Technical Report

UCAM-CL-TR-65
ISSN 1476-2986

Number 65

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Introduction to the programming language "Ponder"

Mark Tillotson

May 1985

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-65*

Introduction to the programming language "Ponder"

Mark Tillotson
Cambridge University Computer Laboratory

Technical Report
May 1985

Table of Contents

Chapters and sections marked '*' are more involved
and should perhaps be skipped on a first reading

The functional language 'Ponder'

Introduction

This document is primarily an introduction to the programming language 'Ponder'. A section at the end gives details about the VAX™ UNIX™ implementation of Ponder, available from Cambridge.

The language was designed by Jon Fairbairn, at Cambridge, and is an attempt to design a programming language with as few built-in constructs as possible, and yet to be as flexible and expressive as possible.

It is a purely *applicative* language, supporting *higher-order functions* with *normal order evaluation* and a *polymorphic type system*. The power of this approach is demonstrated by the ability to define, in Ponder, constructs (such as **If-Then-Else-Fi** and **Case-Esac**) that would be regarded as integral parts of most other programming languages.

Similarly basic data-types, such as *pairs* and *lists*, can all be defined within the language. The combination of simplicity and straight-forward extensibility give Ponder both the clear semantics desired in a programming language, and great expressive freedom. The model of computation is one that is not sequential, and could thus exploit parallelism, although the current implementations are on sequential machines.

The only qualification required of the reader is a general familiarity with a high-level programming language. Some of the sections (marked with '*') are more advanced, and should perhaps be skipped on a first reading if you are not familiar with the ideas of functional programming. Hopefully the expert reader will not find the pace too slow.

The elements of Ponder

Objects and names

In Ponder, as in other functional languages, each fragment of program has a *value* associated with it. The act of executing a program becomes simply the *evaluation* of the expression denoted by the entire program. There is no concept of a variable in the sense of a value-container, and consequently no notion of assignment.

A program consists of an expression, made up of names and icons (constants) and combined by function application and function definition. This expression is also annotated with *type* information (which is checked for consistency by the compiler).

To make the language easier to use and read, the programmer can define syntax (purely grammatical constructs), such as *infix operators*, to tailor the appearance of the language to the problem in hand (often greatly improving the readability of the program). This is merely a cosmetic facility—one could (if so desired) get by without ever using the operator syntax.

Every value has a type

Superimposed on the basic mechanism of the language (expression evaluation and function representation) is a polymorphic type system, capable of fully checking that the type of each object matchs the context of each of its uses (for instance you may not add a number to a text string). The checking handles functions of arbitrary *order*, and is all performed at compilation time (ie. it is achieved solely by a static analysis of the program text).

Despite being a *strong* type system, it endevours not to restrict the kinds of functions that can be expressed in any way, and largely succeeds. There is a notation within Ponder for expressing types, which tries to provide the most power with the least mechanism.

Thus every object in Ponder has both a value and a type. For the moment, we shall ignore the type system, and consider objects and expressions:

How do we write down an object in Ponder? How do we give objects names? What do they look like?

The four syntactic elements

A Ponder program is made up of a sequence of elements, each one of which is either *an icon*, *a name*, *a bold name* or *a special symbol.*

Icons

Icons (constants) define explicitly the value they represent, and have known type. There are three varieties of icon: *character*, *string* and *integer*. A character icon is written as a single quote mark followed by the character in question:

```
'a      'b      'Z      '@
```

Certain characters (which may be difficult to type in or to read) can be denoted by an 'escape' convention, where instead of using the character itself, another quote mark, followed by some printable character, is used:

```
''n     ''l     ''^     '''     ''"
```

These are the icons for 'newline', 'line-feed', 'bell', single quote and double quote respectively.

A string icon denotes a (possibly empty) sequence of characters, and is written by placing double quote marks around the sequence. Again the characters within the sequence may be escaped:

```
"Hello world'n"
```

In order to allow long strings of characters to be broken up as an aid to readability, there is a further escape convention: A single quote followed by layout (spaces, newlines, tabs etc.) is ignored within a string icon:

```
"Hello '              wor'      ld'n"
```

is the same string as in the previous example, only written a different way.

Integer icons represent non-negative integers, and are written simply as non-null sequences of decimal digits, (the only base allowed), like this:

```
4676232     1     42     00000009
```

Names

The next kind of element is a *name*. Names can be used to denote objects, and there are two ways of doing this (see under declarations). They are written in the following form: An *italic* letter followed by zero or more italic letters, italic digits or italic hyphens. Thus the following examples are all valid names:

abc *a2* *long-name-* *x----1*

Bold names

Bold names and *specials* are used for syntactic purposes, and never represent objects directly (however *operators* represent function objects indirectly). Bold names are similar to names in appearance, except that they are in BOLD ROMAN letters, digits and hyphens. Thus

ABC **A2** **Long-Name-** **X----1**

are valid bold names.

Special symbols

Special symbols are made from characters that are not letters or digits. The characters from which specials can be made fall into two categories, *singletons* and *multiples*. Singletons can only form single character specials, and the singleton characters are:

( ) [ ] { } , ; ∀ → ≜

A multiple is a sequence of one or more of the following characters:

$ % & + − * / = ~ | \ <
> : . ? @ !

Is unusual to want more than about three such characters in a multiple, for reasons of readability. Some of the possible specials have a predefined and unalterable meaning in the language. These are:

( ) [ ] , ; : . ≜ → ∀

Comments

A *comment* is a piece of text included in the program, but not part of the program. Comments are used for the human programmer (or some software tool) to add explanatory or other information where relevant—they are ignored totally by the compiler. In Ponder comments are introduced by an underscore character (_), and continue until the end of the line.

Real character sets

Most computers have a rather restricted character set and, for instance, do not know the difference between italic and bold roman type-faces. This means that there needs to be a convention for representing names, bold names and certain of the specials on real machines. From now on this description will use this convention, and all examples of Ponder will appear in a typewriter-like way (`like this in fact`). Names are represented by a sequence of *lower-case* letters, digits and underscores, starting with a letter. Repeated underscores are treated as equivalent to a single underscore (again for readability). Thus the names introduced before would look like:

```
abc         a2          long_name_
x_1         x_____1
```

Bold names are exactly like names, except that the initial letter **must** be upper case, and subsequent letters **may** be upper case. Note that upper and lower case letters are distinct, so that '`LET`' and '`Let`' are not the same bold name. The same rule about underscores applies. The singletons are represented by:

```
(   )   [   ]          ,   ;
```

and the multiples are taken from:

```
#   $   %   &   +   -   *   /       =
~   |   <   >   :       .       ?       @   !
```

The specials $\forall$, $\triangleq$ and $\rightarrow$ are represented by !, == and -> respectively (unless those characters are actually available in the character set). Comments begin only at underscores that are not part of a name or bold name. To avoid problems, it is wise to have comments start at the beginning of a line, or be preceded by some white-space.

3

Declarations, functions, expression evaluation

Declaring names

A name can be introduced to represent the value of some expression by means of the Let declaration:

```
Let   fred  ==   "the value that fred signifies";
```

The left hand side is either a name, or a sequence of names separated by commas. Thus:

```
Let   one, two, three  ==  1, 2, 3;
```

Having defined a name, it can be used within the rest of the program in place of the expression it represents, without *any* change in the meaning of the program (subject to scope rules)—this property is known as *referential transparency*—equals can be substituted for equals. The ability to be able to give a name to any object in this way is a major source of expressive power. The value of

```
Let  fred  ==  "Let there ";  fred @ "be light"
```

is the same as

```
"Let there " @ "be light"
```

(where @ is the string concatenation operator).

In many computer languages the use of 'abstraction' of this sort, where some entity in the language is given an equivalent name, is not fully general—there are things expressible in the language that cannot be given names. In Ponder any expressible thing can be named.

How functions are represented

A function representation looks like this:

```
Int x  ->  x * x * x
```

which is the representation for the function that takes an integer argument, and returns its cube (assuming * is defined as an infix operator for the standard function '`int_times_int`'). The syntax of function representations is, in general:

Type-spec *name* `->` *expr*

The type specification **Type-spec** is for the purposes of the type checker (which needs to know the type each parameter is supposed to be). The *name* is a *formal parameter* to the function, and it shows where within the *expr* the actual argument goes (whenever the function is applied to something). The result of such a function applied to some expression *expr2* is the value of a copy of *expr* with every occurrence of *name* replaced by *expr2* (subject to the correct interpretation of 'every occurrence'—the problem is when nested function representations use the same name). More precisely, every occurrence of the *name* that is not *within the scope* of an inner function (or '**Let**') declaration, is replaced. More about this later.

One can write a function of several arguments like this:

```
Int x  ->  (Int y  ->  (Int z  ->  x + y + z))
```

'->' associates to the right, so we can omit the parentheses, and write:

```
Int x  ->  Int y  ->  Int z  ->  x + y + z
```

(which is nicer).

It is often useful to package a function up in a name, so that it can be applied to different arguments at different places in the program, without having to write its representation out every time. This is not a problem—since everything is an object (including any function), we simply use the **Let** declaration:

```
Let  add_3_numbers  ==
  Int  x  ->  Int  y  ->  z  ->  x + y + z;
add_3_numbers 4 5 (add_3_numbers 1 2 3)
```

Here we have used '`add_3_numbers`' twice, which has result $4+5+1+2+3$.

