{T}_1$ through $\mathtt{T}_n$, then the tuple value $[\mathtt{v}_1\ \ldots\ \mathtt{v}_n]$ has the type $[\mathtt{T}_1\ \ldots\ \mathtt{T}_n]$. (In particular, the value `[]` has the type `[]`. Although the value `[]` and its type are written using the same sequence of characters, there is no danger of confusion: it will always be clear whether a given expression should be regarded as a value or a type.)

2. Each channel may be used to send and receive values of exactly one type. If this type is `T`, then the type of the channel is `^T`, read "channel of `T`."

For example, in the expression

```
run (x?[] = () | x![])
```

the channel `x` has type `^[]`. In

```
run w?[a] = a?[] = ()
```

a has type ^[] and w has the type ^[^[]].

Rule 2 embodies an essential restriction: a Pict channel may not be used in one place to send a signal and in another to send a channel. Although (as in any type system) this restriction excludes some reasonable and possibly useful programs, relaxing it would mean using a dynamic flow analysis for typechecking a program. If each channel is used, throughout its lifetime, to carry values of the same shape, then well-typedness becomes a static property and a wide range of well-understood algorithmic and theoretical ideas can be applied in the engineering of the type system.

It is often convenient to make up abbreviations for commonly used types. The declaration

```
type X = T
```

makes the symbol X stand for the type T in what follows.

1.6 Channel Creation

New channel names are introduced by the new declaration:

```
new x:^[]
run (x![] | x?[] = ())
```

The declaration new x:^T creates a fresh channel, different from any other channel, and makes the name x refer to this channel in the scope of the declaration. The values sent and received on x will have the type T.

The keywords new, run, and type all introduce *declaration clauses*; we will see a few other kinds of declaration clauses later on. A Pict program is simply a sequence of declaration clauses, where the scope of variables introduced in one clause is all of the following clauses.

It is also possible to prefix any process expression with a sequence d of private declarations: (d e) is a process expression in which the scope of the variables introduced by d is just e. So the previous example could just as well have been written

```
run
  (new x:^[]
   (x![] | x?[] = ()))
```

or even:

```
run
  (new x:^[]
   run x![]
   x?[] = ())
```

Two new declarations binding the same channel name may be used in different parts of the same program:

```
run ( (new x:^[] (x![] | x?[] = ()))
    | (new x:^[] (x![] | x?[] = ())))
```

There is no danger of confusion between the two communications on x: the output in the first line can only synchronize with the input in the same line, and likewise for the second line. Two declarations of the same name may even have overlapping scopes. In this case, the inner binding hides the outer one. For example, this program does *not* print "Got it":

```
run
  (new x:^[]
   ( (new x:^[] x![])
   | x?[] = print!"Got it?"))
```

It is often useful to communicate a channel outside of the scope of the `new` declaration where it was created. For example, the program fragment

```
run (new x:^[]
     ( z!x
     | x?[] = print!"Continuing..."))
```

creates a channel `x`, sends it along `z` to some colleague in the outside world, and then waits for a response along `x` before continuing. In the π-calculus literature, the possibility of names escaping beyond the scope of their declarations is called *scope extrusion.*

Now we have enough constructs to show a more interesting example. The process `b?[t f] = t![]` reads two channels from the channel `b` and transmits a signal along the first one. This is a reasonable encoding of the boolean value `true`, since (like Church's encoding of booleans in λ-calculus) it gives an outside observer the ability to select between two courses of action by interrogating this process along `b` and choosing its next action based on whether a response arrives along `t` or `f`; the value `false` can be encoded by the analogous process `b?[t f] = f![]`. If some other process wants to test whether it is running in parallel with the first or the second of these, all it has to do is transmit a pair of channels and then listen to see where the response appears:

```
new b:^[^[] ^[]]
new t:^[]
new f:^[]

run {- The process "false" -}
    b?[t f] = f![]

run {- The testing process -}
    ( b![t f]
    | t?[] = print!"It´s true"
    | f?[] = print!"It´s false" )
```

```
It´s false
```

Before we continue, let's tidy up this program by making the scopes of the `new` declarations as small as possible. The only global channel is `b`; the other two are local to the testing process:

```
new b: ^[^[] ^[]]

run {- The process "false" -}
    b?[t f] = f![]

run {- The testing process -}
    (new t:^[]
     new f:^[]
     ( b![t f]
     | t?[] = print!"It´s true"
     | f?[] = print!"It´s false" ))
```

(Since the output of this process is the same as the previous one, it is not shown. From now on, we often omit output from examples.) Also, since the expression `^[^[] ^[]]` is unwieldy to type repeatedly, let us introduce the abbreviation

```
type Boolean = ^[^[] ^[]]
```

for the remainder of the chapter. The declaration at the head of our example program can then be written:

```
new b: Boolean
```

1.7 Process Abstractions

We can think of the clause labelled with the comment "`{- the process "false" -}`" in this program as "the process `false` *located at* `b`," since `b` is the channel over which its truth value can be tested. But there is nothing essential in the choice of the name `b`: we might just as well locate the value `true` or `false` at any other channel, or at many different channels at various points in a large program. Pict provides a convenient notation for such situations. If we declare

```
def tt[b:Boolean] = b?[t f] = t![]
and ff[b:Boolean] = b?[t f] = f![]
```

then `tt![b]` can be used in the rest of the program as an abbreviation for the expression `b?[t f] = t![]`. (The resemblance of the notations for output and for instantiation of abbreviations such as `tt` is not accidental, as we shall see in following chapters.)

Similarly, we can define a generic testing process:

```
def test[b:Boolean] =
    (new t:^[] (new f:^[]
      ( b![t f]
      | t?[] = print!"It´s true"
      | f?[] = print!"It´s false")))
```

Now the "main program" is:

```
new b:Boolean
run ( ff![b]
    | test![b])
```

```
It´s false
```

This encoding of the booleans values is not completely satisfying. Each of our booleans can interact only once over its location channel; then it is used up. To fix this, we could make the testing process restart an equivalent boolean at the same location:

```
def test[b:Boolean] =
      (new t:^[]
       new f:^[]
       ( b![t f]
       | t?[] = (print!"It´s true" | tt![b])
       | f?[] = (print!"It´s false" | ff![b]) ))
```

But this is a lot of work to impose on every testing process. It is simpler to do the same thing in the definitions of `tt` and `ff` themselves by replicating the inputs on `b`:

```
def tt[b:Boolean] = b?*[t f] = t![]
and ff[b:Boolean] = b?*[t f] = f![]
```

1.8 Records

So far, we have seen two basic forms of values: channels and tuples. These two are usually taken as primitive in the π-calculus, and it is possible to do a surprising amount of programming with nothing else. However, for some of the programs we shall want to write in Pict, other forms of values are also helpful. In particular, for experimenting with concurrent objects we shall want to use structured values with named fields, i.e. records.

Because records in Pict programs are often fairly large, we use a hybrid keyword-parenthesis syntax for their "brackets." For example, a record with two fields, `l` and `m`, with the values `a` and `b`, is written:

```
(record l=a m=b)
```

Records, like tuples, are values — i.e., they may be sent and recieved along channels. To transmit our record along the channel `c`, we write, for example:

```
c ! (record l=a m=b)
```

During pattern-matching, the fields of records may be accessed by name. For example, if the input expression

```
c ? (record l=x m=y) = e
```

succeeds in communicating with the output just above, then the variable `x` in the body `e` is bound to the value `a` and the variable `y` is bound to `b`.

The types of record values are also written with "keyword brackets." The type of empty records is `(record)`. If the types of `a` and `b` are `A` and `B`, respectively, then the type of the value `(record l=a m=b)` is `(record l:A m:B)` and the type of the channel `c` is `^(record l:A m:B)`.

1.9 Booleans

The encoding of the booleans that we have seen in this chapter is important as an illustration of the π-calculus's expressive power: it indicates that just the primitives of input, output, replicated input, parallel composition, and channel creation are enough to faithfully encode other familiar and useful structures. We shall see much more of this in later chapters, as we move from the Pict core to the full language. However, it is much more convenient to program with booleans using the familiar `if...then...else...` syntax.

Pict provides two built-in boolean values, written `true` and `false`; their type is written `Bool`. The conditional construct is written (for example):

```
if false then x![y] else x![z]
```

1.10 Exercises

Instructions for running the Pict compiler can be found in the Pict Compiler Manual.

1.10.1 Exercise [Recommended]: *Define a negation operation on the encoded type* `Boolean`*. If* `b` *is the location of some boolean value (i.e., either the process* `tt![b]` *or the process* `ff![b]` *is present in the environment) and* `c` *is a fresh channel, then running* `notB![b c]` *should attach to* `c` *a process representing the negation of* `b`*. [Solution on page 93.]*

1.10.2 Exercise [Recommended]: *Define a conjunction operator* `andB` *on the encoded booleans. If* `b1` *and* `b2` *are the locations of two boolean processes and* `c` *is a fresh channel, then* `andB![b1 b2 c]` *should create a boolean process representing the logical conjunction of* `b1` *and* `b2` *and locates it at* `c`*. [Solution on page 93.]*

1.10.3 Exercise [Recommended]: *In the previous two exercises, the encoded boolean operators "returned their results" by creating a new boolean process located at a certain channel provided as an argument. In effect, it was the caller's responsibility to allocate the storage for the new boolean value computed by each call to* `notB` *or* `andB`*. Another possible arrangement would be to make the operators themselves allocate this storage. For example, if* `b` *is a boolean and* `res` *is a channel of type* `^Boolean`*, then* `notB![b res]` *creates a process that will send along* `res` *the location of a boolean value representing the negation of* `b`*. Rewrite your solutions to the previous exercises using this protocol for returning results. [Solution on page 93.]*

What advantages does each scheme have over the other?

1.10.4 Exercise: *Write a pair of operators for converting between the type* `Bool` *of primitive booleans and the type* `Boolean` *of encoded booleans:*

```
def boolBoolean[b:Bool r:^[Boolean]] =
  ...
def booleanBool[b:Boolean r:^[Bool]] =
  ...
```

Chapter 2

Core Language

Defining a full-scale language rigorously requires some work. Our approach is to present first a very small, untyped sublanguage, called *core Pict*, and explain the meaning of other language constructs by means of translations into this core. Chapter 1 introduced (a slight superset of) the core informally, relying on english descriptions, examples, and exercises to convey a practical understanding of the language. We now develop it precisely. The present chapter deals with the syntax and operational semantics of the core; Chapter 5 covers the foundations of the type system.

The presentation in this chapter and Chapter 5 is somewhat more technical than is usually found in programming language tutorials: I believe it is easier to understand a complex object from a precise description than from a gentle but vague tour of its features. The extra work involved in getting through this part should be repaid by a clear understanding of fundamental principles that will make the rest of the language easy to grasp.

2.1 Overview

The complete definition of core Pict semantics recaputilates the structure of a compiler:

$$\begin{tikzcd}
\boxed{\text{ascii program text}} \arrow[d, "\text{lexical analysis}"] \\
{} \arrow[d, "\text{parsing}"] \\
\boxed{\text{abstract syntax tree (Section 2.2)}} \arrow[d, "\text{scope resolution}"] \\
\boxed{\text{nameless abstract syntax (Section 2.3)}} \arrow[d, "\text{execution (Section 2.4)}"] \\
{}
\end{tikzcd}$$

First, the source text of the program is transformed by a lexical analyzer into a sequence of *tokens*: identifiers, keywords, and so on. If the program is well formed, this sequence of tokens matches the *concrete syntax* grammar of Pict in exactly one way, allowing the parser to transform it into an *abstract syntax* tree. The scope resolution phase identifies free occurrences of variables with the points at which they are bound, simplifying the description of later phases. The data structure generated during scope resolution is called a *nameless* abstract syntax tree because each variable occurrence includes a pointer directly to its corresponding binder. The nameless abstract syntax tree is then executed.

In an actual implementation of Pict, the execution phase has a number of subphases: optimisation, intermediate code generation, optimisation of the intermediate code, native code generation, linking, and finally execution on a real machine. Fortunately, we do not need to treat all of these formally! Instead, we directly define a notion of *reduction of abstract syntax trees* that defines precisely, but at a fairly high-level, the set of allowable behaviors for every program. The compiler's job is to produce a machine-code program whose actual behavior exactly mimics some behavior in this set.

In Chapter 5, we shall add one new phase to this diagram, verifying that the program is well typed before erasing the type information and executing.

$$\begin{tikzcd}
\boxed{\text{program text}} \arrow[d, "\text{lexical analysis}"] \\
{} \arrow[d, "\text{parsing}"] \\
{} \arrow[d, "\text{scope resolution}"] \\
\boxed{\text{nameless abstract syntax}} \arrow[d, "\text{typechecking (Chapter 5)}"] \\
{} \arrow[d, "\text{execution}"] \\
{}
\end{tikzcd}$$

Though the execution phase operates on *typed* programs, the operational semantics is written in such a way that type annotations in a program do not affect its behavior. We use this fact to justify presenting the type system in a series of steps in later chapters. Each step enriches the set of type annotations that may be added to programs and thus allows a larger set of programs to be verified as type-correct. The behavior of programs at run time is completely insensitive to these refinements of the type system.

The definition of the full Pict language in Chapter 6 adds one more phase, which performs the translation from high-level to core abstract syntax.

$$\begin{tikzcd}
\boxed{\text{program text}} \arrow[d, "\text{lexical analysis}"] \\
{} \arrow[d, "\text{parsing}"] \\
{} \arrow[d, "\text{scope resolution}"] \\
{} \arrow[d, "\text{typechecking}"] \\
\boxed{\text{bare (high-level) abstract syntax}} \arrow[d, "\text{translation (Chapter 6)}"] \\
\boxed{\text{bare (core) abstract syntax}} \arrow[d, "\text{execution}"] \\
{}
\end{tikzcd}$$

Intuitively, this translation phase may be thought of as happening either before or after typechecking, since there is little difference between the typing rules for the high-level forms and for their translations. In fact, we place it afterwards. This allows the typechecker to generate much more comprehensible error messages, and also gives the typechecker the freedom to place certain convenient restrictions on the high-level typing rules that would not arise automatically from the low-level rules. (The most important example of this is that `def` clauses define output-only channels, whereas their translation into a `new` and a replicated input would allow an ordinary channel type.)

2.2 Syntax

We now turn to defining the syntax of core Pict.

2.2.1 Notational Conventions

For describing syntax, we rely on a meta-syntactic notation similar to the Backus-Naur Form commonly used in language definitions. Each class of syntactic expressions has a name like *Val* or *Proc*. For each class C, the grammar contains a clause of the form

$$\begin{array}{lll} C & = & X \\ & & Y \\ & & Z \end{array}$$

where X, Y, and Z are the possible forms that members of the class C may have. Each line consists of a sequence of subexpressions, which may be either literal symbols like `!` or `new` or the names of syntactic classes like *Proc* or *Val*. Literal symbols are set in typewriter font.

An expression of the form `X Y Z` ... `X Y Z` stands for any list of zero or more instances of the sequence `X Y Z`. For example, the production

$$\begin{array}{lll} \mathit{TupleVal} & = & \texttt{[}\ \mathit{Val} \ldots \mathit{Val}\ \texttt{]} \end{array}$$

stands for

TupleVal	=	`[` *Val* `]`
		`[` *Val* *Val* `]`
		`[` *Val* *Val* *Val* `]`

and so on. An expression of the form ⟨ `X Y Z` ⟩ stands for either zero or one occurrences of `X Y Z`.

The class *Id*, which is defined by the lexical analyzer, contains identifiers like `x`, `five`, and `help_me´please`. It is described precisely in the "Lexical Conventions" section of the Pict Definition [PT96b].

2.2.2 Concrete Syntax

Figure 2.1 gives the concrete syntax of core Pict, excluding only the syntax of type expressions (which we defer to Chapter 5). Some notes about this definition...

- The grammar is a bit larger than it strictly needs to be, since there are several syntactic categories with just one production in their definitions. It is written in this way to leave growing room for extending these syntactic categories with other constructs later. For example, we need to distinguish the syntactic category *Name* from the category *Id* of identifiers, even though they are defined here to be identical because we will later add some constructs that will be names but not identifiers (cf. Section 4.1.1).

- Record values are built up from the constructors `(record)` and `(... with ...)` instead of being constructed in a single step, as we did in the previous chapter. For example, the two-field record `record l=a m=b end` is more properly written `(((record) with l=a) with m=b)`; the multi-field syntax is provided as a derived form. This is because the formal treatment of records is cleaner when they are built up field-by-field.

- There are some ways in which this syntax is more permissive than the examples in Chapter 1 suggested.

  - In the output expression $\mathtt{v}_1\mathtt{!v}_2$, both $\mathtt{v}_1$ and $\mathtt{v}_2$ are allowed to be arbitrary values, not just identifiers. This means that processes like `[]!x` are syntactically legal and must be weeded out during typechecking. (In principle, since the core abstract syntax also includes these forms, we might ask how such a nonsensical expression behaves. The answer will be "it doesn't," i.e. none of the evaluation rules allow any interaction between such a process and the rest of the program.)
  - Definitions and patterns are both defined in terms of the syntactic class `Abs` of process expressions prefixed by patterns. This means that a process definition need not take a tuple of arguments; it may match its arguments using an arbitrary pattern. It also means that variables bound by input expressions may be annotated with types. This point will be taken up again in Section 5.7; for the rest of this chapter, type annotations are ignored anyway.
  - All type annotations are optional. However, to avoid complicating the formal definitions that follow we shall usually behave as if they were required, relying on the *type inference* algorithm (cf. Chapter 9) to fill in an appropriate type expression wherever one is omitted.
  - We allow a value expression `v` to be annotated with an explicit type declaration by writing `(v : T)`. The typechecker verifies that `T` is really the type of `v` and flags an error if not.

- Conversely, some constructs, like `def`, that we used in Chapter 1 are omitted from the core language. We shall see in Chapter 6 how these can be recovered as derived forms.

TopLevel	=	`run` *Proc*	Whole program
Proc	=	*Val* `!` *Val*	Output (message)
		Val `?` *Abs*	Input prefix
		Val `?*` *Abs*	Replicated input prefix
		`(` *Proc* `|` *Proc* ... `|` *Proc* `)`	Parallel composition
		`(` *DecProc* `)`	Local declarations
		`if` *Val* `then` *Proc* `else` *Proc*	Conditional
DecProc	=	*Dec Proc*	Local declaration
Dec	=	`new` *Name* $\langle$: *Type* $\rangle$	Channel creation
		`type` *Id* `=` *Type*	Type abbreviation
Abs	=	*Pat* `=` *Proc*	Process abstraction
Pat	=	*Name* $\langle$: *Type* $\rangle$	Name
		`(` `record` *Id* `=` *Pat* `)`	Record pattern
		`[` *Pat* ... *Pat* `]`	Tuple pattern
Val	=	*Name*	Name
		`true`	Boolean constant
		`false`	Boolean constant
		`[` *Val* ... *Val* `]`	Tuple
		`(record)`	Empty record
		`(` *Val* `with` *Field* `)`	Record extension
		`(` *Val* : *Type* `)`	Explicit type constraint
Name	=	*Id*	Identifier
Field	=	*Id* `=` *Val*	Record field

Figure 2.1: Core language syntax

- Even though we have already seen that the booleans can be encoded using the other constructs of the core, we include them directly in the core syntax. This is really just a matter of technical convenience: they add almost no complexity to the description of the core language and having them as primitives makes other parts of the language description cleaner.
- For the moment, a program is just the keyword `run` followed by a process expression `e`. Its behavior is the behavior of `e`. For the rest of the chapter, though, we ignore programs and deal only with processes.

2.3 Scoping

Since Pict has several syntactic categories, the scoping of variables is not quite so simple as in the λ-calculus (or the pure π-calculus, for that matter), where one can say, "The x in $\lambda x.\ e$ is a binding occurrence whose scope is e," and be done with it. Instead, we need to identify the syntactic categories that can create new name bindings — *Dec* and *Pat* — and, wherever one of these categories is used in the abstract syntax, specify the scope of the names that it creates.

2.3.1 Definition: The rules for name creation and scoping are as follows:

- In a process expression of the form `(d e)`, the scope of names created by `d` is `e`.
- A declaration clause of the form `new x:T` creates the name `x`.
- A type declaration clause of the form `type X = T` creates the type name `X`.
- In an abstraction of the form `p = e`, the scope of names created by `p` is `e`.
- A pattern of the form `x:T` creates the name `x`.
- A pattern of the form $[\mathtt{p}_1\ \ldots\ \mathtt{p}_n]$ creates all of the names created by the subpatterns $\mathtt{p}_1$ through $\mathtt{p}_n$. The sets of names created by the subpatterns must be disjoint.
- A pattern of the form `(record l=p)` creates the names created by `p`.

In the remaining "phases" of the core language definition (the operational semantics, typechecking, and translation rules), we assume that every expression or program under consideration is *closed*, in the sense that each occurrence of a variable can be uniquely associated with a binder, either within the expression (in a private `new` declaration, etc.) or in the surrounding context. Also, we assume that the names of bound variables are always different, silently renaming variables if necessary to achieve this form. Technically, these conventions can be captured by transforming an abstract syntax tree to a *nameless form* in which free occurrences of variables are replaced by direct pointers to the corresponding binding occurrences [dB72]. We shall not perform this translation formally, though, since it would result in unwieldy notation.

We use the notation $FV(\mathtt{e})$ for the set of variables appearing free in an expression `e`. Formally, $FV(\mathtt{e})$ can be thought of as the set of (binding occurrences of) variables appearing in `e` whose corresponding binding occurrences are not within `e`. Similarly, $BV(\mathtt{d})$ stands for the set of variables bound (created) by the declaration `d`.

2.4 Operational Semantics

The operational semantics of Pict programs is presented in two steps. First, we define a *structural congruence* relation $\mathtt{e}_1 \equiv \mathtt{e}_2$ on process expressions; this relation captures the fact that, for example, the order of the branches in a parallel composition has no effect whatsoever on its behavior. Next, we define a *reduction relation* $\mathtt{e}_1 \rightarrow \mathtt{e}_2$ on process expressions, specifying how processes evolve by means of communication.

2.4.1 Structural Congruence

Structural congruence plays an important technical role as a device for simplifying the statement of the reduction relation. For example, we intend that the processes `(x!v | x?y = e)` and `(x?y = e | x!v)` both reduce to $\{\mathtt{x} \mapsto \mathtt{v}\}\mathtt{e}$. By making these two structurally congruent, we can get away with writing the reduction rule just for the first case and stipulating, in general, that if `e` contains some possibility of communication, then any expression structurally congruent to `e` has the same possible behavior.

The first two structural congruence rules state that parallel composition is commutative

$$(\mathtt{e}_1 \mid \mathtt{e}_2) \equiv (\mathtt{e}_2 \mid \mathtt{e}_1) \qquad \text{(STR-COMM)}$$

and associative:

$$((\mathtt{e}_1 \mid \mathtt{e}_2) \mid \mathtt{e}_3) \equiv (\mathtt{e}_1 \mid (\mathtt{e}_2 \mid \mathtt{e}_3)) \qquad \text{(STR-ASSOC)}$$

The third rule, often called the rule of *scope extrusion* in the π-calculus literature, plays a crucial role in the treatment of channels.

$$\frac{\mathtt{x} \notin FV(\mathtt{e}_2)}{((\mathtt{new\ x{:}T\ e}_1) \mid \mathtt{e}_2) \equiv (\mathtt{new\ x{:}T\ (e}_1 \mid \mathtt{e}_2))} \qquad \text{(STR-EXTRUDE)}$$

Informally, this rule says that a declaration can always be moved toward the root of the abstract syntax tree ("always," because the precondition is always satisfied when the rule is read from left to right[1]). For example, the process expression

```
((new y x!y) | x?z = z![])
```

may be transformed to:

```
(new y (x!y | x?z = z![]))
```

It is precisely this rule that allows the new channel `y` to be communicated outside of its original scope.

