Programming in the Pi-Calculus

A Tutorial Introduction to Pict

(Pict Version 4.1)

Benjamin C. Pierce

Computer Science Department
Indiana University
Lindley Hall 215
Bloomington, Indiana 47405-4101
USA
`pierce@cs.indiana.edu`

March 22, 1998

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, is becoming popular as a theoretical foundation for a broad class of concurrent computations. The goal in Pict is to identify and support 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 `pierce@cs.indiana.edu`. Comments, suggestions, and bug reports are also welcome. (Please send these to both `pierce@cs.indiana.edu` and `dnt@dcs.gla.ac.uk`.) Suggestions for improvement of the documentation are especially welcome.

Citations

The best bibliographic single citation for Pict is [PT97a]. Others that may be of interest are [PT95, PT97b, Tur96] and [Pie96] (this document).

Contents

1 Processes and Channels **5**
1.1 Simple Processes . . . 6
1.2 Channels and Communication . . . 7
1.3 Values and Patterns . . . 8
1.4 Types . . . 9
1.5 Channel Creation . . . 9
1.6 Process Definitions . . . 11
1.7 Running the Compiler . . . 12
1.8 Primitive Booleans . . . 13
1.9 Records . . . 14

2 Core Language Semantics **15**
2.1 Overview . . . 15
2.2 Syntax . . . 17
2.2.1 Notational Conventions . . . 17
2.2.2 Concrete Syntax . . . 17
2.3 Scoping . . . 19
2.4 Type Reconstruction . . . 20
2.5 Operational Semantics . . . 20
2.5.1 Structural Congruence . . . 20
2.5.2 Substitution and Matching . . . 22
2.5.3 Reduction . . . 22

3 Subtyping **24**
3.1 Input and Output Channels . . . 24
3.2 Subsumption . . . 25
3.3 Responsive Output Channels . . . 26

4 Simple Derived Forms **28**
4.1 Primitive Values . . . 28
4.1.1 Symbolic Identifiers . . . 28
4.1.2 Numbers . . . 28
4.1.3 Characters and Strings . . . 31
4.2 Derived Forms for Declarations . . . 31
4.2.1 Declaration Sequences . . . 32
4.2.2 `run` Declarations . . . 32
4.3 Parallel Composition . . . 32

4.4 Programs 33
4.5 Larger Programs 33
4.5.1 Importing Other Files 33
4.5.2 Separate Compilation 33
4.5.3 Library Modules 34
4.6 Exercises 34

5 Toward a Programming Language 38
5.1 Complex Values 38
5.1.1 "Continuation-Passing" Translation 38
5.1.2 Value Declarations 40
5.1.3 Application 40
5.2 Derived Forms for Abstractions 41
5.3 Sequencing 42

6 Simple Concurrent Objects 44

7 Advanced Language Features 47
7.1 Lists 47
7.2 Polymorphism 48
7.3 Abstract Types 49
7.4 User-defined Type Constructors 50
7.5 Recursive Types 51

A Solutions to Selected Exercises 53

Bibliography 56

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; conversations with Robin have contributed specific insights too numerous to list. 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, PS97]. Didier Rémy worked with Pierce on the original PIC compiler [PRT93] (on which an early version of the present Pict compiler was based) 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 [PS96]. Dilip Sequeira contributed both code and ideas to an early implementation of type inference and record type checking. Kevin Millikin and Philip Wadler gave us helpful comments on the formal definition.

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, at the Computer Laboratory, University of Cambridge, with support from CONFER and from the British Science and Engineering Research Council, and at Indiana University.

Chapter 1

Processes and Channels

The π-calculus of Milner, Parrow, and Walker [MPW92] bears many similarities 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, from Landin's ISWIM [Lan66] and McCarthy's LISP [McC78] to modern languages such as ML, Scheme, and Haskell.

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 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 is a 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 directly support both higher-order programming and natural encodings of primitive datatypes. The ability to pass channels themselves as values between processes — the defining characteristic of the π-calculus — 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 discussion continues in the concurrency community as to what should constitute a general theory of concurrency. Nevertheless, we've reached a good point to begin experimenting. If Lisp (or, if you prefer, ML, 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 is our attempt to learn the answer.

This chapter offers an informal introduction to a fragment of the Pict language closely related 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 adds some more complex translation rules yielding convenient high-level syntactic constructs such as function application. Chapter 6 develops an extended example, showing how reference cell objects can be programmed in Pict.

The full Pict language offers a number of features not discussed in this brief tutorial. See the *Pict Language Definition* [PT97b] for a formal description of the entire language.

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

The simplest thing that a process can actually *do* is to cause an observable event in the outside world. For example, a process of the form `print!"abc"` causes the string `abc` to be printed on the standard output stream.

```
    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 and process definitions 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 is completed when 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![]` is now inside the body of the input `x?[] = ...` and so cannot take place until after the input 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 a *channel* can be passed from the sender to the receiver as part of the act of communication.

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

```
Got it!
```

As we shall see, this ability is a primary source of the π-calculus's (and Pict's) expressive power.

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 variable* 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 quite 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 Values and Patterns

More generally, each communication step involves the atomic transmission of a single *value* from sender to receiver. In the fragment of Pict we are considering at the moment, 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 $\mathtt{[v}_1\mathtt{...v}_n\mathtt{]}$ 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 the moment 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 $\mathtt{[p}_1\mathtt{...p}_n\mathtt{]}$ 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 $\mathtt{[p}_1\mathtt{...p}_n\mathtt{]}$ and `v` has the form $\mathtt{[v}_1\mathtt{...v}_n\mathtt{]}$ 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 $\Delta_1 \cup \cdots \cup \Delta_n$.

Two additional forms of patterns are often useful in programming:

- A *wildcard pattern*, written `_` (underscore), matches any value but yields no bindings.
- A *layered pattern* `x@p` matches whatever `p` matches, yielding all the bindings that `p` yields, and also yields a binding of `x` to the whole value matched by `p`.

1.4 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. Like ML, Pict is *statically typed*, in the sense that this check occurs at compile time; program that passes the compiler cannot encounter a situation at run time 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 type-inference mechanism. But there is no need to discuss all of these at once (some of them will be introduced one by one in later chapters; the rest can be found in the *Pict Language Definition*). 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 $\mathtt{[v_1 \ldots v_n]}$ has the type $\mathtt{[T_1 \ldots T_n]}$. (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`" or "channel carrying `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 with any type system simple enough to be tractable) this restriction excludes some reasonable and perhaps even useful programs, relaxing it would mean using a dynamic flow analysis for typechecking programs. 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 techniques 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.

1.5 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` must 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 each 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 possibility 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
```

1.6 Process Definitions

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![]`. Note that the parameters of a definition must be explicitly annotated with their expected types.

