Technical Report

UCAM-CL-TR-75
ISSN 1476-2986

Number 75

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Design and implementation of a simple typed language based on the lambda-calculus

Jon Fairbairn

May 1985

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 1984 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Gonville & Caius College.

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

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

Despite the work of Landin and others as long ago as 1966, almost all recent programming languages are large and difficult to understand. This thesis is a re-examination of the possibility of designing and implementing a small but practical language based on very few primitive constructs.

The text records the syntax and informal semantics of a new language called Ponder. The most notable features of the work are a powerful type-system and an efficient implementation of normal order reduction.

In contrast to Landin's ISWIM, Ponder is statically typed, an expedient that increases the simplicity of the language by removing the requirement that operations must be defined for incorrect arguments. The type system is a powerful extension of Milner's polymorphic type system for ML in that it allows local quantification of types. This extension has the advantage that types that would otherwise need to be primitive may be defined.

The criteria for the well-typedness of Ponder programmes are presented in the form of a natural deduction system in terms of a relation of generality between types. A new type checking algorithm derived from these rules is proposed.

Ponder is built on the λ-calculus without the need for additional computation rules. In spite of this abstract foundation an efficient implementation based on Hughes' super-combinator approach is described. Some evidence of the speed of Ponder programmes is included.

The same strictures have been applied to the design of the syntax of Ponder, which, rather than having many pre-defined clauses, allows the addition of new constructs by the use of a simple extension mechanism.

Acknowledgements

I am indebted to the following for reading proofs of this thesis: Bill Clocksin, Inder Dhingra, Jackie Hawkins, Pete Hutchison, William Stoye, Sarah Woodall and Stuart Wray. All mistakes are, however, my own work. Particular thanks are due to Mike Gordon, my supervisor, who has tolerated my eccentricities throughout my research and who read the earlier drafts that I didn't dare show anyone else.

Thanks also to the Computer Laboratory for a friendly (and pollen-free) environment, and to the Science and Engineering Research Council of Great Britain for funding the research.

Contents

Foreword

In 1949 work was completed on EDSAC [OANAR 1949], a computer that had 2kbytes of store, consumed 12kW of electricity, occupied an entire room and cost approximately £40 000 for the components alone. Today it is possible to buy a machine with a greater memory capacity, which is battery powered and will fit in a pocket, for around £80†. By comparison, the development of software technology has been slow. Fortran became available around 1956, and Lisp and Algol in 1960. Unlike the computers of 1956 or 1960, Fortran and Lisp are still in use. The development time for programmes has decreased little, so the cost of the computer has become insignificant beside that of the programme.

The development of programmes is slow because programming languages are inadequate. What constitutes a good language? Something is good if it serves its purpose well. At first thought, one might assume that the only purpose of a programming language is to codify an algorithm for interpretation by a computer. But the programme that is written once by a human, read once by a machine and then used heavily is an rare oddity. Human nature predisposes us towards the re-use of things, rather than the creation of new things. Even if a problem is so simple that it can be coded correctly in one attempt it is likely that the programme will be used again for some other purpose. Most programmes are read by humans, either to discover why they do not work, or to see whether they will perform some new task. A good programming language must communicate an algorithm to a human reader with the minimum possibility of confusion. A good programming language will allow the application of old solutions to new data without fuss.

Although none of the readily available programming languages achieves this, I have been asked "Surely Fortran is suitable for scientific programming?" sufficiently often that I feel obliged to mention it. If Fortran were suitable, scientists would use it for the communication of formulae and descriptions of processes. Instead they use mathematical notation. Mathematics is more concise, and easier to read than Fortran.

The conclusion is that programming languages should resemble mathematical notation. Recent exploration of this idea has resulted in the concept of functional programming. Unfortunately, the implementation of functional languages has not yet reached a level of efficiency that would make them a commercially viable alternative to 'traditional' languages. One part of this thesis describes new techniques that go some way towards efficient implementation.

The majority of this thesis is devoted to the design of a simple programming language. An objection to conventional languages is that they are hard to comprehend. I do not mean that programmes written in these languages are necessarily unintelligible (although this is often a consequence). The languages themselves are incomprehensible. This is the result of the tendency of designers to construct languages from the best bits of previous languages, without regard for the coherence of the whole. Current functional languages suffer less from this, but nonetheless are often ad-hoc collections of goodies. My intention has been to design a language in which complexity is replaced by flexibility, but which allows the

† This comes to about 36/- for components when converted to 1949 values.

use of a notation similar to mathematics. A consequence of this design methodology is that the core of the language is small and readily susceptible to formal description.