[1]Indeed, since we have already performed scope resolution when the structural congruence rules are invoked, we are justified in assuming that the precondition *always* holds. We adopt this view formally, but retain the precondition as a reminder.

2.4.2 Substitution and Matching

To define precisely what happens when two processes communicate, we need some notation for matching values against patterns.

A *substitution* is a finite map from variables to values. The empty substitution is written $\{\}$. A substitution mapping just the variable `x` to the value `v` is written $\{\mathtt{x} \mapsto \mathtt{v}\}$. If σ_1 and σ_2 are substitutions with disjoint domains, then $\sigma_1 \cup \sigma_2$ is a substitution that combines the effects of σ_1 and σ_2. A substitution σ can be extended to a function from values to values by applying σ to variables that fall in its domain and leaving the rest of the value unchanged. For example, applying the substitution $\sigma = \{\mathtt{x} \mapsto \mathtt{a}\} \cup \{\mathtt{y} \mapsto \mathtt{[]}\}$ to the value `[b [x] x y]`, written σ(`[b [x] x y]`), yields the value `[b [a] a []]`.

2.4.2.1 Definition: When a value `v` is successfully matched by a pattern `p`, the result is a substitution $\mathit{match}(\mathtt{p},\mathtt{v})$, defined as follows:

$$\begin{array}{ll}
\mathit{match}(\mathtt{x:T}, \mathtt{v}) & = \{\mathtt{x} \mapsto \mathtt{v}\} \\
\mathit{match}(\mathtt{[p_1 \ldots p_n]}, \mathtt{[v_1 \ldots v_n]}) & = \mathit{match}(\mathtt{p_1},\mathtt{v_1}) \cup \cdots \cup \mathit{match}(\mathtt{p_n},\mathtt{v_n}) \\
\mathit{match}(\mathtt{(record{:}T\ l{=}p)}, \mathtt{(record\ \ldots l{=}v \ldots)}) & = \mathit{match}(\mathtt{p},\mathtt{v}) \\
\mathit{match}(\mathtt{\{|X{<}U|\}p}, \mathtt{\{|T|\}v}) & = \{\mathtt{X} \mapsto \mathtt{T}\} \cup \mathit{match}(\mathtt{p},\mathtt{v}) \\
\mathit{match}(\mathtt{(rec{:}T\ p)}, \mathtt{(rec{:}S\ v)}) & = \mathit{match}(\mathtt{p},\mathtt{v})
\end{array}$$

If `v` and `p` do not have the same structure, then $\mathit{match}(\mathtt{p},\mathtt{v})$ is undefined.

The first clause is the base case of the definition: it states that a variable pattern matches any value and yields a substitution mapping that variable to the whole value. The remaining clauses traverse the structure of the pattern and the value in parallel, comparing their outermost constructors for consistency and then invoking *match* recursively to match corresponding substructures.

2.4.3 Reduction

The reduction relation $\mathtt{e} \rightarrow \mathtt{e}'$ may be read as "The process `e` *can evolve* to the process $\mathtt{e}'$." That is, the semantics is nondeterministic, specifying only what *can* happen as the evaluation of a program proceeds, not what *must* happen. Any particular execution of a Pict program will follow just one of the possible paths.

The most basic rule of reduction is the one specifying what happens when an input prefix meets an output atom:

$$\frac{\mathit{match}(\mathtt{p},\mathtt{v}) \text{ defined}}{\mathtt{(x!v\ |\ x?p\ =\ e)} \rightarrow \mathit{match}(\mathtt{p},\mathtt{v})(\mathtt{e})} \qquad \text{(Red-Comm)}$$

Similarly, when a replicated input prefix meets an output atom, the result is a running instance of the input's body *plus* another copy of the replicated input itself:

$$\frac{\mathit{match}(\mathtt{p},\mathtt{v}) \text{ defined}}{\mathtt{(x!v\ |\ x?{*}p\ =\ e)} \rightarrow (\mathit{match}(\mathtt{p},\mathtt{v})(\mathtt{e})\ \mathtt{|\ x?{*}p\ =\ e)}} \qquad \text{(Red-RComm)}$$

A conditional expression reduces in one step to either its `then` part or its `else` part, depending on the value of the boolean guard:

$$\texttt{if true then } \mathsf{e}_1 \texttt{ else } \mathsf{e}_2 \rightarrow \mathsf{e}_1 \qquad \text{(Red-If-T)}$$

$$\texttt{if false then } \mathsf{e}_1 \texttt{ else } \mathsf{e}_2 \rightarrow \mathsf{e}_2 \qquad \text{(Red-If-F)}$$

The next two rules allow reduction to occur under declarations and parallel composition:

$$\frac{\mathsf{e}_1 \rightarrow \mathsf{e}_2}{(\texttt{new x } \mathsf{e}_1) \rightarrow (\texttt{new x } \mathsf{e}_2)} \qquad \text{(Red-Dec)}$$

$$\frac{\mathsf{e}_1 \rightarrow \mathsf{e}_3}{(\mathsf{e}_1 \mid \mathsf{e}_2) \rightarrow (\mathsf{e}_3 \mid \mathsf{e}_2)} \qquad \text{(Red-Prl)}$$

The body of an input expression, on the other hand, *cannot* participate in reductions until after the input has been discharged.

The structural congruence relation captures the distributed nature of reduction. Any two subprocesses at the "top level" of a process expression may be brought into proximity by structural manipulations and allowed to interact.

$$\frac{\mathsf{e}_1 \equiv \mathsf{e}_2 \rightarrow \mathsf{e}_3 \equiv \mathsf{e}_4}{\mathsf{e}_1 \rightarrow \mathsf{e}_4} \qquad \text{(Red-Str)}$$

In closing, it is worth mentioning that we have done here only part of the work involved in giving a really complete definition of the semantics of Pict. For one thing, we have not talked about the fact that any reasonable implementation of this operational semantics must schedule processes for execution *fairly.* A short discussion of fairness in Pict appears in [PT96a].

For another thing, we have only specified the behavior of *closed programs*, with no connections to the outside world. Of course, real Pict programs do have external connections (such as the `print` channel and, using the libraries provided with the compiler, other external facilities such as file systems and X servers). Peter Sewell has shown how the simple semantics presented here can be extended to model the externally observable behavior of processes.

Chapter 3

Subtyping

Chapter 1 introduced the essentials of Pict's type system: values are assigned types describing their structure; in particular, the types of channels specify the types of the values that they carry. In this chapter, we discuss some refinements of this basic type system. Chapter 5 will have the job of turning the intuitions gleaned here into precise definitions.

3.1 Input and Output Channels

Channel types serve a useful role in ensuring that all parts of a program use a given channel in a consistent way, completely eliminating the possibility of pattern matching failure at run time. Of course, pattern matching failure is just one kind of bad behavior that programs may exhibit; especially in concurrent programs, the minefield of possible programming mistakes is vast: there may be unintended deadlocks, race conditions, and protocol violations of all kinds. Ultimately, one might wish for static analysis tools capable of detecting all of these errors — perhaps even capable of verifying that a program meets an arbitrary specification (expressed, for example, in some modal logic). But the technology required to do this is still some distance away.

Fortunately, there are some simple ways in which bare, structural channel types can be enriched so as to capture useful properties of programs while remaining within the bounds established by current technology. One of the most important of these in Pict is based on the distinction between input and output capabilities for channels.

In practice, it is relatively rare for a channel to be used for both input and output in the same region of a program; the usual case is that some parts of a program use a given channel only for reading while in other regions it is used only for writing. For example, in the boolean example in Section 1.7, the boolean processes `tt` and `ff` only read from the channel `b`, while the client process `test` only writes to `b`.

Pict captures this observation by providing two refinements of the channel type `^T`: a type `!T` giving only the capability to write values of type `T` and a type `?T` giving only the capability to read values of type `T`. For example, the `Boolean` type `^[^[] ^[]]` can be refined in two different ways

```
type ClientBoolean = ![^[] ^[]]
type ServerBoolean = ?[^[] ^[]]
```

expressing the different views of a boolean channel from the perspective of clients (`test`) and servers (`tt` and `ff`):

```
def tt[b:ServerBoolean] = b?[t f] = t![]
```

```
and ff[b:ServerBoolean] = b?[t f] = f![]

def test[b:ClientBoolean] =
      (new t:^[] (new f:^[]
        ( b![t f]
        | t?[] = print!"It´s true"
        | f?[] = print!"It´s false" )))

new b: Boolean
run ( ff![b]
    | test![b] )
```

Note that the `new` declaration at the bottom gives `b` the general type `Boolean`, rather than `serverBoolean` or `ClientBoolean`: since the main program must send `b` to both `ff` and `test`, it must begin with both read and write capabilities for `b`.

The types `ClientBoolean` and `ServerBoolean` can be refined still further: the server processes `tt` and `ff` both read a pair of values `[t f]` from `b` and send a signal on either `t` or `f`. Since `tt` and `ff` never read from the response channels `t` and `f`, there is no reason to give them read capability: it suffices to send along `b` just the write capabilities for the response channels:

```
type Boolean       = ^[![] ![]]
type ClientBoolean = ![![] ![]]
type ServerBoolean = ?[![] ![]]
```

Again, in the definition of `test`, we must still create `t` and `f` with both read and write capabilities

```
def test[b:ClientBoolean] =
      (new t:^[] (new f:^[]
        ( b![t f]
        | t?[] = print!"It´s true"
        | f?[] = print!"It´s false" )))
```

because both are needed here: the write capabilities are sent along `b` (and clearly, `test` can't send capabilities that doesn't have!), while the read capabilities are used directly.

Output-only channel types are also useful for capturing the fact that process definitions introduce channels that may only be used for output. Although the same syntax is used to invoke definitions as to send on ordinary channels, a program like

```
( def c[] = ()
  c?[] = () )
```

does not make sense (and the code generator exploits this fact!). This is captured in Pict by giving `c` the type `![]` instead of `^[]`.

3.2 Subsumption

The well-typedness of the programs in the previous section depends in several places on the observation that a value of type `S` can sometimes be passed along a channel of type `^T` (or `!T`) even though `S` and `T` are not identical types. For example, the channel `test` has type `![ClientBoolean]`, but in the expression `test![b]`, the argument `b` is declared with type `Boolean`, not `ClientBoolean`; we argued that the application should still be allowed because `Boolean` is "better than" `ClientBoolean`. In other words, a value of type `Boolean` can be *regarded as* an element of type `ClientBoolean` without any risk of failure at runtime. We say that `Boolean` is a *subtype* of `ClientBoolean` and write `Boolean` < `ClientBoolean`.

For any type `U`, the type `^U` is a subtype of the types `!U` and `?U`, but `!U` and `?U` themselves are unrelated: neither is a subtype of the other.

By the same reasoning, the tuple type `[^U]` is a subtype of `[!U]` and `[?U]`; in the example, we incur no risk of failure by passing the singleton tuple `[b]` of type `[Boolean]` along the channel `test`, which nominally requires an argument of type `[ClientBoolean]`. In general, if each $\mathtt{S}_i$ is a subtype of the corresponding $\mathtt{T}_i$, then the tuple type $\mathtt{[S_1\ \ldots\ S_n]}$ is a subtype of $\mathtt{[T_1\ \ldots\ T_n]}$.

3.2.1 Exercise [Easy]: *Is*
`S = [^A [^B ?C] ^D ^[^E]]` *a subtype of* `T = [?A [!B ?C] !D ?[^E]]`*? According to what we have said so far, should* `S` *be a subtype of* `T´ = [?A [!B ?C] !D ^[?E]]`*?*

Finally, when writing programs involving subtyping, it is occasionally convenient to have some type that is a supertype of every other type — a maximal element of the subtype relation, functioning as a kind of "don't care" type. We call this type `Top` in Pict.

3.2.2 Exercise [Recommended]: *Rewrite your solutions to the exercises in Section 1.10, using* `!` *and* `?` *instead of* `^` *wherever possible.*

3.2.3 Exercise [Recommended]: *We have seen that the tuple constructor is* monotone *in the subtype relation: if each* $\mathtt{S}_i < \mathtt{T}_i$*, then* $\mathtt{[S_1,\ldots,S_n]} < \mathtt{[T_1,\ldots,T_n]}$*. What about channel types?*

1. *What relation should hold between* `S` *and* `T` *in order for* `!S` *to be a subtype of* `!T`*? For example, would it be correct to allow* `!S` < `!T` *if* `S` *and* `T` *are not identical but* `S` < `T`*? What about when* `T` < `S`*?*

2. *What relation should hold between* `S` *and* `T` *in order for* `?S` *to be a subtype of* `?T`*?*

3. *What relation should hold between* `S` *and* `T` *in order for* `^S` *to be a subtype of* `^T`*?*

Chapter 4

Simple Derived Forms

We now introduce several convenient higher-level programming constructs in the form of predefined channels and simple syntactic sugar. By the end of the chapter, we will have built up enough syntactic forms to do some more serious programming.

4.1 Primitive Values

Like most high-level programming languages, Pict provides special syntax and special treatment in the compiler for a few built-in types, including booleans, characters, strings, and numbers. We sketch some of the facilities available for manipulating these basic types and show how they can be understood in terms of encodings in the core language.

4.1.1 Symbolic Identifiers

When we come to deal with numbers and arithmetic expressions in the next section, it will be convenient to use standard symbolic names for operations like addition. Pict supports *symbolic identifiers* consisting of strings of symbols from the set `!`, `^`, `?`, `#`, `~`, `*`, `%`, `/`, `\`, `+`, `-`, `<`, `>`, `=`, `&`, `@`, `|`, `$`, and , (excluding those strings that are reserved words). A symbolic identifier may be used as an ordinary *Name* by enclosing it in parentheses:

```
    run (new +++:^[]
          ( +++![]
          | +++?*-- = print!"two together"
          | (new +-=&&%:^[]
              ( +-=&&% ?* ** = +++!**
              | +-=&&%![]
              ))))

two together
two together
```

4.1.2 Numbers

Like the booleans, numbers and arithmetic operations can, in principle, be represented in core Pict via a "Church-like" encoding. Such encodings are nice theoretical exercises, illustrating the power of the core language, but they are too inefficient to be useful in practice. As usual in functional languages based on the λ-calculus, we need to add some primitive values to the π-calculus. However, we want to maintain the *illusion* that these primitive values are actually implemented as processes: a program that computes with numbers should not be able to tell whether they are represented as Church numerals or as machine integers. More to the point: a human being *reasoning about* a program that computes with numbers should not have to think about their concrete representation.

A number n can be thought of as a process "located at" a channel `n`; it can be interrogated over `n` to gain information about its value. Higher-level arithmetic operations like `(+)` can be implemented as processes that interrogate their arguments and construct a new number as result. But if a program using numbers always manipulates them by means of processes like `(+)` rather than interrogating them directly, then we can simply think of the channel `n` as *being* the number n. This done, we can introduce a special set of "numeric channels" and an efficient reimplementation of `(+)`, and no one will be able to tell the difference.

This handy abuse of notation depends on a certain choice of *protocol* for `(+)` and similar operations — a certain pattern of passing arguments and receiving results. When we work just with encoded "Church numerals," there are two reasonable choices:

1. The process `+![m n r]` gets information from the processes located at `m` and `n` and uses it to implement the behavior of a process encoding their sum, which is henceforth located at `r`. To add three numbers `l`, `m`, and `n` with this protocol (locating the result at `s`), we would write:

```
(new r:^Int
 (new s:^Int
  ( +![l m r]
  | +![r n s]
  | ...
  )))
```

2. The process `+![m n r]` gets information from `m` and `n`, uses it to implement the behavior of a new process located at a fresh channel `o`, and sends `o` along `r`. To add three numbers this way (and send the result on `s`), we would write:

```
(new r:^Int
 (new s:^Int
  (       +![l m r]
  | r?o = +![o n s]
  | ...
  )))
```

If we want to think of the set of "numeric channels" as fixed and given, then the second protocol works fine but the first does not: the expression `(new r  +![m n r])` creates a brand new channel `r` and then locates some number at `r`. We therefore prefer the second.

With this decision behind us, the language design task of adding primitive values is straightforward. We just need to extend our grammar for values — those entities that can be sent on channels — to include integer constants like `123`. Negative integers are prefixed with a `~`. (N.b. this is a tilde, not a hyphen!)

The Pict distribution comes with a *standard prelude* that defines arithmetic operations like (+), boolean operations like not, and many other useful primitive facilities. These are listed in the Pict Standard Libraries Manual [PT96c]. To add two numbers and print the result, for example, we can write:

```
    run (new r:^Int
          ( +![2 3 r]
          | r?z = printi!z ))
5
```

The printi operation here plays the same role for integers as print does for strings. Note that the result of (+) that is read along r is an individual integer, not a tuple.

A more interesting example, which also illustrates the use of the if construct, is the factorial function. (Don't be alarmed by the length of this example! The derived forms introduced in Chapter 6 make it possible to express such programs much more concisely.)

```
    run
    (new fact: ^[Int !Int]
     (fact?*[n r] =
       (new br:^Bool
         ( {- calculate n=0 -}
           ==![n 0 br]
         | {- is n=0? -}
           br?b =
             if b then
               {- yes: return 1 as result -}
               r!1
             else
               {- no... -}
               (new nr:^Int
                 ( {- subtract one from n -}
                   -![n 1 nr]
                 | nr?nMinus1 =
                     {- make a recursive call to compute fact(n-1) -}
                     (new fr:^Int
                       ( fact![nMinus1 fr]
                       | fr?f =
                           {- multiply n by fact(n-1) and send the
                              result on the original result channel r -}
                           *![f n r]
                       ))))))
     | (new r:^Int
         ( fact![5 r]
         | r?f = printi!f ))
     ))
120
```

4.1.2.1 Exercise [Recommended]: *Use the numeric and boolean primitives to implement a simple algorithm for calculating the fibonacci function:*

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \qquad \textit{when } n > 1 \end{aligned}$$

4.1.3 Characters and Strings

Besides booleans and integers, Pict provides the built-in types `Char` and `String`, with special syntax for values of these types. Character constants are written by enclosing a single character in single-quotes, as in `'a'`. Similarly, string constants are written by enclosing a sequence of zero or more characters in double-quotes. In both strings and character constants, special characters like double- and single-quote are written using the following *escape sequences*:

`\'`	single quote
`\"`	double quote
`\\`	backslash
`\n`	newline (ascii 13)
`\t`	tab (ascii 8)

The escape sequence `\ddd` (where `d` denotes a decimal digit) denotes the character with ascii code `ddd`.

The standard prelude provides a number of operations for manipulating characters and strings. For example, the operation `intString` converts an integer to its string representation, and the operation `(+$)` concatenates strings. Using these, the predefined operation `printi` can be expressed in terms of `print`:

```
    new printi:^Int
    run printi?*i =
         (new r1
          ( intString![i r1]
          | r1?s = print!s ))

    run printi!5
```

```
5
```

Indeed, `print` itself is defined in terms of an even lower-level predefined channel called `pr`. Instead of sending just a string along `pr`, clients must send a pair of a string and a completion channel, which `pr` uses to acknowledge that the string has actually been sent to the standard output stream. This facility is useful when programs need to send many strings to the output in a fixed order.

```
    new done: ^[]
    run ( pr!["Once " done]
        | done?[] =
            ( pr!["Paumanock" done]
            | done?[] =
                pr![", ...\n" done] ))
```

```
Once Paumanock, ...
```

To define `print`, we just ignore the completetion signal returned by `pr`.

```
    new print:^String
    run print?*s = (new r  pr![s r])
```

It is convenient to make the type `Char` a subtype of the type `Int`, so that any character value can implicitly be regarded as the integer representing its ASCII code. For example:

```
    run printi!'a'
```

```
97
```

4.2 Derived Forms for Declarations

In this section, we extend the syntactic category of declarations with a number of handy constructs. (Readers familiar with Standard ML [MTH90] will recognize our debt to its designers here.)

4.2.1 Declaration Sequences

First, to avoid proliferation of parentheses in a sequence of declarations like

```
(new x (new y (new z ...)))
```

we allow a *Proc* to be preceded by a sequence of declaration clauses within a single set of parentheses:

Proc	=	...	
		(*DecProc*)	Local declarations
DecProc	=	*Dec DecProc*	Composite declarations
		Dec Proc	Local declaration

Translating such process back into the core language is easy:

$$\texttt{(d}_1\ \texttt{...}\ \texttt{d}_n\ \ \texttt{e)} \Rightarrow \texttt{(d}_1\ \texttt{...}\ \texttt{(d}_n\ \texttt{e))} \qquad \text{(Tr-DecSeq)}$$

4.2.2 run Declarations

In sequences of declarations, it is often convenient to start some process running in parallel with the evaluation of the remainder of the declaration. For example, one often wants to define some server process via a replicated input and then start a single copy running. We introduce the declaration keyword run for this purpose. (The keyword `fork` might have been more intuitive, but we find our programs are easier to read if all declaration keywords are three characters long!) Once a declaration sequence has been translated into a nested collection of individual declarations, this run declaration may be translated into a simple parallel composition:

$$\texttt{(run e}_1\ \ \texttt{e}_2\texttt{)} \Rightarrow \texttt{(e}_1\ \texttt{|}\ \texttt{e}_2\texttt{)} \qquad \text{(Tr-Run)}$$

For example, the process

```
(run print!"twittering"
 run print!"rising"
 print!"overhead passing")
```

is transformed first by Tr-DecSeq to

```
(run print!"twittering"
 (run print!"rising"
  print!"overhead passing"))
```

then by Tr-Run into

```
  ( print!"twittering"
  | (run print!"rising"
     print!"overhead passing"))
```

and finally, by TR-RUN again, into:

```
  ( print!"twittering"
  | ( print!"rising"
    | print!"overhead passing"))
```

Or, since no particular ordering or translations is specified, we might take a different path after the first step and translate

```
(run print!"twittering"
 (run print!"rising"
  print!"overhead passing"))
```

by TR-RUN into

```
(run print!"twittering"
 ( print!"rising"
 | print!"overhead passing"))
```

and finally (as before) into:

```
  ( print!"twittering"
  | ( print!"rising"
    | print!"overhead passing"))
```

The fact that both sequences of translations reach the same core-language program is no accident: the translation rules are carefully designed so that this will *always* be the case.

4.2.3 Process Definitions

The process definitions that we used in Chapter 1 can be translated into the core language as follows:

$$\begin{array}{c}\texttt{(def } x_1\ p_1 \texttt{ = } e_1 \ \ldots \texttt{ and } x_n\ p_n \texttt{ = } e_n\ \ \texttt{e)} \Rightarrow \\ \texttt{(new } x_1 \ \ldots \texttt{ (new } x_n \texttt{ (}x_1\texttt{?*}p_1 \texttt{ = } e_1 \texttt{ | } \ldots \texttt{ (}x_n\texttt{?*}p_n \texttt{ = } e_n \texttt{ | e))))}\end{array} \qquad \text{(TR-DEF)}$$

For example, the expression

```
(def x[i] = printi!i
 x![3])
```

is translated by TR-DEF into:

```
(new x
 (run x ?* [i] = printi!i
  x![3]))
```

By TR-RUN, this expression is further translated into the core-language expression:

```
(new x
 ( x ?* [i] = printi!i
 | x![3] ))
```

Here, the systematic "pun" between channels and process abstractions is apparent. The declarations

```
def tt[b:Boolean] = b?*[t f] = t![]
and ff[b:Boolean] = b?*[t f] = f![]
```

and

```
new tt:^Boolean
new ff:^Boolean
run tt?*b = b?*[t f] = t![]
run ff?*b = b?*[t f] = f![]
```

are operationally equivalent. However, it is useful to make a small refinement in the way we assign types to definitions. Instead of giving the definitions `tt` and `ff` the type `^Boolean`, we use the type `!Boolean`. This ensures that when a channel is introduced by a `def` clause, the body of the `def` is the only process that can ever read from the definition channel itself. (This refinement will be made precise in Section 6.1.)