Functions in Ponder are evaluated *before* their arguments. This is what *normal order evaluation* means, and it allows you to write functions that never evaluate some of their arguments. This is necessary if conditional functions are to be specifiable in Ponder (a conditional can take a truth value and return one of two other arguments dependant upon the condition, but without *evaluating* both).

The 'raw' syntax of expressions

The 'raw' syntax of Ponder consists of names, icons, and the two operations of function application and function representation, which we have already met. The syntax of these should perhaps be clarified before we procede.

Function application is denoted by writing the function to the left of the argument. The whole application may be enclosed in parentheses if wanted, and this makes it possible to group expressions (with more than one application) in any way. Thus:

```
(abc  def)  ( ( (ghi  jkl)  mno)  (pqr  stu) )
```

Function application associates to the left, so that the middle group could have been written '`(ghi jkl mno)`' without changing the meaning. This convention of associating to the left tends to remove the need for many of the parentheses. The above example could also have been written:

```
abc  def  (ghi  jkl  mno  (pqr  stu))
```

Function representation, on the other hand, associates to the right, so that:

```
Int  x  ->  (Int  y  ->  x / y)
```

and

```
Int  x  ->  Int  y  ->  x / y
```

are the same function representation.

Function representation has a lower grouping power than application, so that where both occur, function representations may have to be enclosed in parentheses:

```
(Int  x  ->  Int  y  ->  x / y)  alpha  beta
```

(if the parentheses were omitted, then we would have a function representation with "`x / y  alpha  beta`" as its body).

Declarations made with 'Let' have the least grouping power of all, so that we can go:

```
Let  fred  ==  Int x  ->  Int y  -> x / y;  fred  alpha beta
```

without using any parentheses at all. Ponder programs usually have only a modest number of parentheses, which makes for good readability. In fact it is possible to define new *bracket operators*, which make '(' and ')' redundant if so desired.

The whole program is an expression

An entire Ponder program is just one big expression, and this will normally be a sequence of 'Let' and other declarations, followed typically by a concise 'main expression' which draws upon these definitions. When the program is compiled, the compiler will look not just at your program, but will read a *prelude file* first, which contains type definitions for the built-in functions provided by the Ponder runtime-system, as well as some useful syntax and function representations. The philosophy is to provide a minimum of useful things in the *standard prelude.*

The compiler compiles the entire program to return something of type 'File-action', which is a special type that the *runtime system* knows about. The runtime system takes the compiled version of the Ponder program, and evaluates the program's main expression, and uses the file-actions that result as instructions to affect the 'outside world'. Typically this will involve putting some text on your terminal, or to a file. More about input and output can be found in a later section, and it is ignored for now.

Evaluation of a function applied to an argument

```
(Int x -> Int y -> Int z -> x + y + z) 4 5 23
```

The above expression consists of the function (that we have just met) applied to 4, 5 and 23. One should think of the process of evaluation of the expression as follows: The 4 is an actual argument for the formal parameter, x. It is taken up and substituted for every occurrence (that is in scope) of x within the *body* (right hand side) of the function:

```
(Int y -> Int z -> 4 + y + z) 5 23
```

The process is repeated with the two other arguments to give:

```
(Int z -> 4 + 5 + z) 23
(4 + 5 + 23)
32
```

The idea of scope

It is important to know where in a program the definition of a given name has meaning—what its *scope* is. Typically a program will appear as a list of declarations, separated by ';', and with an expression at the end that denotes

the result of the program, making use of the definitions. Now declarations associate to the right:

```
Let  x  ==  12;
 ( Let  y  == 34;
   ( Let  x  ==  x + y;   x * y
   )
 )
```

(Parentheses have been added to make explicit the scope of the declarations). Note that the scope of a declaration does **not** include its own right-hand-side, nor does it include the scope of any inner declaration which uses the *same name*. Thus in the above example, the value of the whole program is 46 * 34, because `x * y` is in the scope of `x = 46` and `y = 34`. This is in turn because '`Let x == x + y; ...`' is in the scope of `x = 12` and `y = 34`.

At first sight it may seem as though it is impossible to define a name recursively (in terms of its own value) in Ponder. There is, however, a special form of **Let** used for recursive definitions, namely '**Letrec**'. (It is not strictly necessary, since there *is* a way to define recursive objects using just **Let**).

```
Letrec  so_on  ==  so_on;
```

This is a definition for a name directly in terms of itself, and if you try to make use of the value of '**so_on**' then you will have to wait forever, since the value of '**so_on**' is defined to be '**so_on**', which is defined to be '**so_on**', *ad infinitum*. This example is not quite correct Ponder—a type ought to be specified.

Recursive definition, once mastered, is a very powerful tool for writing programs, and is the natural way to specify any form of repetition in a applicative language.

More about functions

We have met functions, but not in full detail. For instance how do you write the definition of a recursive function? How is the *type* of a function written?

How to write the type of a function

The type of a function that takes an argument of type 'X' and returns type 'Y' is written:

```
X  ->  Y
```

Note that the use of '->' here is actually different from its use in a function representation, but that it is natural for the syntax of these two uses to look the same.

Functions are just normal objects

It is important to be clear that functions are not special kinds of objects, they are normal objects with just the same 'status' as any other. It will be harder to realise the full import of this if you are only accustomed to programming languages where this is not the case. There is no restriction at all as to whether you may pass functions as parameters to other functions, return them as results, or put them in data structures. The only restrictions are those imposed by the type system, which effectively means you have know what you are doing, and thus use each function in a consistent manner.

Functions take only one argument

Functions in Ponder are very simple in that they all take just one argument, and return one result. This the very simplest model of functions that can exist.

For instance the function:

```
Int  x  ->  Int  y  ->  x*x - y*y
```

is really regarded as two functions, each taking one argument:

```
Int  x  ->  (Int  y  ->  x*x - y*y)
```

where the outer function itself returns a function (the inner one).

Functions take any number of arguments

Having just said that functions always take one argument, it should be made clear that you don't have to think about functions in this way, since defining functions that mimic functions of several arguments is straightforward, as the above example illustrates.

Sometimes you will want a function that takes a variable number of arguments in different circumstances. This cannot be done directly as a Ponder function. What you should do in this case is pass the arguments wrapped up in some suitable data structure, such as a *list* or a *union*.

Recursive functions and how to define them

A recursively defined function is one whose definition makes use of its own definition. This is, of course, what '**Letrec**' is for:

```
Letrec  gcd  ==  Int  x  ->  Int  y  ->  Int:
  If  x < y
    Then  gcd y x
  Elif  y = 0
    Then  x
    Else  gcd y (x Rem y)
  Fi;
Letrec  diff  ==  Int x  ->  Int  y  ->  Int:
  If  x < y  Then  diff y x  Else  x-y  Fi;
```

The first of these examples is a function for returning the greatest common divisor of two integers, and the second returns the absolute difference between two integers. There are both defined in terms of themselves, and both use a conditional (the **If-Then-Else-Fi** syntax) to test for the simple case. For a recursively defined function ever to return a result, there has to be some condition which allows it to return immediately and avoid infinite recursion.

Often you will find that instead of just one recursive function, you will want to define several recursive functions, mutually. The way to specify such a group of functions is to use the multiple form of declaration with '**Letrec**'.

```
Letrec  f, g  ==
  ( Int n  ->  Int:    g (g n) ),
```

```
( Int m  ->  Int:    f (g (f m)) );
```

The example is not very meaningful, but shows the syntax needed. Note that the precedence of ',' compared to '->' means that the function representations themselves need parentheses, and that the types have to be *cast*—made explicit—for each function, so that the type-checker can analyse the definition.

Higher-order functions

A higher-order function is one that returns some other function as its result, or returns a higher-order function as result.

Since it is possible to pass functions as arguments and results freely, we can write functions that return functions that have been built up from simpler functions.

```
Let  twice  ==  (Int -> Int) fn  ->
  (Int n  ->  fn (fn n));
```

'twice' is something that takes an integer function, **fn**, and returns another integer function, being **fn** applied twice in succession.

```
Let  quad  ==  twice square;
Let  identity  ==  twice minus_int;
```

Here a higher order function is used to define, very compactly, functions in terms of other functions.

To be fully usable, the type checker must allow useful higher-order functions to have correct types. This mechanism is *polymorphism* and it is described in the section on types.

Higher order functions are very useful for doing things in a systematic manner to a whole data-structure. A higher-order function, which knows about traversing the structure in question, is passed a second function which knows about the type of object that resides in the data structure. The higher-order function can apply the function to every part of the structure, as required. The higher-order traversal function need know nothing of what is in the structure. The other function need know nothing of the actual data-structure—convenient separation is thus obtained.

Types

The type of an object, roughly speaking, is information saying what kind of value it may have, and what operations are possible on that kind of value.

An object of type 'Int' thus might have the value 3 or -27, but not "Hello". The operations on integers include multiplication, but 'String's cannot be multiplied in the conventional sense. Type-checking involves ensuring that objects in the program are used only as they should be—types of functions and arguments are always consistent (You are not allowed to say '3 * "Hello"'—unless you have added a new meaning to '*').

For types to be consistent, the type of the argument in each function application must be that that is required by the function, or something "more general". Thus if 'print_int' is of type 'Int -> String', only the first of these three expressions is correctly 'typed':

```
(print_int  7)
(print_int  "32")
(print_int  print_int)
```

Types met so far

There are some built-in types that the compiler knows about, because it has to deal with icons of these types—'Int', 'Char' and 'String'.