An aspect of programming that is different from mathematics is that a programme must be completely rigorous and formal, because a computer will eventually interpret it. This can be a disadvantage in that where a human might overlook a simple mistake because it is nonsense, the computer will go ahead and produce nonsense. The same rigour can be turned to advantage if we impose a discipline of data types, and have the computer check that programmes adhere to it.

An argument that a minimal programming language is viable would carry no weight if it could not be implemented. My research is therefore divided into two parts: the design of a minimal language called Ponder, and its implementation. Accordingly this thesis is divided into four parts:

(*i*) an introduction to the concepts of functional languages and to Ponder in particular,
(*ii*) a description of the design process that lead to the present language,
(*iii*) a description of the implementation of Ponder, and
(*iv*) a critical examination of the result.

1

Introduction

Background

This chapter presents the basic concepts of functional programming, together with a survey of previous work in this area. A more rigorous treatment of the mathematical aspects of the calculi can be found in [Barendregt 1980].

1. Foundations

1.1. λ-calculus

1.1.1. Functions

The term *function* in functional programming corresponds closely with the mathematical idea of a function. A function is a recipe that can be used to calculate a *result* when given some *arguments*. An example is the function that expresses the relationship between the volume of a sphere and its radius. Traditional mathematical notation has no explicit representation for functions; we might write $v(r) = \frac{4}{3}\pi r^3$ as a definition of the function v, but have no means of writing down the value of v.

In his 1941 paper [Church 1941], Church proposed the λ-calculus as a formal notation for functions. In this notation the function for v is written $\lambda r.\frac{4}{3}\pi r^3$. The '$\lambda r.$' indicates that r is the name of the *parameter* of the function, and is said to *bind* r. A function can be *applied* to an argument, the notation for which is juxtaposition. For example $(\lambda r.\frac{4}{3}\pi r^3)3$ is an expression for the volume of a sphere of radius 3.

Although the notation is for functions of one argument, the use of functions whose result is a function allows us to simulate functions with any number of parameters. The volume of a rectangular parallelpiped is a function of three parameters: its length, breadth and width. In the usual notation $v_{\text{box}}(l, b, w) = l \times b \times w$. This can be represented in λ-notation as $v_{\text{box}} = \lambda l.\lambda b.\lambda w.l \times b \times w$. An application appears thus: v_{box} 10 20 15 is an expression for the volume of a box 10 by 20 by 15. In this application v_{box} is applied to 10, and the result is a function for the volume of a box of length 10 in terms of its breadth and width, which is then applied to the subsequent arguments. The representation of a function of more than one argument in this way is termed Currying after Haskell B Curry, although Frege [Frege 1960] seems to have anticipated the idea.

1.1.2. Free and Bound Variables

In the function $\lambda w.l \times b \times w$, w is the name of the parameter, and is bound. The names l and b are not bound, and are said to be *free*. If we add $\lambda X.$ at the beginning of an expression, all the free occurrences of X become bound, but bound occurrences are unaffected. Thus in the expression $\lambda x.x$, x is bound, and so in $\lambda x.\lambda x.x$ the initial λx binds no instances of x.

For historical reasons the names of parameters are also referred to as *variables*.

1.1.3. Reduction

We can interpret λ-expressions as rules for the computation of the result of the application of functions. The major step in the application of a function to an argument is the substitution of the argument for all the free occurrences of the parameter in the body of the function. To apply $\lambda x.x + x$ to 3, substitute 3 for free occurrences of x in $x + x$ (which is the body in this case) to get $3 + 3$. Notice that in the application of $\lambda x.\lambda x.x$ to 3 there are no free occurrences of x in the body, so the result is $\lambda x.x$. This rule of substitution is called *β-reduction*.