Notice that TR-DEF makes all of the process definitions in the same declaration

```
def p[x:X y:Y] = e
and q[z:Z w:W] = f
and ...
```

mutually recursive: `e` may refer to `q` and `f` may refer to `p`. Note that only the first clause is introduced by `def`; all the others begin with `and` to show that they form a single interdependent collection.

To close this section, here is a last kind of declaration clause that is sometimes handy in programming...

4.2.4 Local Declarations

A `local` declaration allows some bindings to be established, used in the definitions of another set of bindings, and then hidden. For example, in the expression

```
local (
 val x = []
) in (
 val y = [x x]
)
val w = [y y y]
z!w
```

the scope of `x` is just the binding of `y`, while the scopes of `y` and `w` include the whole remainder of the expression.

The `local` declaration is sometimes slightly puzzling to those who have not encountered it before. Whereas most of the other derived forms are intended to make certain programming idioms easier to write, `local` plays the opposite role, making certain (erroneous) programs *impossible* to write. For example, suppose that we write a definition `d` that uses another, private, definition `p` to do some of its work:

```
def p[] = print!"the myriad thence-aroused words"
def d[] = (p![] | p![])
run d![]
```

```
the myriad thence-aroused words
the myriad thence-aroused words
```

Although we intend that the only uses of `p` are the two in `d`, there is nothing here to prevent `p` from being used accidentally later in the program. The intended invariant — that the string `"the myriad thence-aroused words"` will always be printed an even number of times — may not be preserved:

```
    run d![]
    ...
    run p![]
```

```
the myriad thence-aroused words
the myriad thence-aroused words
the myriad thence-aroused words
```

By putting the definition of `p` in a `local` scope, we render all uses of `p` after the `end` of the `local` erroneous:

```
    local (
      def p[] = print!"the myriad thence-aroused words"
    ) in (
      def d[] = (p![] | p![])
    )
    run d![]
    run p![]
```

```
example.pi:9.7: Unbound name: p
```

Since its only effect is on the visibility of variable bindings, the `local` declaration form is only important during the scope resolution phase of compilation. Afterwards, when all variable occurrences have been turned into direct pointers to their binders, `local` plays no role whatsoever; it is "translated" by simply throwing it away:

$$\texttt{(local (d}_1\ \texttt{...}\ \texttt{d}_m\texttt{) in (d}_{m+1}\ \texttt{...}\ \texttt{d}_n\texttt{) e)} \Rightarrow \texttt{(d}_1\ \texttt{...}\ \texttt{(d}_m\ \texttt{(d}_{m+1}\ \texttt{...}\ \texttt{(d}_n\ \texttt{e))))} \qquad \text{(TR-LOCAL)}$$

4.3 N-ary Parallel Composition

Strictly speaking, the core language syntax only allows two processes to be composed in parallel. We generalize this to arbitrary numbers (≥ 2) of processes:

$$\frac{n > 2}{\texttt{(e}_1\ \texttt{|}\ \texttt{...}\ \texttt{|}\ \texttt{e}_n\texttt{)} \Rightarrow \texttt{(e}_1\ \texttt{|}\ \texttt{...}\ \texttt{(e}_{n-1}\ \texttt{|}\ \texttt{e}_n\texttt{))}} \qquad \text{(TR-PRL)}$$

4.4 Inaction

Many presentations of the π-calculus include a "null process" that has no observable behavior. In Pict, this process is written `()`. (The notation reminds us that it is the unit for the parallel composition operator.) Its (non-)behavior can be defined by translating it into a process that creates a fresh channel and immediately tries to communicate on it:

`()` $\Rightarrow$ `(new x  x![])` (TR-NULL)

4.4.1 Exercise [Easy]: *Check that the process* `(new x x![])` *is really inert — i.e., that for any process* `e`*, the composite* `e | (new x x![])` *has exactly the possible behaviors that* `e` *has.*

4.5 Programs

Now that we have introduced `run` as a form of declaration clause, we can relax the restriction that a program should consist of the keyword `run` followed by an arbitrary process. From now on, a program in Pict will be an arbitrary declaration sequence. Its meaning is obtained by surrounding it by `(... ())` and using the translation rules obtain an expression in the core language.

For example, the complete program

```
run print!"Yet I murmur"
```

is understood by the Pict compiler as the process expression

```
(run print!"Yet I murmur"
 ())
```

which, by TR-RUN and TR-NULL, is translated to the core-language process:

```
(print!"Yet I murmur" | (new x x![]))
```

4.6 Importing Other Files

This section is somewhat out of date with the current language. If all you want to do is to import library files, the instructions here will work fine. If you want to write your own imported files, then you'll need to look at the Makefiles in the Examples directory of the Pict installation to see how to generate a `.px` *file for each* `.pi` *file. (The* `.px` *files are a step toward true separate compilation, which will arrive in a future release.)*

Although Pict does not (yet) support true separate compilation or modules, it does provide a rudimentary facility for breaking large programs up into smaller pieces and storing the pieces in separate files for easy reuse.

A Pict program is organized as several files, each containing a declaration sequence preceded by a number of `import` clauses. Each `import` clause has the form

```
import "name"
```

where `name` is an absolute or relative pathname (the suffix `.pi` is added automatically). If a relative pathname is used, both the current directory and a central directory of Pict library files are searched. Imports may be nested (that is, imported files may themselves begin with `import` clauses).

Semantically, the first occurrence of an import clause for a given filename means exactly the same as if the whole imported file had been included at the point where the `import` appears. Subsequent occurrences of the same import clause have no effect.

4.7 Exercises

4.7.1 Exercise [Recommended]: *Write a process accepting requests on a channel "submit" of type* `^Submission`, *where*

```
type Submission = [Worker Int ![]]
```

and

```
type Worker = ![![]].
```

Each submission consists of a trigger channel for a "worker process," an integer priority (which we will ignore until we get to 4.7.2), and a completion channel for the submission request. A worker process is triggered by sending it a completion channel, on which it signals when it is finished.

The constraints that must be satisfied by your solution are:

1. *Only one worker process should be scheduled at a time. If a new* `submit` *request arrives while a previously submitted worker is still working, the new worker must be delayed.*
2. *Each submission request should be acknowledged by sending an empty tuple along the provided completion channel. If the submitted worker cannot be scheduled for execution, the submission should be acknowleged without waiting for it to complete. (The completion channel should be thought of as an acknowledgement from the job scheduler to the client that the worker has been accepted and will eventually be scheduled.)*

Your solution should have the form

```
import "tester.pi"
def submit [w:Worker p:Int r:![]] =  ...
run test![submit]
```

where `tester.pi` *contains the following code:*

```
{- A simple process that wastes some time and then signals completion -}
def delay[n:Int c:![]] = if (== n 0) then c![] else delay![(-- n) c]

{- Two simple worker processes -}
def worker1[c:![]] =
  (new done:^[]
   ( pr!["Worker 1 starting\n" done]
   | done?[] =
      ( delay![20 done]
      | done?[] =
         ( pr!["Worker 1 working\n" done]
         | done?[] =
            ( delay![20 done]
            | done?[] =
               (pr!["Worker 1 working\n" done]
               | done?[] =
                  ( delay![20 done]
                  | done?[] =
                     ( pr!["Worker 1 finished\n" done]
                     | done?[] =
                        c![]
                     ))))))))

def worker2[c:![]] =
  (new done:^[]
   ( pr!["Worker 2 starting\n" done]
   | done?[] =
      ( delay![20 done]
      | done?[] =
         ( pr!["Worker 2 working\n" done]
         | done?[] =
            ( delay![20 done]
            | done?[] =
               (pr!["Worker 2 working\n" done]
               | done?[] =
                  ( delay![20 done]
                  | done?[] =
                     ( pr!["Worker 2 finished\n" done]
                     | done?[] =
                        c![]
                     ))))))))

{- Try out a proposed definition of "submit" by starting a worker process,
   submitting two other workers with increasing priorities, and then finishing
   the first worker and seeing what happens. -}
def test[submit:!Submission] =
  (new done:^[]
   def testWorker[c:![]] =
     ( pr!["Test worker starting\n" done]
     | done?[] =
        ( submit![worker1 1 done]
        | done?[] =
            ( submit![worker2 2 done]
            | done?[] =
               pr!["Test worker finishing\n" c]
            )))
   new testdone:^[]
   submit![testWorker 3 testdone]
  )
```

For comparison, here is a trivial implementation of `submit` *that ignores the constraint that only one worker at a time should be allowed to execute.*

```
import "tester"

def submit[wor:Worker pri:Int c:![]] =
   ( c![]
   | (new done:^[]
      wor![done]))

run test![submit]
```

```
Test worker starting
Worker 1 starting
```

```
Test worker finishing
Worker 2 starting
Worker 1 working
Worker 2 working
Worker 1 working
Worker 2 working
Worker 1 finished
Worker 2 finished
```

Your solution, which takes this constraint into account, should produce output like this:

```
Test worker starting
Test worker finishing
Worker 1 starting
Worker 1 working
Worker 1 working
Worker 1 finished
Worker 2 starting
Worker 2 working
Worker 2 working
Worker 2 finished
```

[Solution on page 94.]

4.7.2 Exercise [Recommended]: *Now make your job scheduler execute workers in priority order. If several submissions are outstanding, the one with the highest priority must always be honored first. For example, submitting* `testWorker` *as before should now result in:*

```
Test worker starting
Test worker finishing
Worker 2 starting
Worker 2 working
Worker 2 working
Worker 2 finished
Worker 1 starting
Worker 1 working
Worker 1 working
Worker 1 finished
```

[Solution on page 94.]

Chapter 5

Core Type System

The type system of Pict has its roots in the theoretical literature on type systems for the π-calculus [Mil91, Tur96, PS93, Gay93] and for functional languages, among which the most immediate predecessors to Pict are ML [GMW79, MTH90, WAL^{+}89], QUEST [Car91], and AMBER [Car86]. The type inference technique introduced in Chapter 9 is similar to the ones used in LEGO [Pol90], λ-PROLOG [NM88], and Cardelli's implementation of $F_{\leq}$ [Car93]). The treatment of subtyping and higher-order polymorphism is based on recent papers on static type systems for object-oriented languages [Car84, Bru94, CCH^{+}89, CHC90, PT94, FM94, etc.] and the λ-calculus $F^{\omega}_{\leq}$ [Car90, Mit90, PS96, Com94]. The rules for input/output modalities for channel types come from Pierce and Sangiorgi's type system for the pure π-calculus [PS93]; polymorphic channels have been investigated in detail in Turner's Ph.D. thesis [Tur96]. The rules for records are new, but may be regarded as a simplified fragment of the systems described by Cardelli and Mitchell [CM91]. An early version of the Pict type system was described in [PRT93].

5.1 Notation

If a value contains free variables, its type depends on the types of these variables; for example, the value `[x y]` has the type `[^[] ^[]]` if `x` and `y` both have type `^[]`. This dependance must be made explicit in the rules defining the typing relation: we say that a value `v` has some type `T` under a specific set of assumptions ?, written $? \vdash \texttt{v} \in \texttt{T}$.

Formally, a *typing context* ? is a list of typing assumptions for a collection of (distinct) variables. For example,

$$? = \texttt{x} : \texttt{^[]},\ \texttt{y} : \texttt{^[]}$$

is a context assigning types to the variables `x` and `y`. The *domain* of ? is the set $\{\texttt{x},\texttt{y}\}$. The metavariables ? and Δ range over contexts. The empty context is written $\bullet$. If ? and Δ are contexts with disjoint domains, then $? + \Delta$ is a context containing the combined assumptions of ? and Δ.

The type system of Pict is defined by a collection of rules of the form

$$\frac{\text{premise} \qquad \text{premise} \qquad \cdots \qquad \text{premise}}{\text{conclusion}}$$

where the conclusion and premises are *statements* of the following forms:

$? \vdash \mathtt{v} \in \mathtt{T}$	value `v` has type `T` under assumptions ?
$? \vdash \mathtt{S} < \mathtt{T}$	`S` is a subtype of `T` under ?
$? \vdash \mathtt{d} \Rightarrow \Delta$	declaration `d` is well formed under ? and yields bindings Δ
$? \vdash \mathtt{p} \in \mathtt{T} \Rightarrow \Delta$	pattern `p` requires type `T` under ? and yields bindings Δ
$? \vdash \mathtt{a} \in \mathtt{T}$	abstraction `a` is well formed and accepts type `T`
$? \vdash \mathtt{e}\ \mathit{ok}$	process expression `e` is well-formed under ?

The first two kinds of statement are familiar from type systems for functional languages. The third is used for checking Pict declaration sequences. Since a declaration sequence cannot be sent over a channel, it does not itself have a type; however, it may give rise to a collection of variable bindings for some following scope, and we need to keep track of the types of these variables. The "type" of a declaration is therefore a typing context. Similarly, a pattern binds some variables and thus gives rise to a context; however, a pattern *also* has a type, since it can only match values of a certain shape. An abstraction likewise requires an argument of a certain shape. Finally, a process expression yields neither bindings nor a value; it is simply either well formed or not. (A process is well formed in a given context if all its input- and output-subexpressions respect the typings of the channels over which communication occurs.)

5.2 Type Abbreviations

The typing of declarations of the form `type X = T` can be dealt with once and for all: in the presence of such a declaration, there is *no difference whatsoever* between the type expressions `X` and `T`, i.e. an expression `{ type X = T in e }` is exactly equivalent to $\{\mathtt{X} \mapsto \mathtt{T}\}\mathtt{e}$. Internally, the typechecker does actually keep track of abbreviations instead of immediately substituting them away, but this is done only to improve the efficiency of typechecking and allow more comprehensible error reporting.

5.3 Values

We now proceed to detailed typing rules for the various syntactic categories of Pict. In this chapter, we give typing rules just for the core language defined in Chapter 2. Rules for derived syntactic forms are discussed in Chapter 6.

The first two rules for values capture the informal description of value typing that we saw in Section 1.5. If the current context contains the binding `x:T` for the variable `x`, then the type of `x` is `T` in this context:

$$?_1, \mathtt{x{:}T}, ?_2 \vdash \mathtt{x} \in \mathtt{T} \qquad \text{(V-VAR)}$$

If the values $\mathtt{v}_1$ through $\mathtt{v}_n$ have the types $\mathtt{T}_1$ through $\mathtt{T}_n$, then the tuple value $[\mathtt{v}_1\ \ldots\ \mathtt{v}_n]$ has the tuple type $[\mathtt{T}_1\ \ldots\ \mathtt{T}_n]$.

$$\frac{? \vdash \mathtt{v}_1 \in \mathtt{T}_1 \qquad \ldots \qquad ? \vdash \mathtt{v}_n \in \mathtt{T}_n}{? \vdash [\mathtt{v}_1 \ldots \mathtt{v}_n] \in [\mathtt{T}_1 \ldots \mathtt{T}_n]} \qquad \text{(V-TUPLE)}$$

Records require a rule for each of the two basic record constructors. The empty record has an empty record type, while a record `v` of type `T` extended with a new field `l` of type `U` has the extended type `(T with l:U)`.

$$\frac{? \vdash \mathtt{v}_1 \in \mathtt{T}_1 \qquad \dots \qquad ? \vdash \mathtt{v}_n \in \mathtt{T}_n}{? \vdash (\mathtt{record}\ \mathtt{l}_1\mathtt{=v}_1\mathtt{...l}_n\mathtt{=v}_n) \in (\mathtt{record}\ \mathtt{l}_1\mathtt{:T}_1\mathtt{...l}_n\mathtt{:T}_n)} \tag{V-Rcd}$$

We allow types of values to be *promoted* according to the subtype relation: if `v` is a value of type `S` and `S` is a subtype of `T`, then `v` also has type `T`.

$$\frac{? \vdash \mathtt{v} \in \mathtt{S} \qquad \mathtt{S} < \mathtt{T}}{? \vdash \mathtt{v} \in \mathtt{T}} \tag{V-Sub}$$

This rule embodies a principle of "safe substitutability" that captures the intended meaning of the subtype relation: the statement $\mathtt{S} < \mathtt{T}$ means that an element of `S` can always be used in a context where an element of `T` is required. The job of Section 5.4 will be to axiomatize this relation.

Finally, it is sometimes convenient to explicitly annotate a value with its expected type. (In Chapter 9 we will see how the behavior of the type inference algorithm can be controlled using such annotations.) The expression `v:T` behaves the same as `v` and has type `T`, as long as `v` itself has type `T`.

$$\frac{? \vdash \mathtt{v} \in \mathtt{T}}{? \vdash (\mathtt{v:T}) \in \mathtt{T}} \tag{V-Coerce}$$

(Of course, `v` itself may have other types besides `T`; in particular, it may have some type `S` that is a proper subtype of `T`. But `v:T` will *not* have the type `S` in this case.)

5.4 Subtyping

Having shown how to assign types to values, we must now define the "safe substitutability" relation. Statements of the form $? \vdash \mathtt{S} < \mathtt{T}$ may be read as "under assumptions ?, every element of `S` may safely be regarded as an element of `T`."[1]

The subtype relation consists of two structural rules

$$? \vdash \mathtt{T} < \mathtt{T} \tag{S-Refl*}$$

$$\frac{? \vdash \mathtt{S} < \mathtt{U} \qquad ? \vdash \mathtt{U} < \mathtt{T}}{? \vdash \mathtt{S} < \mathtt{T}} \tag{S-Trans*}$$

stating that it is reflexive and transitive, plus one or more rules for each type constructor. (By convention, a * at the end of a rule's name signals that this is not the final version of the rule: we will encounter other features of the type system later that will lead us to refine it in some way. The Pict Language Definition [PT96b] gives the final versions of all the typing rules.)

`Top` is a maximal type:

$$? \vdash \mathtt{S} < \mathtt{Top} \tag{S-Top*}$$

The rule for comparing tuple types requires that the two tuples have the same arity but allows the types of the elements to vary:

[1]Actually, the assumptions Γ play no role in subtyping at this stage. But it is convenient to carry them along since, in Chapter 13, we will enrich contexts with *subtyping assumptions* in addition to the assumptions about types of free identifiers that they carry here.

$$\frac{? \vdash \mathtt{S}_1 < \mathtt{T}_1 \qquad \ldots \qquad ? \vdash \mathtt{S}_n < \mathtt{T}_n}{? \vdash \mathtt{[S}_1 \ldots \mathtt{S}_n\mathtt{]} < \mathtt{[T}_1 \ldots \mathtt{T}_n\mathtt{]}} \qquad \text{(S-TUPLE)}$$

For example, if `Char` < `Int`, then `[Char Char]` < `[Char Int]`.

For records, we allow the types of the elements to vary, as with tuples, but also allow new elements to be added on the right-hand side. (In other words, the "smaller" record can have more fields, but the common fields must have the same names and appear in the same order):

$$\frac{\text{for each } i \leq m,\ ? \vdash \mathtt{S}_i < \mathtt{T}_i}{? \vdash \mathtt{(record\ l}_1\mathtt{:S}_1 \ldots \mathtt{l}_m\mathtt{:S}_m \ldots \mathtt{l}_n\mathtt{:S}_n\mathtt{)} < \mathtt{(record\ l}_1\mathtt{:T}_1 \ldots \mathtt{l}_m\mathtt{:T}_m\mathtt{)}} \qquad \text{(S-RCD)}$$

Note that this subtyping rule is quite a bit more restrictive than the usual one for record types, which also allows fields to be reordered. The more restrictive rule is adopted in Pict for the sake of implementation efficiency.

Finally, we have a collection of rules describing the subtyping behavior of the channel type constructors `^`, `!`, and `?`. The constructor `^` is invariant in the subtype relation:

$$\frac{? \vdash \mathtt{S} < \mathtt{T} \qquad ? \vdash \mathtt{T} < \mathtt{S}}{? \vdash \mathtt{\hat{}S} < \mathtt{\hat{}T}} \qquad \text{(S-CHAN)}$$

That is, `^S` is a subtype of `^T` only when `S` and `T` are equivalent.

The constructors `!` and `?` have more interesting subtyping behavior: `?` is covariant and `!` is contravariant. Operationally, this captures the observation that, for example, if a given channel `x` is being used in a given context only to read elements of type `T`, then it is safe to replace `x` by another channel `y` carrying elements of type `S`, as long as any element that is read from `y` may safely be regarded as an element of `T` — that is, as long as `S` is a subtype of `T`.

$$\frac{? \vdash \mathtt{T} < \mathtt{S}}{? \vdash \mathtt{!S} < \mathtt{!T}} \qquad \text{(S-OCHAN)}$$

$$\frac{? \vdash \mathtt{S} < \mathtt{T}}{? \vdash \mathtt{?S} < \mathtt{?T}} \qquad \text{(S-ICHAN)}$$

Finally, `^T` is a subtype of both `?T` and `!T`. That is, we are allowed to forget either the capability to read or the capability to write on a channel: a channel that can be used for both input and output may be used in a context where just one capability is needed.

$$? \vdash \mathtt{\hat{}T} < \mathtt{?T} \qquad \text{(S-CHANICHAN)}$$

$$? \vdash \mathtt{\hat{}T} < \mathtt{!T} \qquad \text{(S-CHANOCHAN)}$$

5.5 Declarations

A `new` declaration returns a binding for the newly defined channel using the declared type, which must be a channel type.

$$\frac{? \vdash \mathtt{T} < \mathtt{{}^\wedge U}}{? \vdash \mathtt{new\ x{:}T} \Rightarrow \mathtt{x{:}T}} \qquad \text{(D-New)}$$

5.6 Type Inference

Before showing the typing rules for patterns, a few words about type inference are in order.

So far, we have seen two constructs that introduce new variable bindings: `new` declarations and input expressions. In `new` declarations, we have used explicit annotations to tell the typechecker the types of the new variables; variables bound by input expressions have not been annotated.

In reality, *all* binding constructs in Pict carry type annotations. If we wish, we can write the "boolean server processes" from Chapter 1 in fully annotated form:

```
new tt: ^[Boolean]
run tt ?* [b:Boolean] = b?[t:^[] f:^[]] = (t![] | tt![b])
new ff: ^[Boolean]
run ff ?* [b:Boolean] = b?[t:^[] f:^[]] = (f![] | ff![b])
```

On the other hand, Pict also allows all type annotations to be omitted. The booleans can be written with no types at all:

```
new tt
run tt?*[b] = b?[t f] = (t![] | tt![b])
new ff
run ff?*[b] = b?[t f] = (f![] | ff![b])
```

Whenever a type annotation is omitted, the Pict parser supplies a fresh *unification variable* in place of the missing type. As typechecking proceeds, these unification variables are *instantiated* to actual types.

Unfortunately, in general it is not possible to automatically instantiate unification variables in an optimal way. There are Pict programs that pass the typechecker if some of their binders are explicitly annotated with appropriate types, but that fail to typecheck when the annotations are omitted because the typechecker guesses wrong when it instantiates the unification variables. Chapter 9 will have more to say about understanding and controlling the type inference algorithm; for now a simple rule will suffice:

> All variables bound by `new` declarations must be explicitly annotated with their expected types. Variables bound by input expressions need not be annotated.