The type of a function whose argument is of type 'A' and whose result is of type 'B' is denoted by the type-expression:

```
A  ->  B
```

For functions of several arguments '->' associates to the right, as in function representations. Thus the type of

```
Int x  ->  Bool y  ->  (Char -> Bool):        ...      ...
```

is 'Int -> Bool -> Char -> Bool'. Since the order in which a function takes its arguments is significant, 'Int -> Bool -> Int' and 'Bool -> Int -> Int' are distinct types.

Casts

In Ponder, any expression may be written with a 'cast' type, this being an assertion that the type of the expression is the one written. Casts do not enable the type-system to be bypassed. Sometimes casts are needed to allow the type checker to start analysis, as in some recursive definitions. Often it improves readability, and it is good style to make the result type of every function explicit with a cast.

The syntax of a cast is very simple:

Type : *expression*

and the type acts a bit like a prefix operator, but note that there is no effect on the value of the expression, it is solely an operation on types. As an example of a cast, I shall redefine the '**so_on**' example from one of the earlier examples:

```
Letrec  so_on  ==  so_on;
```

would not only not terminate, it would not pass the type-checker. For the type of '**so_on**' to be determined, the type-checker wants to know the type of everything inside its defining expression (which includes '**so_on**').

The way to allow the type checker to handle this is:

```
Letrec  so_on  ==  Int:  so_on;
```

(though there is no reason why such an object should be limited to integers).

Polymorphic types

So far we have learnt of types for expressing functions that take arguments of known type. However there is a whole class of powerful and useful functions that take arguments of *arbitrary* type, or of a range of types subject to some *constraint*. An example would be the function that takes a list of *things*, and a predicate (truth-valued function) for things of the kind in the list, and and returns 'true' if every thing yielded 'true' to the predicate. Another example is the function 'middle_one' which takes three items of some type, and a comparison function for that type, and returns the median (middle) of the three items.

Languages like Pascal make the definition of such functions possible, but do not allow a type to be specified sufficiently precisely to enable the type system to detect invalid uses of such functions. However, Ponder, like other

polymorphically typed languages, allows such types to be defined and type checked. Here are the type-expressions for the two functions just described:

```
!T.  List [T]  ->  (T -> Bool)  ->  Bool
!T.  T  ->  T  ->  T  ->  (T -> T -> Bool)  ->  T
```

(remember that '!' is really meant to be a '∀' symbol, pronounced '*for all*').

In these types a *universally quantified* type name, (here T), was introduced by ∀, meaning that the type name T can stand for any type at all in the subsequent type expression. If such a quantified type name is used more than once, then it means that the type (whatever it may be) is the same in all occurences. A more complicated type expression, using two quantified types might be:

```
!M, N.  (M -> N)  ->  (N -> M)  ->  Int
```

It is possible to specify very complex relationships between the types in such type-expressions, and the type-checker will endevour to let through all types that are meaningful.

Type generators and declarations

We have now seen the various kinds of types that can be specified, but we do not yet have a way to name types, or a way of defining types in terms of themselves (recursively). Ponder provides both of these with the *type generator* declaration. A type generator is written as a bold name optionally followed by a list (in square brackets) of parameter types.

Type generators may be in defining or applied occurrences. When type generators are defined, their parameters are simply bold names, which are type-variables introduced for the scope of the definition. When type generators are applied, the parameters may be any type expressions, and these are effectively substituted into the definition to yield the type denoted by the applied generator.

```
List [T]
```

might be a defined *or* an applied occurrence of the single-parameter type-generator '**List**'.

```
Set [Int -> Int]
```

is an applied occurence of the type generator '**Set**'.

There are two forms of type-generator declaration, recursive and non-recursive:

```
Type  Binary_Op [T]  ==  T -> T -> T;
```

is a non-recursive definition of the type of a general binary operator. For instance the type of '`int_plus_int`' can be written '`Binary_Op [Int]`'. An example of the recursive form of generator definition is:

```
Rectype  List[ Thing]  ==
         Option [Pair [ Thing, List [ Thing]]];
```

This defines '`List [`*Thing*`]`' to be an optional pair of *Thing* and a `List` of *Thing*. `Option` and `Pair` are applied type generators which are assumed to be defined, and are in fact standard types in Ponder.

Recursive types of this sort can be used to define infinite data types, as in this example. There is, however, a restriction placed on such definitions: Any occurrence of the type generator (being defined) on the right-hand side of the declaration must look *just* like the left-hand side of the declaration. For instance the type definition of `List` given was valid, whereas the following definition is not:

```
Rectype  Funny [T]  ==  ...    Funny [List [Char]]  ...    ;
```

The restriction means that type generators that are defined in terms of themselves must be defined *exactly* in terms of themselves. This restriction makes type-checking tractable in the general case.

The definition of type 'Pair' *

Earlier it was mentioned in passing that the type '`Pair`' can be defined within Ponder. We have now met enough Ponder to be able to do this. The following way is by no means the only possible one, and may be hard to follow, but it does illustrate how data types can be defined in a functional type system.

```
Type  Pair [Left, Right]  ==
      !U.  (Left -> Right -> U)  ->  U;
```

Which only makes sense, when the following definitions of the primitive operations on pairs are defined in terms of '`Pair`'.

```
Let  pair  ==  !L, R. L left -> R right ->  Pair [L, R]:
       (!U. (L -> R -> U) select -> select left right);
Let  left  ==  !L. R.  Pair [L, R] p  ->  L:
       p (L left  ->  R right  ->  L: left);
Let  right  ==  !L, R.  Pair [L, R] p  ->  R:
       p (L left  ->  R right  ->  R: right);
```

The type 'Pair' is represented by a function that takes three things: a left part, a right part, and an unpacking function. The pair cannot be evaluated until an unpacking function is supplied, so it can freely be passed around as an object until we want to get at the inside of the pair. Two unpacking functions, 'left' and 'right', are provided for this. Throughout the definitions suitable polymorphic types are given to allow pairs of any two types.

A similar set of definitions can be made for any data type, such as *unions* and *records*. It is often simplest to define records in terms of a few pairs, in order to avoid too many verbose type and selector definitions. With this in mind the *standard prelude* contains a definition for '><' as a type infix operator for 'Pair'. One can, therefore, use types like:

```
Int  ><  Bool  ><  Bool  ><  List [String]
```

to provide 'records'. (The >< symbol is meant to look like the Cartesian product operator, $\times$)

The multiple declaration again

This multiple form of declaration is implemented by the Ponder compiler knowing about the type 'Pair'. It goes against the philosophy of Ponder somewhat to have such things as 'Pair' built in to the language in this way, but the syntax is very useful, allowing, for instance, the definition of mutually recursive functions to have a reasonably neat appearance. The example we met before of multiple declarations, namely

```
Let  one, two, three  ==  1, 2, 3;
```

is equivalent to:

```
Let  three  ==  1, 2, 3;
Let  one    ==  left  three;
Let  two    ==  left  (right three);
Let  three  ==  right (right three);
```

where comma has special meaning on the left hand side, but is a normal infix operator for 'pair' on the right-hand side. In both circumstances it is right-associative.

The complete syntax for type specifications *

This is included here for reference. It is informal, and only covers the type expressions and declarations in Ponder.

```
type              ::=  quantifier  type  |  type_map
```

```
quantifier      ::=  '!'   bold_name_list  '.'
type_map        ::=  solid_type  |
type_map  type_infix_op  type_map
solid_type      ::=  type_name  |  '('  type  ')'  |
applied_type_gen
applied_type_gen  ::=  bold_name  '['  solid_name_list  ']'
type_decl       ::=  'Type'  generator_name  '=='  type  |
'Rectype'  generator_name  '=='  type
capsule_decl    ::=  'Capsule'  type_decl
seal_capsule    ::=  'Seal'  bold_name
generator_name  ::=  bold_name  |
bold_name  '['  bold_name_list  ']'
type_infix_decl  ::=  'Typeinfix'  type_infix_op
'=='  bold_name
bold_name_list  ::=  bold_name  |
bold_name  ','  bold_name_list
solid_type_list ::=  solid_type  |
solid_type  ','  solid_type_list
type_name       ::=  bold_name
new_operator    ::=  bold_name  |  special
type_infix_op   ::=  bold_name  |  special
```

Some interesting types

The type generator 'Option'

This is actually a built-in type, but it could be defined in the prelude if one wanted. The concept is very simple: an option of some type (T) is a type that is either one of the values of T (it is present), or 'nil' (in which case it is absent). Hence the definition of `List` we saw earlier as:

```
Rectype  List [T]  ==  Option [Pair [T, List [T]]];
```

It is quite common to have some data structure in which some component may be absent, and '`List`' is only one of many types that can be built using '`Option`'. There are several functions and a useful piece of syntax defined to be used with options:

```
null   ==  !  T.  Option [T]  ->  Bool  _ (true if nil)
```

```
opt_in  ==  !  T.  T  ->  Option [T]     _ (makes an option)
opt_out  ==  !  T.  Option [T]  ->  T    _ (extracts the
_ value of type T, or fails)
If   <value of type  Option [T]>      _ (if nil, then the
Is    <function   T  ->  S>          _  else-part is
Else  <some value of type S>         _  returned,
Fi                              _  otherwise the function
_  is applied to the value)
```

The type of a list length function

The type of a function that takes a list of items and returns the number of items in the list:

```
!T.  List [T]  ->  Int
```

Such a function might be defined as follows:

```
Letrec  length  ==  !T.  List [T] list  ->  Int:
  If  list  Is  T head, List [T] tail
    ->  1 + length tail
    Else  0
  Fi;
```

A tree of nodes and leaves

If 'Node' is the type of objects to go in the nodes of a binary tree, and 'Leaf' is the type for the leaves, then we can express the type of an arbitrary tree of such objects (using the standard infix forms of 'Pair' and 'Union') as:

```
Rectype  Tree [Leaf, Node]  ==  Leaf  \./
   (Tree [Leaf, Node]  ><  Node  ><  Tree [Leaf, Node])
   \./  Dot;
```

(All unions are expressed as

```
A  \./  B  \./     .....          \./  Dot
```

where Dot denotes the end of the union list.)