Although β-reduction does all the real work, it is insufficient to describe function application. The problem occurs when a function is applied to a name. For the notation to be well behaved, the meaning of a function should be independent of the names used: $\lambda x.x$ means the same as $\lambda y.y$. But suppose that we apply $\lambda x.\lambda y.2 \times x + y$ to y. Unmodified β-substitution would give $\lambda y.2 \times y + y$, in which both occurrences of y are bound. If the function had been written $\lambda x.\lambda z.2 \times x + z$, the result would have had y as a free variable, and must be different. Such name clashes are avoided by the requirement that no free occurrence of a name in an argument may become bound when the argument is substituted. This requires us to rename any variables in the function that would cause binding to happen. The renaming process is called *α-reduction* or *α-conversion*.

1.2. Combinators

Combinatory logic [Curry 1958] is an alternative formalism to λ-calculus in which all functions are represented in terms of certain primitive functions, called combinators. A combinator is a function that has no free variables.

All expressions that may be represented in the λ-calculus with no free variables may be expressed in terms of combinations of the two combinators **S** and **K**:

$$\mathbf{S}\ f\ g\ h \triangleq f\ h\ (g\ h) \qquad (s)$$

$$\mathbf{K}\ a\ b \triangleq a \qquad (k)$$

($\triangleq$ means 'is defined as'). Alternatively they may be defined in the notation of λ-calculus: $\mathbf{S} = \lambda f.\lambda g.\lambda h.f\ h\ (g\ h)$ and $\mathbf{K} = \lambda a.\lambda b.a$. To show that these are the only combinators needed, it is sufficient to give an algorithm to translate λ-terms into combinators. As a first example, define **I** as **S K K**. Now **I** $x = x$:

$$\begin{aligned} \mathbf{I}\ x &= \mathbf{S}\ \mathbf{K}\ \mathbf{K}\ x && \text{By definition} \\ &= \mathbf{K}\ x\ (\mathbf{K}\ x) && \text{By } (s) \\ &= x && \text{By } (k) \end{aligned}$$

Which suggests that $\mathbf{I} = \lambda x.x$, because it gives the same answer as $\lambda x.x$ for any argument[†].

[†] Functions that always give the same answer for the same argument are said to be *extensionally* equal.

This gives us the first rule of the algorithm:

$$\begin{array}{rll} \lambda x.x & \text{becomes} & \mathbf{I} \\ \lambda x.y & \text{becomes} & \mathbf{K}\ y \quad \text{if } x \text{ not free in } y \\ \lambda x.y\ z & \text{becomes} & \mathbf{S}\ (\lambda x.y)\ (\lambda x.z) \end{array}$$

To achieve a complete conversion, it is necessary to apply these rules to the terms of the expression until no λs remain.

Notice that all combinatory expressions may be converted to λ-expressions by the substitution of the λ-expressions for **S**, **K**, and **I**.

1.2.1. Composition

Another combinator that deserves special notice is the composition operator. The common name for this combinator is **B**:

$$\mathbf{B}\ f\ g\ h \triangleq f\ (g\ h)$$

As a notational convenience, **B** is often written as the infix operation $\circ$. If $double \triangleq \lambda x.x+x$, then $double \circ double$ is a function that quadruples its argument: $(double \circ double)\ 2$ is **B** *double double* 2 is *double* (*double* 2) is *double* 4 is 8.

Backus [Backus 1978] proposed a variable-free programming style, and a language, FP, based on combinators. Although all programmes may be written using **S** and **K**, the result would not be readable. FP consists of a set of more expressive combinators, such as composition.

1.3. Recursion

Neither the λ-calculus nor combinatory logic has an obvious mechanism for the definition of recursive functions. Nevertheless both are capable of expressing functions that are defined in terms of themselves. The key observation is that we can define functions that partially perform the required recursion. The traditional example is the factorial function:

$$factorial\ n \triangleq \begin{cases} 1 & \text{if } n = 0 \\ n \times factorial\ (n-1) & \text{otherwise} \end{cases}$$

Notice that if the parameter f of the function *fac* defined by

$$\begin{array}{l} fac \triangleq \lambda f.\lambda n.if\ n = 0 \\ \qquad\qquad\qquad then\ 1 \\ \qquad\qquad\qquad else\ n \times f(n-1) \end{array}$$

is the factorial function, then the resulting expression is equivalent to the factorial function (by β-substitution); *fac factorial* = *factorial*. Consider also fac_0, defined by $fac_0 \triangleq$ *fac splat* where *splat* is an expression whose evaluation does not terminate (see 1.4 below)

fac_0 approximates the factorial function in that $fac_0\ 0 = factorial\ 0$ and it terminates for no other value of the argument. Now $fac_1 = fac\ fac_0$ is a better approximation: it has the same value as *factorial* for 0 and 1, and terminates nowhere else. Similarly $fac_2 = fac\ fac_1$ is better still in that it works for all values of n up to 2.