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
```

One final refinement that we might make is to reduce the number of irrelevant variable bindings by using the wildcard pattern `_` instead of named variables:

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

1.7 Running the Compiler

To execute a Pict program, first place it in a file `prog.pi` and then compile and execute this file with the Pict compiler:

```
pict prog.pi
```

Running the compiler this way produces a file `a.out`, which is executed automatically the first time, and which you can thereafter execute directly, without recompiling the original program:

```
./a.out
```

Alternatively, you can give the Pict compiler an explicit output file

```
pict -o prog prog.pi
```

and run the resulting executable yourself:

```
./prog
```

1.7.1 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`*. For example, running the following program should produce the output* `It's true`*:*

```
type Boolean = ^[^[] ^[]]
def tt[b:Boolean] = b?[t f] = t![]
and ff[b:Boolean] = b?[t f] = f![]
def test[b:Boolean] =
  (new t:^[] new f:^[]
   ( b![t f]
   | t?[] = print!"It's true"
   | f?[] = print!"It's false"))
```

```
def notB[b:Boolean c:Boolean] = <your definition>

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

```
It's true
```

[Solution on page 53.]

1.7.2 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 53.]*

1.7.3 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 is 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`*.*

```
def notB[b:Boolean res:^Boolean] =
 (new c:Boolean
  run c?[t f] = b![f t]
  res!c)

new b:Boolean
new res:^Boolean
run ( tt![b]
    | notB![b res]
    | res?c = test![c])
```

```
It's false
```

Rewrite your solution to exercise 1.7.2 using this protocol for returning results. *[Solution on page 53.]*

1.8 Primitive 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]
```

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

```
def bool2Boolean[b:Bool    r:^[Boolean]] = ...
def bool2Boolean[b:Bool    r:^[Boolean]] = ()
def boolean2Bool[b:Boolean r:^[Bool]   ] = ...
def boolean2Bool[b:Boolean r:^[Bool]   ] = ()
```

1.9 Records

It is sometimes convenient to give names to the elements of a tuple. This is accomplished in Pict by associating *labels* with the elements, each label consisting of an alphanumeric identifier and an `=` sign to separate it from the element itself. Such tuples are called *records*.

```
run x![a=false b=true c=[]]
```

A pattern matching a record must have the same labels in corresponding positions:

```
run x?[a=p b=q c=r] = if q then print!"Got true" else print!"Got false"
```

```
Got true
```

Similarly, the type of a tuple with labeled fields includes appropriate labels:

```
new x : ^[a=Bool b=Bool c=[]]
```

In fact, labeled and unlabeled fields may be mixed in the same record:

```
new y : ^[Bool b=Bool []]
run y![false b=true []]
run y?[p b=q r] = if q then print!"Got true" else print!"Got false"
```

Tuples are just the special case where all of the labels are blank. From now on, we use the word "record" for both tuples and labeled records.

For explicitly labeled fields of records, we allow the usual "dot notation" for projecting fields:

```
run x![a=true b=false]
run x?r = if r.a then print!"true" else print!"false"
```