The natural numbers from first principles

By making use of the properties of type-generator 'Option' one can simulate the non-negative integers:

```
Rectype  Natural  =  Option [Natural];
```

'Natural' may either be nil (zero) or the successor to another 'Natural'. We can use such a type to perform (very inefficiently) arithmetic operations:

```
Let  is_zero  ==  Natural  n  ->  null n;
Let  successor  ==  Natural  n  ->  Natural:
  opt_in  n;
Let  previous  == Natural  n  ->  Natural:
  opt_out  n;
Letrec  nat_plus_nat  ==  Natural  a  ->
                          Natural  b  ->  Natural:
  If  a  Is  Natural  a_minus_1
    ->    nat_plus_nat  a_minus_1  (successor b)
    Else  b
  Fi;
Letrec  nat_eq_nat  ==  Natural  a  ->
                        Natural  b  ->  Bool:
  If  a  Is  Natural  a_minus_1
    ->
      If  b  Is  Natural  b_minus_1
        ->    nat_eq_nat a_minus_1 b_minus_1
        Else  false
      Fi
    Else  false
  Fi;
```

Lambda Calculus *

λ-calculus was devised by the mathematician Alonzo Church circa 1930. It was an attempt to produce a theory of mathematical functions, and a notation for all computable functions. It is very simple and very general, and is computationally as powerful as all other schemes of computation.

Syntax of the lambda calculus

It has a very simple syntax (indeed it can be thought of as purely textual, since its semantics are so simple): A *λ-expression* is either a *variable name*,

fred

or an *application*,

(*lambda-expr*1 *lambda-expr*2)

(where *lambda-epxr1*, *lambda-expr2* are any λ-expressions)
or an *abstraction*

(λ*name* . *body-expr*)

(where *name* is a variable name, and *body-expr* is a lambda-expression forming the body of the *lambda function*)

Semantics of the lambda calculus

The evaluation of a lambda expression is defined in terms of a process of *reduction*, which can be informally described as follows: the actions outlined in the next paragraph are performed repeatedly indefinitely, or until a point is reached where no further reduction can be done:

Find the left-most abstraction, **a**, in the current lambda-expression. If it occurs *applied* to some lambda-expression, **e**, then do the following: Replace the whole lambda expression by one in which **a** is overwritten by a copy of its body in which every *free* occurrence of its bound variable is replaced by **e**.

By *free* occurrence it is meant one that properly goes with the abstraction in question, and does not belong to an inner abstraction which happens to use the same variable name, as in this example:

$$\lambda a \,.\, (b\, a\, (\lambda a \,.\, a\, b))$$

This is exactly equivalent in effect to

$$\lambda a \,.\, (b\, a\, (\lambda c \,.\, c\, b))$$

Ponder semantics in terms of lambda calculus *

In order to clarify the evaluation of Ponder programs the various constructs we have met will be explained by giving rules for conversion from Ponder into equivalent lambda-expressions. Note that all type information is lost in this transformation.

The let declaration

The Ponder form below:

`Let` *name* `==` *expr* `;` *expr2*

translates to the lambda-expression:

$$(\lambda name \,.\, expr2)\quad expr$$

Function application

The general case of one expression applied to another:

expr *expr2*

translates to the lambda-expression:

$$(expr\quad expr2)$$

Function representation

The function representation:

Type *name* `->` *expr*

translates to the lambda-expression:

$$(\lambda name \,.\, expr)$$

Note the equivalence between introducing a name with '`Let`' and with function representations:

```
Let  n  ==  e1 ;  e2
```

is exactly the same in meaning as

```
( Type  n  ->  e2 )  e1
```

Casts

The cast in Ponder:

Type : *expr*

translates to the lambda-expression:

expr

Recursive definitions with 'Letrec'

This can be defined in terms of a special lambda-expression:

`Letrec` *name* `==` *expr* ; *expr2*

translates to

$$(\lambda name\ .\ expr2)(Y\,(\lambda name\ .\ expr))$$

where

$$Y \quad = \quad \lambda f\ .\ f(\ (\lambda g\ .\ f(g\ g))\ (\lambda g\ .\ f(g\ g))\ )$$

(You might like to entertain yourself by checking that this actually works!)

Evaluation order and laziness

Recall the object I defined earlier, called '**so_on**'. I have not yet written it with the most general type it could have:

```
Letrec  so_on  ==  !T.  T:  so_on ;
```

Here it has any type at all. In fact there are no useful *objects* of this type, since there are no values that are general enough to passed to *any* function. Since '**so_on**' will never finish evaluating, and thus never yield a value, it is type-correct for it to be passed to any function—it cannot yield a type-incompatible value, and thus can be of type '`!T. T`'.

Normal order and applicative order

Note that in many programming languages the act of passing an argument to a function causes the argument to be evaluated. Thus something that behaved like '**so_on**' could not be passed as an argument without provoking a non-terminating computation—even if the value were not going to be used.

Defining "If-Then-Else"

Also note that in these programming languages it would quite alright to say '`IF condition THEN so_on ELSE other`'. If the condition were false then '**so_on**' would not be evaluated. In a functional language everything is done with functions, so we need to be able to define such a 'conditional function', which takes a condition (of type `Bool`) and two alternatives of the same type. It should return the appropriate alternative *without* attempting to evaluate the other.

It is clear that such a function could not be defined *by the programmer* in a language such as Pascal. Thus the conditional *has* to be an integral part of these languages. Ponder, however, was designed so that things like conditional functions could be defined by the programmer. For this reason it had to have an evaluation strategy that did not evaluate arguments before they were passed—evaluation of arguments is done by *functions*, and only if they *need* to know the values.

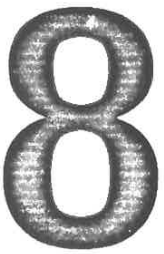
This is the *normal order* evaluation scheme, also termed 'call by need'. It is an established fact that (in programs where things like '**so_on**' are being

passed around) normal order evaluation will lead to program termination if such termination is possible—in other words normal order means that there will *never* be an evaluation of an unnecessary expression.

Lazy evaluation

We thus have a strategy that avoids evaluation of things that never need be evaluated, but it is possible to do better than this—it would be advantageous to performance, never to evaluate the same expression twice. In the scheme of *fully lazy evaluation* the expressions that are passed around as arguments to functions are overwritten with their values whenever some function first requires to evaluate them. Subsequent uses of such expressions see the value rather than the unevaluated expression, and thus all further evaluation is bypassed.

For such a scheme to work there is an all-important condition—there shoud be no *side-effects* in the language. This ensures that the value of each expression is the same each time it is evaluated, and that evaluation of one expression cannot alter the value of any other expression. By using overwriting in this way, program evaluation can be seen to be similar to simplifying a mathematical expression, in that each stage in the process produces a new version of the expression, which is equivalent to the previous version, and can thus replace it. The evaluation of a Ponder program always tends away from unevaluated expression, and towards 'hard' data, ie characters for output to the outside world. The need to output such data is used as the 'driving force' for evaluation.



Defining prefix, infix and bracket operators

We have already seen infix operators, such as + and *, and it is very easy to define your own, or to add new meanings to ones already defined (unless they are specials or bold names that are built-in to the language, such as 'Letrec' or '->').

'Overloading' operators

You may associate as many functions with the same prefix, infix or bracket operator as you wish.

The way the compiler decides which definition to use when it meets the use of an operator is very simple - it uses the latest definition that is type-correct. For instance you can have '=' defined to mean some test for 'equality' for several types, and then use '=' throughout the program—the compiler will select the appropriate definition for you, if one exists, and the program will be more concise and readable as a consequence.

Prefix operators

A prefix operator is ones which act on one item, which it immediately precedes. Thus when say 'twice seven' we are using 'twice' as a prefix operator in English. Prefix operators in Ponder (unlike English) have more binding power than any infix operator, and more binding power than function application. Thus to apply a prefix operator to some expression, you should place the expression in parentheses. The way to define a bold name or special to be a prefix operator is this:

```
Prefix    symbol  ==   some expression ;
```

The expression will need to be a some function (taking at least one argument).

Infix operators: priority and association

To be able to use a bold name or special as an infix operator, it must have both a *priority* and an *association direction* assigned to it. A priority is a number

between 1 and 9, and signifies the relative grouping power of the operator relative to other infix operators. In the normal rules for arithmetic, * has a higher grouping power than +. *Larger* priority-numbers mean *less* grouping power, equal priority implies equal grouping power. When the same infix operator occurs next to itself (as in `x + y + z`) then it can either *associate* to the *left* or to the *right* (+ associates to the left, of course).