This leads us to the means of introducing recursion. The final approximation to *factorial* would be $fac_\infty = fac\ (fac\ (fac(\ldots)))$, with an infinite number of applications of *fac*. Suppose we have a function **Fix** such that **Fix** $fac = fac$ (**Fix** fac). Intuitively **Fix** $fac = fac\ (fac\ (fac(\ldots)))$.

Can a **Fix** be defined? We can arrive at an expression for **Fix** by reasoning similar to that by which we came to decide that it is needed. Suppose that we can define a function Θ such that $\Theta\ \Theta =$ **Fix**. We would then be able to define a new version of **Fix** as $\lambda f.f\ (\Theta\ \Theta\ f)$. But as yet we have no expression for Θ so take it out as a parameter, giving $\Theta_0 \triangleq \lambda\theta.\lambda f.f\ (\theta\ \theta\ f)$. By β-reduction we have that $\Theta_0\ \Theta = \lambda f.f\ (\Theta\ \Theta\ f)$, which, as we have already seen is equal to **Fix**. i.e. $\Theta_0\ \Theta =$ **Fix** $= \Theta\ \Theta$, which suggests that $\Theta_0 = \Theta$. We now have

$$\textbf{Fix} \triangleq \Theta\ \Theta$$
$$\Theta \triangleq (\lambda\theta.\lambda f.f(\theta\ \theta\ f))$$

because

$$\begin{aligned}
\textbf{Fix}\ F &= \Theta\ \Theta\ F && \text{Definition of } \textbf{Fix}\\
&= (\lambda\theta.\lambda f.f(\theta\ \theta\ f))\ \Theta\ F && \text{Definition of } \Theta\\
&= (\lambda f.f(\Theta\ \Theta\ f)\ F && \beta\text{-reduction}\\
&= F\ (\Theta\ \Theta\ F) && \beta\text{-reduction}\\
&= F\ (\textbf{Fix}\ F) && \text{Definition of } \textbf{Fix}
\end{aligned}$$

1.3.1. Recursion Equations

Another formalism for functions is recursion equations, in which a function is defined recursively by a case analysis of values of its arguments. For example,

$$f\ 0 = 1$$
$$f\ (succ\ n) = succ\ n \times f\ n$$

defines the factorial function. Turner has defined several languages based on this principle, including SASL and KRC [Turner 1982].

1.4. Evaluation Order

In the example of composition, I said that *double* (*double* 2) is *double* 4, having reduced *double* 2 first. I could equally have said that *double* (*double* 2) is (*double* 2) + (*double* 2), reducing the first application of *double* first. There are several possible orders of reduction (in both combinatory logic and λ-calculus), but if a sequence of reductions terminates it will give the same answer (up to α-conversion) as any other sequence of reductions

that terminates. (This is known as the Church-Rosser property—a recent reference is [Rosser 1982]).

In traditional programming languages the most common order of evaluation is *applicative* order, where the arguments of a function are evaluated before the body. Applicative order has the disadvantage that it does not necessarily terminate, even though an expression has a value. Consider $\mathbf{K}\ (2+2)\ (\mathbf{S}\ \mathbf{I}\ \mathbf{I}\ (\mathbf{S}\ \mathbf{I}\ \mathbf{I}))$. In applicative order, the arguments are evaluated first; $(2+2)$ gives 4, but $(\mathbf{S}\ \mathbf{I}\ \mathbf{I}\ x) = x\ x$ so $(\mathbf{S}\ \mathbf{I}\ \mathbf{I}\ (\mathbf{S}\ \mathbf{I}\ \mathbf{I})) = (\mathbf{S}\ \mathbf{I}\ \mathbf{I}\ (\mathbf{S}\ \mathbf{I}\ \mathbf{I}))$, which is the same as before. If the $\mathbf{K}$ is applied first, the whole expression reduces to $(2+2)$ and then to 4. Indeed, if the leftmost reducible expression (sometimes called *redex*) is always reduced first, the result is guaranteed to be found if it exists [Barendregt 1980]. This is called *normal* order reduction.