```
true
```

Chapter 2

Core Language Semantics

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; a formal presentation of the type system can be found in the Pict Language Definition [PT97b].

Most of the material in this chapter is repeated, in less formal terms, elsewhere. It can be skimmed on a first reading.

2.1 Overview

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

$$
\begin{array}{c}
\boxed{\text{ascii program text}} \\
\downarrow \ \text{lexical analysis} \\
\boxed{\text{sequence of tokens}} \\
\downarrow \ \text{parsing (Section 2.2)} \\
\boxed{\text{abstract syntax tree}} \\
\downarrow \ \text{scope resolution (Section 2.3)} \\
\boxed{\text{abstract syntax with unique names}} \\
\downarrow \ \text{type reconstruction (Section 2.4)} \\
\boxed{\text{explicitly typed abstract syntax}} \\
\downarrow \ \text{execution (Section 2.5)}
\end{array}
$$

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 a new abstract syntax tree where each variable binder is named differently from any other binder elsewhere in the program. A type reconstruction pass walks over this tree, checking that it satisfies the typing rules of the language and filling in any type annotations that have been omitted by the programmer to yield a new abstract syntax tree with the types of all its subphrases explicitly given. This final 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 give 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.

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. This justifies giving a formal treatment of the operational semantics while remaining informal (in this document) about the type system.

The definition of the full Pict language in Chapters 4 and 5 adds two more phases, which perform the translation from high-level to core abstract syntax.

$\boxed{\text{program text}}$

$\downarrow$ lexical analysis (see *Definition*

$\downarrow$ parsing

$\boxed{\text{abstract syntax}}$

$\downarrow$ simple derived forms (Chapters 4 and 5)

$\boxed{\text{simpler abstract syntax}}$

$\downarrow$ scope resolution

$\downarrow$ type reconstruction

$\boxed{\text{explicitly typed (high-level) abstract syntax}}$

$\downarrow$ type-directed derived forms (Chapters 4 and 5)

$\boxed{\text{core abstract syntax}}$

$\downarrow$ execution

The derived forms are divided into two separate phases. Some of them are so simple that they can be performed directly by the parser, thus eliminating the need to deal with them in the scope resolution or type reconstruction phases. Others (in particular, the CPS-conversion described in Chapter 5) require that scopes be resolved and missing type annotations inserted before they can do their work.

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. The possible forms of each production are listed on successive lines. Keywords are set in typewriter font. An expression of the form $X \ldots X$ denotes a list of zero or more occurrences of X. The expression $\langle empty \rangle$ denotes an empty production.

2.2.2 Concrete Syntax

We have said that the top level of a compilation unit (i.e., a file containing Pict source code) is a series of `import` statements followed by a series of declarations. For present purposes, though, it is convenient to consider programs in a much more restricted form: just a single `run` declaration with a process expression in its body.

Program	=	`run` *Proc*	Program

A process expression can have several forms:

Proc	=	*Val* `!` *Val*	Output atom
		Val `?` *Abs*	Input prefix
		`( )`	Null process
		`(` *Proc* `|` *Proc* `)`	Parallel composition
		`(` *Dec* *Proc* `)`	Local declaration
		`if` *Val* `then` *Proc* `else` *Proc*	Conditional

Note that the syntax given her is a bit more restrictive than what we saw in Chapter 1: it allows only binary parallel composition and only one local declaration in front of a process. The more permissive forms will be recovered in Chapter 4 as derived forms.

On the other hand, there are also some ways in which this syntax is more permissive than the examples in Chapter 1 suggested. In particular, in the output expression $\mathtt{v}_1\mathtt{!}\mathtt{v}_2$, 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.)

Input processes are defined in terms of the syntactic class `Abs` of process expressions prefixed by patterns

Abs	=	*Pat* `=` *Proc*	Process abstraction

where a pattern can be a variable, a record of (labeled or unlabeled) patterns, or a layered or wildcard pattern.

Pat	=	*Id* *RType*	Variable pattern
		`[` *Label* *Pat* ... *Label* *Pat* `]`	Record pattern
		`_` *RType*	Wildcard pattern
		Id *RType* `@` *Pat*	Layered pattern

Every variable, wildcard, and layered pattern includes space for an optional type annotation.

RType	=	⟨*empty*⟩	Omitted type annotation
		: *Type*	Explicit type annotation

In the typechecking rules, operational semantics, and derived forms, we assume that all these annotations have been provided explicitly. It is the job of the *type reconstruction* phase to fill in omitted annotations, if possible, before these later phases need them.

The syntactic class of values—things that can be sent along channels—includes constants, variables, records, and field projections. For technical reasons, we syntactically restrict the "left hand side" of a field projection to be either a variable or another field projection by introducing a separate syntactic class of *paths*.

Val	=	*Const*	Constant
		Path	Path
		`[` *Label Val* ... *Label Val* `]`	Record

A path is a variable followed by zero or more field projections.

Path	=	*Id*	C	Variable
		Path `.` *Id*	C	Record field projection

Constant values include the booleans `true` and `false` and literal strings that we have already seen, plus integers and character constants (these last are discussed in Chapter 4):

Const	=	*String*	String constant
		Char	Character constant
		Int	Integer constant
		`true`	Boolean constant
		`false`	Boolean constant

Type expressions correspond to the possible shapes of values: channels, constants

Type	=	`^` *Type*	Input/output channel
		`Bool`	Boolean type
		`String`	String type
		`Int`	Integer type
		`Char`	Character type
		`[` *Label Type* ... *Label Type* `]`	Record type

A declaration can be a channel definition, a process definition, or a type definition.

Dec	=	`new` *Id* `:` *Type*	Channel creation
		`def` Id_1 Abs_1 `and ... and` Id_n Abs_n	Recursive definition ($n \geq 1$)
		`type` *Id* `=` *Type*	Type abbreviation

Note that the bodies of definitions are abstractions—the same syntactic form as the bodies of input processes. This means, in particular, that a process definition can actually use an arbitrary pattern, not just a tuple. Also, the type annotations on the pattern in a process definition are syntactically optional. However, because of the way the type reconstruction propagates information, it turns out that they can almost never be recovered during the type reconstruction phase, and must therefore be provided explicitly.

A group of definitions introduced by `def` and separated by `and` can be mutually recursive—i.e., each of the bodies can refer to any of the defined names.

The `run` declarations of Chapter 1 (except for the single outermost one) are missing from this syntax. They will be reintroduced in Chapter 4 as derived forms. Also, type declarations are treated informally in this document. In the *Language Definition* they are actually presented as derived forms, but their formalization depends on language features that fall outside the scope of this tutorial.

Finally, as we have seen, a field label (in a record pattern, record value, or record type) can be either an identifier followed by an `=` sign, or else empty.

Label	=	$\langle empty \rangle$	Anonymous field
		Id `=`	Labeled field

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 variable pattern of the form `x:T` creates the name `x`.
- A record pattern of the form $[\mathtt{l}_1\mathtt{p}_1 \ldots \mathtt{l}_n\mathtt{p}_n]$ 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 layered pattern of the form `x:T@p` creates all of the names created by the subpattern `p`, plus `x` (which must be distinct from the names created by `p`).

In the remaining phases of the core language definition (the operational semantics, typechecking, and translation rules), we assume that the names of bound variables are always different, silently renaming variables if necessary to achieve this form. A formal definition of this translation process can be found in the *Pict Language Definition.*

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 Type Reconstruction

In this tutorial, the typechecking rules and type reconstruction algorithm are treated informally (they appear in full glory in the *Definition*). Fortunately, the type system corresponding to the part of the language we are convering here is quite straightforward.

Type reconstruction, too, is based on a few simple ideas:

- During reconstruction, type information can be propagated either from subphrases to the larger phrases that contain them ("synthesis") or from larger phrases to their constituents ("checking"). Synthesis mode is used to *calculate* the type of a phrase, while checking mode is used to *verify* that a phrase has some expected type.
- In synthesis mode, type annotations are generally required; in checking mode, an omitted annotation can often be filled in by using the expected type at that point.
- All bound variables must have their types determined at the point of binding, either by an explicit type annotation or by appearing in a checking context. This means that later occurrences always have a known type, whether they are encountered in a checking or a synthesis context.