There is a construct for defining the priority and left/right associativity of a new infix operator, and it goes like this:

```
Priority  5  Rem  Associates  Left;
Priority  6  +    Associates  Left;
Priority  8  =    Associates  Left;
Priority  8  ::      Associates  Right;
```

There should only be *one* such definition for each infix operator, and the default is priority 1, left associativity.

There can be any number of declarations that give a meaning to the operator, by associating a function with it:

```
Infix  +  ==  Int -> Int -> Int:    int_plus_int;
Infix  =  ==  Int -> Int -> Bool:    int_eq_int;
Infix  =  ==  Char -> Char -> Bool:    char_eq_char;
```

It is also possible to define an infix operator for a type-generator with the following declaration:

`Typeinfix` **Symbol** == **Bold-Name** ;

Any function bound to an infix operator must, of course, take at least two arguments. A type generator can only be bound to an infix operator if it has precisely two parameters.

Bracket operators

Bracket operators are like prefix operators, in that they apply to one expression, but they syntactically consist of two parts *surrounding* the expression. This means that you do not have to add ordinary parentheses round the expression, and it can improve the readability—the correspondance between matching opening and closing brackets can be accentuated. In other respects bracket operators are just like prefix operators. Here is how you might define '`Begin-End`' in Ponder:

```
Bracket  Begin  End  ==  !T. T  t -> t;
```

(where '`t -> t`' is an identity or 'do nothing' function).

Definitions in the standard prelude *

While you can write you own prelude, and ignore the definitions provided by the standard one, it is probably not useful to do this. There are certain built-in functions of the runtime system that need to be defined in dummy declarations before they can be used, and certain types must be declared, such as Bool, File-action, before the language is very usable.

The following is a list compiled from the standard prelude, and it briefly lists everything in the prelude.

Types

`Char`, `Option`, `Int` and `Pair` are all types which are known to the compiler, but the following types and operations are defined in the prelude:

```
Bool                    _ with operations 'not', 'And', 'Or'
Union [A, B]
String  ==  List [Char]
File_action  ==         _ something hidden
```

Operators and type operators

The prefix operators (together with the functions bound to them) that are defined in the prelude are:

```
=   char_eq_char    <   c char_lt_char     >   c char_gt_char
~=  char_ne_char    <=  c char_le_char     >=  c char_ge_char
=   int_eq_int      <   c int_lt_int       >   c int_gt_int
~=  int_ne_int      <=  c int_le_int       >=  c int_ge_int
-     minus_int
Not   not_int
Not   not
```

The function 'c' used above is a function that swaps over the roles of the two arguments that follow it—called the 'C combinator'. Its representation is:

```
Let  c  ==  !Tx, Ty, Trf.     (Ty -> Tx -> Trf) fn  ->
                              Tx  x  ->
                              Ty  y  ->   Trf:  f y x;
```

The infix operators defined in the prelude are:

```
O     compose
=     int_eq_int        <     int_lt_int        >     int_gt_int
~=    int_ne_int        <=    int_le_int        >=    int_ge_int
=     char_eq_char      <     char_lt_char      >     char_gt_char
~=    char_ne_char      <=    char_le_char      >=    char_ge_char
+     int_plus_int      -     int_minus_int     *     int_times_int
%     int_over_int      Rem   int_rem_int
&               int_and_int
|               int_or_int
Shift_left      int_shift_left_int
Shift_right     int_shift_right_int
::                    list
@           append
Elem        elem
Else    Then    Elif    Is      And     Or
In      Out     *>      ||    _  These are all to provide syntax
_  see the section on case clauses
```

The function 'compose' that is bound to 'O' is function composition, and has the representation:

```
Let  compose  ==  !A, B, C.  (B -> A) ba  ->
                             (C -> B) cb  ->
                             C        c   ->
      ba (cb c);
_  (f O g) x    is thus equivalent to    f (g x)
```

The bracket operators defined in the prelude are:

```
If      Fi      ==  !T. If_clause [T]  if  ->  T: if
Case    Esac    ==  i
Begin   End     ==  i
$<      $>      ==  i
```

The type-infix operators defined in the prelude are:

```
><          Pair
\./         Union
```

Syntax

The syntactic constructs defined in the prelude are:

```
If      condition     _ (As many 'Elif's as you like)
  Then    expr1
Elif    condition2
  Then    expr2
  Else    expr3
Fi
 If    option type  Is   function  Else   default  Fi
            _  The above can look very readable, as in:
       If  list  Is  Char first, String rest
        ->  first ::  first ::  fn rest
        Else  nil
     Fi
 Case    expr
 In     predicate1    *>     expr1
  ||     predicate2    *>     expr2
  ||     ...     ...     *>   ...     ...
  ||     ...     ...     *>   ...     ...
  Out    default
Esac
 Case    union-typed-expr
  In   is_1   *>     Type1    t   ->     expr1
  ||   is_3   *>     Type3    t   ->     expr3
  ||   ...     *>   ...      ->   ...      ...
  ||   ...     *>   ...      ->   ...      ...
  Out    something else
Esac
 Begin                 _ or  $<
   ...     ...
   ...     ...
   ...     ...
End                   _ or  $>
```

The two 'case' constructions are worthy of further discussion, and there is a section dedicated to this later on.

Useful functions

The functions provided by the prelude are the following, listed with any operator

they are bound to. The bodies of many of these functions have been omitted, and it is left as an exercise to the reader to fill them in:

```
_ The identity function
Let  i  ==  !T. T  t -> t;
_ function composition
Let  compose  ==  !A,B,C. (B -> A) ba ->
(C -> B) cb ->
C   c ->  ba (cb c);
_  The 'C' combinator
Let  c  ==  !A, B, C. (B -> A -> C)  f  ->
A        a  ->
B        b  ->  C : f b a;
_  boolean not
Let  not  ==  Bool a  ->
If  a  Then  false  Else  true  Fi;
_  appending lists
-
_          append  (a, b, c)  (d, e)
_       =  (a, b, c, d, e)
Letrec  append  ==  !T. List [T]  list1  ->
List [T]  list2  ->  List [T]:
_  mapping a function onto every element of a list
-
_         map  funct  (a, b, c, d)
_      =  (funct a, funct b, funct c, funct d)
Letrec  map  ==  !A, B.  (A -> B) function  ->
List [A] list       ->  List [B]:
_ gathering lists together with binary operations
-
_        right_gather  op  end  (a, b, c)
_     =  op a (op b (op c end))
-
_        left_gather  op  start  (a, b, c)
_    =   op (op (op start a) b) c
Letrec  right_gather  ==  !A, B.  (B -> A -> A)  bin_op  ->
A   initial ->  A:
Letrec  left_gather  ==  !A, B.  (B -> A -> B)  bin_op  ->
B   initial ->  B:
Let  gather  ==  right_gather;
_  filter  -  takes a predicate and a list, and returns only
```

```
_                 those members that satisfy the predicate
Letrec  filter  ==  !A.  (A -> Bool)  predicate  ->
List [A]   list        ->  List [A]:
_  reverse a list
Letrec  reverse  ==  !A.  List [A] list  ->  List [A]:
_  length of a list
Letrec  length  ==  !A. List [A] list  ->  Int:
_  Take the first n elements of the list
Letrec  first  ==  Int  n  ->
!A. List [A] list  ->  List [A]:
_  Extract the n'th element of a list
Letrec  elem  ==  Int  n  ->  (!A. List [A]  ->  A):
_  Determine the position of a character
_  within a string (0 if not present)
Let  position_of_char_in  ==  String  str  ->
Char  char  ->  Int:
_  converting integers to decimal strings for printing
Let  digits  ==  "0123456789";
Let  print_int  ==  Int  n  ->  String:
_  input and output functions:
Let  terminal_input_list  ==  String :
opt_out  (get_file  "*");
Let  string_to_terminal   ==  String  str  ->
File_action :
opt_out  (make_file  "*")  str;
Let  print_to_terminal    ==  String  str  ->
List [File_action] :
string_to_terminal  str  ::    nil;
```

10

Generality of types, 'Capsules'

The relation 'is more general than'

Now it has been explained that a program is type-correct if, for every function application, `(expr1 expr2)`, `expr1` has type `A -> R`, `expr2` has type `B`, and `B` "is more general than" `A`.

This is only roughly what happens in type-checking—there really is a relation between types, "more general than", but the result type of a function application may actually be infered from the function and argument types:

```
Let  demo  ==  !A, B.  A aa ->  (A -> B) ba ->
  .........     ;       _ demo has type   !A,B. A -> (B -> A)
 Let  fn  ==  Int i  ->  Bool:
  ......  ;            _ fn has type      Int -> Bool
 demo  12  fn          _ is deduced as    Bool
```

Some of the properties of the "more general" relation, which I shall denote $\subseteq$, are:

$$A \subseteq A, \text{ for all types } A$$

$$!T.T \subseteq A, \text{ for all types } A$$

$$A \text{ -> } B \subseteq C \text{ -> } D \iff B \subseteq D \wedge A \supseteq C$$

If the last rule seems counter-intuative as regards the direction of the relation between tha arguments, remember that the relation, $\subseteq$, should be read "can be used as an argument instead of".