1.5. Types

It is accepted by many that type-checking is an important aid to the production of correct programmes. A counter argument is that the type systems used in conventional languages have been too restrictive—a procedure definition must specify the type of its argument even if it doesn't matter what that type is.

The language ML, originally designed as a metalanguage for a proof assistant [Gordon 1979] includes a type system that partially overcomes this problem. A programme in ML is prepared largely without the incorporation of type information, but the ML compiler infers the types of expressions automatically [Milner 1978]. An expression is not required to have just one type; $\lambda x.x$ will be inferred to have all types of the form $\alpha \rightarrow \alpha$ (where α is a type variable). This frees programmes from the restrictions of rigid type-checking, but still ensures that they contain no type-errors.

HOPE [Burstall 1980] is a pure functional language that has the same types as ML, except that the HOPE compiler does not do the same kind of type-inference. Instead, the programmer is required to specify the type of each function. This allows function names to be overloaded on the type of their arguments—this means that the same name may be used for a number of different functions. For example, 'plus' is the common name of the function to add two numbers, but how the addition is to be performed depends on the type of numbers to be added. The HOPE compiler uses the type information to detect which function is appropriate to perform the operation on a particular type of argument.

2. Implementation Techniques

The implementation of functional languages can conveniently be divided into two areas—software and hardware. Although most of the software techniques were originally intended for 'conventional' computers, many of them are useful in conjunction with specialised hardware.

2.1. Software

2.1.1. Lazy Evaluation

Although normal order reduction always finds the answer, it often does more work than necessary. The normal order reduction of *double* (*double* 2) gives (*double* 2) + (*double* 2), and then (*double* 2) will be evaluated twice. This problem is ameliorated by call by need [Wadsworth 1971], (also known as lazy evaluation [Henderson 1976]). Rather than copying expressions, lazy evaluation uses pointers to remember the values of expressions, so that *double* (*double* 2) reduces to $\Pi + \Pi$, where Π is a pointer to (*double* 2). When Π is evaluated, the expression becomes $4 + 4$; the effect has been to reduce both instances of $(2 + 2)$ simultaneously.

There has been some confusion in functional programming literature between normal order and laziness. Rather than use the two terms to mean the same thing I will use normal order for the order, and reserve laziness for the technique of remembering the value of subexpressions. I will also use 'normal order semantics' as a name for any reduction order that is guaranteed to produce the answer if normal order does.

2.1.2. Bigger Combinators

A major problem in the implementation of λ-based functional languages is that β-substitution is a costly process. Combinators have the seductive property that all reductions are merely re-orderings of parameters. In his seminal papers [Turner 1979, 1979′], Turner proposed the use of combinators as an implementation technique for functional languages. He observed, however, that the expression of a function in terms of **S**, **K** and **I** alone is generally much larger than the original expression. As an attempt to counteract this problem, he introduced a number of new combinators.

$$\begin{aligned}
\mathbf{C}\ f\ g\ h &\triangleq f\ h\ g \\
\mathbf{B}'\ a\ b\ c\ d &\triangleq a\ b\ (c\ d) \\
\mathbf{C}'\ a\ b\ c\ d &\triangleq a\ (b\ d)\ c \\
\mathbf{S}'\ a\ b\ c\ d &\triangleq a\ (b\ d)\ (c\ d)
\end{aligned}$$

These are then incorporated into programmes by the following optimisations:

$$\begin{aligned}
\mathbf{S}\ (\mathbf{K}\ a)\ b &\text{ becomes } \mathbf{B}\ a\ b \\
\mathbf{S}\ a\ (\mathbf{K}\ b) &\text{ becomes } \mathbf{C}\ a\ b \\
\mathbf{S}\ (\mathbf{B}\ a\ b)\ (\mathbf{K}\ c) &\text{ becomes } \mathbf{C}'\ a\ b\ c \\
\mathbf{S}\ (\mathbf{B}\ a\ b)\ c &\text{ becomes } \mathbf{S}'\ a\ b\ c
\end{aligned}$$

After these optimisations, the resulting combinator expressions are generally smaller, but still quadratic in the size of the input programme.