5.7 Patterns

With this background in mind, the rules for typechecking patterns are very simple. Each pattern has a type describing the shape of the values that it can match, and also gives rise to a set of variable bindings.

A variable `x` appearing in a pattern with the type annotation `T` matches any value of type `T` and gives rise to a binding for the variable `x`.

$$? \vdash \texttt{x:T} \in \texttt{T} \Rightarrow \texttt{x:T} \tag{P-Var}$$

A tuple pattern $\texttt{[p}_1 \ \ldots \ \texttt{p}_n\texttt{]}$ has the type $\texttt{[T}_1 \ \ldots \ \texttt{T}_n\texttt{]}$, where the $\texttt{T}_i$'s are the types of its elements, and gives rise to a set of bindings including all the bindings from its subpatterns.

$$\frac{? \vdash \texttt{p}_1 \in \texttt{T}_1 \Rightarrow \Delta_1 \qquad \ldots \qquad ? \vdash \texttt{p}_n \in \texttt{T}_n \Rightarrow \Delta_n}{? \vdash \texttt{[p}_1 \ldots \texttt{p}_n\texttt{]} \in \texttt{[T}_1 \ldots \texttt{T}_n\texttt{]} \Rightarrow \Delta_1, \ \ldots, \ \Delta_n} \tag{P-Tuple}$$

Similarly, a record pattern of the form `(record l=p)` has a type of the form $\texttt{(record l}_1\texttt{:T}_1\texttt{)}$, where `T` is the type of `p`, and gives rise to the same set of bindings as `p` does.

$$\frac{? \vdash \texttt{p} \in \texttt{T} \Rightarrow \Delta}{? \vdash \texttt{(record l=p)} \in \texttt{(record l:T)} \Rightarrow \Delta} \tag{P-Rcd}$$

5.8 Abstractions

We expect an input expression `v?p = e` to be well formed if:

1. `v` is an input channel, i.e., it has type `?S` for some `S`;
2. the pattern `p` has the same type `S`; and
3. the body `e` is well formed under the bindings arising from `p`.

Since we have separated input expressions syntactically into an input prefix `v?` and an abstraction `p=e`, we handle these three conditions in two different typing rules. The typing rule for abstractions handles conditions (2) and (3) together.

$$\frac{? \vdash \texttt{p} \in \texttt{T} \Rightarrow \Delta \qquad ?, \Delta \vdash \texttt{e} \ \mathit{ok}}{? \vdash \texttt{p = e} \in \texttt{T}} \tag{A-Abs}$$

5.9 Processes

Condition (1) is handled by the typing rule for input expressions:

$$\frac{? \vdash \texttt{v} \in \texttt{?S} \qquad ? \vdash \texttt{a} \in \texttt{S}}{? \vdash \texttt{v?a} \ \mathit{ok}} \tag{E-In}$$

A similar rule applies to replicated inputs.

$$\frac{? \vdash \texttt{v} \in \texttt{?S} \qquad ? \vdash \texttt{a} \in \texttt{S}}{? \vdash \texttt{v?*a} \ \mathit{ok}} \tag{E-RIn}$$

Symmetrically, an output expression `v!v´` is well formed if `v` has a channel type `!S`, for some `S`, and `v´` has type `S`.

$$\frac{? \vdash \mathtt{v}_1 \in \mathtt{!S} \qquad ? \vdash \mathtt{v}_2 \in \mathtt{S}}{? \vdash \mathtt{v}_1\mathtt{!}\mathtt{v}_2 \ \mathit{ok}} \tag{E-Out}$$

The parallel composition of two processes is well formed in a given context if both parts are.

$$\frac{? \vdash \mathtt{e}_1 \ \mathit{ok} \qquad ? \vdash \mathtt{e}_2 \ \mathit{ok}}{? \vdash \mathtt{(e}_1 \ \mathtt{|} \ \mathtt{e}_2\mathtt{)} \ \mathit{ok}} \tag{E-Prl}$$

Finally, a process prefixed by a declaration is checked in a context extended by the bindings provided by the declaration.

$$\frac{? \vdash \mathtt{d} \Rightarrow \Delta \qquad ?, \Delta \vdash \mathtt{e} \ \mathit{ok}}{? \vdash \mathtt{(d\ e)} \ \mathit{ok}} \tag{E-Dec}$$

Chapter 6

Toward a Programming Language

Chapters 1 to 5 introduced the syntax, operational semantics, and first-order type system of the Pict core language, as well as some of the simpler derived forms. We now proceed with (and, by the end of the chapter, conclude) the task of defining a high-level programming notation based on these foundations.

6.1 Derived Typing Rules

In Chapter 4, we introduced several new syntactic forms for processes and declarations. In this chapter, we will see many more. Adding all this extra syntax raises a question: Under what circumstances is a program involving derived forms well typed? For example, what assumptions do we need about `e` and `f` in order for the process `(run e f)` to be well typed?

The behavior of such a program is found by translating the derived `run` syntax into the core language via the rule Tr-Run:

$$\texttt{(run } \texttt{e}_1 \;\; \texttt{e}_2\texttt{)} \Rrightarrow \texttt{(}\texttt{e}_1 \texttt{ | } \texttt{e}_2\texttt{)} \qquad \text{(Tr-Run)}$$

The process resulting from the translation is `(e | f)`, which is well typed in a given context ? just when `e` and `f` are. We could check the typing of every program in this way, by translating to core syntax and using the typing rules already given, but it is convenient to give extra *derived typing rules* for the high-level derived forms, allowing us to calculate the same typings directly. For `run`, we add a new declaration rule:

$$\frac{? \vdash e \; ok}{? \vdash \texttt{run e} \Rightarrow \bullet} \qquad \text{(D-Run)}$$

Together with the rule E-Dec for processes prefixed by declarations, this rule implies that `(run e f)` will be well typed in ? just when `e` and `f` are.

Typing rules for the other derived declaration forms from Section 4.2 are equally easy to obtain. From the rule Tr-Def for mutually recursive process definitions, for example, we get the following rule.

$$\frac{\text{for each } i \quad ?, \texttt{x}_1\texttt{:\^{}T}_1, \ldots, \texttt{x}_n\texttt{:\^{}T}_n \vdash \texttt{a}_i \in \texttt{T}_i}{? \vdash \texttt{def x}_1 \;\texttt{a}_1 \texttt{ and } \ldots \texttt{ and x}_n \;\texttt{a}_n \Rightarrow \texttt{x}_1\texttt{:\^{}T}_1, \ldots, \texttt{x}_n\texttt{:\^{}T}_n} \qquad \text{(D-Def*)}$$

This rule is perhaps easiest to grasp operationally: to typecheck a set of mutually recursive definitions introduced with `def...and...`, we first typecheck the patterns $\texttt{p}_1$ through $\texttt{p}_n$ to obtain a list $\texttt{T}_1$

through $\mathtt{T}_n$ of types for the definitions. Then, we verify that the body of each definition is well formed in a context where all of the definitions are visible. Finally, the `def...and...` declaration itself yields bindings for all of the definitions with the given types.

It is useful to slightly tighten the derived typing rule for process abstractions so as to forbid any process besides the body of a process definition from reading on its request channel, since a real program would not ordinarily do this except by accident.

$$\frac{\text{for each } i \quad ?,\, \mathtt{x}_1{:}!\mathtt{T}_1, \ldots, \mathtt{x}_n{:}!\mathtt{T}_n \vdash \mathtt{a}_i \in \mathtt{T}_i}{? \vdash \mathtt{def}\ \mathtt{x}_1\ \mathtt{a}_1\ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathtt{x}_n\ \mathtt{a}_n \Rightarrow \mathtt{x}_1{:}!\mathtt{T}_1, \ldots, \mathtt{x}_n{:}!\mathtt{T}_n} \qquad \text{(D-DEF)}$$

We take this rule as the official one. The Pict typechecker actually runs before the translation rules, so it can "see" the higher-level syntactic forms before they are translated away. This allows us to choose typing rules for constructs like `def` that are stronger — exclude more programs — than would be the case if we translated first and typechecked second.

In the typing of declaration sequences, the body of the second declaration can "see" the names bound by the first. This rule shows clearly the intended scoping of variables: the scope of variables bound by $\mathtt{d}_1$ includes $\mathtt{d}_2$ (second hypothesis), and the whole declaration creates a scope containing the bindings from both $\mathtt{d}_1$ and $\mathtt{d}_2$ (conclusion).

$$\frac{? \vdash \mathtt{d}_1 \Rightarrow \Delta_1 \qquad ?,\, \Delta_1 \vdash \mathtt{d}_2 \Rightarrow \Delta_2}{? \vdash \mathtt{d}_1\ \mathtt{d}_2 \Rightarrow \Delta_1,\, \Delta_2} \qquad \text{(D-DECSEQ)}$$

The fact that TR-LOCAL translates a `local` declaration into a simple declaration sequence indicates that all the "action" is in the scoping behavior of `local`, which can be read off from the typing rule:

$$\frac{? \vdash \mathtt{d}_1 \Rightarrow \Delta_1 \qquad ?,\, \Delta_1 \vdash \mathtt{d}_2 \Rightarrow \Delta_2}{? \vdash \mathtt{local}\ (\mathtt{d}_1)\ \mathtt{in}\ (\mathtt{d}_2) \Rightarrow \Delta_2} \qquad \text{(D-LOCAL)}$$

A process that does not behave cannot misbehave, so the derived typing rule for `()` is trivial:

$$? \vdash \mathtt{()}\ \mathit{ok} \qquad \text{(E-NULL)}$$

6.2 Complex Values

So far, all the value expressions we have encountered have been built up in an extremely simple way, using just variables, channels (including built-in channels such as `true`, `(+)`, `5`, and `"hello"`), tuples of values, and records of values. These *simple values* are important because they are exactly the entities that can be passed along channels and participate in pattern matching.

6.2.1 "Continuation-Passing" Translation

In programming, it is very common to write an expression that computes a simple value and immediately sends it along some channel. For example, the process `(new n x!n)` creates a fresh channel `n` and sends it off along `x`. More interestingly,

```
run (def f[x res] = +![x x res]
     y!f)
```

creates a local definition `f` and sends its "request channel" along `y`.

An alternative syntax for such expressions, which can often make them easier to understand, puts the whole value-expression *inside* the output: `x!(new n n)`. In general, it is useful to allow such expressions in any position where a simple value is expected. Formally, we extend the syntactic category of values with declaration values of the form `(d v)`. We use the term *complex value* for an expression in the extended syntax that does not fall within the core language.

When we write `x!(new n n)`, we do not mean to send the *expression* `(new n n)` along `x`. A complex value is always evaluated "strictly" to yield a simple value, which is substituted for the complex expression.

In introducing complex values, we have taken a fairly serious step: we must now define the meaning of a complex value occurring in any position where simple values were formerly allowed. For example, the nested expression `x![23 (new x x) (new y y)]` must be interpreted as a core language expression that creates two new channels, packages them into a simple tuple along with the integer `23` and sends the result along `x`.

To interpret arbitrary complex values, we introduce a general "continuation-passing" translation. Given a complex value `v` and a continuation channel `c`, the expression $[\![\mathtt{v} \rightarrow \mathtt{c}]\!]$ will denote a process that evaluates `v` and sends the resulting simple value along `c`. We then introduce translation rules for process expressions containing complex values. For example, the rule

$$\frac{\mathtt{v}_1 \text{ or } \mathtt{v}_2 \text{ complex}}{\mathtt{v}_1\mathtt{!}\mathtt{v}_2 \;\Rightarrow\; \mathtt{(new\ c\ (}[\![\mathtt{v}_1 \rightarrow \mathtt{c}]\!]\ \mathtt{|}\ \mathtt{c?x\ =\ }[\![\mathtt{v}_2 \rightarrow \mathtt{x}]\!]\mathtt{))}} \qquad \text{(TR-OUT)}$$

translates an output $\mathtt{v}_1\mathtt{!v}_2$ into a process expression that first allocates a fresh continution channel `c`, evaluates $\mathtt{v}_1$, waits for its result to be sent along `c`, and then evaluates $\mathtt{v}_2$, sending the result directly along the channel `x` that resulted from the evaluation of $\mathtt{v}_1$. Input processes containing complex values are translated similarly:

$$\frac{\mathtt{v} \text{ complex}}{\mathtt{v?a} \;\Rightarrow\; \mathtt{(new\ c\ (}[\![\mathtt{v} \rightarrow \mathtt{c}]\!]\ \mathtt{|\ c?x\ =\ x?a))}} \qquad \text{(TR-IN)}$$

$$\frac{\mathtt{v} \text{ complex}}{\mathtt{v?*a} \;\Rightarrow\; \mathtt{(new\ c\ (}[\![\mathtt{v} \rightarrow \mathtt{c}]\!]\ \mathtt{|\ c?x\ =\ x?*a))}} \qquad \text{(TR-RIN)}$$

The continuation passing translation itself is defined by induction on the syntax of value expressions:

$$\begin{array}{lcl}
[\![\mathtt{x} \rightarrow \mathtt{c}]\!] & = & \mathtt{c!x} \\
[\![\mathtt{k} \rightarrow \mathtt{c}]\!] & = & \mathtt{c!k} \qquad \text{if } \mathtt{k} \text{ is a constant} \\
[\![\mathtt{(d\ v)} \rightarrow \mathtt{c}]\!] & = & \mathtt{(d\ }[\![\mathtt{v} \rightarrow \mathtt{c}]\!]\mathtt{)} \\
[\![\mathtt{[v_1 \ldots v_n]} \rightarrow \mathtt{c}]\!] & = & \mathtt{(new\ c_1\ \ldots\ (new\ c_n} \\
 & & \quad \mathtt{(}[\![\mathtt{v_1} \rightarrow \mathtt{c_1}]\!]\ \mathtt{|\ c_1?x_1\ =\ \ldots} \\
 & & \qquad \mathtt{(}[\![\mathtt{v_n} \rightarrow \mathtt{c_n}]\!]\ \mathtt{|\ c_n?x_n\ =} \\
 & & \qquad\quad \mathtt{c![x_1 \ldots x_n]))))}
\end{array}$$

Conditional values are handled similarly:

$$\begin{array}{lcl}
[\![\mathtt{if\ v_1\ then\ v_2\ else\ v_3} \rightarrow \mathtt{c}]\!] & = & \mathtt{(new\ c_1\ (}[\![\mathtt{v_1} \rightarrow \mathtt{c_1}]\!]\ \mathtt{|\ c_1?x_1\ =} \\
 & & \quad \mathtt{if\ x_1\ then\ }[\![\mathtt{v_2} \rightarrow \mathtt{c}]\!]\mathtt{\ else\ }[\![\mathtt{v_3} \rightarrow \mathtt{c}]\!]\mathtt{\ ))}
\end{array}$$

To typecheck processes containing value declarations, we use one new derived typing rule:

$$\frac{? \vdash \mathtt{d} \Rightarrow \Delta \qquad ?, \Delta \vdash \mathtt{v} \in \mathtt{T}}{? \vdash \mathtt{(d\ v)} \in \mathtt{T}} \tag{V-Dec}$$

This rule captures the intuition that, if we use a value of the form `(d v)` in a context that requires a value of type `T`, it will eventually be translated into a core-language process in which the body `v` is sent along some channel carrying elements of `T`. So the type of the whole `let`-value is actually just the type of its body.

6.2.1.1 Exercise [Recommended]: *What is the fully simplified form of the process* `[]![]`*? What is its type?*

6.2.2 `val` Declarations

Since complex value expressions may become long and may involve expensive computations, it is convenient to introduce a new declaration form that evaluates a complex value. For example, `(val x = (new n [n n]) e)` binds `x` to the result of evaluating `(new n [n n])` and then executes `e`. Formally, `val` declarations are translated into the core language using the continuation-passing translation as above:

$$\mathtt{(val\ p\ =\ v\ \ e)} \Rightarrow \mathtt{(new\ c\ (}\llbracket \mathtt{v} \rightarrow \mathtt{c} \rrbracket \mathtt{\ |\ c?p\ =\ e))} \tag{Tr-Val}$$

$$\frac{? \vdash \mathtt{v} \in \mathtt{T} \qquad ? \vdash \mathtt{p} \in \mathtt{T} \Rightarrow \Delta}{? \vdash \mathtt{val\ p\ =\ v} \Rightarrow \Delta} \tag{D-Val}$$

Since the expression on the left of the `=` can be an arbitrary pattern, a `val` declaration can be used to bind several variables at once. For example, `val [x y] = [23 [a]]` binds `x` to `23` and `y` to `[a]`. Note that, when a `val` declaration `(val p = v e)` is translated into the core language, the body `e` appears after an input prefix. This fact implies that `val` declarations are *strict* or *blocking*: the body cannot proceed until the bindings introduced by the `val` have actually been established.

6.2.3 Application

Of course, allowing declarations inside values represents only a minor convenience; the usefulness of this extension would not by itself justify all of the foregoing machinery — the distinction between complex and simple values, etc. But having established the basic pattern of simplifying complex value expressions by transformation rules, we can apply it to a much more useful extension.

In value expressions, we allow the *application* syntax $\mathtt{(v\ v_1\ \ldots\ v_n)}$. For example, if we define a `double` function by

```
def double [s:String r:!String] = concat![s s r]
```

(where `concat` is string concatenation), then, in the scope of the declaration, we can write `(double s)` as a value, dropping the explicit result channel `r`. For example,

```
run print!(double "soothe")
```

causes "soothesoothe" to be sent along the built-in channel print.

We define the meaning of application values by adding a clause to the definition of the continuation-passing translation:

$$\begin{array}{lcl} [\![\texttt{(v |}\texttt{T}_1\texttt{...T}_n\texttt{| v}_1\texttt{...v}_n\texttt{)} \to \texttt{c}]\!] & = & \texttt{(new d (new c}_1 \texttt{ ... (new c}_n \\ & & \quad \texttt{(}[\![\texttt{v} \to \texttt{d}]\!] \texttt{ | d?f =} \\ & & \quad\quad \texttt{(}[\![\texttt{v}_1 \to \texttt{c}_1]\!] \texttt{ | c}_1\texttt{?x}_1 \texttt{ = ...} \\ & & \quad\quad\quad \texttt{(}[\![\texttt{v}_n \to \texttt{c}_n]\!] \texttt{ | c}_n\texttt{?x}_n \texttt{ =} \\ & & \quad\quad\quad\quad \texttt{f!\{|T}_1\texttt{|\}...\{|T}_n\texttt{|\}[x}_1\texttt{...x}_n \texttt{ c]))))))} \end{array}$$

Operationally, this rule encodes the intuition that the implicit final parameter in an application is the continuation of the function being invoked — the place where the function's result should be sent in order for the rest of the computation to proceed.

6.2.3.1 Exercise [Recommended]: *What core-language program results from applying the translation rules to the following process expression?*

```
x![(* (+ 2 3) (+ 4 5))]
```

6.2.3.2 Exercise [Recommended]: *Rewrite your solution to exercise 4.1.2.1 using application syntax.*

6.2.4 Records

Record projections like (v.l) are allowed as convenient abbreviations for pattern matching:

$$\texttt{(v:T.l)} \Rightarrow \texttt{(val (record:T l=x) = v x)} \qquad \text{(Tr-Proj)}$$

$$\frac{\Gamma \vdash \texttt{v} \in \texttt{T} \qquad \Gamma \vdash \texttt{T} < \texttt{(record l}_1\texttt{:T}_1\texttt{...l}_n\texttt{:T}_n\texttt{)}}{\Gamma \vdash \texttt{(v:T . l}_i\texttt{)} \in \texttt{T}_i} \qquad \text{(V-Proj)}$$

We have now convered the most complex derived forms in Pict. It remains to discuss a few useful extensions of the syntax of processes, abstractions, and patterns.

6.3 Derived Forms for Abstractions

Although Pict's core language and type system do not distinguish between "real functions" and "processes that act like functions," it is often useful to write parts of Pict programs in a functional style. This is supported by a small extension to the syntactic class of abstractions, mirroring the ability to omit the names of result parameters in applications (Section 6.2.3). We replace a process definition of the form

```
def f[a1:A1 a2:A2 a3:A3 r:R] = r!v
```

where the whole body of the definition consists of just an output of some (complex) value on the result channel, by a "function definition" that avoids explicitly giving a name to r:

```
def f (a1:A1 a2:A2 a3:A3) = v
```

To avoid confusion with ordinary definitions (and for symmetry with application), we change the square brackets around the list of arguments to parentheses.

Formally, this is captured by the following translation rule for abstractions:

$$\texttt{(p}_1\texttt{...p}_n\texttt{) : T = v} \Rightarrow \texttt{[p}_1\texttt{...p}_n\texttt{ r:!T] = r!v} \qquad \text{(TR-VABS*)}$$

The typing rule for ordinary abstractions (Section 5.8) yields the following derived rule:

$$\frac{? \vdash p_1 \in \mathrm{T}_1 \Rightarrow \Delta_1 \ldots ? \vdash p_n \in \mathrm{T}_n \Rightarrow \Delta_n \qquad ?, \Delta_1, \ldots, \Delta_n \vdash v \in \mathrm{T}}{? \vdash \texttt{(p}_1\texttt{...p}_n\texttt{) : T = v} \in \texttt{[T}_1\texttt{...T}_n\texttt{!T]}} \qquad \text{(A-VABS*)}$$

Note that the type annotation for the implicit result parameter is given between the `)` and the `=`. This allows us to easily annotate a function with its result type, as in:

```
def plus (x:Int y:Int) : Int = (+ x y)
```

This notation is heavily used in the Pict User Manual. For example, the documentation of `+` is

```
def + (x:Int y:Int) : Int
```

which stands for:

```
def + [x:Int y:Int result:!Int]
```

Since anonymous process declarations like `(def x () = e x)` are often useful, we provide a special form of value allowing the useless `x` to be omitted:

$$\texttt{\textbackslash a} \Rightarrow \texttt{(def x a x)} \qquad \text{(TR-ANONABS)}$$

$$\frac{? \vdash \texttt{a} \in \texttt{T}}{? \vdash \texttt{\textbackslash a} \in \texttt{!T}} \qquad \text{(V-ANONABS)}$$

Anonymous abstractions are used quite heavily in Pict programs. For example, the standard library provides the operation `for`, which takes two integers `min` and `max`, a channel `f` of type `![Int Sig]`, and a completion channel `done`, and successively sends each integer between `min` and `max` to `f`, waiting each time for `f` to signal back before proceeding with the next. When `f` returns for the last time, `for` signals on `done`.

```
def for [min:Int max:Int f:![Int Sig] done:Sig] =
 (def loop x =
    if (<= x max) then
      (new c
       ( f![x c]
       | c?[] = loop!(+ x 1) ))
    else
      done![]
   loop!min
 )
```

The most common use of `for` is to pass it an anonymous abstraction for `f`, as in:

```
    run (new done
         ( for![1 4
                \[x c] = (printi!x | c![])
                done]
         | done?[] = print!"Done!") )

1
2
3
4
Done!
```

Another important use of anonymous process abstractions is in fields of records, where they can be thought of as the method bodies of an object:

```
    val r = (record
      one = \[] = print!"Low hangs the moon"
      two = \[] = print!"O it is lagging"
    )

    run ((r.one)![] | (r.two)![])

Low hangs the moon
O it is lagging
```

The connection with objects is developed in later chapters.

6.4 Sequencing

One very common form of result is a *continuation signal*, which carries no information but tells the calling process that its request has been satisfied and it is now safe to continue. For example, the standard library includes the output operation pr, which signals on its result channel when the output has been accomplished. So a sequence of outputs that are intended to appear in a particular order can be written:

```
    run
     (val [] = (pr "the ")
      val [] = (pr "musical ")
      val [] = (pr "shuttle")
      () )

the musical shuttle
```

(Note that pr, unlike print, does not append a carriage return to the string that is output.)