In particular, the patterns in process definitions are analyzed in a synthesis context (since we do not know yet what shapes of values their are expected to match), while patterns in input expressions are analyzed in a checking context, since the type of the channel from which the input is taken determines the shape of the values that the pattern must match. The parameters to a process definition must therefore be explictly typed, while the parameters of an input expression need not be.

2.5 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.

For present purposes, a program is just the keyword `run` followed by a process expression `e`. Its behavior is the behavior of `e`.

2.5.1 Structural Congruence

Structural congruence plays an important technical role in 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 adding a general stipulation that, when `e` contains some possibility of communication, any expression structurally congruent to `e` has the same possible behavior.

The first three 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

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

and that the null process `()` is an identity for parallel composition.

$$(\mathtt{e} \mid ()) \equiv \mathtt{e} \qquad \text{(STR-NULL)}$$

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

$$\frac{BV(\mathtt{d}) \cap FV(\mathtt{e}_2) = \emptyset}{((\mathtt{d}\ \mathtt{e}_1) \mid \mathtt{e}_2) \equiv (\mathtt{d}\ (\mathtt{e}_1 \mid \mathtt{e}_2))} \qquad \text{(STR-EXTRUDE)}$$

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.

Similarly, two adjacent `new` declarations can always be swapped (since, by the conventions introduced in Section 2.3, they must introduce channels with different names), and a `new` may be swapped with a `def` (or even a `def` with a `def`) as long as the body of the `def` does not use the name defined by the `new`. These cases are captured by a general rule for exchanging adjacent declarations:

$$\frac{BV(\mathtt{d}_1) \cap FV(\mathtt{d}_2) = \emptyset \qquad BV(\mathtt{d}_2) \cap FV(\mathtt{d}_1) = \emptyset}{(\mathtt{d}_1\ (\mathtt{d}_2\ \mathtt{e})) \equiv (\mathtt{d}_2\ (\mathtt{d}_1\ \mathtt{e}))} \qquad \text{(STR-SWAPDEC)}$$

Finally, two adjacent `def` clauses may be merged into one:

$$\frac{\begin{array}{c}(\{\mathtt{x}_1, \ldots, \mathtt{x}_m\} \cap \{\mathtt{x}_{m+1}, \ldots, \mathtt{x}_n\} = \emptyset \\ (FV(\mathtt{a}_1) \cup \cdots \cup FV(\mathtt{a}_m)) \cap \{\mathtt{x}_{m+1}, \ldots, \mathtt{x}_n\} = \emptyset\end{array}}{\begin{array}{c}(\mathtt{def}\ \mathtt{x}_1\mathtt{a}_1\ \ldots\ \mathtt{and}\ \mathtt{x}_m\mathtt{a}_m\ (\mathtt{def}\ \mathtt{x}_{m+1}\mathtt{a}_{m+1}\ \ldots\ \mathtt{and}\ \mathtt{x}_n\mathtt{a}_n\ \ \mathtt{e})) \equiv \\ (\mathtt{def}\ \mathtt{x}_1\mathtt{a}_1\ \ldots\ \mathtt{and}\ \mathtt{x}_m\mathtt{a}_m\ \mathtt{and}\ \mathtt{x}_{m+1}\mathtt{a}_{m+1}\ \ldots\ \mathtt{and}\ \mathtt{x}_n\mathtt{a}_n\ \ \mathtt{e})\end{array}} \qquad \text{(STR-COALESCE)}$$

Reading this rule in the other direction, it says that a single compound `def...and` clause may be split into two, so long as the bodies of definitions in the first part do not depend on names defined by the second part.

[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.

2.5.2 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 []]`.

In order to support planned extensions of the language, we want to maintain the syntactic property that the head of every path is a variable, not an explicitly constructed record. To maintain this property during reduction, we need to make sure that, when we substitute a record value for a variable during a communication step, any projection expressions with this variable as their head are reduced at the same time by projecting out the appropriate field:

$$\{\mathtt{x} \mapsto \mathtt{[}\ldots\mathtt{l}_i\mathtt{v}_i\ldots\mathtt{]}\}\mathtt{x.l}_i \quad = \quad \mathtt{v}_i$$

This refined version of substitution is the one used in the rules that follow.

2.5.2.1 Definition: When a value `v` is successfully matched by a pattern `p`, the result is a substitution $\{\mathtt{p} \mapsto \mathtt{v}\}$, defined as follows

$$\begin{array}{rcl}
\{\mathtt{x:T} \mapsto \mathtt{v}\} & = & \{\mathtt{x} \mapsto \mathtt{v}\} \\
\{\mathtt{_:T} \mapsto \mathtt{v}\} & = & \{\} \\
\{\mathtt{(x:T@p)} \mapsto \mathtt{v}\} & = & \{\mathtt{x} \mapsto \mathtt{v}\} \cup \{\mathtt{p} \mapsto \mathtt{v}\} \\
\{\mathtt{[l_1fp_1\ldots l_n fp_n]} \mapsto \mathtt{[l_1fv_1\ldots l_n fv_n\ldots]}\} & = & \{\mathtt{fp}_1 \mapsto \mathtt{fv}_1\} \cup \cdots \cup \{\mathtt{fp}_n \mapsto \mathtt{fv}_n\}
\end{array}$$

If `v` and `p` do not have the same structure, then $\{\mathtt{p} \mapsto \mathtt{v}\}$ 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.

2.5.3 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:

$$\frac{\{\mathtt{p} \mapsto \mathtt{v}\} \text{ defined}}{\mathtt{(x!v\ \mid\ x?p = e)} \rightarrow \{\mathtt{p} \mapsto \mathtt{v}\}\mathtt{(e)}} \qquad \text{(Red-Comm)}$$

The rule for instantiating definitions is similar, except that the "input side" is some clause of a definition, not a simple input prefix.

$$\frac{\{\mathtt{p}_i \mapsto \mathtt{v}\} \text{ defined}}{\begin{array}{l}\quad(\mathtt{def}\ \mathtt{x}_1\mathtt{p}_1 = \mathtt{e}_1\ \ldots\ \mathtt{and}\ \mathtt{x}_n\mathtt{p}_n = \mathtt{e}_n\ \ (\mathtt{x}_i!\mathtt{v}\ |\ \mathtt{e}))\\ \to (\mathtt{def}\ \mathtt{x}_1\mathtt{p}_1 = \mathtt{e}_1\ \ldots\ \mathtt{and}\ \mathtt{x}_n\mathtt{p}_n = \mathtt{e}_n\ \ (\{\mathtt{p}_i \mapsto \mathtt{v}\}(\mathtt{e}_i)\ |\ \mathtt{e}))\end{array}} \qquad \text{(RED-DEF)}$$

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

$$\mathtt{if\ true\ then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \to \mathtt{e}_1 \qquad \text{(RED-IF-T)}$$

$$\mathtt{if\ false\ then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \to \mathtt{e}_2 \qquad \text{(RED-IF-F)}$$

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

$$\frac{\mathtt{e}_1 \to \mathtt{e}_2}{(\mathtt{d}\ \mathtt{e}_1) \to (\mathtt{d}\ \mathtt{e}_2)} \qquad \text{(RED-DEC)}$$

$$\frac{\mathtt{e}_1 \to \mathtt{e}_3}{(\mathtt{e}_1\ |\ \mathtt{e}_2) \to (\mathtt{e}_3\ |\ \mathtt{e}_2)} \qquad \text{(RED-PRL)}$$

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.

$$\frac{\mathtt{e}_1 \equiv \mathtt{e}_2 \to \mathtt{e}_3 \equiv \mathtt{e}_4}{\mathtt{e}_1 \to \mathtt{e}_4} \qquad \text{(RED-STR)}$$

In closing, it is worth mentioning that we have done here only a 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 [PT97a].

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 [Sew96].

Chapter 3

Subtyping

We have already 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 an important refinement of this basic type system.

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, 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 a good distance away.

Fortunately, there are some simple ways in which our simple channel types can be enriched so as to capture useful properties of programs while remaining within the bounds established by current typechecker 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 1.6, 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 Boolean       = ^[^[] ^[]]
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.

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]`. More generally, if each $\mathtt{S}_i$ is a subtype of the corresponding $\mathtt{T}_i$, then the record type $[\mathtt{l}_1\mathtt{S}_1 \ldots \mathtt{l}_n\mathtt{S}_n]$ is a subtype of $[\mathtt{l}_1\mathtt{T}_1 \ldots \mathtt{l}_n\mathtt{T}_n]$.

That is, the labels in the two record types must match in each position (either both must be blank or both must be the same explict label) and the types of corresponding fields must be in the subtype relation. More generally yet, we allow the smaller type to have some extra fields on the right, so that $\texttt{[l}_1\texttt{S}_1\texttt{...l}_m\texttt{S}_m\texttt{...l}_n\texttt{S}_n\texttt{]}$ is a subtype of $\texttt{[l}_1\texttt{T}_1\texttt{...l}_m\texttt{T}_m\texttt{]}$. Note, though, that we do not allow fields to be reordered or extra fields to be added in the middle of the common ones. (In the absence of this restriction, separate compilation is much more difficult.)

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.

3.2.1 Exercise [Recommended]: *Rewrite your solutions to the exercises in Chapter 1, using* `!` *and* `?` *instead of* `^` *wherever possible.*

3.2.2 Exercise [Recommended]: *We have seen that the tuple constructor is* monotone *in the subtype relation: if each* $\texttt{S}_i < \texttt{T}_i$*, then* $\texttt{[S}_1\texttt{,...,S}_n\texttt{]} < \texttt{[T}_1\texttt{,...,T}_n\texttt{]}$*. 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`*?*

3.3 Responsive Output Channels

In fact, we take the refinement of channel types a step further in Pict, identifying one case of communication that is so common as to deserve special treatment.

Channels created by `def` clauses (as opposed to `new` clauses) have two important properties: (1) There is always a receiver (the body of the `def`) available, and (2) all communications are received by the *same* receiver—the same `def` body. These properties turn out to be extremely useful in compilation: sending on a channel that is known to have been created by a `def` can be implemented essentially as a branch, whereas general communication requires quite a bit more checking of channel status bits, manipulation of queues, etc. But, since channels can be passed along other channels before being used for communication, it is not possible to tell statically which outputs will actually be communicating with definitions when they are executed, just by looking at the program's structure. Instead, we use the type system.

A channel created by a `def` has a type of a special form, written `/T` and pronounced "*responsive* channel carrying elements of type `T`." When such channels are sent along other channels, the type system tracks this fact. For example, from the type of `x` in the following program, the output process `a!false` knows that it is communicating with a `def`.