2.1.3. Super-combinators

The term 'super-combinator' was coined by Hughes [Hughes 1982] as a name for generalised combinators chosen for a particular programme. Expressions translated to combinators by Turner's improved algorithm are quite small, but each combinator reduction performs only a small amount of work, resulting in poor running speed. Super-combinators provide a method of making the combinators used in a programme as large as is consistent with the preservation of the semantics of the original expression. Hughes' paper describes reasoning behind the selection of combinators in detail. Here I shall give only an outline of the algorithm.

By analogy with free variables, Hughes defines a *free expression* to be an expression that does not contain a bound variable. In $\lambda a.f\ b\ (a\ (b\ (c\ d)))$ all of $b, c, d, f, f\ b$ and $b\ (c\ d)$ are free expressions. A free expression of a λ-abstraction is a *maximal* free expression if it is not part of any larger free expression. The maximal free expressions of the above example are $f\ b$ and $b\ (c\ d)$.

λ-abstractions are converted to super-combinators by 'lifting out' all of the maximal free expressions as parameters. The λ-abstraction is then replaced by an application of the resulting combinator to the maximal free expressions. Taking the same example again, $\lambda a.f\ b\ (a\ (b\ (c\ d)))$ is converted to $\alpha\ (f\ b)\ (b\ (c\ d))$ with $\alpha \triangleq \lambda fb.\lambda bcd.\lambda a.fb\ (a\ bcd)$†.

Hughes describes an improvement on this method that revolves around the numbering of bound variables: the *level* of a bound variable is given by the number of enclosing lambdas. In $\lambda f.(\lambda g.(\lambda i.i)f\ (g\ g))(\lambda h.f\ (h\ h))$ the level of f is 1, of both g and h is 2 and of i is 3. The level of a sub-expression is then the greatest of the levels of all its free variables. In the context of the above λ-expression, $\lambda i.i$ has level 0 (it has no free variables), $(h\ h)$ has level two and $(\lambda h.f\ (h\ h))$ has level one.

The translation of a λ-expression to super-combinators proceeds recursively. Each innermost λ-abstraction is translated to super-combinators until there are no remaining λ-abstractions. An innermost λ-abstraction is translated by lifting out its maximal free expressions as before, but they are sorted into increasing order of level before the application is constructed. If the levels of b, c, d and f are 1, 2, 3 and 4 respectively, $\lambda a.f\ b\ (a\ (b\ (c\ d)))$ will be converted to $\alpha\ (b\ c\ d)\ (f\ b)$ with $\alpha \triangleq \lambda bcd.\lambda fb.\lambda a.fb\ (a\ bcd)$. This ordering of parameters ensures that the maximal free expressions of the resulting combinator application will be as large as possible.

A similar technique, λ-lifting, is in use at Göteborg [Johnsson 1984], but as it does not analyse maximal free expressions, it does not always preserve laziness.

2.1.4. Programme Transformation

Another alternative is to transform the functional programme into an equivalent imperative one that exploits conventional hardware more efficiently. Burstall and Darlington [Burstall 1977] presented a technique for the semi-automatic transformation of programmes.

Programme transformation is not confined to implementations on conventional machines. The compilers for Alice [Darlington 1981] also use this technique, and it is

† *fb* and *bcd* are single names.

likely that there would be benefits from the its application to programmes for SKIM [Clarke 1980].

Programme transformation may not be a completely automatic process. This leads to the problem that a record of how to transform each particular programme must be kept. Without such a record, the modification of the original programme would necessitate the re-application of the transformer, a task that one would prefer to be automatic. Darlington [Darlington 1983] has suggested that a good way to keep this record is to write the programme transformations in some language, and keep this transformer-programme with the original.

2.2. Hardware

Attempts to implement functional programming languages on conventional machines often give rise to the observation that the architecture of the machine is not really suitable for the task. Recent research has shown that given hardware specially designed for the task, functional languages can be made to run much more efficiently than on ordinary computers.

The hardware techniques fall into three categories: string, acyclic graph and general graph reduction. In string reduction the programme is represented by its text and reduction is performed directly according to the reduction rules. An example may be found in [Mago 1980]. In acyclic graph reduction the programme is represented as a graph, essentially the syntax tree of the programme text, but with common sub-expressions shared. Alice [Darlington 1981] is a parallel processor based on this principle. Finally, general graph reduction allows the programme graph to include cycles, (which usually represent recursive objects), and SKIM [Clarke 1980] uses graph representations of combinator expressions to evaluate functional programmes.