Since pr sends back an empty tuple to signal completion, its result channel has type ![]. We adopt this convention throughout the standard libraries, and introduce a global type abbreviation

```
    type Sig = ![]
```

which we use to mark channels that are used for signalling completion (or some other condition) rather than for exchanging data. The type of pr, then, is ![String,Sig]. In the documentation for the standard libraries in the User Manual, it is listed in the form

```
    def pr (s:String) : []
```

which means the same as pr ∈ ![String Sig].

The idiom "invoke an operation, wait for a signal as a result, and continue" appears so frequently that it is worth providing some convenient syntax. Whenever v is a value expression whose result is an empty tuple, the expression v; is a declaration clause whose effect is to evaluate v, throw away the result, and then continue with its body. Like all declaration clauses, sequential declarations can appear in sequences, and can be mixed with other declaration clauses in arbitrary ways.

```
    run ((pr "Following ");
         val you = "you, "
         (pr you);
         (pr "my brother.\n");
         () )
```

```
Following you, my brother.
```

Formally, we can use a val construct to wait for the evaluation of v to finish before proceeding with e.

$$\texttt{v;} \Rightarrow \texttt{val [] = v} \qquad \text{(TR-SEMI)}$$

$$\frac{? \vdash \texttt{v} \in \texttt{[]}}{? \vdash \texttt{v;} \Rightarrow \bullet} \qquad \text{(D-SEMI)}$$

In the Pict libraries, many basic operations (like pr) return a null value so that the caller can detect when they are finished. Even in situations where the caller does not care, the null result value must still be accepted and thrown away.

6.5 Complex Patterns

Finally, it is convenient to define some additional forms of patterns. The *wildcard pattern* _, which matches any value, can be used to throw away part of a compound value that is not needed. A layered pattern is used to give a name to the whole of a compound value while also allowing parts of it to be matched more specifically. For example, the pattern x@[_ y _] matches the value [a b c], yielding the substitution {x ↦ [a,b,c], y ↦ b}. We also allow multi-field record patterns.

The meaning of these patterns is most easily defined by a translation function similar to the one we used for complex values. The expression ⟦p ← x⟧ denotes a declaration sequence that matches the complex pattern p against the value of the variable x.

$$
\begin{array}{lcl}
[\![\texttt{y:T} \leftarrow \texttt{x}]\!] & = & \texttt{val y:T = x} \\
[\![\texttt{\_:T} \leftarrow \texttt{x}]\!] & = & \texttt{val y:T = x} \quad \text{with } \texttt{y} \text{ fresh} \\
[\![\texttt{y:T@p} \leftarrow \texttt{x}]\!] & = & \texttt{val y:T = x } [\![\texttt{p} \leftarrow \texttt{x}]\!] \\
[\![\texttt{\{|X<U|\}p} \leftarrow \texttt{x}]\!] & = & \texttt{val \{|X<U|\}y=x}[\![\texttt{p} \leftarrow \texttt{y}]\!] \\
[\![\texttt{[p}_1 \ldots \texttt{p}_n\texttt{]} \leftarrow \texttt{x}]\!] & = & \texttt{val [x}_1 \ldots \texttt{x}_n\texttt{]=x} \\
 & & \quad [\![\texttt{p}_1 \leftarrow \texttt{x}_1]\!] \ \ldots \ [\![\texttt{p}_n \leftarrow \texttt{x}_n]\!] \\
[\![\texttt{(record:T l}_1\texttt{=p}_1 \ldots \texttt{l}_n\texttt{=p}_n\texttt{)} \leftarrow \texttt{x}]\!] & = & \texttt{val (record:T l}_1\texttt{=x}_1\texttt{)=x } [\![\texttt{p}_1 \leftarrow \texttt{x}_1]\!] \ \ldots \\
 & & \quad \texttt{val (record:T l}_n\texttt{=x}_n\texttt{)=x } [\![\texttt{p}_n \leftarrow \texttt{x}_n]\!] \\
[\![\texttt{(rec:T p)} \leftarrow \texttt{x}]\!] & = & \texttt{val (rec:T y) = x } [\![\texttt{p} \leftarrow \texttt{x}]\!]
\end{array}
$$

We rewrite every complex pattern p in an abstraction as follows:

$$\frac{\texttt{p}\ \text{complex}}{\texttt{p = e} \Rightarrow \texttt{x = (}[\![\texttt{p} \leftarrow \texttt{x}]\!]\ \texttt{e)}} \qquad \text{(Tr-Abs)}$$

(Of course, complex patterns may also occur in `val` declarations; these can be eliminated by using Tr-Val to convert the declaration into a process involving an input. Note that there is no danger of circularity here, as all `val` declarations created by the translation rules, including the definition of $[\![\texttt{p} \leftarrow \texttt{x}]\!]$, are simple.)

Part II

Advanced Topics

Chapter 7

Simple Concurrent Objects

As an example of many of the derived forms described in Chapter 6, let us see how a simple *reference cell* abstraction can be defined in Pict.

A reference cell can be modeled by a process with two channels connecting it to the outside world — one for receiving `set` requests and one for receiving `get` requests. For example, suppose that our cell holds an integer value and that it initially contains `0`. Then its behavior can be defined like this:

```
new contents: ^Int          {- Create a local channel holding current contents -}
run contents!0              {- "Initialize" it by sending 0 -}

def set [v:Int c:Sig] =     {- Repeatedly read 'set' requests... -}
    contents?_ =            {- discard the current contents... -}
    (contents!v | c![])     {- install new contents and signal completion -}

def get [res:!Int] =        {- Repeatedly read 'get' requests... -}
    contents?v =            {- read the current contents... -}
    (contents!v | res!v)    {- restore contents and signal result -}
```

The current value of the cell is modeled by a waiting sender on the channel `contents`. The process definitions `set` and `get` must be careful to maintain the invariant that, at any given moment, there is at most one process waiting to send on `contents`; furthermore, when no instances of `get` or `set` are currently running, there should be exactly one sender on `contents`. To protect against other processes reading or writing `contents` and possibly destroying the invariant, it is good practice to wrap the declaration of `contents` in a `local` block so that it is visible only within the definitions of `get` and `set`.

```
local (
  new contents:^Int
  run contents!0
) in (
  def set [v:Int c:Sig] =
      contents?_ =
      ( contents!v | c![] )
  def get [res:!Int] =
      contents?v =
      ( contents!v | res!v )
)
```

We can test that our cell is behaving as we expect by sending a few requests and printing the results. Note the use of application and sequencing syntax.

```
run ((prNL (intString (get)));
     (set 5);
     (prNL (intString (get)));
     (set ~3);
     (prNL (intString (get)));
     ())
```

```
0
5
~3
```

(The operation `prNL` is a version of `pr` that prints its string argument followed by a newline character.)

This definition is fine if all we need is a single reference cell, but it would be awkward to have to repeat it over and over, choosing different names for the `set` and `get` channels of each whenever we needed a new reference cell. As we did for booleans, we can encapsulate it in a process definition that, each time it is invoked, generates a fresh reference cell and returns the `set` and `get` channels to the caller as a pair.

```
def refInt [res: ![![Int Sig] ![!Int]]] =
(new contents:^Int
 run contents!0
 def set [v:Int c:Sig] = contents?_ = ( contents!v | c![] )
 def get [res:!Int]    = contents?v = ( contents!v | res!v )
 res![set get]
)
```

Now we can build multiple reference cells and use them like this:

```
val [set1 get1] = (refInt)
val [set2 get2] = (refInt)

run ((set2 5);
     (prNL (intString (get1)));
     (prNL (intString (get2)));
     ())
```

```
0
5
```

But it is not very convenient to have to bind two identifiers each time `refInt` is invoked. A cleaner solution is to bind a single identifier to the whole pair returned by `refInt`:

```
val ref1 = (refInt)
val ref2 = (refInt)
```

Moreover, if we modify `refInt` to return a two field record instead of a two-element tuple, then we can simply use record field-projection syntax to extract whichever request channels we need:

```
def refInt [res: !(record set:![Int Sig] get:![!Int])] =
(new contents:^Int
 run contents!0
 def set [v:Int c:Sig] = contents?_ = ( contents!v | c![] )
 def get [res:!Int]    = contents?v = ( contents!v | res!v )
```

```
     res!(record set=set get=get) )

    val ref1 = (refInt)
    val ref2 = (refInt)

    run (((ref2.set) 5);
         (prNL (intString ((ref1.get))));
         (prNL (intString ((ref2.get))));
         ())
```

```
0
5
```

The header of `refInt` will be easier to read if we move the long type of its result to a separate type definition:

```
    type RefInt = (record
      set:![Int Sig]
      get:![!Int]
    )
```

Finally, for a final touch of syntactic polish, we can move the definitions of `set` and `get` directly into the fields of the record that is being returned.

```
    def refInt [res:!RefInt] =
    (new contents:^Int
     run contents!0
     res ! (record
              set = \[v:Int c:Sig] = contents?_ = ( contents!v | c![] )
              get = \[res:!Int]    = contents?v = ( contents!v | res!v )
           ))
```

and make the result `res` anonymous by making `refInt` a value abstraction instead of a process abstraction:

```
    def refInt () : RefInt =
    (new contents:^Int
     run contents!0
     (record
        set = \[v:Int c:Sig] = contents?_ = ( contents!v | c![] )
        get = \[res:!Int]    = contents?v = ( contents!v | res!v )
     ))
```

What we have done, in effect, is to introduce a *function* (`refInt`) that creates reference cell *objects*, each consisting of

- a "server process" with some internal state that repeatedly services requests to query and manipulate the state, while carefully maintaining a state invariant, even in the presence of multiple requests, and
- two request channels used by clients to request services, packaged together in a record for convenience.

Active objects of this kind, reminiscent of (though lower-level than) the familiar idiom of *actors* [Hew77, Agh86] (also cf. [Nie92, Pap91, Vas94, PT95, SL96, Var96, NSL96]), seem to arise almost inevitably when programming in a process calculus. They are widely used in Pict's libraries.

Chapter 8

Polymorphism

From here to the end of the document is still under (re)construction. Some sections are incomplete; others use inconsistent syntax. Don't believe everything you see! (Even the inset program examples are often broken.)

We have completed a full tour of Pict's core language, core type system, and derived forms. As regards the behavior of programs, we have now seen the whole language. However, there is more to say about the type system — i.e., about the rules that determine which programs are allowed to compile, and where these programs must be annotated with type information to help the typechecker do its job. The next few chapters treat various advanced aspects of the type system in more detail.

On a first reading, it may suffice to read just Section 8.1 and Chapter 9: these contain most of the information necessary to begin writing larger programs.

8.1 Overview

The practical utility of *polymorphism* — the ability to write a single piece of code that deals uniformly with data of varying types — is well established in the programming languages community. For example, suppose we want to define a reference cell abstraction like the one in the previous chapter, except that it should hold a boolean instead of an integer:

```
type RefBool = (record
  set:![Bool Sig]
  get:![!Bool]
)

def refBool () : RefBool =
(new contents:^Bool
 run contents!false
 def set [v:Bool c:Sig] = contents?_ = ( contents!v | c![] )
 def get [res:!Bool]    = contents?v = ( contents!v | res!v )
 (record set=set get=get) )
```

This definition is identical to the other, except that `Bool` has been substituted for `Int` throughout and the initial value has been changed to `false`. Clearly, rather than writing yet another such definition each time we need a reference cell holding a new type, it is better to define, once and for all, a generic reference cell type

```
type (Ref X) = (record
  set:![X Sig]
  get:![!X]
)
```

and a polymorphic constructor that can build reference cells of any type.

```
def ref (|X| init:X) : (Ref X) =
(new contents:^X
 run contents!init
 def set [v:X c:Sig] = contents?_ = ( contents!v | c![] )
 def get [res:!X]    = contents?v = ( contents!v | res!v )
 (record set=set get=get) )
```

Intuitively, when we invoke `ref` we must send it both a type `X` and an initial value of type `X`, as well as an implicit result channel; it then constructs a reference cell whose `set` and `get` operations expect and return values of type `X`, and sends it back. In other words, the channel `ref` can be thought of as carrying triples consisting of a type and two channels. We write its type as `!{|X|}[X !(Ref X)]`, placing the type component in front in a separate set of brackets to separate it from the two ordinary (value) components.

In order to build a reference cell, we must send a request along the `ref` channel — that is, we must actually construct an element of the type `{|X|}[X !(Ref X)]`. This is done by extending the syntax of tuple values as we did with tuple types, allowing the type component to be added in brackets at the front

```
new res: ^(Ref Bool)
run ref!{|Bool|}[false res]
run res?r1 = ( ((r1.set) true); () )
```

or, more concisely (using application notation):

```
val r1 = (ref |Bool| false)
run ( ((r1.set) true); () )
```

The Pict library includes polymorphic reference cells like these, as well as several other useful polymorphic datatypes: lists, arrays, hash tables, dictionaries, etc. These are described in the User Manual. Our task for the rest of this chapter is to extend the earlier formal definitions of Pict's typing relation to account for polymorphism.

8.2 Syntax

Formally, we treat polymorphic communications like `ref ! {|Bool|}[false res]` by extending the syntax of tuple values so that each tuple consists of an optional sequence of types followed by a sequence of values:

```
TupleVal = ['[:' <Type ...> ':]'] '[' <Val ...> ']'
```

This definition extends the earlier one, since the sequence of types and its enclosing brackets are optional.

Next, we must extend the definition of typing contexts ? so that they not only describe typing assumptions for variables, but also maintain a list of the type variables that are currently being used. From now on, a typing context will be a list of

1. typing assumptions like `x:T` (as before), and
2. type variables, like `A`.

In each typing assumption `x:T`, we require that `T` only contain type variables already mentioned to the left of this binding. So

$?_{ok}$ = `A, x:!A, B, t:[A A B]`

is a good context, while

$?_{bad}$ = `x:!A, A, t:[A A B]`

is not.

Certain type identifiers such as `Bool` and `^` are treated specially by the compiler: they are predefined in the standard basis and it is illegal to redefine them in programs. However, most of the identifiers supplied by the standard basis, including symbolic identifiers, may freely be redefined.

Having extended the syntax of tuple values, we must also extend the syntax of tuple patterns:

```
TuplePat = [´[:´ <Id ...> ´:]´] ´[´ <Pat ...> ´]´
```

For example, we may use such patterns in value declarations

```
val {|X|}[i:X r:^(Ref X)] = {|Bool|}[false res]
```

or, as in the previous section, in abstractions:

```
def ref (|X| init:X) : (Ref X) = ...
def ref (|X| init:X) : (Ref X) =
```

8.3 Typing Rules

The typing rule for the extended form of tuple values is: *[V-Tuple**?]* That is, a polymorphic tuple consisting of the sequence `[:`S_1 ... S_n`:]` of types and the sequence `[`v_1 ... v_n`]` of values can validly be given the polymorphic type `[:`X_1 ... X_m`:][`T_1 ... T_n`]`, in which the concrete types S_1 through S_m are replaced by the variables X_1 through X_m, provided that the actual types of the values v_1 through v_n match the abstracted types T_1 through T_n after the substitution of the S's for the X's. For example, if `res` $\in$ `^(Ref Bool)`, then the value `[:Bool:][false res]` has type `[:X:][X, (Ref X)]`, since `false` has type $\{$`X` $\mapsto$ `Bool`$\}$`X` = `Bool` and `res` has type $\{$`X` $\mapsto$ `Bool`$\}$`(^(Ref X))` = `^(Ref Bool)`.

The corresponding typing rule for polymorphic patterns is: *[P-Tuple**?]* That is, the pattern `[:`A_1 ... A_m`:][`p_1 ... p_n`]` matches values of the polymorphic type `[:`x_n ... x_m`:][`T_1 ... T_n`]`, yielding bindings for the type varibles A_1 to A_m as well as the bindings Δ_1 through Δ_n, provided that each pattern p_i matches values of type T_i and yields bindings Δ_i.

Readers familiar with typed λ-calculi will recognize the similarity of these rules to the standard introduction and elimination rules for existential types (e.g. [CW85, MP88]).

For example, the declaration

```
val v = (  {|Int|} [5 \c = printi!c ]
         : {|A|}[A !A] )
```

defines a value `v` of type `{|A|}[A !A]`, which can be read as, "`v` is a package containing some type `A`, an element of `A`, and a channel on which element of `A` can be sent." This value can be matched against the pattern

```
    val {|A|}[x:A c:!A] = v
```

to yield bindings for a type variable A, a value x of type A, and a channel c of type !A. Now, at this point, the only interesting thing we can do with x is to send it along c:

```
    run c!x
```

```
5
```

In particular, trying to use x as an integer fails to typecheck:

```
    run printi!x
```

Notice that we do *not* need to extend the definition of the pattern matching function *match* to deal with this new form of pattern, because all we have added is type information, which will be erased as part of the step from abstract syntax to untyped abstract syntax. The matching function only deals with untyped computations; the present extension to patterns only enriches the typing relation without affecting behavior.

From the rule T-Tuple** for polymorphic tuples and the translation rule Tr-App for applications, we obtain a more powerful derived typing rule for applications:

$$\frac{\begin{array}{c} ? \vdash \mathtt{v} \in \mathtt{![A_1 \;\ldots\; A_m \,|\; T_1 \;\ldots\; T_n \;\; !T]} \qquad \{\mathtt{A}_1, \ldots, \mathtt{A}_m\} \cap FV(\mathtt{T}) = \emptyset \\ \text{for each } j \qquad ? \vdash \mathtt{v}_j \in \{\mathtt{A}_1, \ldots, \mathtt{A}_m \mapsto \mathtt{S}_1, \ldots, \mathtt{S}_m\}\mathtt{T}_j \end{array}}{? \vdash (\mathtt{v} \;\; |\mathtt{S}_1 \;\ldots\; \mathtt{S}_m|\;\; \mathtt{v}_1 \;\ldots\; \mathtt{v}_n) \in \mathtt{T}} \qquad \text{(V-App**)}$$

For example, we can invoke a polymorphic definition with application syntax like this:

```
    def f {|A|}[a1:A a2:A c:!A] = c!a2

    run printi ! (f |Int| 333 444)
```

```
444
```

Or even, omitting the type argument and letting the type inference algorithm fill it in:

The case of infix applications requires special treatment. An expression of the form $\mathtt{v}_1$ `infixid` $\mathtt{v}_2$ is translated (via Tr-InfId) into `infixid[`$\mathtt{v}_1$ $\mathtt{v}_2$`]`. If `infixid` is a polymorphic channel, then we may want to write an analogous infix expression that, intuitively, translates into `infixid[:`$\mathtt{S}_1$ ... $\mathtt{S}_m$`:][`$\mathtt{v}_1$ $\mathtt{v}_2$`]`, but there is no obvious place to put the type arguments `[:`$\mathtt{S}_1$ ... $\mathtt{S}_m$`:]` in the infix form. We handle this by making a small extension to the syntax of infix expressions, allowing $\mathtt{v}_1$ `infixid @ [:`$\mathtt{S}_1$ ... $\mathtt{S}_m$`:]` $\mathtt{v}_2$, where the `@` signals that the list of types that follow it are arguments to `infixid`, not part of $\mathtt{v}_2$. The derived typing rules for infix identifiers and general infix values are now: *[V-InfId**?]* *[V-InfV**?]* The explicitly annotated forms are rare in programs, since (in infix expressions like elsewhere) type arguments can normally be inferred by the type inference algorithm, and need not be supplied explicitly by the programmer.

Chapter 9

Type Inference

Most modern programming languages provide some form of *type inference* to relieve the programmer of the burden of writing huge amounts of explicit type information. In the best case (as in ML, for example [Mil78, DM82]), it can be shown that the type inference method is *complete*, in the sense that it accepts any program whose *explicitly typed form* (with all type annotations explicitly included in the text) is well typed. However, complete type inference algorithms exist only for a restricted class of type systems based on predicative (ML-style) polymorphism.

As type systems go, Pict's is fairly ambitious. In particular, it includes both impredicative (System-F-style) polymorphism and subtyping. In the presence of these features, there appears to be no hope of finding complete a type inference algorithm (or even a well-behaved semi-algorithm). Instead, the Pict compiler includes an *incomplete*, but highly useful, partial type inference facility.

This chapter explains how the partial type inference algorithm works, warns about some potentially surprising aspects of its behavior, and suggests a programming style that will avoid most surprises.

Throughout the chapter, the examples use the "`set types`" switch of the compiler to display the types that it infers for pieces of program text. The first line in the program

```
now (set types)
val d@[b c] = [true 5]
```

sets the internal compiler flag `types`, so that the types of the variables introduced by the declaration in the second line are printed during compilation:

```
c : Int
b : Bool
d : [Bool Int]
```

9.1 Behavior

The main advantage of Pict's type inference algorithm is that it is quite simple. Similar ideas have been used in a number of programming languages and theorem provers; our implementation is based on one used by Cardelli [Car93] in his implementation of System $F_{\leq}$.[1]

We begin by introducing *unification variables*, which stand for "unknown types." A unification variable can be introduced explicitly in a program by writing `Any` in place of a type expression.

[1]Technically, our algorithm differs from Cardelli's mainly in the places where we generate fresh unification variables during typechecking and in the fact that our underlying type system is based on $F_{\leq}^{\omega}$, not $F_{\leq}$, which requires us to consider more interactions between subtyping and instantiation of unification variables.

```
    new x:^Any

x : ^Any4
```

Each occurrence of the keyword `Any` generates a different unification variable named `AnyNNN`, where `NNN` is a unique numeric index. Once introduced, a unification variable behaves just like any other type. For example, it may become part of the type of another value:

```
    new a:^Any
    val b = [a a a]
    val c = b

a : ^Any6
b : [^Any6 ^Any6 ^Any6]
c : [^Any6 ^Any6 ^Any6]
```

When, during typechecking, a subtyping constraint of the form `AnyNNN` < `T` or `T` < `AnyNNN` is encountered, the unification variable `AnyNNN` becomes *instantiated* to the type `T`. From this point on, `AnyNNN` and `T` are completely equivalent. For example, if we introduce a channel `n` with a unification variable in its type and then try to send an integer along `n`, the unification variable gets instantiated to `Int`.

```
    new n:^Any
    run n!5
    val m = n

n : ^Any8
m : ^Int
```

This example illustrates an important point. The `set types` switch causes the type of each binding to be printed *at the moment at which the binding is made.* The type that is printed may therefore contain unification variables that later become instantiated to concrete values. Thus, if the type of the same binding were printed at a later point in the compilation, it might be different.

Unification variables are automatically inserted by the parser at each point in a program where a type annotation is expected but none is provided. In principle, *all* type annotations may be omitted in this way.

```
    new p
    def f [i j] = ()

p : ^Any11
f : ![Any13 Any12]
```

Unification variables in the types of process definitions are treated just like unification variables in the types of channels. They are inserted in the headers of `def` clauses when types of parameters are omitted, and they become instantiated as typechecking proceeds. For example, in

```
    def g[i j] = j!i
    val d = (g false)
    val g´ = g

g : ![Any17 !Any17]
d : Bool
g´ : ![Bool !Bool]
```

the definition of `g` creates a new unification variable. The body `j!i` instantiates the type of `j` to a channel type, but does not determine what type it carries, except that this type should be the same as the type of i. The next line uses g by sending along it a pair of a boolean and a result channel (of unknown type); this has the effect of instantiating both the type of i in the definition of g and the type of the result channel. The third line shows that the type of g is now fully instantiated.