```
new x : ^[/Bool]
def d b:Bool = if b then print!"True" else print!"False"
run x![d]
run x?[a] = a!false
```

```
False
```

The subtyping relation allows `/T` $<$ `!T`, so that responsive channels can be used, if desired, as ordinary channels, forgetting their special properties.

```
new y : ^[!Bool]
run y![d]
run y?[a] = a!false
```

```
False
```

Allowing subtyping in the other direction, promoting ordinary channels to responsive ones, would clearly be unsound. But there are some cases where we may want to use an ordinary output channel where a responsive channel is required. For example, the Pict standard library provides a primitive called `pr` that (like `print`) prints a string on the standard output stream, but then sends a signal along another channel to indicate that it has completed its work. The type of `pr` is `/[String /[]]`—that is, `pr` is itself a responsive channel, and it expects its two arguments to be a string and another responsive channel (carrying data-less signals). We can invoke `pr` by sending it a channel created by a definition...

```
def d [] = print!"done"
run pr!["pr... " d]
```

```
pr... done
```

...but if we try to send it an ordinary channel, from which we can later read to obtain the signal sent by `pr`, we get a typechecking failure:

```
new c : ^[]
run pr!["pr... " c]
run c?[] = print!"done"
```

```
example.pi:2.20:
Expected type does not match actual
since ^[] is not a subtype of Sig since ^[] and /[] do not match
```

In order to allow programs like the last one (which is arguably easier to read, since the "continuation" of the call to `pr` appears after the call itself), Pict provides an operator `(rchan ...)` than can be used to coerce an ordinary output channel into a responsive channel:

```
new c : ^[]
run pr!["pr... " (rchan c)]
run c?[] = print!"done"
```

```
pr... done
```

(In fact, `rchan` can be defined in Pict, using the high-level syntactic forms introduced in Chapter 5.)

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

It is 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).

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

```
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:Int  +![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`.

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 [PT97c]. To add two numbers and print the result, for example, we can write:

```
def r z:Int = printi!z
run +![2 3 r]
```

```
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 5 make such programs much more concise.)

```
run
(def fact [n:Int r:!Int] =
   (new br:^Bool
     ( {- calculate n=0 -}
       ==![n 0 (rchan 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 (rchan 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 (rchan 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:*

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

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 `int.toString` 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`:

```
def printi i:Int =
      (new r1 : ^String
       ( int.toString![i (rchan r1)]
       | r1?s = print!s ))

run printi!5
```

```
5
```

Indeed, `print` itself is defined in terms of the lower-level predefined channel `pr`: we just ignore the completetion signal returned by `pr`.

```
def print s:String =
      (def 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, as in the examples in Chapter 1, we avoid proliferation of parentheses in a sequence of declarations like

```
(new x:A (new y:B (new z:C ...)))
```

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

```
(new x:A new y:B new z:C ...)
```

Processes taking advantage of this more liberal syntax are translated back into the core language during compilation, simply by reinserting the dropped parentheses. In the *Definition*, this is captured by the following rule:

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

(We will not show all of the translation rules for the derived forms we discuss; see the *Definition* for full details.)

4.2.2 `run` Declarations

Again, as we saw in Chapter 1, 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 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 programs 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. For example, the process

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

is equivalent to the following core-language process:

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

4.3 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 composed in parallel, translating "high level" process expressions like

```
(x![] | y![] | z![] | w![])
```

into:

```
(x![] | (y![] | (z![] | w![])))
```

4.4 Programs

Now that we have (re)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. In the full language, a Pict program an arbitrary declaration sequence. Its meaning is obtained by surrounding it by `(... ())` and using the translation rules for declarations to 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 is equivalent to the core-language process

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

which, in turn, is structurally congruent to the process:

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

4.5 Larger Programs

The Pict compiler provides a simple facility for breaking up large programs into parts, storing the parts in separate files, and compiling these files separately.

4.5.1 Importing Other Files

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 (not including the suffix `.pi`). 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.

4.5.2 Separate Compilation

Before a file `f.pi` can be imported (using `import "f"`) by other files, it must be processed by the Pict compiler, yielding a file `f.px`. The command-line flag `-set sep` tells the compiler that the file it is processing is a separately compiled module, not the main program.

For example, if the file `f.pi` contains

```
def x[] = print!"...endlessly rocking\n"
```

and the file `g.pi` contains

```
import "f"
run x![]
```

then we compile `g.pi` into an executable file `g` in two steps:

```
pict -set sep -o f.px f.pi
pict -o g g.pi
```

4.5.3 Library Modules

The Pict distribution includes a library of precompiled modules implementing a variety of data structures, interfaces to operating system facilities, and other services. These are described in full in the *Pict Libraries Manual* [PT97c]. A few of the most basic libraries, including all of the facilities described so far, are imported by default; however, most must be imported explicitly if they are needed.

4.6 Exercises

4.6.1 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 4.6.2), 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"
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:![]] =
  (def c1 b:Bool =
    if b then
```

```
      c![]
    else
      (def c2 i:Int = delay![i c]
       -![n 1 c2])
   ==![n 0 c1])

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

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

```
        ( submit![worker1 1 (rchan done)]
        | done?[] =
            ( submit![worker2 2 (rchan done)]
            | done?[] =
               pr!["Test worker finishing\n" (rchan c)]
            )))
   new testdone:^[]
   submit![testWorker 3 (rchan 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
Worker 2 starting
Test worker finishing
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 54.]

4.6.2 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 54.]

Chapter 5

Toward a Programming Language

Chapters 1 to 4 introduced the syntax and operational semantics 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, essentially conclude) the task of defining a high-level programming notation based on these foundations.

5.1 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 `pr` and `+`) and records of values. These *simple values* are important because they are the entities that can be passed along channels and participate in pattern matching.