The more formal definition of the relation is not included here—see the Technical Report on the language if you are interested in this:

PONDER AND ITS TYPE SYSTEM, Cambridge University Computer Laboratory, Tech. Report, 31.

A NEW TYPE CHECKER FOR A FUNCTIONAL LANGUAGE, Cambridge University Computer Laboratory, Tech. Report, 53.

Sealing definitions in a capsule

There is a facility in Ponder for 'sealing' type definitions within a 'Capsule'. This allows the definition of abstract types to be kept 'locked up' in such a way that the objects and functions defined inside are the only ones that can take advantage of the internal implementation of the type (in terms of other types). It is a mechanism to provide a means of disciplining the use of a type, and so should be used where possible.

Various types that we have met are defined in capsules in the standard prelude, including 'Bool', which may only be manipulated by the operations of *and*, *or*, *not*, and is the only type that may appear within the **If-Then-Else-Fi** syntax as a condition. It is not possible, short of redefining 'Bool', to make use of its actual definition, which is:

```
Capsule Type Bool == !T. T -> T -> T;
```

This is the syntax for opening a capsule, with the keyword 'Capsule' just before a 'Type' or 'Rectype' declaration. A capsule must be sealed after all the definitions relevant to it have been made—otherwise they might 'spill' over into the rest of the program. To seal a capsule type, one goes:

```
Seal Bool;
```

You can define any number of capsules simultaneously, and they do not have to nest in any particular way—all that is required is that you seal each capsule that you open, for safety.

The effect of sealing a capsule is to disconnect the type name concerned from its definition, and thereby make every function and object defined in terms of the named type *also* disconnected from the definition of the sealed type. This has the required effect of abstracting the type from the details of its definition, and prevents you from subsequently exercising any inside knowledge you may have of the sealed type's internals.

Input and output in Ponder

Functional languages, without the notion of updatable variables, do not map particularly well into the world of conventional filing systems, where the contents (or value) of files are inherently changeable.

The most straightforward way to fit a functional language onto such a conventional filing system is to regard the whole program as a function from one state of the filing system to a new state of the filing system (in which some of the files have been updated).

Fileactions

This is the approach taken by the Ponder system. The whole Ponder program has type '`List [File_action]`'. A file-action is conceptually a function from one state of the filing system to new state, in which one of the files has been changed in some way (either 'made', 'appended' or 'deleted'). Thus when a Ponder program is run (evaluated), the file system is mapped from its original state to a state in which the file-actions (that are the result of the program) have all been applied, in sequence, to the original state.

There is also a means by which a Ponder program can examine the contents of a file (from the *original* state of the filing system).

Thus a Ponder program can alter files in a way dependant on their original contents, thereby allowing fully general input and output. I shall now introduce the functions that are built into Ponder and its runtime system which provide this form of input and output.

Reading the contents of a file

The first such function is that provided for input from a file, called '`get_file`'.

```
Let  get_file  ==  String  name  ->
                   Option [String]:     ....   ;
```

This is a function that takes a file name (a String), and returns an option of its contents. If the filename is valid, and the file can be read, then the contents is returned as an 'opted-in' string, else 'nil' is returned.

There are some special meanings to certain filenames: '`get_file "*"`' returns the standard input as its result, rather than the contents of a given file. '`get_file "**"`' returns the latter half of the command-line-option-string as its result (if the implementation supports such things).

Altering files

There are several functions that return file-actions for use by the program.

```
Let  make_file  ==  String name  ->
                    Option [String  ->
                    File_action]:  ....  ;
Let  append_to_file  ==  String name  ->
                         Option [String  ->
                         File_action]:    ....    ;
Let  delete_file  ==  String name  ->
                      Option [File_action]:      ....    ;
```

'`make_file`' takes a string, which it interprets as a filename. It returns nil if that file cannot be made, else it returns a function from string to file-action, which will be able to put a string into the file in question, overwriting any previous contents (well, probably).

'`append_to_file`' is very like '`make_file`', except that the file-action concerned is one that appends the string to the end of any existing file.

'`delete_file`' is a function that takes a filename and returns an optional file-action for deleting the file, which is only returned if the file is likely to be successfully deleted.

To make life easier there are several functions defined for using the standard input and output character streams:

```
Let  terminal_input_list  ==  String:
   opt_out (get_file "*");
Let  string_to_terminal   ==  String str  ->
   opt_out (make_file "*") str;
Let  print_to_terminal    ==  String str  ->
   string_to_terminal str ::  nil;
```

One could also define the following function to access the option string (this is not included in the prelude though):

```
Let  option_list   ==  String:    opt_out (get_file "**");
```

Due to certain 'difficulties' in fortelling the future state of a real multi-user filing system, and the possibility of files being 'protected' or 'in-use', the implementation cannot guarantee that every file-action issued will succeed.

The implementation is, however, true to the notion of the filing system changing 'atomically' when the program is run—'`get_file`' can never see a file that is created by the program, because all output files are written to hidden temporary files, and these are put in their proper place after the program has stopped running.

To demonstrate the use of the functions, here are couple of small, but type-correct Ponder programs:

```
print_to_terminal
  (opt_out (get_file "fred")  @  terminal_input_list)
opt_out (make_file  "temp") "Some contents for temp"  ::
opt_out (delete_file "temp")
```

12

Case-clauses and Unions

Since there are two distinct versions of the 'Case-Esac' syntax defined in the standard prelude, and they are fairly complex, this whole section is devoted to them. The simplest kind switches upon the value of some expression, according to a series of predicates.

The other kind of case clause uses predicates that return *Options* rather than Booleans. In fact both the predicates and the right-hand-sides are functions.

An extension of the latter kind of case clause (by defining appropriate predicates), allows a union-case-clause, which selects according to the actual type of some union-valued expression.

The boolean-predicate version looks like this:

```
Case   e
  In   p1  *>  v1
  ||   p2  *>  v2
  Out    v3
Esac
```

and this syntax ('In, ||, *>, Out' are infix operators) is shorthand for:

```
( switchon_choice  p1
   ( v1 ,
      ( switchon_choice  p2
         ( v2 , out_part  v3 )
      )
   )
)    e
```

switchon_choice takes a predicate, and a pair containing a value and a function, and a selecting value. If the predicate returns true for the selecting value, then the value from the pair is returned, else the result is the function applied to the selecting value—continuing the case selection. out_part is a function that is used to terminate the chain of tests: it simply returns the default value, whatever the selecting value is.

The definitions of the various operators and functions involved (taken from the standard prelude) are:

```
Let  switchon_choice  ==  !Sel.     (Sel -> Bool)  pred ->
                    Res thing1,  (Sel -> Res)  thing2  ->
                                 Sel  selecting_value  ->
  If  pred  selecting_value
    Then   thing1
    Else   thing2  selecting_value
  Fi;
Priority  9  In  Associates  Right;
Priority  9 Out  Associates  Right;
Priority  9  *>  Associates  Right;
Priority  9  ||  Associates  Right;
Let  out_part  ==  !Sel.    Sel  selecting_value  ->
                   !Res.    Res  default_result   ->
                default_result;
Infix  In   ==  !Sel.    Sel  selecting_value      ->
                       (Sel -> Res) case_clause  ->
             case_clause  selecting_value;
Infix  Out  ==  !Sel, Res.    Sel previous_value  ->
                            Res default          ->
             previous_value , default;     _ wraps up in a Pair
Infix  *>  ==  switchon_choice;
Infix  ||  ==  !A, B.  A a -> B b ->  a, b;
Bracket  Case  Esac  ==  i;
```

The second kind of case-clause uses a different kind of 'predicate' and 'right-hand-side'—the syntax is the same. The function 'option_choice' is used instead of 'switchon_choice':

```
Let  option_choice  ==
    !Sel, W, Res.    (Sel  ->  Option [W])  pred  ->
                     (W  ->  Res)  func          ->
                     (Sel -> Res)  case_clause  ->
                     Sel   selecting_value     ->  Res:
      If   pred  selecting_value   Is   W  which
        ->
          func  which
        Else
          case_clause  selecting_value
      Fi;
```

```
Infix  *>  ==  option_choice;
```

These definitions are very flexible, and by defining option-valued functions from 'Union' types to options of a member type, one can achieve selection dependant on the type and value of a Union. The actual definitions for doing this are hard to follow, so I only include an example of its use:

```
Type  Un ==  Alpha \./ Beta \./ Gamma \./ Delta \./ Dot ;
Let  u  ==  String  str  ->  Un :
   .........
   .........  ;
Case  u "some string"
  In  is_1  *>  Alpha a  ->    thing
  ||  is_3  *>  Gamma g  ->    another thing
Out     yet another thing
Esac
```

Union types

There are some functions defined for creating unions, 'in_1, in_2, in_3' being functions to take something of a member type, and turn it into union type. The functions 'next_in next_is' are available to generate as many of 'in_x is_y' as required:

```
is_7  ==  next_is  is_6
in_4  ==  next_in  in_3
```

and so forth.

When a union of some types is specified, there should be a 'dot' at the end of the list:

```
Type   Fred  ==  Tom \./ Dick \./ Harry \./ Dot;
```

The consequence of this 'dot' is that the same union-injection functions are available to every kind of union—which is useful.

Extracting the type from a Union object should be done by the appropriate kind of case-clause.