In straightforward programs, this simple process of generating unification variables and instantiating them when they meet nontrivial constraints yields predictable behavior and allows most type annotations to be omitted. However...

9.2 Misbehavior

It is important to realize that the process of instantiating unification variables is *greedy*: at the first moment when any nontrivial constraint is applied to a unification variable, it is instantiated so as to satisfy the constraint in the simplest possible way. No attempt is made to choose a "best possible" instantiation. This means that the typechecker may make a bad decision, instantiating a unification variable in such a way that the remainder of the program will fail to typecheck. For example, the built-in type Char is a subtype of Int, so the program

```
new c:^Int
run c!'a'
run c!5
```

is well typed. But if the type annotation on c is omitted in the first line, the constraint arising from the output in the second line will cause the type of c to be instantiated to ^Char, so that the output in the third line will fail, even though it would also have been possible to instantiate the type of c to ^Int in the first place.

```
new c
run c!'a'
run c!5
```

```
example.pi:1.10: Parse error
```

9.2.1 Exercise [Recommended]: *Write a program* e *such that:*

1. *When all of the variable binders in* e *are explicitly annotated with types,* e *typechecks.*
2. *When* none *of the variable binders in* e *is annotated with explicit types,* e *also typechecks.*
3. *Annotating some binders in* e *but not others with explicit types makes* e *fail to typecheck.*

[Solution on page 96.]

In polymorphic definitions, the situation is even worse: the type inference algorithm's greedy behavior essentially *prevents* it from making sufficiently general instantiations when bound type variables are involved. As an example of what can happen, suppose we write a polymorphic definition of a reference cell, as in Section 8.1, but omit the type annotations.

```
def ref (|X| init) =
(new contents
 run contents!init
 (record
    set = \[v c] = contents?_ = ( contents!v | c![] )
    get = \[res] = contents?v = ( contents!v | res!v )
 ))
```

After analyzing the body of `ref`, the unification variables in the header are constrained so that the type of the initial value is the same unification variable as the argument to the `set` method and the result of the `get` method of the returned reference cell.

```
ref : !{X}[Any17 !(record set : ![Any17 ![]] get : ![!Any17])]
```

When `ref` is applied to the initial value `true` to create a boolean reference cell, the single remaining unification variable becomes instantiated to `Bool`, yielding a result of the expected type.

```
    val r = (ref true)
```

```
r : (Ref Bool)
```

The type that is printed is equivalent to `Ref Bool` — i.e., it is formed by substituting `Bool` for the parameter `X` in the body of the definition of `Ref`. In general, the typechecker does not attempt to automatically "collapse" type definitions when printing types.

But a side-effect of this last instantiation is that the unification variable in the type of `ref` becomes instantiated to `Bool`

```
    val ref´ = ref
```

```
ref´ : !{X}[Bool !(record set : ![Bool ![]] get : ![!Bool])]
```

which means that the definition of `ref` is not actually polymorphic, as we intended. If we now try to create an integer reference cell, the application of `ref` fails.

```
    val i = ref[999]
```

```
example.pi:2.12: Parse error
```

The step where things go wrong here is when the type inference algorithm instantiates the unification variable in the type of `ref` to `Bool`. Looking at the program, we can see that a better choice would have been to instantiate it to `X`, so that the definition of `ref` would be as polymorphic as possible: then both applications of `ref` would have succeeded. But this observation is beyond the capabilities of a simple greedy algorithm.

9.3 Playing It Safe

Having looked at a few examples of both good and bad behavior, we now propose a programming style that allows most type annotations to be omitted while avoiding puzzling type errors arising from the incompleteness of type inference. In general, greedy type inference works fine in programs (or sections of a program) that are completely *monomorphic* — i.e. where each channel is used to carry values of exactly one type. When either subtyping or polymorphism is involved, unification variables may be instantiated in an insufficiently general way. Based on this observation, we suggest the following principles.

For a given binding occurrence of a variable `x` in a pattern (e.g. a variable in a `val` declaration, a parameter to a `def`, or a bound variable in an input prefix), decide what is the *expected type* of `x` — the type that it would be given if all type annotations were included in the program. Now:

1. When the pattern in which `x` is bound is polymorphic and the expected type `T` contains an occurrence of one of the bound type variables, then the annotation `x:T` should be written explicitly.

   For example, write

```
def ref (|X| init:X) : Ref X = ...
```

instead of:

```
def ref (|X| init) : Ref X = ...
```

2. If `x` is used in the body with different types (or with any type different from `T`), then the annotation `x:T` should be written explicitly.

   For example, if `x` is used to send both integers and characters in `e`, then write

   ```
   def f (x:!Int) = e
   ```

   instead of:

   ```
   def f (x) = e
   ```

3. If `T` is a record type and `x` appears in the body in a field projection expression like `x.l`, then the annotation `x:T` should be given explicitly. (This principle is actually a special case of the previous one.)

4. Otherwise (i.e., if `T` does not refer to type variables bound in the same pattern as `x` and if `x` is always used with the same type throughout its scope), then the annotation `:T` may be omitted.

This advice can be summarized even more succinctly:

Use type annotations in the headers of polymorphic definitions and whenever a variable is used with different types because of subtyping.

As a matter of style, we often annotate arguments to definitions even if the annotations could be inferred, since this helps make programs self-documenting.

We should emphasize (in case it isn't obvious) that the type inference facility of Pict is experimental. We are still searching for a deep understanding of how to design this kind of algorithms and talk about their theoretical properties. We would be grateful for feedback from users on their experiences with the present algorithm.

Chapter 10

Choice

The behavior of the primitives described in this chapter corresponds to the `Events` library provided with the implementation, but names have been changed in some cases.

One facility that Pict inherits from the pure $-pi$-calculus still remains to be discussed: the *choice* or *summation* operator. It appears last both because many useful programs can be written without it and because its status in process calculi like the $-pi$-calculus — and even more so in programming languages — is still a matter of controversy.

10.1 Programming with Choice

A choice expression allows one course of action to be chosen, nondeterministically, from several alternatives. Those not chosen are immediately aborted. Some presentations of the $-pi$-calculus allow a choice between arbitrary processes: executing the expression `P+Q+R` will have the effect of choosing one of `P`, `Q`, and `R` (assuming that each is ready to take a step), allowing it to proceed, and throwing away the other two. Here, we allow only a restricted form, called *input-only choice*, where the processes `P`, `Q`, and `R` must each start with an input action. This variant is substantially simpler to implement, and suffices for many situations where choice is needed in practice.

Because the underlying mechanisms needed to implement the choice operator are somewhat expensive, we distinguish the ordinary channels we have seen so far from *input event channels*, which are created and manipulated by separate means. An input event channel is created using the predefined operation `buildEventChan` operator. If `c` is an ordinary channel, then `(buildEventChan c)` is an input event channel that can be used to read values sent along `c`. For example, writing

```
now (set types)
new c: ^Int
val cev = (buildEventChan c)
```

creates a new input event channel `cev` that can be used to read values send along `c`. Note that the type of `c` is `(EChan Int)`, not `^Int` or `?Int`: input event channels are completely distinct from ordinary channels.

The syntax of input expressions involving input event channels is somewhat different from (and heavier than) than the syntax of ordinary input: we use the infix operator `=>` instead of `?`, put an explicit abstraction on the right-hand side instead of just listing the bound variables, and wrap the input action with the predefined operation `sync`:

```
    run ( c!999
        | sync!(=> cev \i = printi!i))
```

```
999
```

This program sends the value `999` on the ordinary channel `c` and, in parallel, waits to read a value from the input event channel `cev`, binds `i` to the value received, and prints it.

In general, several choices may be provided in the same `sync`, separated by the infix operator `$`.

```
    new c:^Int new d:^[]
    val cev = (buildEventChan c)
    val dev = (buildEventChan d)
    run ( c!999
        | d![]
        | sync!($
            (=> cev \i = printi!i)
            (=> dev \[] = print!"on the prong of a moss-scallop'd stake")
          ))
```

```
999
```

(We use `$` instead of `+` because `+` is already used for integer addition.)

Operationally, an expression of the form `sync!(`S_1 `$` S_2 `$ ...)` (where each S_i has the form `(`x_i `=> \`y_i `=` e_i`))` behaves as follows:

- If any of the input actions is ready to proceed — i.e., some other process is already waiting to perform an output on the same event channel x_i — then one of the ready ones is chosen arbitrarily, the communication occurs, and the corresponding suffix e_i is allowed to continue. The rest of the `S`'s are thrown away.

- If none of the inputs can proceed now, then the choice waits until one or more become ready; at this point, one is chosen (nondeterministically), the communication occurs, and the rest are thrown away.

After an input event channel has been attached to an ordinary channel `c` using `buildEventChan`, there is normally no need to perform ordinary (low-level) inputs on `c`. The Pict library supports this common case by providing a function `newEventChan` that creates a fresh channel, attaches an input event channel, and returns the two as a pair:

```
    def newEventChan (|X|) : [!X (EChan X)] =
     (new send:^X
      [send (buildEventChan send)]
     )
```

This allows us to write

```
    val [c:!Int cev:(EChan Int)] = (newEventChan)
```

instead of:

```
    new c:^Int
    val cev = (buildEventChan c)
```

10.1.1 Exercise [Recommended]: *The* dining philosophers problem *is a common example in textbooks on concurrent programming. One variant goes like this:*

> *Imagine a dinner party at a Chinese restaurant where five philosophers have met to eat and think together. They find themselves seated around a circular table, each with a plate of noodles. But due to a miscommunication with the restaurant manager, there are only five chopsticks, each one placed between two of the philosophers; in order to eat, a philosopher must pick up both of the chopsticks, one after the other in either order. After eating, she returns both chopsticks to the table and thinks for a while, until she becomes hungry again.*

The point of the exercise is to prevent starvation: if, by bad luck, each philosopher were to pick up her left hand chopstick (say) at the same moment and then simply wait for the chopstick on her right to become free, no one would ever be able to eat again. To prevent this situation, they must all behave politely: if someone picks up a chopstick and then discovers that the other one is already being used, she should put down the first and go back to thinking for a while.

1. *Construct a table arrangement with four philosophers and four chopsticks. Each chopstick is represented by a* semaphore *process that communicates alternately on two event channels,* `grab` *and* `release`*. Each philosopher is a process that repeatly cycles through the following sequence of actions: thinking, grabbing either the left or the right chopstick, grabbing the other chopstick (but releasing the first one if the other is not available), eating, and releasing both chopsticks.*

   During the "thinking" phase, each philosopher should wait for a random interval by invoking the process `delay`*:*

   ```
   def delay () = (for 1 (randomInt 20) \(i) = [])
   ```

2. *What happens to the behavior of your solution if* `(randomInt 20)` *in the definition of* `delay` *is replaced by the constant* 20*? Why?*

3. *[Optional] Generalize your program so that the number of philosophers can be given as a parameter.*

[Solution on page 97.]

10.2 Simple Objects Using Choice

We saw in Chapter 7 how the basic facilities of Pict can be used to construct simple concurrent objects. The key idiom there was the use of a private channel as a "lock" to prevent simultaneous access to the shared data by the critical section of more than one method. A different style of concurrent objects can be built using events to ensure that only one message at a time is accepted by an object, so that the internal state need not be locked.

Instead of ordinary channels, objects written in this style use event channels as the ports by which they receive requests from clients. For example, a reference cell object is built by creating two event channels,

```
def refIntE () : RefInt = (
  val [sendSet set] = (newEventChan)
  val [sendGet get] = (newEventChan)
```

starting a "server" process to handle requests on them,

```
  def processRequests v =
        sync!($
          (=> set \[v´:Int c:Sig] = ( processRequests!v´ | c![] ))
          (=> get \[res:!Int]     = ( processRequests!v | res!v ))
        )
  run processRequests!0
```

and returning a record of the “sender sides” of the event channels.

```
 (record set=sendSet get=sendGet)
)
```

The fact that the clients themselves are passed only the sender sides means that the difference in implementation is invisible from the outside: these reference cells work the same as the ones in Chapter 7.

The “server loop” `processRequests` is parameterized on the current contents `v` of the reference cell. At the beginning, we start a single instance, passing it the initial value `0`. Each time it runs, `processRequest` chooses between accepting a `set` or a `get` message; in each case, it sends some appropriate response to the client and invokes a fresh copy of `processRequest` (passing the clients new value `v´` as the new current value in the case of `set`).

10.3 Example: Priority Queues

One simple concurrent data structure, illustrating some common Pictish programming idioms, is the priority queue, a standard example from the concurrent object-oriented programming literature (e.g. [Jon93, Wal94]). A priority queue represents a collection of elements of some ordered set — say, for simplicity, the integers. The elements are stored in sorted order, and operations are provided for adding and removing elements, for asking whether the set of elements is empty, and for obtaining the smallest element, removing it from the queue at the same time. The type of priority queue objects, then, is:

```
type PQueue = (record
  add: ![Int Sig]
  remove: ![Int Sig]
  first: ![!Int]
  isEmpty: ![!Bool]
)
```

Notice that unlike lists, for example, none of the priority queue operations accepts or returns a priority queue, so the type need not be recursive. We will consider recursively typed data structures in Chapter 12.

The behavior of priority queues is expressed by providing an operation for building a new one. Calling `newPQueue` results in the creation of an object with two internal states: `empty`, which represents an empty queue, and `(full h t)`, representing a queue whose smallest element is `h`, where `t` is another queue representing the rest of the elements.

```
def newPQueue () =
 (val [addS add] = (newEventChan)
  val [removeS remove] = (newEventChan)
  val [firstS first] = (newEventChan)
  val [isEmptyS isEmpty] = (newEventChan)
```

```
    val q = (record add=addS remove=removeS first=firstS isEmpty=isEmptyS)

    def empty[] =
      sync!($
         (=> isEmpty \[r] = (r!true | empty![]))

         (=> add \[h c] = (full![h (newPQueue)] | c![]))
      )

    and full[h:Int t:PQueue] =
      sync!($ >
         (=> isEmpty \[r] = (r!false | full![h t]))

         (=> add \[newh c] =
           if (>= newh h) then (((t.add) newh); (full![h t] | c![]))
           else (((t.add) h); (full![newh t] | c![])))

         (=> remove \[oldh c] =
           if (== oldh h) then
             if ((t.isEmpty)) then
               (empty![] | c![])
             else
               (full![((t.first)) t] | c![])
           else
             (((t.remove) oldh); (full![h t] | c![])))

         (=> first \[r] =
           if ((t.isEmpty)) then (empty![] | r!h)
           else (full![((t.first)) t] | r!h))
      )

    run empty![]
    q)
```

In the `empty` state, a queue responds only to the `isEmpty` and `add` messages. The latter causes it to enter the `full` state with the given element as the first and a fresh, empty `PQueue` as the rest. In the `full` state, the message handlers need to compare the first element with the element being added or removed and behave accordingly.

Now, let us check our work by creating a `PQueue`

```
val q = (newPQueue)
```

and testing its behavior:

```
run
  (((q.add) 8);
   ((q.add) 5);
   ((q.add) 13);
   ((q.add) 2);
   ((q.remove) 5);
   (prInt ((q.first))); (pr " ");
   (prInt ((q.first))); (pr " ");
   (prInt ((q.first))); (pr " ");
   ())
```

2 8 13

10.3.1 Exercise [Easy]: *What happens if* `remove![x,c]` *is sent to a queue that does not contain the element* `x`*?*

10.3.2 Exercise [Moderate]: *This queue implementation is not very concurrent. For example, the* `remove` *operation does not signal to the client until the element has actually been found and deleted. Rewrite* `newPQueue` *to achieve as much concurrency as possible, while maintaining the property that sequences of calls on the queue operations will always be performed in the correct order.*

10.3.3 Exercise [Research problem]: *Give a rigorous argument that your refined implementation meets some reasonable specification of its behavior. (This exercise is difficult because solving it entails inventing a notation for specifications, writing the specification itself, and finding a convincing way of arguing that the program corresponds to the specification.)*

10.3.4 Exercise [Moderate]: *Pict's standard library includes several primitives that can return* exceptions *if they are invoked with unreasonable arguments. Use these to extend your implementation of queues so that* `remove` *can signal an error (using a client-specified handler) if the named element is not in the queue.*

10.3.5 Exercise [Moderate]: *Refine this implementation so that the elements of a* `PQueue` *can be drawn from any ordered set.*

10.3.6 Exercise [Optional]: *Our encoding of the boolean values* `true` *and* `false` *as −pi-calculus processes in Section 1.7 bears some resemblance to Church's encoding of the booleans in λ-calculus:*

$$\begin{array}{lcl} tt & = & \lambda t.\lambda f.t \\ ff & = & \lambda t.\lambda f.f \end{array}$$

Of course, the −pi-calculus encoding does not work exactly like Church's. In particular, the notation of λ-calculus allows t or f to be "returned as a result" simply by placing it in a certain syntactic position. The −pi-calculus, on the other hand, has no primitive concept of returning results: it has to be programmed explicitly by means of signals on certain channels that tell other processes when to proceed.

Church also gave an encoding of the natural numbers and the arithmetic operations on them:

$$\begin{array}{lcl} zero & = & \lambda s.\lambda z.\, z \\ one & = & \lambda s.\lambda z.\, s(z) \\ two & = & \lambda s.\lambda z.\, s(s(z)) \\ & & \\ succ & = & \lambda n.\lambda s.\lambda z.\, s\,(n\,s\,z) \\ plus & = & \lambda m.\lambda n.\lambda s.\lambda z.\, m\,s\,(n\,s\,z) \end{array}$$

Of course, we have seen that the notion of function application, which is primitive in the λ-calculus, can also be encoded in the −pi-calculus. But the encoding is a bit complicated, so it is worthwhile to look for simpler ways of building processes that can be used to represent numbers. Can you find one?

10.3.7 Exercise [Optional]: *If your solution to the previous exercise used events, can you find a way to do without them?*

Hint: the simplest solution using only the constructs of the core language seems to be the following: The natural number n is represented by a process that reads two channels `s` *and* `z` *and sends n signals (empty tuples) along* `s`*, waiting after each one for an acknowledgement from the receiver, followed by one signal along* `z`*. The types of the operators involved may be defined as follows:*

```
type Signal = ^[]
type AckReq = ^Signal
type Number = ^[AckReq Signal]

def zero [n:Number]                       {- Create a proc representing 0 located at n -}
def succ [m:Number  n:Number]             {- Create a proc representing m+1 at n -}
def plus [m:Number n:Number o:Number]     {- etc. -}
```

[Solution on page 100.]

10.4 Programming with Events

John Reppy [Rep92, Rep90, Rep88, Rep91, Rep95] has argued that the programming facilities offered by the pure $-pi$-calculus and many related concurrent programming languages are inadequate in an important way: they provide insufficient support for *abstraction* in concurrent code. The main culprit is the restriction to *first-order choice*, where the branches of a summation must all be phrases beginning with an input or output action. This means, for example, that the implementation of a dining philosopher process will necessarily contain a summation expression with two symmetric branches written out in full: the branches cannot be abstracted as a common "try to get one chopstick and then the other" operation. This duplication of similar code quickly leads to errors. (Even in small programs: an earlier version of these notes contained a bug in the dining philosophers example that was introduced by making a change in one branch and forgetting to change the other!) Reppy's solution is to analyze high-level communication actions as compositions of more primitive actions. His slogan for the resulting programming style is *higher-order concurrency.*

The key concept in Reppy's scheme is an *event*: an input or output action that has been specified (which channel is to be used, what values are to be transmitted, and what to do next) but to which no committment has yet been made. In the present, somewhat simplified, scheme, we consider only input events.[1]

Events can only occur on event channels, never on ordinary channels. The basic operation for creating an event is

```
(=> [c v])
```

where `c` is an input event channel (created with `newEventChan` or `buildEventChan`) and `v` is a continuation value, usually of the form `\p = ...`. The result of this expression is a value `ev` of type `Event`. Events are just another form of value, so we also allow variables ranging over events as well as channels carrying events, functions returning events, etc. Intuitively, `ev` can be thought of as a *description* of an act of communication on the event channel `c`. To actually perform the communication, we *synchronize* on `ev` using the operation `sync`:

```
sync!ev
```

[1]Another difference is that Reppy's events are fully integrated with the basic mechanisms of channel creation and I/O on channels. All channels in CML are event channels. See [Rep95] for further discussion of this point.

The choice construct of π-calculus is generalized to events using the operation `$`. If `ev1` and `ev2` are event values, then

```
$[ev1 ev2]
```

is an event value describing the action of *either* performing `ev1` or performing `ev2`.

10.4.1 Exercise [Recommended]: *Rewrite your solution to Exercise 10.1.1 so that the case where a philosopher picks up her left-hand chopstick and then tries to get the one on the right is implemented by exactly the same code as the case where she first gets the right-hand chopstick and then tries on the left. (I.e., write a single definition of the chopstick-grabbing behavior, parameterized on the channels for the right and left chopsticks, and instantiate this definition twice to give the branches of the choice.) [Solution on page 99.]*

10.5 Implementing Events

Our principal reason for separating ordinary channels from event channels is that the latter can actually be implemented in terms of the former. Indeed, the primitives described in the previous sections are actually implemented as a library module in Pict.

The basic abstraction used in the implementation of events is a `Lock`:

```
type Lock = ^[!Bool]
```

A lock is queried by sending it a boolean result channel. The first time this happens, it sends back the value `true` over the result channel; for all future requests, it returns the value `false`. In other words, the first process that tries to "obtain" a lock by querying its value receives `true`, indicating that it has succeeded; all later attempts fail.

```
def newLock () =
 (new lock:Lock
  run lock?[r] = (r!true | lock?*[r] = r!false)
  lock)
```

In the simplest implementation, an input event channel connected to the ordinary channel `c` can be represented by `c` itself. So the type of input event channels is

```
type (EChan X) = ^X
```

and the `buildEventChan` function can be implemented like this:

```
def buildEventChan (|X| c:^X) : (EChan X) = c
```

A receive operation on an input event channel may in general be enclosed in a choice. In this case, there is a race going on between all of the receivers in a choice event. The first one that is ready to proceed is allowed to do so; the rest become garbage. This race is mediated by a single lock, which is created by the `sync` operation and distributed to all the individual `receiveEvents`. In other words, an event is triggered by passing it a lock.

```
type Event = !Lock
```

The `sync` operation takes an event and triggers it with a freshly created lock.

```
def sync e = e!(newLock)
```

The $ operation takes two events and creates a new event that, when it receives its lock, transmits it to both component events.

```
    def $ (e1:Event e2:Event) : Event =
      \lock =
        (e1!lock | e2!lock)
```

Finally, the `receiveEvent` operation takes the "receiver end" of an event channel `c` and a continuation `receiver` that is waiting to accept a data value over `c`. It first waits for a lock; then it waits for a value to arrive on `c`. Once a value `v` arrives, it checks to see whether the lock has already been taken — i.e., whether it has won or lost the race. If it wins (gets the value `true` from the lock), it sends `v` along to the receiver. If it loses, it re-sends `v` along `c`, so that another receiver has the chance to pick it up.

```
    def receiveEvent (|X| c:(EChan X) receiver:!X) : Event =
      \lock =
        c?v =
        if (lock) then
          receiver!v
        else
          c!v
```

Let us test this implementation by creating two events, both reading from the channel `c`; one prints the received value twice, the other prints it four times.

```
    new csend:^Int
    val c = (buildEventChan csend)
    val ev1 = (receiveEvent c \i = (printi!i | printi!i))
    val ev2 = (receiveEvent c \i = (printi!i | printi!i | printi!i | printi!i))
    val ev3 = ($ ev1 ev2)
    run (csend!5 | sync!ev3)
```

```
5
5
```

10.5.1 Exercise [Moderate]: *Extend this implementation of events to provide* synchronous *communication, so that a sender on an event channel is notified when the communication has taken place.*

10.5.2 Exercise [Difficult]: *Suppose that a priority queue object* `q` *has been implemented using event channels as in Section 10.3, and that a client process repeatedly invokes the methods* `add`, `first`, *and* `isEmpty` *of* `q` *without invoking* `remove`. *Will such a program behave correctly if it is allowed to run for a long time? If not, how can the implementation of events be corrected? [Hint on page 101.]*

Chapter 11

Record Manipulation

$$\frac{? \vdash \mathtt{v} \in \mathtt{T} \qquad ? \vdash \mathtt{v} \in \mathtt{U} \qquad ? \vdash (\mathtt{T\ with\ l:U}) \in \mathtt{Type}}{? \vdash (\mathtt{v:T\ with\ l=v}) \in (\mathtt{T\ with\ l:U})} \tag{V-With}$$

Note, in the second rule, the precondition that the type `T` of `v` must be a subtype of the empty record type — i.e., it must be some record type. This avoids deriving nonsensical statements like $? \vdash$ `(5 with l=6)` $\in$ `(Int with l:Int)`.