5.1.1 "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:T x!n)` creates a fresh channel `n` and sends it off along `x`. More interestingly,

```
run (def f[x:Int res:/Int] = +![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:T 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:T n)`, we do not mean to send the *expression* `(new n:T 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:A x) (new y:B 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 $[\![\mathtt{v} \to \mathtt{c}]\!]$ 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

$$\frac{\Gamma \vdash \mathtt{v}_1 \in \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{v}_2 \in \mathtt{T}_2}{[\![\mathtt{v}_1\mathtt{!v}_2]\!] = \mathtt{(def\ c_1\ x_1:T_1\ =\ (def\ c_2\ x_2:T_2\ =\ x_1!x_2}\quad [\![\mathtt{v}_2 \to \mathtt{c}_2]\!])\quad [\![\mathtt{v}_1 \to \mathtt{c}_1]\!])} \qquad \text{(CPS-OUT)}$$

translates an output $\mathtt{v}_1\mathtt{!v}_2$ into a process expression that first allocates a fresh continution channel c, next 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 $\mathtt{v}_1$. The notation $[\![\mathtt{v} \to \mathtt{c}]\!]$ stands, intuitively, for a process that calculates v and sends the resulting simple value along c. The premises $\Gamma \vdash \mathtt{v}_1 \in \mathtt{T}_1$ and $\Gamma \vdash \mathtt{v}_s \in \mathtt{T}_2$ are calls to the typechecker, which calculate the types $\mathtt{T}_1$ and $\mathtt{T}_2$ that should appear in the annotations of the two defs. (In general, $\Gamma \vdash \mathtt{v} \in \mathtt{T}$ is read "Under the assumptions Γ (which give the types of all the free variables in v), the value v has type T."

Input processes and conditionals containing complex values are translated similarly:

$$\frac{\Gamma \vdash \mathtt{v} \in \mathtt{T}}{[\![\mathtt{v?p=e}]\!] = \mathtt{(def\ c\ x:T\ =\ x?p=}[\![\mathtt{e}]\!]\quad [\![\mathtt{v} \to \mathtt{c}]\!])} \qquad \text{(CPS-IN)}$$

$$[\![\mathtt{if\ v\ then\ e_1\ else\ e_2}]\!] = \mathtt{(def\ c\ x:Bool\ =\ if\ x\ then}\ [\![\mathtt{e}_1]\!]\ \mathtt{else}\ [\![\mathtt{e}_2]\!]\quad [\![\mathtt{v} \to \mathtt{c}]\!]) \qquad \text{(CPS-IF)}$$

The continuation passing translation itself is defined by induction on the syntax of value expressions. Variables and constants are easy to handle, since they already represent simple values:

$$[\![\mathtt{x} \to \mathtt{c}]\!] = \mathtt{c!x} \qquad \text{(CPS-VAR)}$$

$$[\![\mathtt{k} \to \mathtt{c}]\!] = \mathtt{c!k} \qquad \text{(CPS-CONST)}$$

The translation of values prefixed by declarations is also straightforward:

$$[\![\mathtt{(new\ x:T\ v)} \to \mathtt{c}]\!] = \mathtt{(new\ x:T}\ [\![\mathtt{v} \to \mathtt{c}]\!]) \qquad \text{(CPS-NEWV)}$$

$$[\![\mathtt{(def\ x_1a_1\ \ldots\ and\ x_na_n\quad v)} \to \mathtt{c}]\!] = \mathtt{(def\ x_1}[\![\mathtt{a}_1]\!]\ \ldots\ \mathtt{and\ x}_n[\![\mathtt{a}_n]\!]\quad [\![\mathtt{v} \to \mathtt{c}]\!]) \qquad \text{(CPS-DEFV)}$$

$$[\![\mathtt{(run\ e\quad v)} \to \mathtt{c}]\!] = ([\![\mathtt{e}]\!]\ \mid\ [\![\mathtt{v} \to \mathtt{c}]\!]) \qquad \text{(CPS-RUNV)}$$

The only slightly complex case is records, where each of the fields is CPS-converted separately, from left to right. To keep the size of the rule manageable, we rely on an auxiliary definition that recurses down the list of fields, transforming each one; we give it the list of fields and set a marker (written $|$) at the beginning of the list of fields to indicate that none of them have been translated yet.

$$\frac{\Gamma \vdash \mathtt{[l_1fv_1 \ldots l_nfv_n]} \in \mathtt{[l_1FT_1 \ldots l_nFT_n]}}{[\![\mathtt{[l_1fv_1 \ldots l_nfv_n]} \to \mathtt{c}]\!] = [\![\,|\,\mathtt{l_1fv_1 \ldots l_nfv_n} \to \mathtt{c}]\!]} \qquad \text{(CPS-RECORD)}$$

There are two rules for CPS-converting lists of field values, depending on whether the marker $|$ is at the end of the list of fields, indicating that all fields have already been converted, or a value field, which must be CPS-converted. In the latter case the field following the marker is CPS-converted by creating a definition for a new channel $\mathtt{c}_i$ whose body is formed by recursively CPS-converting the remaining fields and starting a process to send the value for the ith field along $\mathtt{c}_i$.

$$[\![\mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_n\mathtt{fv}_n \mid \to \mathtt{c}]\!] = \mathtt{c!}[\mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_n\mathtt{fv}_n] \qquad \text{(CPSF-DONE)}$$

$$\frac{\Gamma \vdash \mathtt{v}_i \in \mathtt{T}_i}{\begin{array}{c}[\![\mathtt{l}_1\mathtt{fv}_1 \ldots \mid \mathtt{l}_i\mathtt{v}_i \ldots \mathtt{l}_n\mathtt{fv}_n \to \mathtt{c}]\!] = \\ \mathtt{(def\ c}_i\ \mathtt{x}_i\mathtt{:T}_i\ \mathtt{=}\ [\![\mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_i\mathtt{x}_i \mid \mathtt{l}_{i+1}\mathtt{fv}_{i+1} \ldots \mathtt{l}_n\mathtt{fv}_n \to \mathtt{c}]\!] \quad [\![\mathtt{v}_i \to \mathtt{c}_i]\!]\mathtt{)}\end{array}} \qquad \text{(CPSF-VALUE)}$$

Using the mechanism of CPS-conversion, it is also easy to allow conditional value expressions and CPS-convert them to processes:

$$\begin{array}{c}[\![\mathtt{if:T\ v\ then\ v}_1\ \mathtt{else\ v}_2 \to \mathtt{c}]\!] = \\ \mathtt{(def\ d\ x:Bool\ =\ if\ x\ then}\ [\![\mathtt{v}_1 \to \mathtt{c}]\!]\ \mathtt{else}\ [\![\mathtt{v}_2 \to \mathtt{c}]\!]\ [\![\mathtt{v} \to \mathtt{d}]\!]\mathtt{)}\end{array} \qquad \text{(CPS-IFV)}$$

5.1.2 Value 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:T [n n]) e)` binds `x` to the result of evaluating `(new n:T [n n])` and then executes `e`. Formally, `val` declarations are translated into the core language using the same continuation-passing translation as above:

$$[\![\mathtt{(val\ p\ =\ v\quad e)}]\!] = \mathtt{(def\ c\ p\ =}\ [\![\mathtt{e}]\!]\quad [\![\mathtt{v} \to \mathtt{c}]\!]\mathtt{)} \qquad \text{(CPS-VAL)}$$

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.

Since declarations can also appear in value expressions, we also need to add a clause to the definition of CPS-conversion for values:

$$[\![\mathtt{(val\ p\ =\ v}_1\quad \mathtt{v}_2\mathtt{)} \to \mathtt{c}]\!] = \mathtt{(def\ d\ p\ =}\ [\![\mathtt{v}_2 \to \mathtt{c}]\!]\quad [\![\mathtt{v}_1 \to \mathtt{d}]\!]\mathtt{)} \qquad \text{(CPS-VALV)}$$

5.1.3 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 $\mathtt{(v\ v}_1\ \ldots\ \mathtt{v}_n\mathtt{)}$. For example, if we define a `double` function by

```
def double [s:String r:/String] = +$![s s r]
```

(where +$ 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:

$$[\![\texttt{(v } \texttt{l}_1\texttt{fv}_1\texttt{...}\texttt{l}_n\texttt{fv}_n\texttt{)} \rightarrow \texttt{c}]\!] = [\![\texttt{v![}\texttt{l}_1\texttt{fv}_1\texttt{...}\texttt{l}_n\texttt{fv}_n\ \texttt{c]}]\!] \tag{Cps-App}$$

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.

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

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

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

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.

5.2 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 5.1.3). We replace a process definition of the form

```
def f[a1:A1 a2:A2 a3:A3 r:/T] = 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):T = v
```

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

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:

$$\texttt{\textbackslash a} \Rightarrow \texttt{(def x a x)} \tag{Tr-AnonAbs}$$

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 /[]], 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 /[]] done:/[]] =
 (def loop x:Int =
    if (<= x max) then
      (new c : ^[]
       ( f![x (rchan 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![])
            (rchan 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 = [
  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 continued in Chapter 6.

5.3 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 responsive channels that are used for signalling completion (or some other condition) rather than for exchanging data. The type of `pr`, then, is `![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`.

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

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.

Chapter 6

Simple Concurrent Objects

As an example of many of the derived forms described in the previous chapters, 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`.