13 Example programs

Some short examples

These two examples are simple functions, the first of which is in the standard prelude, and is a list-manipulating function: The function that takes a list and a function, and returns a new list in which each item of the original list has been passed to the function.

```
Let  map  ==  !Old, New.    (Old -> New) function  ->
              (List [Old] -> List [New]):
  Begin
    Letrec  map_function  ==  List [Old]  old_list  ->
                              List [New]:
      If  old_list  Is  Old first,  List [Old] rest
        ->
          function  first  ::    map_function  rest
        Else
          nil
      Fi;
    map_function
  End;
```

This illustrates several points of programming style. The definition of `map_function` is hidden inside the definition of `map`, and passed as result. Otherwise `function` would have to be an explicit parameter in the recursive definition, and this creates clutter.

Use of the `If ... Is ... -> ... Else ... Fi` syntax for the testing of an optional data type saves testing for `nil` with `null`.

Definition of `map` as a higher order function, such that no list parameter is passed to it, and it instead returns a function requiring a list.

Use of a cast to allow the first line of the function to indicate its type clearly—this reduces the need for comments explaining the circumstances under which a function may be used.

Reversing letters in words

This is a program to read the input, treat it as a series of words (separated by newlines and spaces), and maps it to the output with every word reversed.

```
Letrec flatten == List [String] wordlist -> String:
  If wordlist Is String word, List [String] rest
    ->
      If rest Is String word2, List [String] rest2
        ->
          word @ " " @ flatten rest
        Else
          word
      Fi
    Else
      nil
  Fi;
Let parse == String input -> List [String]:
  Begin
    Letrec parser == String input ->
                     String word ->
                     List [String] words -> List [String]:
      If input Is Char first, String rest
        ->
          If first = ' Or first = ''n
            Then
              parser rest nil (word :: words)
            Else
              parser rest (first :: word) words
          Fi
        Else
          reverse words
      Fi;
    parser input nil nil
  End;
print_to_terminal ((flatten O parse) terminal_input_list)
```

Sorting

Here are two different sort programs, 'bubble-sort' and 'quick-sort', to show how one can do such (traditionally imperative) algorithms in a functional way.

```
Letrec  print_int_list ==  List [Int] l  ->  List [Char]:
  If   l  Is  Int first,  List [Int] rest
    ->
      print_int first  @  ", "  @  print_int_list rest
    Else
      "'n"
  Fi;
Letrec  get_min  ==  List [Int] l  ->  List [Int]:
  If  null l
    Then  nil
    Else
      Let  h, t  ==  head l, get_min (tail l);
      If  null t
        Then l
        Else
          Let  ht, tt  ==  head t, tail t;
          If  h < ht
            Then  h ::  t
            Else  ht ::  h ::  tt
          Fi
      Fi
  Fi;
Letrec bubble_sort ==  List [Int] l  ->  List [Int]:
  If  null l
    Then  nil
    Else
      Let  nl  ==  get_min l;
      Let  h, t  ==  head nl, tail nl;
      h ::  bubble_sort t
  Fi;
print_to_terminal (
  print_int_list (
    bubble_sort (
      10::  9::  8::  7::  6::  5::  4::  3::  2::  1
  )))
```

Here is quicksort:

```
Let  get_median  ==  List [Int] l ->
  If  l  Is  Int m, List [Int] rest
    ->
```

```
      opt_in  (filter (< m) rest,
               (m::  filter (= m) rest),
               filter (> m) res)
    Else
      nil
  Fi;
Letrec  quick_sort  ==  List [Int]  l  ->  List [Int]:
  If get_median l Is List [Int] l,List [Int] m,List [Int] r
    ->
      quick_sort l  @  m  @  quick_sort r
    Else
      nil
  Fi;
```

It is worth noting how concise such a program can be, aided partly by the availability of flexible list-operations, and partly by the user-defined syntax.

A desk calculator

This is quite a large program, but will be given in full, since it demonstrates how to write an interactive program, and because you may want to extend it to produce a version of your own.

```
__ A reverse polish desk calculator
Let  number_of_digit  ==  Char ch  ->
 $<
    Let  p  ==  position_of_char_in digits ch;
    If  p = 0
      Then  10
      Else  p - 1
    Fi
 $>;
Let  bell       == ''^;
Let  backspace  == ''b;
Let  rubout     == ''d;
Priority  5  Is_in  Associates  Right;
Infix        Is_in  ==  Char x  ->
                        String s  ->
                        position_of_char_in s x > 0;
_ read an item, reflecting characters as we go.
_ uses valid_characters and terminators to decide
```

```
_ whether or not to accept a character.
Letrec  read_item  ==  String so_far        ->
                       String valid_chars   ->
                       String terminators   ->
                       String input         ->
                       String >< String >< String:
_    (   Item >< remainder of input >< reflections    )
 $<
    Let  ch, rest  ==  head input, tail input;
    If  null input
      Then  so_far, input, nil
    Elif  ch Is_in terminators
      Then  so_far, input, nil
    Elif  ch Is_in valid_chars
      Then
        Let  new, rest, ref  ==
          read_item (ch ::  so_far) valid_chars
                                    terminators
                                    rest;
        new, rest, (ch ::  ref)
    Elif  (ch = rubout) And (Not null so_far)
      Then
        Let  new, rest, ref  ==
          read_item (tail so_far) valid_chars
                                  terminators
                                  rest;
        new, rest, (backspace ::  ' ::  backspace ::  ref)
      Else
        Let  new, rest, ref  ==
          read_item so_far valid_chars terminators rest;
        new, rest, (bell ::  ref)
    Fi
 $>;
_ But that produces the item backwards, and involves
_ a variable that the user needn't know about,
_ so redefine it:
Let  read_item  ==  String valid_chars  ->
                    String terminators  ->
                    String input        ->
 $<
```

```
    Let  item, rest, ref  ==
      read_item nil valid_chars terminators input;
    reverse item, rest, ref
 $>;
_ this assumes that its argument only contains digits:
Letrec  convert_string_to_int  ==  Int so_far  ->
                                   String s  ->  Int:
  If  null s
    Then  so_far
    Else  convert_string_to_int
            (10 * so_far + number_of_digit (head s))
            (tail s)
  Fi;
Let  string_to_int  ==  String s  ->  Int:
  convert_string_to_int 0 s;
Let  read_int  ==  String terminators  ->
                   String input        ->
                   Option [Int] >< String >< String:
  Begin
    Let  item, ref, rest  ==
      read_item digits terminators input;
    If  null item
      Then  nil
      Else  opt_in (string_to_int item)
                _ this is certainly only digits
    Fi, ref, rest
  End;
_ Now get down to the real programme:
Letrec  factorial  ==  Int n  ->  Int:
  If  n < 1
    Then  1
    Else  n * factorial (n - 1)
  Fi;
Let  null_operations   ==  " 'e'l";
Let  operators         ==  "+-*%=pdqs!:'n'r";
Let  valid_characters  ==
     null_operations @ operators;
Let  print_stack  ==  List [Int] stack  ->  String:
  gather (append 0 (Int item -> print_int item @ "'n"))
         "[end]'n"
```

```
          stack;
_ Perform the operation associated with the character 'op'
_ on the appropriate elements of the original_stack to get
_ a String of reflected characters, and a new stack:
Let  perform_operation  ==  Char op  ->
                    List [Int] original_stack  ->
                    String >< List [Int]:
  If  op Is_in null_operations
    Then  (op::  nil), original_stack
  Elif  Not (op Is_in operators)
    Then  "'^", original_stack
  Elif  null original_stack
    Then  "'n++ Stack empty'n", original_stack
    Else
      Let  right_op, stack  ==
        head original_stack, tail original_stack;
      If  op Is_in "='n'r"
        Then  ('= ::  ''n ::
                  (print_int right_op @ " ")),
              original_stack
      Elif  op = 'p
        Then  "pop!'n", stack
      Elif  op = 'd
        Then  "dup!'n",
              (right_op ::  original_stack)
      Elif  op = 's
        Then  ("stack contents:'n" @
                print_stack original_stack),
              original_stack
      Elif  op = '!
        Then  ('!  ::  nil), (factorial right_op ::  stack)
      Elif  null stack
        Then  "'n++ Only one item on stack'n",
               original_stack
        Else
          Let left_op, stack  ==  head stack, tail stack;
            If  op = '+
              Then  "+", (left_op + right_op ::  stack)
            Elif  op = '-
              Then  "-", (left_op - right_op ::  stack)
```

```
            Elif  op = '*
              Then  "*", (left_op * right_op ::  stack)
            Elif  op = '%
              Then
                If  right_op = 0
                  Then  "Can''t divide by zero'n",
                        original_stack
                  Else  "%" , (left_op % right_op ::  stack)
                Fi
            Elif  op = ':
              Then
                If  right_op = 0
                  Then  "Can''t remainder by zero'n",
                        original_stack
                  Else  ":", (left_op Rem right_op ::  stack)
                Fi
              Else  "'^", original_stack
            Fi
      Fi
  Fi;
_ The calculator itself:
Letrec  calculate  ==  String input  ->
                       List [Int] stack  ->
                       String:
  If  input  Is  Char first,  String rest
    ->
      If  first = 'q
        Then  "quit'nCalculator finished'n"
      Elif  first  Is_in  digits
        Then
          Let  n, rest_of_input, reflection  ==
            read_int valid_characters input;
          reflection @ calculate rest_of_input
                       If  n  Is  Int value
                         ->  value ::  stack
                         Else  stack
                       Fi
        Else
          Let  ref, stack  ==
            perform_operation  first  stack;
```

```
            ref  @  calculate  rest  stack
  Fi;
Let  initial_stack  ==  nil;
print_to_terminal ("Reverse Polish calculator'n" @
       calculate terminal_input_list initial_stack)
```

14

Implementation Techniques *

The lambda-calculus can be directly implemented by simulation of the process of textual substitution, augmented with mechanism for dealing with types like integers and lists in a special way. Such textual implementations are bound to suffer from severe inefficiency if naïvely done.