Since we have chosen a primitive formulation of records, building multi-field records by adding single fields to the empty record, we need several rules to capture the relations between different record types. First, the order of extensions does not matter, so long as they are for different labels:

$$\frac{\mathtt{l} \neq \mathtt{j}}{? \vdash ((\mathtt{S\ with\ l:U})\ \mathtt{with\ j:V}) < ((\mathtt{S\ with\ j:V})\ \mathtt{with\ l:U})} \tag{S-Swap}$$

On the other hand, if a record is extended by the same label twice, only the second extension has any effect. In other words, duplicate extensions may freely be dropped

$$? \vdash ((\mathtt{S\ with\ l:U})\ \mathtt{with\ l:V}) < (\mathtt{S\ with\ l:V}) \tag{S-Dup1}$$

or added:

$$? \vdash (\mathtt{S\ with\ l:V}) < ((\mathtt{S\ with\ l:U})\ \mathtt{with\ l:V}) \tag{S-Dup2}$$

Since the only thing that can be done with a record is to add fields or extract existing fields by pattern matching, the empty record can be regarded as a supertype of any record type: Finally, given two record types `S` and `T` such that `S` $<$ `T`, we may extend both `S` and `T` by the *same* field `l` while preserving subtyping, so long as the types assigned to `l` themselves fall in the subtype relation:

$$\frac{? \vdash \mathtt{S} < \mathtt{T} \qquad ? \vdash \mathtt{U} < \mathtt{V}}{? \vdash (\mathtt{S\ with\ l:U}) < (\mathtt{T\ with\ l:V})} \tag{S-With}$$

Chapter 12

Recursive Data Structures

Like most programming languages, Pict offers the capability to build and manipulate recursive data structures like lists and trees. For example, consider integer lists. Since the tail of such a list is also a list, its type is something like

```
type IntList = (record
  empty: ![!Bool]
  head:  ![!Int]
  tail:  ![!IntList]
)
```

in which the type being defined appears on both the left and the right of the `=`. Such *recursive types* have received considerable attention in the literature (e.g. [CC91, AC93, Car89, BM92, Ghe93, MPS86, Wan87, TW93, AP90, KPS93]), and many different technical treatments have been proposed. Because the type system of Pict is already rather complex and recursive types tend to be used only in small sections of code, we have chosen one of the simplest alternatives, where the "folding" and "unfolding" of the recursion must be managed explicitly by the programmer.

12.1 Recursive Types

Let us begin with the simplest possible example where recursive types are required. Suppose we want to write a program in which a channel is used to send itself:

```
... c!c ...
```

What must the type of `c` be, in order for this program to be well typed? Since it appears on the left of a `!`, it must certainly begin with a `!` or a `^`. Moreover, the type of the values carried along `c` is the same as the type of `c` itself, so the initial `!` or `^` should be applied to some type beginning with another `!` or `^`, which (by the same argument) should be applied to a type beginning with a `!` or `^`, and so on. One valid type for `c` would therefore be something like:

```
c : ^(^(^(^(^(^(...))))))
```

As a finite abbreviation for this infinitely long type expression, we introduce the notation

```
(rec X where X = ^X)
```

which can be read intuitively as "a type of the form `^X`, where `X` stands for the type itself." However, the expression `c!c` now presents a slight problem. Up to this point, we have required that the left-hand part of an output expression should have a channel type; but `c` does not have a channel type: its type begins with `rec`, not `!` or `^`. There are two basic approaches to resolving this difficulty:

1. We can refine the rules that the compiler uses for typechecking, so that when a value with a type like `(rec X where X = T)` appears in a position where a channel type is required, the recursive type is automatically *unfolded* to expose the structure of its body.

   Here, the type `(rec X where X = ^X)` would be unfolded to `^(rec X where X = ^X)`, which does indeed begin with a channel type constructor. Moreover, the type carried on the channel is `(rec X where X = ^X)`, which exactly matches the type of the value `c` that we are trying to transmit, and so the whole output expression `c!c` typechecks successfully.

2. We can add an explicit coercion `unfold` that transforms a value with a recursive type into one with a nonrecursive type by performing one manual unfolding step.

   In the present example, if `c` has the type `(rec X where X = ^X)`, then `(unfold c)` has the type `^(rec X where X = ^X)`.

The first alternative is clearly smoother for the programmer: the recursive types are always unfolded out of the way by the compiler, and the programmer never has to explicitly do anything about them. However, this kind of automatic unfolding can have serious consequences for the efficiency of typechecking.[1] Moreover, the theoretical foundations of type systems combining this automatic unfolding with the other features present in Pict (e.g. subtyping and polymorphism) are only beginning to receive satisfactory treatment (cf. [AC93]). We have therefore chosen the second alternative for Pict.

Actually, both unfolding and folding annotations are needed. To see why, observe that, since the `new` constructor requires the type of each fresh channel to begin with a `^`, the declaration

```
new c: (rec X where X = ^X)
```

will actually not be accepted by the compiler. We must declare it first with a channel type:

```
new c´: ^ (rec X where X = ^X)
```

To give `c` the type `(rec X where X = ^X)`, we must essentially perform the opposite of the unfolding operation above. For this purpose, we introduce the annotation `fold`:

```
val c = (rec : (rec X where X = ^X) c´)
```

The type annotation between the keyword `fold` and the value `c´` tells the typechecker which recursive type we unfolded to obtain the current type of `c´`. Since, in general, the type of `c´` might contain several `Rec` expressions (or possible even none), there is no way for the typechecker to "guess" this information. Having obtained the value `c` with the desired type, we can now write a program that sends `c` along itself like this:

```
run c´!c
```

[1]This is a subtle point. In languages with ML-style type inference, adding recursive types can actually improve the performance of typecheckers, since it has the effect of *forcing* the implementor to use a graph unification algorithm, which can take advantage of implicit sharing between type expressions. However, in a language with type operators, automatic unfolding of recursive types may lead to an explosion in the number of cases that must be considered by the subtype-checking algorithm.

Finally, all of the above examples can be made more readable by introducing a type abbreviation for the recursive type:

```
type Omega = (rec X where X = ^X)
new c´: ^Omega
val c = (rec:Omega c´)
run c´!c
```

Formally:

$$\frac{? \vdash \mathtt{S} \rightsquigarrow \mathtt{T} \qquad ? \vdash \mathtt{v} \in \mathtt{T}}{? \vdash \mathtt{(rec:S\ v)} \in \mathtt{S}} \tag{V-Rec}$$

$$\frac{\mathtt{S} \rightsquigarrow \mathtt{T} \qquad ? \vdash \mathtt{p} \in \mathtt{T} \Rightarrow \Delta}{? \vdash \mathtt{(rec:S\ p)} \in \mathtt{S} \Rightarrow \Delta} \tag{P-Rec}$$

12.1.1 Exercise [Easy]: *The following untyped Pict program generates a typechecking error because the typechecker tries to instantiate a unification variable in the type of* `c` *to a type expression containing the very same unification variable.*

```
def c[p] = p?x = c![x]
```

```
example.pi:1.22:
value´s type does not match type of channel
since [Any5] is not an instance or subtype of [?Any5]
since the types of field 1 are not compatible since Any5 is not a subtype of
   ?Any5
since the occur check failed when instantiating Any5 with ?Any5:
Cyclic instantiation
```

Add type annotations to this program so that it compiles. [Solution on page 101.]

12.2 Lists

Now let us return to the example of lists and see how to implement a set of polymorphic constructors for simple list cell objects using recursive types. Pict's standard library includes a similar package.

We adopt the view of Lisp and ML: a list is either a special terminating object `nil` or it is a "cons object" containing a "head element" and a pointer to the "tail" of the list. Every list object supports an `empty` message, which returns `true` if it is `nil` and `false` if it is a cons object, plus two messages `head` and `tail` for interrogating cons objects. (These messages are also part of the interface of empty lists, since it is convenient to give them the same type as nonempty lists, but attempting to send these two messages to an empty list will simply result in deadlock.)

```
type (List X) =
  (rec L where L =
    (record
      empty: ![!Bool]
      head:  ![!X]
      tail:  ![!L]
    ))
```

The list constructor `nil` is very simple. Its only parameter is a type argument `X`, which determines the type `List X` of its result. Its body builds a simple object whose methods are all abstractions: the method `empty` immediately returns `true`, while the methods `head` and `tail` do nothing (and return nothing). The record of methods is enclosed in a `fold` annotation to transform its type from `(record ...)` to the recursive type `(List X)`.

```
def nil (|X|) : (List X) =
 (rec
  (record
    empty = \() = true
    head = \[r] = ()
    tail = \[r] = ()
  ))
```

The list constructor `cons` is almost equally simple: it takes a head `h` and a tail `t`, where `t` is a list with elements of type `X` and `h` is an element of type `X`. As for `nil`, the record of methods is prefixed with a `fold` to give it the proper typing.

```
def cons (|X| h:X t:(List X)) : (List X) =
 (rec
  (record
    empty = \() = false
    head = \() = h
    tail = \() = t
  ))
```

We can now build lists as usual with `nil` and `cons`

```
val (rec:(List Int) l) = (cons 1 (cons 2 (cons 3 (nil))))
```

and interrogate them by sending messages:

```
run ((prInt ((l.head))); ())
```

```
1
```

Note that the `unfold` is needed here to coerce the recursive type `List X` to a record type, so that the field projection operator can be used. This is the counterpart of the `fold` annotation in the definitions of `nil` and `cons`.

12.2.1 Exercise [**Moderate**]:

1. *Implement a* `reverse` *operation that, given a list, yields a new list with the elements in the opposite order.*

2. *Implement versions of the* `nil` *and* `cons` *constructors yielding list objects that respond to* `reverse` *requests.*

Compare these two implementations.

12.2.2 Exercise [**Moderate**]: *Construct an implementation of* mutable list *objects, with methods* `setHead` *and* `setTail` *for modifying the head and tail, in addition to the three methods of* `List`*:*

```
type (MList X) =
  (rec L where L =
    (record
      empty:   ![!Bool]
      head:    ![!X]
      tail:    ![!L]
      setHead: ![X Sig]
      setTail: ![L Sig]
    ))
```

Chapter 13

Higher-Order Polymorphism

In previous chapters, we have used *parameterized type declarations* like

```
type (Ref X) = (record
  set:![X Sig]
  get:![!X]
)
```

in which what is defined is not a single type, but a family of types, one for each value of `X`. In types, we allowed expressions like `(Ref Int)` and `(Ref Bool)`. But not *all* such type expressions are meaningful. We surely do not want to allow expressions like `(Ref Ref)` or `(Bool Bool)`. To eliminate such nonsensical type expressions, we need to add another layer of consistency checking to the type system that performs "typechecking of type expressions." In this chapter, we develop the necessary machinery.

The treatment here is intended only to illustrate the use of higher-order polymorphism in programming, not to discuss all aspects of the type system in detail or to do justice to the theoretical issues involved. Background reading can be found in [Car91, PDM89, Car90, PS96, HP95, Com94].

13.1 Overview

If we are going to do "typechecking of type expressions," then clearly we need to introduce a language of "types of type expressions." To keep the terminology from getting out of hand, it helps to give a different name to this level of classification: we call them *kinds*. Figure 14.1 (adapted from [Car91]) should help to see the relation between the levels of values, types, and kinds. The level of values contains familiar entities like `5`, `true`, `ref[5]`, the tuple `[5,5]`, and the channel `c`. The level of types contains *proper types* like `Int`, `Bool`, `(Ref Int)`, `[Int Int]`, and `^Int`, as well as *type operators* like `Ref`, `List`, and `funX = ^[X X]`. The proper types classify values, in the sense that entities at the level of values may *inhabit* proper types: `5` inhabits `Int`, etc. In the same sense, kinds classify types: all the proper types inhabit the kind `Type`; type operators accepting one proper type parameter and yielding a proper type (like `Ref`) inhabit the kind `(Type->Type)`; type operators taking two proper type arguments and yielding a proper type inhabit the kind `(Type->(Type->Type))`; and so on.

Just as, in Section 5.3, we gave a set of rules defining which values inhabit which types, we must now give rules defining when type expressions inhabit kinds. These appear in detail in Section 14.2.3 below. Most of them look something like this:

$$\frac{? \vdash \texttt{T} \in \texttt{Type}}{? \vdash \texttt{\^{}T} \in \texttt{Type}} \qquad (\text{K-CHAN})$$

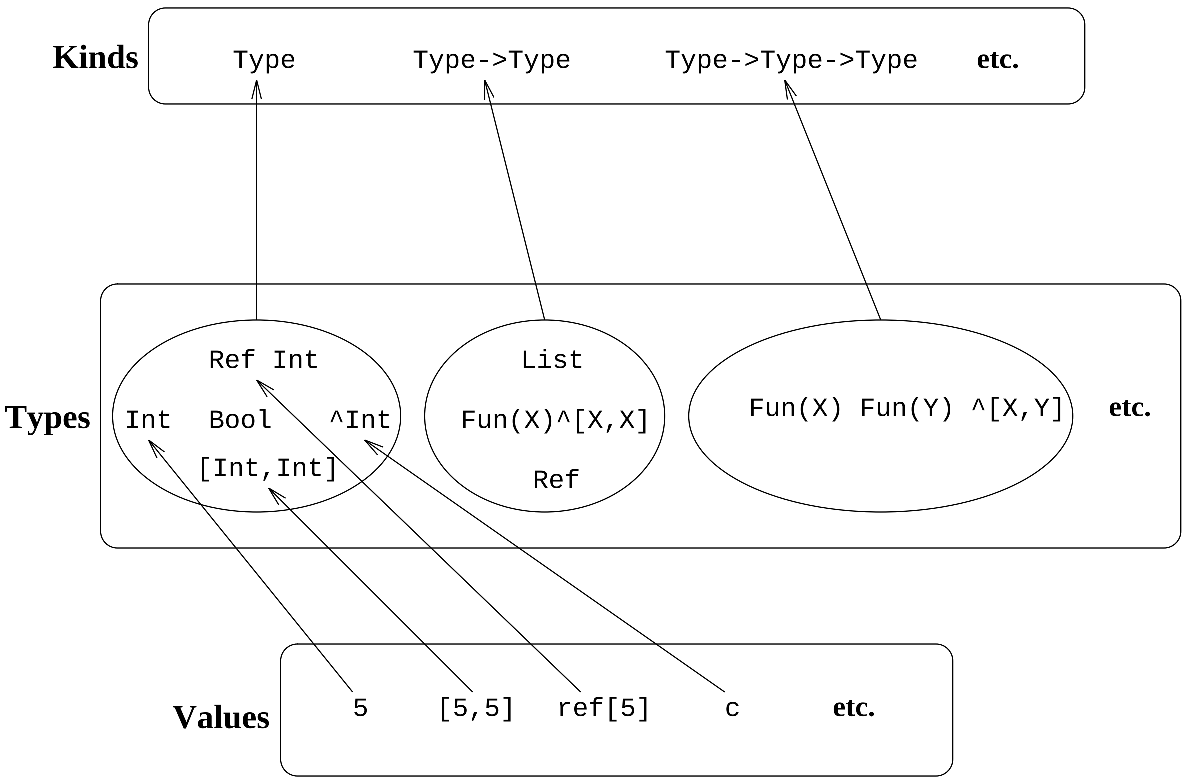

Figure 13.1: The Universes of Kinds, Types, and Values

That is, if the type expression `T` is a proper type, then so is `^T`.

Three of the kinding rules deserve special attention. First, we allow type variables of arbitrary kind. In the typing context, each type variable is associated with both its upper bound (c.f. Chapter 13) and its kind (which must also be the kind of its upper bound). The kind of a type variable, then, is just whatever kind is declared for it in the prevailing context:

$$\frac{?_1 \vdash \mathtt{T} \in \mathtt{K}}{?_1, \mathtt{X{<}T}, ?_2 \vdash \mathtt{X} \in \mathtt{K}} \qquad \text{(K-TVar)}$$

The well-formedness of type expressions like `Ref Int`, in which a type operator is applied to an argument, is controlled by this rule, familiar from the simply typed λ-calculus:

$$\frac{? \vdash \mathtt{S} \in \mathtt{K}_1\mathtt{->K}_2 \qquad ? \vdash \mathtt{T} \in \mathtt{K}_1}{? \vdash \mathtt{(S\ T)} \in \mathtt{K}_2} \qquad \text{(K-Arrow-E)}$$

The rule for kinding of type operators themselves also comes from the λ-calculus.

$$\frac{?, \mathtt{X{<}Top{:}K}_1 \vdash \mathtt{T} \in \mathtt{K}_2}{? \vdash \mathtt{fun\ X{:}K}_1\ \mathtt{=\ T} \in \mathtt{K}_1\mathtt{->K}_2} \qquad \text{(K-Arrow-I)}$$

Formally, a type operator is a type expression (its *body*) prefixed with an *abstraction* of the form $\mathtt{Fun(X{:}K_1)}$. If the body has kind $\mathtt{K}_2$, then the whole operator has kind $\mathtt{K_1\text{->}K_2}$.

Type declarations like

```
type (Ref X) = (record set:![X Sig] get:![!X])
```

with parameters on the left-hand side of the `=` are actually just abbreviations for simple definitions like:

```
type Ref = fun X = (record set:![X Sig] get:![!X])
```

In general,

```
type (A X:K) = T
```

stands for:

```
type A = fun X:K = T
```

As a small example of programming with type operators, recall our definition of list objects from Section 12.2:

```
type (List X) =
  (rec L where L =
    (record
      empty: ![!Bool]
      head:  ![!X]
      tail:  ![!L]
    ))
```

Although we had no trouble defining constructors with these types, the resulting lists were a little inconvenient to use, since it was necessary to use an `unfold` annotation to remove the outer `Rec` constructor before sending any message to a list. It might be more convenient to perform this outer unfolding once and for all, defining the type of lists as

```
type (List X) =
  (record
    empty: ![!Bool]
    head:  ![!X]
    tail:  ![!(rec L where L = (record empty:![!Bool] head:![!X] tail:![!L]))]
  )
```

so that at least `empty` and `head` could be used without `unfold`. But this definition is a bit hard to read; for example, the type of `head` actually appears twice. An easier way to obtain the same effect is to define an auxiliary operator

```
type (ListA L X) =
  (record
    empty: ![!Bool]
    head:  ![!X]
    tail:  ![!L]
  )
```

and then write

```
type (List X) = (ListA (rec L where L = (ListA L X)) X)
```

to recover the original definition of `List`.

13.2 Typing Rules

This final section summarizes the inference rules defining the higher-order part of Pict's type system. The rules for the whole language can be found in the User Manual.

13.2.1 Preliminaries

In the fully typed Pict language, every type binder has a kind declaration. The concrete syntax allows kind declarations to be omitted, replacing missing ones by `Type`.

In many places in the definitions below, we need premises to ensure that a proper kinding discipline is respected (e.g. $\texttt{S} < \texttt{Top(K)}$ only when $\texttt{S} \in \texttt{K}$, etc.). But for readability, these are kept to a minimum: we maintain the invariant that if the conclusion of a subtyping derivation is a well-formed statement (having a well-formed context, mentioning only well-kinded types, etc.), then conclusions of all of its subderivations are similarly well formed.

13.2.2 Context Well-formedness

Since contexts contain type expressions, we need to begin by introducing rules for checking that all the type expressions in a given context are well formed.

$$\vdash \bullet\ \mathit{ok} \qquad \text{(C-Empty)}$$

$$\frac{? \vdash \texttt{T} \in \texttt{K} \qquad \texttt{X} \notin \mathit{dom}(?)}{\vdash ?,\texttt{X<T}\ \mathit{ok}} \qquad \text{(C-TVar)}$$

$$\frac{? \vdash \texttt{T} \in \texttt{Type} \qquad \texttt{x} \notin \mathit{dom}(?)}{\vdash ?,\texttt{x:T}\ \mathit{ok}} \qquad \text{(C-Var)}$$

13.2.3 Kinding

As we discussed above, the kinding rules assign kinds to type expressions.

$$\frac{?_1 \vdash \texttt{T} \in \texttt{K}}{?_1,\texttt{X<T},?_2 \vdash \texttt{X} \in \texttt{K}} \qquad \text{(K-TVar)}$$

$$\frac{?,\texttt{X<Top:K}_1 \vdash \texttt{T} \in \texttt{K}_2}{? \vdash \texttt{fun X:K}_1\ \texttt{=}\ \texttt{T} \in \texttt{K}_1\texttt{->K}_2} \qquad \text{(K-Arrow-I)}$$

$$\frac{? \vdash \texttt{S} \in \texttt{K}_1\texttt{->K}_2 \qquad ? \vdash \texttt{T} \in \texttt{K}_1}{? \vdash \texttt{(S T)} \in \texttt{K}_2} \qquad \text{(K-Arrow-E)}$$

$$? \vdash \texttt{Top:K} \in \texttt{K} \qquad \text{(K-Top)}$$

$$\frac{? \vdash \texttt{T} \in \texttt{Type}}{? \vdash \texttt{\^{}T} \in \texttt{Type}} \qquad \text{(K-Chan)}$$

$$\frac{? \vdash \texttt{T} \in \texttt{Type}}{? \vdash \texttt{?T} \in \texttt{Type}} \qquad \text{(K-IChan)}$$

$$\frac{? \vdash \texttt{T} \in \texttt{Type}}{? \vdash \texttt{!T} \in \texttt{Type}} \qquad \text{(K-OChan)}$$

$$\frac{?, \texttt{X}_1\texttt{<Top:K}_1, \ldots, \texttt{X}_n\texttt{<Top:K}_n \vdash \texttt{T}_i \in \texttt{K}_i \text{ for each } i}{? \vdash \texttt{(rec X}_j \texttt{ where X}_1\texttt{:K}_1\texttt{=T}_1 \ \ldots\ \texttt{X}_n\texttt{:K}_1\texttt{=T}_n\texttt{)} \in \texttt{K}_j} \qquad \text{(K-Rec)}$$

$$\frac{? \vdash \texttt{T}_1 \in \texttt{Type} \qquad \ldots \qquad ? \vdash \texttt{T}_n \in \texttt{Type}}{? \vdash \texttt{[T}_1 \ldots \texttt{T}_n\texttt{]} \in \texttt{Type}} \qquad \text{(K-Tuple)}$$

13.2.4 Subtyping

We also need to extend the subtyping relation to the new type constructors, `Fun` and application.

$$\frac{?, \texttt{X<Top:K} \vdash \texttt{S} < \texttt{T}}{? \vdash \texttt{fun X:K = S} < \texttt{fun X:K = T}} \qquad \text{(S-Abs)}$$

$$\frac{? \vdash \texttt{S}_1 < \texttt{T}_1 \qquad ? \vdash \texttt{S}_2 < \texttt{T}_2 \qquad ? \vdash \texttt{T}_2 < \texttt{S}_2}{? \vdash \texttt{(S}_1\ \texttt{S}_2\texttt{)} < \texttt{(T}_1\ \texttt{T}_2\texttt{)}} \qquad \text{(S-App)}$$

These rules constitute the simplest possible extension of subtyping to a higher-order setting: the subtype relation on kind `Type` is lifted pointwise to higher-kinds.