```
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 )
```

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 (int.toString (get)));
     (set 5);
     (prNL (int.toString (get)));
     (set -3);
```

```
        (prNL (int.toString (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 (int.toString (get1)));
     (prNL (int.toString (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: /[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![set=set get=get] )

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

run ((ref2.set 5);
```

```
      (prNL (int.toString (ref1.get)));
      (prNL (int.toString (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 = [
  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 ! [
          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
 [
    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, etc.]), seem to arise almost inevitably when programming in a process calculus. They are widely used in Pict's libraries.

Chapter 7

Advanced Language Features

The full Pict language includes a number of features that cannot be treated fully in a short tutorial. This chapter surveys some of the most useful through a series of examples. See the *Pict Language Definition* for complete details.

7.1 Lists

The standard libraries of Pict include support for a variety of common data structures. Among the most important is the library `Std/List`, which defines basic operations for constructing and manipulating lists. For example, the following program constructs a list of integers and then prints its second element:

```
import "Std/List"
val l = (cons 6 (cons 7 (cons 8 nil)))
run print ! (int.toString (car (cdr l)))
```

```
7
```

The first line makes the definitions in `Std/List` available in the current compilation unit. The second uses the functions `cons` and `nil` to construct the list `l`. The third uses the function `cdr` to select the tail of `l`, then `car` to select its head, and prints the integer returned from `car`.

The next program defines a process abstraction that, when sent a list of integers, prints its second element:

```
def print2ndInt [l: (List Int)] =
  if (null l) then
    print!"Null list"
  else if (null (cdr l)) then
    print!"Null tail"
  else
    print!(int.toString (car (cdr l)))

run print2ndInt![(cons 6 (cons 7 (cons 8 nil)))]
```

```
7
```

(We elide the `import` clause from here on, to reduce clutter.)

To prevent the proliferation of parentheses in expressions like `(cons 6 (cons 7 (cons 8 (nil))))`, Pict provides a special syntax for repeated applications of the same function to a sequence of arguments (often called "folding"). The last line in the program above can be written

```
run print2ndInt![(cons > 6 7 8 nil)]
```

In general, (f > a_1 a_2 ... a_n a) means the same as (f a_1 (f a_2 ... (f a_n a))). The expression (f < a a_1 a_2 ... a_n) stands for (f (f (f a a_1) a_2) ... a_n), the analogous "right fold" of `f` over `a` and a_1 through a_n.

7.2 Polymorphism

Of course, we can build lists of values other than integers. For example, the program above can be rewritten to build and destruct a list of strings like this:

```
def print2ndString [l: (List String)] =
  if (null l) then print!"Null list"
  else if (null (cdr l)) then print!"Null tail"
  else print!(car (cdr l))

run print2ndString![(cons > "one" "two" "three" nil)]
```

```
two
```

Indeed, except for the type declaration in the pattern and the channel (`printi` or `print`) used for printing, the definitions of `print2ndInt` and `print2ndString` are identical. We may wish to combine them into a single definition by taking both the type of the elements and an appropriate printing function as parameters:

```
def print2nd [#X l:(List X) p:/[X /String]] =
  if (null l) then print!"Null list"
  else if (null (cdr l)) then print!"Null tail"
  else print!(p (car (cdr l)))
```

The `#` before the parameter `X` indicates that it is a *type parameter*. When this abstraction receives a message, the occurrences of `X` in the types of the parameters `l` and `p` must be replaced, consistently, with whatever type is passed as argument for `X` in the concrete types of whatever values are matched by `l` and `p`. For example, to use `print2nd` on a list of integers, we pass it `Int` for `X` and `int.toString` for `p`.

```
run print2nd![#Int (cons > 6 7 8 nil) int.toString]
```

```
7
```

Similarly, to use `print2nd` on a list of strings, we pass `String` for `X` and the identity function for `p`.

```
run print2nd![#String (cons > "one" "two" "three" nil) \(s:String)=s]
```

```
two
```

For the sake of brevity, the type reconstruction phase of the Pict compiler will attempt to fill in any missing types in an argument list that is passed to `print2nd`.

```
run print2nd![(cons > 6 7 8 nil) int.toString]
run print2nd![(cons > "one" "two" "three" nil) \(s:String):String=s]
```

```
7
two
```

When a type argument is omitted, the compiler attempts to determine its value by examining the types of the rest of the arguments and matching them against the type of the channel on which the arguments are being sent. If it succeeds in determining the omitted type uniquely, all is well. If not, a compile-time error occurs and the programmer must supply the missing type argument explicitly.

(One slightly subtle point illustrated by this example is that, when type arguments are omitted in this way, the typechecker then has less information to work with when synthesizing the types of the other arguments. Here, this shows up in the fact that we are obliged to add the explicit annotation `:String` showing the result type of the identity function, whereas this was inferred automatically before. Details of how all this works can be found in the *Definition*, but there is no need to understand exactly what information is required and what can be omitted: if the typechecker needs to know more, it will indicate exactly where.)

Indeed, we have been using this mechanism all along, whenever we invoked basic operations from the `List` library. Making all the type arguments explicit, list construction actually looks like this:

```
val l = (cons #Int 6 (cons #Int 7 (cons #Int 8 nil)))
```

The fully annotated version of `print2nd` is:

```
def print2nd [#X l:(List X) p:/[X /String]] =
  if (null #X l) then print!"Null list"
  else if (null #X (cdr #X l)) then print!"Null tail"
  else print!(p (car #X (cdr #X l)))
```

7.3 Abstract Types

The kind of polymorphic programming we saw in the previous section is not limited to polymorphic abstractions such as `print2nd`: more generally, Pict allows any tuple (including, of course, tuples with named fields) to include type fields marked by `#` signs; tuple patterns may, in general, include `#` fields as well, introducing dependencies in the types of later fields, as we have seen.

```
val [#X l:(List X) p:/[X /String]]
  = [#String (cons > "one" "two" "three" nil) \(s:String)=s]
val lcdr:(List X) = (cdr l)
run print ! (p (car lcdr))
```

```
two
```

Note that the pattern on the first line here binds not only the variables `l` and `p`, but also the type variable `X`, in the scope of further declarations. In other words (by the ordinary rules of variable binding), the `X` appearing in the types of the bound variables `l` and `p` is a different type from any type mentioned anywhere else in the program.

We can exploit this fact—that patterns with type bindings create fresh types—to build *abstract types* whose elements can be manipulated only by some given set of functions.[1] As, a simple (but

[1]This programming technique directly follows Mitchell and Plotkin's explanation of abstract datatypes in terms of existential types in the lambda-calculus [MP88].

useful) example, the following binding introduces a new *enumeration type* called `Weekday` and two operators, `sameday` and `tomorrow`:

```
val [#Weekday
     monday:Weekday tuesday:Weekday wednesday:Weekday thursday:Weekday
     friday:Weekday saturday:Weekday sunday:Weekday
     sameday:/[Weekday Weekday /Bool]
     tomorrow:/[Weekday /Weekday]]
  =
    [#Int                                {- representation type -}
     0 1 2 3                             {- representations of days -}
     4 5 6
     \(d1:Int d2:Int) = (== d1 d2)       {- same day? -}
     \(d:Int) = (mod (+ d 1) 7)]         {- tomorrow -}
```

After this declaration, we can behave as though `Weekday` were a built-in type with constant elements `monday` through `sunday` and built-in operations `sameday` and `tomorrow`:

```
def weekend(d:Weekday):Bool =
  (|| (sameday d saturday) (sameday d sunday))
```

7.4 User-defined Type Constructors

In Chapter 6, we saw how to define a type `RefInt` of integer reference cell objects and an associated constructor `refInt` for creating new elements of this type.