Informal Description of Ponder

This chapter is a self contained description of Ponder as if the functions defined in the standard prelude were built in. This allows me to give intelligible examples before all of the language has been described. The next chapter, **Example Ponder Programmes** gives definitions of the conditional expression and a large example to show how the syntax defining mechanisms may be used to increase the expressiveness of a programme. The design of the language is described in more formal detail in Part II; readers interested in the technical aspects of this work, rather than the language description, may prefer to skip to Part II. I have also described Ponder as it *is*, rather than as it should be; I have glossed over the deficiencies as far as possible. See the chapter **Analysis** below for a critical examination of the language.

1. What a Ponder Programme is

Ponder is a pure functional language. A Ponder programme therefore has no side effects; it is an expression that may be evaluated to give a result. It is important to bear this in mind when reading the rest of this description.

2. Objects

I call the things that a programme manipulates 'objects'. This includes functions. Indeed, if it were not for the fact that the programme must produce results that may be printed out, the only objects would be functions. As it is, the computer must be able to interpret a programme in such a way as to produce objects that have some intrinsic meaning. The most common means of communication with the outside world is by printed characters.

2.1. Characters

A character is anything that may be typed into the computer as a single key-stroke. A Ponder programme will be represented by a sequence of characters, so we need a way of distinguishing characters that stand for themselves from those used to represent the other pieces of programme. A *character icon* is a representation of a character. Character icons are themselves represented by a quotation mark followed by the character. Thus `'a` is the character icon for the letter a, `'b` represents the letter b and so on.

2.2. Strings

Very frequently, a programme will want to manipulate a sequence of characters, and include some constant string of characters. Strings of characters are indicated by surrounding the characters with quotation marks. An example of a string icon is “38, York Terrace”.

Certain characters can only be included in strings by means of escape characters—a closing quotation mark is a notable example. The following table shows the representation used within string icons for these characters.

Character	Representation
”	'”
'	''
newline	'n

Strings may also be split over long lines:

“this string is on '
two lines”

means the same as

“this string is on two lines”

The rule is that all layout after an apostrophe is ignored.

2.3. Integers

Programmes frequently manipulate numbers. Again, there is a means provided for writing numbers in programmes, called the *integer icon*. An integer icon is simply a sequence of digits, such as 312821. Notice that “312821” is still a string icon, despite its consisting of digits.

3. Declarations

Quite often, a programme will need to use the same thing more than once. If the thing is long, or difficult to remember in detail, such as

“Japanese Haiku/ have seventeen syllables/ five then sev'n then five”

it is useful to be able to give it a name. The *let-declaration* gives an object a name. After the declaration

Let *haiku-description* $\triangleq$ “Japanese Haiku/ have seventeen syllables/ '
five then sev'n then five”;

we may use *haiku-description* in exactly the same way as “Japanese Haiku/ have seventeen syllables/ five then sev'n then five”. The semicolon serves to separate the declaration from any following declaration or expression.

Although a name may consist of several words, the words must be linked together by hyphens, because one name followed by another means something different (see 'expressions' below). Thus *haiku-description* and *haiku--description* are both single names (in fact the same one, because only the words count), but *haiku description* is an expression consisting of two names, *haiku* and *description*, and *haikudescription* is a one word name. A name consists of letters, hyphens and digits in *italics*.

4. Expressions

Expressions in Ponder are built up in two ways: either by the application of a function, or of an operator.

4.1. Function Applications

To apply a function, it is written in front of its argument. So if *sin* stood for the sine function, and *pi* stood for π, *sin pi* would be an expression with value zero. There is a function called *reverse*. For example *reverse* "regal" is "lager". Function application *associates* to the left, and parentheses—'(' and ')'—may be used to group things together. So *reverse* (*reverse* "car") is "car".

4.2. Infix Operators

Infix operators are written in between their arguments, in the way that + is used to add two integers together. Indeed $2 + 2$ has the value 4 as one might expect. Similarly 6×7 is 42.

An important difference between operators and functions is that while a function is just an object and always means the same thing, the meaning of an operator can depend on what its arguments are. This mirrors the normal notation. For the addition of two integers, we write +, and to add two fractions together we also write +, but the algorithm for adding integers is different from the one for fractions. Thus operators (but not function names) can be *overloaded*.