The fact that type expressions may contain applications like `(fun X =![X X] T)` requires another extension of the subtype relation, to express the fact that we want to consider `(fun X = ![X X] T)` to be exactly the same type as `![T T]`. Whenever `T` can be obtained from `S` by a reducing such applications, we make `S` $<$ `T`. In general, `S` and `T` are each subtypes of the other if they are *convertible* in the sense of the λ-calculus:

$$\frac{\texttt{S} =_{\beta\top} \texttt{T}}{? \vdash \texttt{S} < \texttt{T}} \qquad \text{(S-Conv)}$$

Formally, single-step $\beta\top$-reduction, written $\longrightarrow_{\beta\top}$, is the closure (under all of the type constructors) of the following rules:

$$\texttt{(fun X:K = T) S} \longrightarrow_{\beta\top} \texttt{[S/X]T} \qquad \text{(R-Beta)}$$

$$\texttt{Top:(K}_1\texttt{->K}_2\texttt{) S} \longrightarrow_{\beta\top} \texttt{Top:K}_2 \qquad \text{(R-Top)}$$

(The second clause is a technical refinement, to make sure that certain boundary cases like `Top:Type` $<$ `Top:(Type->Type) Int` are included in the subtype relation. C.f. [PS96, Com94].) *Conversion*, written $=_{\beta\top}$, is the transitive, reflexive, and symmetric closure of $\longrightarrow_{\beta\top}$.

Besides these rules for the new constructors, we need more general forms of some rules that have already been introduced, taking kinding into account.

$$\frac{? \vdash \mathtt{S} \in \mathtt{K}}{? \vdash \mathtt{S} < \mathtt{Top:K}} \qquad \text{(S-Top)}$$

$$\frac{? \vdash \mathtt{S}_1 < \mathtt{T}_1 \qquad \ldots \qquad ? \vdash \mathtt{S}_n < \mathtt{T}_n}{? \vdash \mathtt{[S}_1 \ldots \mathtt{S}_n\mathtt{]} < \mathtt{[T}_1 \ldots \mathtt{T}_n\mathtt{]}} \qquad \text{(S-Tuple)}$$

$$\frac{?,\ \mathtt{Y}_1\mathtt{<Top:K}_1,\ \ldots,\ \mathtt{Y}_n\mathtt{<Top:K}_n,\ \mathtt{X}_1\mathtt{<Y}_1,\ \ldots,\ \mathtt{X}_n\mathtt{<Y}_n \vdash \mathtt{S}_i < \mathtt{T}_i \text{ for each } i}{? \vdash \mathtt{(rec\ X}_j\ \mathtt{where\ X}_1\mathtt{:K}_1\mathtt{=S}_1 \ldots \mathtt{X}_n\mathtt{:K}_1\mathtt{=S}_n\mathtt{)} < \mathtt{(rec\ Y}_j\ \mathtt{where\ Y}_1\mathtt{:K}_1\mathtt{=T}_1 \ldots \mathtt{Y}_n\mathtt{:K}_1\mathtt{=T}_n\mathtt{)}} \qquad \text{(S-Rec)}$$

13.2.5 Values and Patterns

At the level of values and processes, we need to refine a few rules that have been introduced before. The V-Tuple rule needs to be extended with kinding annotations for bound type variables in polymorphic patterns:

$$\frac{? \vdash \mathtt{v}_1 \in \mathtt{T}_1 \qquad \ldots \qquad ? \vdash \mathtt{v}_n \in \mathtt{T}_n}{? \vdash \mathtt{[v}_1 \ldots \mathtt{v}_n\mathtt{]} \in \mathtt{[T}_1 \ldots \mathtt{T}_n\mathtt{]}} \qquad \text{(V-Tuple)}$$

This refinement in the typing of tuples yields a new typing rule for applications:

$$\frac{\begin{array}{c} ? \vdash \mathtt{v} \in \mathtt{![|A}_1\mathtt{<U}_1\ \ldots\ \mathtt{A}_m\mathtt{<U}_m\mathtt{|}\ \mathtt{T}_1\ \ldots\ \mathtt{T}_n\ \mathtt{!T]} \qquad \{\mathtt{A}_1, \ldots, \mathtt{A}_m\} \cap FV(\mathtt{T}) = \emptyset \\ \text{for each } i \qquad ? \vdash \mathtt{S}_i \in \mathtt{K}_i \qquad ? \vdash \mathtt{U}_i \in \mathtt{K}_i \qquad ? \vdash \mathtt{S}_i < \mathtt{U}_i \\ \text{for each } j \qquad ? \vdash \mathtt{v}_j \in \{\mathtt{A}_1, \ldots, \mathtt{A}_m \mapsto \mathtt{S}_1, \ldots, \mathtt{S}_m\}\mathtt{T}_j \end{array}}{? \vdash \mathtt{(v\ |S}_1\ \ldots\ \mathtt{S}_m\mathtt{|}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_n\mathtt{)} \in \mathtt{T}} \qquad \text{(V-App)}$$

This, in turn, leads to the general forms of the rules for infix application: *[V-InfId?]* *[V-InfV?]*

Appendix A

Solutions to Selected Exercises

Solution to 1.10.1:

```
def notB[b:Boolean c:Boolean] = c?*[t f] = b![f t]

{- Test -}
new b:Boolean  new c:Boolean
run ( ff![b]
    | notB![b c]
    | test![c])
```

```
It´s true
```

Solution to 1.10.2:

```
def andB[b1:Boolean b2:Boolean c:Boolean] =
  c?*[t f] =
  (new x:^[]
   ( b1![x f]
   | x?[] = b2![t f]))

{- Test -}
new b1:Boolean  new b2:Boolean  new c:Boolean
run ( tt![b1] | tt![b2]
    | andB![b1 b2 c]
    | test![c])
```

```
It´s true
```

Solution to 1.10.3:

```
def notB[b:Boolean res:^Boolean] =
 (new c:Boolean
  run c?*[t f] = b![f t]
  res!c)

def andB[b1:Boolean b2:Boolean res:^Boolean] =
 (new c:Boolean
  run res!c
  c?*[t f] =
    (new x:^[]
        ( b1![x f]
        | x?[] = b2![t f])))

    {- Test -}
    new b1:Boolean  new b2:Boolean  new res:^Boolean
    run ( tt![b1] | tt![b2]
        | andB![b1 b2 res]
        | res?c = test![c])
```

```
It´s true
```

Solution to 4.7.1:

```
    import "tester"

    {- Create a channel to use as a global semaphore for the workers.
       We maintain the invariant that when no worker is working, there is
       exactly one running process of the form lock![].  To grab the
       semaphore, read from lock to use up this process.  To give up the
       semaphore, write on lock. -}
    new lock:^[]
    run lock![]

    {- Now submit just signals acknowledgement immediately to the client,
       but waits for the semaphore before triggering the worker.  The
       channel lock itself is passed to the worker as its completion
       channel. -}
    def submit[wor:Worker pri:Int c:![]] =
      ( c![]
      | lock?[] = wor![lock])

    run test![submit]
```

```
Test worker starting
Test worker finishing
Worker 1 starting
Worker 1 working
Worker 1 working
Worker 1 finished
Worker 2 starting
Worker 2 working
Worker 2 working
Worker 2 finished
```

Solution to 4.7.2:

```
    import "tester"

    {- A global lock for the whole scheduler.  The value carried in the lock is
       the number of worker processes currently awaiting activation. -}
    new lock: ^Int
    run lock!0

    {- A queue implementing a bag holding all the waiting worker processes
       and their priorities -}
    new workers: ^[Worker Int]
```

```
{- Temporary channels for numeric and boolean calculations -}
new br: ^Bool
new ir: ^Int

{- A temporary bag of workers, used while we are scanning -}
new temp:^[Worker Int]

{- Put all the workers from temp back into the main bag of workers -}
def restoreWorkers[] = temp?v = (workers!v | restoreWorkers![])

{- Wait for at least one worker to become available in the bag
   "workers."  Then select the worker from the workers bag with the
   highest priority.  We read all the elements currently in the bag,
   keeping track of the one with the highest priority seen so far.
   When we´ve seen them all, we put all the others back onto the workers
   bag (they are stored in the temporary bag temp in the meantime). -}
def startNextWorker[] =
  ({- Find the worker with the highest priority -}
   def scan[n:Int wor:Worker pri:Int count:Int] =
         ( >>![n 1 br]
         | br?b =
            if b then
              workers?[wor´ pri´] =
               ( >>![pri pri´ br]
               | br?b =
                   if b then
                     ( temp![wor´ pri´]
                     | -![n 1 ir]
                     | ir?i =
                         scan![i wor pri count] )
                   else
                     ( temp![wor pri]
                     | -![n 1 ir]
                     | ir?i =
                         scan![i wor´ pri´ count] )
               )
             else
               ( wor![startNextWorker]
               | restoreWorkers![]
               | -![count 1 lock] )
         )
   workers?[wor pri] =
     lock?count =
       scan![count wor pri count]
  )

{- Start a single copy of startNextWorker executing -}
run startNextWorker![]

{- When a new process arrives, we add it to the bag of waiting workers.
   Also, if no worker is working at the moment, send a hint that another
   one should be started. -}
```

```
    def submit[wor:![![]] pri:Int c:![]] =
      lock?count =
        ( c![] | workers![wor pri] | +![count 1 lock] )

    {- Test what we´ve done -}
    run test![submit]
```

```
Test worker starting
Test worker finishing
Worker 2 starting
Worker 2 working
Worker 2 working
Worker 2 finished
Worker 1 starting
Worker 1 working
Worker 1 working
Worker 1 finished
```

Solution to 6.2.3.2:

```
    def fib[n:Int r:!Int] =
      if (|| (== n 0) (== n 1)) then
        r!1
      else
        r!(+ (fib (- n 1)) (fib (- n 2)))

    run printi!(fib 7)
```

```
21
```

Solution to 9.2.1: [This example is based on an observation by Dilip Sequeira.] The explicitly typed program

```
    new ch : ^Int
    def g [a:Char b:Int] = (ch!a | ch!b)
```

typechecks, as does its type-erasure

```
    new ch
    def g [a b] = (ch!a | ch!b)
```

but the following partially erased version fails:

```
    new ch
    def g [a:Char b:Int] = (ch!a | ch!b)
```

```
example.pi:10.37:
value´s type does not match type of channel
since Int is not an instance or subtype of Char
since Int and Char do not match
```

Solution to 10.1.1: Begin by defining a helpful operation for printing messages:

```
def mesg (name m) = (prNL (+$ > (intString name) " " m))
```

A chopstick is essentially a semaphore: it has a `grab` channel and a `rel` channel, and it communicates over each of these in turn. To grab a chopstick, we perform a receive on its `grab` channel; then (since the send on `grab` was asynchronous and the chopstick needs to be informed that it has been grabbed), we signal on the reply channel provided for this purpose.

```
type Chopstick =
  (record
    grab: (EChan [Sig])
    rel: (EChan [Sig])
  )

val maxcycles = 3

def makechopstick () : Chopstick =
 (new loc
  val [grabSend grab] = (newEventChan)
  val [relSend rel] = (newEventChan)
  def cycle x = if (<> x 0) then ((grabSend); (relSend); cycle!(- x 1)) else ()
  run cycle!maxcycles
  (record grab=grab rel=rel))
```

(The integer parameter to `cycle` makes the chopstick stop responding to requests after a certain number have been handled. This allows the running process to terminate gracefully instead of needing to be killed.)

A philosopher process takes a name (an integer) and two chopsticks as arguments.

```
def delay () = (for 1 (randomInt 20) \(i) = [])

def phil [name:Int l:Chopstick r:Chopstick] =
 (def dine[] =
   ((mesg name "thinking");
    (delay);
    sync!($
      (=> (l.grab) \[c] =
        (run c![]
         (mesg name "got left chopstick");
         sync!($
           (=> (r.grab) \[c] =
             (run c![]
              (mesg name "eating and giving up both chopsticks");
              ( sync!(=> (l.rel) \() = [])
              | sync!(=> (r.rel) \() = [])
              | dine![] )))
            (=> (l.rel) \[c] =
             (run c![]
              (mesg name "giving up left chopstick");
              dine![])))))
      (=> (r.grab) \[c] =
        (run c![]
         (mesg name "got right chopstick");
```

```
             sync!($
               (=> (l.grab) \[c] =
                 (run c![]
                  (mesg name "eating and giving up both chopsticks");
                  ( sync!(=> (l.rel) \() = [])
                  | sync!(=> (r.rel) \() = [])
                  | dine![] )))
                (=> (r.rel) \[c] =
                 (run c![]
                  (mesg name "giving up right chopstick");
                  dine![])))))
        ))

      dine![])
```

Now we can easily build a recursive function for setting the table:

```
  def setTable [n] =
   (val first = (makechopstick)
    def addPhils [p f] =
      if (== n p) then
        phil![p f first]
      else
       (val next = (makechopstick)
        ( phil![p f next]
        | addPhils![(+ p 1) next]))

    addPhils![1 first])
```

To test what we've done, here's a five-philosopher banquet:

```
  run setTable![5]
```

```
1 thinking
2 thinking
3 thinking
1 got left chopstick
4 thinking
5 thinking
2 got left chopstick
1 giving up left chopstick
1 thinking
2 eating and giving up both chopsticks
2 thinking
4 got left chopstick
1 got left chopstick
4 eating and giving up both chopsticks
3 got left chopstick
4 thinking
1 eating and giving up both chopsticks
1 thinking
3 giving up left chopstick
3 thinking
5 got left chopstick
2 got left chopstick
5 eating and giving up both chopsticks
5 thinking
2 eating and giving up both chopsticks
2 thinking
```

```
4 got left chopstick
5 got left chopstick
4 giving up left chopstick
4 thinking
5 giving up left chopstick
3 got right chopstick
5 thinking
3 giving up right chopstick
3 thinking
```

Probably the most subtle aspect of this solution is the use of the randomization element `delay`. If it is omitted, the solution given here will quickly begin to "resonate," with the same sequence of actions repeated over and over. (Depending on what other processes are running at the same time, the repeated sequence may not even involve *any* philosophers getting to eat!) This behavior arises from the fact that the process scheduler in the Pict runtime system makes no attempt to be "fair" by randomizing the order in which processes are scheduled for execution; nor does the events library try to choose fairly between possible receivers on a given event channel.

Solution to 10.4.1:

```
    def phil [name:Int l:Chopstick r:Chopstick] =
     (def tryGrab (f:Chopstick g:Chopstick dine:Sig) : Event =
        (=> (f.grab)
          \[c] = (c![] |
          ((mesg name "got one chopstick");
           sync!($
             (=> (g.grab) \[c] = (c![] |
               ((mesg name "eating and giving up both chopsticks");
                 (sync!(=> (f.rel) \[c] = c![]) |
                 (sync!(=> (g.rel) \[c] = c![]) |
                 dine![])
               ))))
             (=> (f.rel) \[c] = (c![] |
              ((mesg name "giving up one chopstick"); dine![])))))))
      def dine [] =
        ((mesg name "thinking"); (delay);
         sync!($ (tryGrab l r dine) (tryGrab r l dine)))
      dine![]
    )

    run setTable![5]
```

```
1 thinking
2 thinking
3 thinking
1 got one chopstick
4 thinking
5 thinking
2 got one chopstick
1 giving up one chopstick
1 thinking
2 eating and giving up both chopsticks
2 thinking
4 got one chopstick
1 got one chopstick
4 eating and giving up both chopsticks
3 got one chopstick
4 thinking
```

```
1 eating and giving up both chopsticks
1 thinking
3 giving up one chopstick
3 thinking
5 got one chopstick
2 got one chopstick
5 eating and giving up both chopsticks
5 thinking
2 eating and giving up both chopsticks
2 thinking
4 got one chopstick
5 got one chopstick
4 giving up one chopstick
4 thinking
5 giving up one chopstick
3 got one chopstick
5 thinking
3 giving up one chopstick
3 thinking
```

Solution to 10.3.7: [This solution is based on an idea by Hendrik Tews.]

```
def zero [n:Number] =
  n?*[s z] = z![]

def succ [m:Number n:Number] =
  n?*[s z] =
  (new zz:Signal
   ( m![s zz]
   | zz?[] = (new ack:Signal (s!ack | ack?[] = z![]))))

def plus [m:Number n:Number o:Number] =
  o?*[s z] =
  (new zz:Signal
   ( m![s zz]
   | zz?[] = n![s z]))

def test[n:Number] =
 (new s:AckReq
  new z:Signal
  def handleS[] = s?ack = (ack![] | print!"*" | handleS![])
  def handleZ[] = z?[] = ()
  (n![s z] | handleS![] | handleZ![]))

new myZero: Number
new myOne: Number
new myTwo: Number
new myThree: Number
new myFive: Number

run
( zero![myZero]
| succ![myZero myOne]
| succ![myOne myTwo]
| plus![myOne myTwo myThree]
| plus![myTwo myThree myFive]
| test![myFive])
```

*
*
*
*
*

Hint for 10.5.2: A long-running program containing an priority queue object that repeatedly receives messages over some of its request channels but never receives messages over other request channels will eventually exhaust the available heap space. To see why, look and the implementation of `receiveEvent` and consider what will happen if a send never occurs on the channel `c`. Even though the lock may have been taken by another receiver long ago, the process referred to by `receiver` will never be recognized as garbage by the garbage collector.

Solution to 12.1.1: We use the same type `Omega` as before:

```
type Omega = (rec X where X = ^X)
```

The annotated program is:

```
def c[(rec:Omega p)] = p?x = c![x]
```

If we add a `print` to trace the messages received along `p`...

```
def c[(rec:Omega p)] = p?x = (c![x] | print!"Blow")
```

... then we can test the definition like this:

```
new x: ^Omega
{-
run c![(rec x)]

run ( x!(rec x)
    | x!(rec x)
    | x!(rec x))
-}
```

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118), and as DEC Systems Research Center Research Report number 62, August 1990.

[Agh86] Gul A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[AP90] M Abadi and G. D. Plotkin. A PER model of polymorphism and recursive types. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 355–365, 1990.

[BM92] Kim Bruce and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapporte de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language".

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[Car89] Felice Cardone. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 164–178, Stresa, Italy, July 1989. Springer-Verlag.

[Car90] Luca Cardelli. Notes about $\mathrm{F}^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[Car93] Luca Cardelli. An implementation of $F_{<:}$. Research report 97, DEC Systems Research Center, February 1993.

[CC91] Felice Cardone and Mario Coppo. Type inference with recursive types. Syntax and semantics. *Information and Computation*, 92(1):48–80, 1991.

[CCH⁺89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. A preliminary version appeared in TACS '91 (Sendai, Japan, pp. 750–770).

[Com94] Adriana B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F_{\wedge}^{\omega}$ is decidable".

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[FM94] Kathleen Fisher and John Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software, Sendai, Japan*, pages 844–885. Springer-Verlag, April 1994. LNCS 789.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[Ghe93] Giorgio Ghelli. Recursive types are not conservative over $F_{\leq}$. In *Typed Lambda Calculus and Applications*, March 1993.

[Ghe95] Giorgio Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1,2):131–162, 1995.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF.* Springer-Verlag LNCS 78, 1979.

[GP96] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 1996. To appear.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1991.

[Jon93] Cliff B. Jones. A pi-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings of CONCUR'93*, LNCS 715, pages 158–172. Springer-Verlag, 1993.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proceedings POPL '93*, pages 419–428, 1993.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[McC78] John McCarthy. History of Lisp. In *Proceedings of the first ACM conference on History of Programming Languages*, pages 217–223, 1978. ACM Sigplan Notices, Vol. 13, No 8, August 1978.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[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*. Prentice Hall, 1989.

[Mil90] Robin Milner. Functions as processes. Research Report 1154, INRIA, Sofia Antipolis, 1990. Final version in *Journal of Mathematical Structures in Computer Science* 2(2):119–141, 1992.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[Mil92] Robin Milner. Action structures. Technical Report ECS–LFCS–92–249, Laboratory for Foundations of Computer Science, University of Edinburgh, December 1992.

[Mil95] Robin Milner. Calculi for interaction. *Acta Informatica*, 1995. To appear.

[Mit90] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[MP88] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[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.* The MIT Press, 1990.

[Nie92] Oscar Nierstrasz. Towards an object calculus. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, Lecture Notes in Computer Science number 612, pages 1–20. Springer-Verlag, 1992.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In *Proceedings of CONCUR '96*, August 1996.

[NSL96] Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lumpe. Formalizing composable software systems — a research agenda. In *Formal Methods in Open, Object-Based Distributed Systems (FMOODS '96)*, February 1996.

[Pap91] M. Papathomas. A unifying framework for process calculus semantics of concurrent object-based languages and features. In Dennis Tsichritzis, editor, *Object composition Composition d'objets*, pages 205–224. Centre Universitaire d'Informatique, Universite de Geneve, [6] 1991.

[PDM89] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, March 1989. Available through `http://www.cl.cam.ac.uk/users/bcp1000/ftp`.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). A preliminary version appeared in POPL '92.

[Pie96] Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to pict. Available electronically, 1996.

[Pol90] Robert Pollack. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.

[PRT93] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version to appear in *Mathematical Structures in Computer Science*, 1996.

[PS96] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 1996. To appear. A preliminary version appeared in IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), June 1994, and as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94, January 1994.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[PT95] 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.

[PT96a] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. To appear, 1996.

[PT96b] Benjamin C. Pierce and David N. Turner. Pict language definition. Draft report; available electronically through `http://www.cl.cam.ac.uk/users/bcp1000/draft`, 1996.

[PT96c] Benjamin C. Pierce and David N. Turner. Pict standard libraries manual. Available electronically, 1996.

[Rep88] John Reppy. Synchronous operations as first-class values. In *Programming Language Design and Implementation*, pages 250–259. SIGPLAN, ACM, 1988.

[Rep90] John Reppy. *Concurrent Programming with Events.* Cornell University, November 1990. the Concurrent ML Manual (Version 0.9).

[Rep91] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

[Rep92] John Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, June 1992. Technical Report TR 92-1285.

[Rep95] John H. Reppy. First-class synchronous operations. 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. Springer-Verlag, April 1995.

[SL96] Jean-Guy Schneider and Markus Lumpe. Modelling objects in Pict. Technical Report IAM–96–004, Universitaet Bern, Institut fuer Informatik und Angewandte Mathematik, January 1996.

[Tur96] David N. Turner. *The Polymorphic Pi-calulus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1996.

[TW93] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *Proceedings of TAPSOFT '93*, pages 686–701, 1993.

[Var96] Patrick Varone. Implementation of "generic synchronization policies" in Pict. Technical Report IAM–96–005, Universitaet Bern, Institut fuer Informatik und Angewandte Mathematik, April 1996.

[Vas94] Vasco T. Vasconcelos. Typed concurrent objects. In *Proceedings of the Eighth European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, July 1994.

[WAL$^+$89] Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.

[Wal94] David Walker. Algebraic proofs of properties of objects. In *Proceedings of European Symposium on Programming.* Springer-Verlag, 1994.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

Index