```
type RefInt = [
  set=/[Int Sig]
  get=/[/Int]
]

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

We now generalize these definitions to a *parametric type* `Ref` that, like `List`, describes a family of types—`(Ref T)` is the type of reference cells holding elements of `T`—and an associated *polymorphic constructor* `ref` for creating reference cell objects. First, the type constructor `Ref` is defined like this:

```
type (Ref X) = [
  set=/[X Sig]
  get=/[/X]
]
```

The pattern `(Ref X)` on the left of the `=` declares the name `Ref` and gives the name `X` to the parameter type (this occurrence of `X` is a binder, whose scope is the type expression on the right-hand side of the `=`). Put differently, `Ref` is a function from types to types such that, for each type `T`, the instance `(Ref T)` means the same as `[set=/[X Sig] get=/[/X]]`.

With this definition in hand, the polymorphic `ref` function is easy to write:

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

The only real difference from `refInt` is that, here, we must take the initial value of the cell as a parameter to `ref`, since we do not have a uniform way to make up a default value of the parameter type `X`.

7.5 Recursive Types

We can combine what we know—parametric type definitions and polymorphic functions—plus one more feature—so-called *recursive types*[2]—to build an object-oriented version of the predefined `List` package. In this version, we will make each list value carry its own interal functions (methods) for the `null`, `car`, and `cdr` operations instead of writing these operations separately and passing the list as a parameter when they are called.

A list whose elements have type `X`, then, will be a record of three functions: (1) `null`, returning a `Bool`; (2) `car`, returning an element of `X`; and (3) `cdr`, returning a list of the same type. Informally, what we want to write is this:

```
type (OurList X) =
  [null=/[/Bool]
   car=/[/X]
   cdr=/[/(OurList X)]
  ]
```

But this definition is not well formed, since it mentions on the right of the `=` the very type constructor `OurList` that is being defined.

To handle such recursive type definitions, we introduce a type constructor `rec` that shows explicitly that `OurList` is recursively defined:

```
type (OurList X) =
  (rec L =
    [null=/[/Bool]
     car=/[/X]
     cdr=/[/L]
    ])
```

The bound variable `L` stands for the whole type `(rec L = ...)` in the type expression to the right of the `=`.

[2]The formulation of recursive types described here is an interim language feature, not a final solution. Although the required mechanisms are fairly simple to describe, we find them quite unwieldy in practice. (In particular, although it is possible in principle to use this mechanism to define *mutually recursive* types, it is too painful to do this in practice.)

The usual technique of making the typechecker automatically infer the required foldings and unfoldings of recursive types is not workable here, due to the complexity of the rest of the type system (in particular, subtyping and type operators). In our view, the best solution would be based on ML's `datatype` mechanism, where recursive types are combined with variants and explicit constructors are used to mark both variant tags and points where folding or unfolding is required.

There are two ways of building lists: the empty list `nil` (we'll call it `ournil` to avoid confusion with the `List` library) and the constructor `cons` (we'll call it `ourcons`). As in the standard `List` library, these are both polymorphic functions, since we need to use them to create lists with elements of arbitrary types. Here is `ournil`:

```
def ournil (#X) : (OurList X) =
  (rec
    [null = \() = true
     car = \[r:/X] = ()
     cdr = \[r:/(OurList X)] = ()
    ])
```

Note the value constructor `rec` here, which shows the typechecker explicitly that the type of the body (a record type) is to be "folded up" into the recursive type `(OurList X)`. This operation must be performed explicitly: the unfolded record type and the folded recursive type are treated as completely distinct by the typechecker. Similarly, here's `ourcons`:

```
def ourcons (#X hd:X tl:(OurList X)) : (OurList X) =
  (rec
    [null = \()=false
     car = \()=hd
     cdr = \()=tl
    ])
```

Using `ournil` and `ourcons`, we can build a list of two integers like this:

```
val l = (ourcons #Int 3 (ourcons #Int 4 (ournil #Int)))
```

Just as we folded up our record values above into recursively typed values, we must unfold our recursively typed value `l` before we can project its fields—if we try to project the `car` field of `l` itself, a typechecking error occurs:

```
run printi!(l.car)
```

```
example.pi:22.14: expected a record type but found: (OurList Int)
```

The required unfolding is accomplished by the `rec` pattern constructor, which matches an element of a recursive type but whose body pattern matches an element of the unfolded body of the recursive type.

```
val (rec ll) = l
run printi!(ll.car)
```

```
3
```

Just as before, we can drop the explicit type arguments to `ournil` and `ourcons` in the construction of `l`:

```
val l = (ourcons 3 (ourcons 4 ournil))
```

Finally, we can use our "function folding" syntactic sugar to reduce the number of parentheses:

```
val l = (ourcons > 3 4 5 6 7 8 9 ournil)
```

Appendix A

Solutions to Selected Exercises

Solution to 1.7.1:

```
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 1.7.2:

```
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 1.7.3:

```
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 4.6.1:

```
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 4.6.2:

```
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 (rchan br)]
         | br?b =
            if b then
              workers?[wor' pri'] =
               ( >>![pri pri' (rchan br)]
               | br?b =
                   if b then
                     ( temp![wor' pri']
                     | -![n 1 (rchan ir)]
                     | ir?i =
                         scan![i wor pri count] )
                   else
                     ( temp![wor pri]
                     | -![n 1 (rchan ir)]
                     | ir?i =
                         scan![i wor' pri' count] )
               )
             else
               ( wor![startNextWorker]
               | restoreWorkers![]
               | -![count 1 (rchan 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 (rchan 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 5.1.3.2:

```
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
```

Bibliography

[Agh86] Gul A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapporte de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[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.

[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.

[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.

[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.

[MP88] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[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.

[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.

[Pie96] Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. Available electronically, 1996.

[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 in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 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.

[PS97] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *Principles of Programming Languages (POPL)*, 1997. Full version available as INRIA-Sophia Antipolis Rapport de Recherche No. 3042 and as Indiana University Computer Science Technical Report 468.

[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.

[PT97a] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, Computer Science Department, Indiana University, 1997. To appear in Milner *Festschrift*, MIT Press, 1997.

[PT97b] Benjamin C. Pierce and David N. Turner. Pict language definition. Draft report; available electronically as part of the Pict distribution, 1997.

[PT97c] Benjamin C. Pierce and David N. Turner. Pict libraries manual. Available electronically, 1997.

[Sew96] Peter Sewell. Observations on Pict, a nondeterministic programming language. Manuscript, 1996.

[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.

[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.