If you are interested in the implementation techniques for functional languages, then the following papers will be useful in outlining the approachs and compilation strategies.

A NEW IMPLEMENTATION TECHNIQUE FOR APPLICATIVE LANGUAGES, D Turner, Software Practice and Experience, Vol 9 (1979), 31–49.

SUPER-COMBINATORS: A NEW IMPLEMENTATION METHOD FOR APPLICATIVE LANGUAGES, R Hughes, Proc. 1982 ACM Symposium on Lisp and Functional Programming, 1–10.

NON-STANDARD INTERPRETATION AND OPTIMISATING TRANSFORMATIONS FOR APPLICATIVE LANGUAGES, Alan Mycroft, PhD Thesis, University of Edinburgh, CST-15-81 (1981)

STRICTNESS DETECTION IN THE PONDER COMPILER, Stuart Wray, University of Cambridge Computer Lab. Techical Report, 1985 (to appear).

15 Support for Separate Compilation *

The current implementations support separate compilation of libraries of declarations by means of a 'prelude' mechanism.

A prelude is a program consisting of declarations without a main expression—or more accurately one in which the main expression is simply the special keyword 'HOLE'. Each prelude needs to identified by a prelude-name:

```
PRELUDE_NAME  some_name ;
```

Definitions following 'PRELUDE_NAME' (up to the next prelude-name) are associated with that name. There can be several prelude-names within the same program if you wish.

When a proper program or prelude is being compiled, other preludes maybe refered to by putting lines of the form:

```
_ PARENT "file/name"
```

at the start of the program. One can view this like 'GET' in BCPL or '#include' in C, except that the inclusions must be at the start of the program, and the implementation is such that no prelude is ever 'included' twice. There is another dissimilarity with a textual inclusion mechanism—no code is generated for the preludes—they must have been compiled previously so that their code can linked up after compilation.

To recap, the way to compile a program "three" with library preludes "one" and "two" is:

```
Contents of file "one":
PRELUDE_NAME  one;
... declarations ...
...
HOLE
Contents of file "two":
_ PARENT  "one"
PRELUDE_NAME  two;
... further declarations ...
...
PRELUDE_NAME  second_part_of_two;
```

... further declarations ...
...
HOLE
Contents of main program "three":
_ PARENT "one"
_ PARENT "two"
... declarations of main program ...
...
... main expression ...

Note that in the hypothetical example given, prelude "one" should be included for two reasons, both "two" and "three" want it. However it will only be included once.

The mechanism of separate compilation is partly done by the compiler—which knows about prelude-names, and partly by a preprocessor, which interprets the '_ PARENT' comments.

16

VAX™ UNIX™ Ponder users' guide

This is a guide for anyone that wants to know how to drive the compiler for Ponder on the VAX under UNIX, using the Cambridge implementation.

Compiling a program

Before creating and compiling Ponder programs, you will need to set up some sub-directories within your working directory, since the commands that compile and run Ponder programs assume files will be in these sub-directories. They are called "`Program`", "`Lambcode`", "`Object`" and "`Run`", and they are for Ponder programs in source form, lambda-code form, object-module form and linked form respectively. Keeping files in directories like this makes your work directory less cluttered, and makes it easier to keep things tidy.

There is a command "`mkponder`" which is provided to set up these directories within the current directory, and it creates any of the directories mentioned that do not already exist.

New Ponder programs are text files, and you should create them in the "`Program`" directory (they can be in sub-directories of "`Program`").

To compile a program called "`Program/fred`" all you have to type is "`ponder fred`". This produces a lambda-code file, an object file, and finally a linked-executable file, in the appropriate directories, all with the name "`fred`". (Unless, of course, the compilation failed)

Running the compiled version of "`fred`", which will be in "`Run/fred`", is simple—just go "`Run/fred`", and the shell will execute it with the standard input/output attached to your terminal.

Doing things in stages

If you like, you can do the compilation in stages (maybe you just want to check if the program compiles, without going to the expense of code generation and linking. To compile only, the command to use is "`cponder`".

Code-generating a lambda-code file (the output from the compiler), involves using the "`cg-ponder`" command, which leaves the resulting object-module in "`Object`".

Linking an object file to the runtime system, creating an executable version of the program, is done by means of the command is "`link-ponder`". All of these commands follow the convention that you need only quote the name of the program, and not the directory, since this is known. Indeed if you try something like "`link-ponder Object/fred`" it would not work, because it would look for a file "`Object/Object/fred`"! Most of the time you will want to compile all in one operation, as this is less trouble. There is a need, however, to keep the filespace usage under control:

Keeping things tidy

It is useful to know the relative sizes of the various kinds of file: both lambda-code and object modules are related to the size of the original program, and thus large programs will have large object modules, and so forth. Object modules are rather large by comparison with more conventional languages, since the machine architecture is not well suited to the common operations of a combinator reduction machine.

The linked executable files in "`Run`" all include a copy of the entire runtime system, which is around 60K bytes in size (or 35K when 'stripped'), and it is thus wise to keep the number of "`Run`" files down to a minimum—it is very quick and cheap to regenerate them with the "`link-ponder`" command. Perhaps you could include "`rm Run/*`" in your logout script.

Pipes, options and tracing

It is very simple to redirect the standard input and output of a Ponder program, using the normal UNIX C-shell pipe operators. For instance you could use a Ponder program, called "`filt`" as a filter thus:

`cat` *somefile* `| Run/filt | something-else`

The runtime system realises when the standard input/output is to a terminal or not, and adjusts its I/O buffering to take this into account—for file or pipe I/O the unit of transfer is a few hundred bytes (for efficiency), for terminal I/O the terminal is put into character-by-character mode, and the unit of I/O is a single byte. The special UNIX keys recognised when a Ponder program is run are the break character, the "stop-job" character and the literal escape (control-V).

You can pass an option string from the command line to the Ponder program, if you so wish. The expression '`get_file "**"`' returns this option string, and the string returned consists of all the command arguments after a lone '`-`' in the command line, each separated by a single space character.

If there is no lone '-' then the option string is deemed not to be present, and '`get_file`' will return `nil`, since it has result type '`Option [String]`'.

There are several options that are passed to the runtime system, and they control its behaviour in some way. There are 4 numerical arguments that are to do with controlling the allocation of memory for the heap and stack, and examples of their use are:

```
-s12        (set up 12K byte stack)
-h2000      (set up 2Mbyte heap size)
-i30        (set aside 30K bytes spare for file-names,
             buffers etc)
-m50        (designate 50K to be the Minimum actual
             heap size)
```

The defaults for these parameters should normally be adequate, though you may want to increase the heap size from its default of 500K. The stack defaults to the same size as the heap.

There are options to control how 'verbose' the garbage collector is. Normally it says nothing, but '`-v`' makes it verbose, and '`-r`' makes it report (not so much output as when verbose).

There is an option to turn on 'tracing'. When 'tracing' the runtime system outputs a line or two at the start of the run to say how much memory it has, and also outputs a line or two at the end saying how many times the garbage collector was called, how many *combinator reductions* were performed, and similar information. Also if some abnormal end to the run occurs (other than keyboard interrupt), then a *back-trace* of the *stack* at the time of the error will be printed, instead of just the error message. (This may help you to sort out where the error occurred, but may not, since lazy evaluation makes it hard to follow the meaning of this backtrace.) The option to turn tracing on is '`-T`'.

The kinds of errors that can occur at runtime include undefined operations, such as division by zero, taking '`head`' of a null list, doing an '`opt_out`' of nil, but may also be a system *signal* of some kind. It is possible that this is a result of bug in the runtime system or code-generator, and if you think this is so please see the 'reporting problems' section below about who to contact. It may be some other cause, though, such as a UNIX resource allocation becoming exhausted.

17

Installing the system

The standard Ponder system for 4.2 version Berkeley VAX UNIX systems comes on a single magnetic tape. This contains an archive (tar format) of a directory called "Ponder". It is intended that this be recreated as "/usr/local/Ponder", though it does not have to go there (you will need to edit the command scripts if you put it somewhere else).

There are a set of command scripts in "Ponder/commands", and this directory should be added to the PATH variable in the login script of Ponder users.

The rest of the stuff includes the executable versions of the Ponder compiler and code generator, and the ready-to-link version of the runtime system. All references to these files are via the command scripts.

There are some example programs in "Ponder/Program", together with their runnable forms in "Ponder/Run".

The standard prelude is in "Ponder/Standard-prelude/ponder", there is a more commented and readable version in

"Ponder/Standard-prelude/minprel"

and "Ponder/Standard-prelude/makeprel".

Reporting problems

The people to contact are Mark Tillotson (mt21@camsteve) or Jon Fairbairn (jf@camjenny), Cambridge University Computer Laboratory, Corn Exchange Street, Cambridge, CB2 3QG.

It is intended that we wait sometime for problems to be reported, so that we can mend them all before releasing any modified versions.