5. Types

Every object in Ponder has a type. Types are not objects, so they do not have types. A character has type Char, an integer type Int and strings have type String. These types have names, and the convention for writing type-names is similar to that for writing object-names. To distinguish type-names, they are written in sans serif.

Because certain types are related to each other, Ponder has the facility of *parameterised* types. An example is List. We can have a list of integers, written List [Int] or a list

of characters, List [Char] or even a list of lists of characters, List [List [Char]]. Incidentally, a String is just a list of characters, and can equally well be referred to as List [Char].

6. Lists

Lists are common in functional programming. If a programme is to manipulate a sequence of things, it will usually use a list. Lists are constructed by means of the list operator '::'. Thus "violin":: "viola":: "violoncello":: "double bass" is a list of strings. We can join together two lists using *append*, written as an infix operator @. For example (2:: 4:: 6:: 8) @ (3:: 5:: 7:: 9) joins together the two lists to get 2:: 4:: 6:: 8:: 3:: 5:: 7:: 9. *Reverse* is in fact a list manipulating function, so *reverse* (1:: 2:: 3) is 3:: 2:: 1. Incidentally, :: is an overloaded operator: the first :: in 1:: 2:: 3 takes an integer and a list, but the second takes two integers.

Strings are lists, so all the list operators may be used on strings. Thus "ant" @ "acid" is "antacid".

7. Functions

I have mentioned that functions are objects, and that *reverse* is a function, but have not yet said how to write them. A function is just an expression with a parameter. A parameter is a name that has no particular value. Because of this, it is always necessary to say what the type of a parameter is going to be, so that the compiler can check to see that it is correctly used. Suppose we want to square integers.

$\mathsf{Int}\ i \rightarrow i \times i$

is a function[†] to do this. The parameter is called i, and Int says that the parameter is always going to be an integer—otherwise it would not be possible to tell which version of $\times$ must be used. The arrow $\rightarrow$ is just there to separate the parameter from the expression.

We can name objects using let-declarations, and functions are objects, so we can name functions:

Let *square* $\triangleq$ $\mathsf{Int}\ i \rightarrow i \times i$;

after this declaration *square* 3 means 3×3, which evaluates to 9.

A function is just a rule that says how to get to the answer when given an argument. To apply a function, substitute occurrences of the parameter name in the expression after the arrow with the value of the argument. To compute *square* (*square* 3), substitute the argument (*square* 3) for the parameter i in the expression $i \times i$ to get (*square* 3)$\times$(*square* 3). To find the value of this in turn, we need to compute *square* 3 by substituting 3 for i giving 9, as before, so the answer is 9×9, which is 81.

[†] It corresponds to the λ-expression $\lambda i.i \times i$ but includes type information

Often a function depends on more than one thing. If we often want to compute the sum of the square of two integers, we might want to write a function to do it, rather than write it out every time. How do we write a function of more than one argument? Well, functions are objects, so there is no reason why a function cannot return another function as its result. Thus

Let *sum-squares* $\triangleq$ Int $i_1 \rightarrow$ Int $i_2 \rightarrow$ *square* $i_1 +$ *square* i_2;

declares *sum-squares* as function with one parameter i_1, but which returns a function as its result.

If we apply *sum-squares* to 15, what happens? The rule says "substitute the argument for the parameter in the part after the arrow". This gives Int $i_2 \rightarrow$ *square* $15 +$ *square* i_2, a function that calculates the sum of the square of a number and the square of 15. If we apply this in turn to 21, we get *square* $15 +$ *square* 21, which is $225 + 441$, which is 666. Function application associates to the left, so we can write these two applications as *sum-squares* 15 21.

What if it is not known in advance how many numbers there will be? The answer is to use lists. We can write the function to compute the sum of the squares of any number of integers if we make its parameter a list of integers.

Before we do that, let me introduce two more useful functions for lists. *Map* (the name is historical, after 'mapcar' in LISP) applies a given function to every element of a list. Thus

map square (1:: 2:: 3:: 4)

is

square 1:: *square* 2:: *square* 3:: *square* 4

which is the list (1:: 2:: 9:: 16), and

map reverse ("no":: "god":: "saw":: "a":: "mined":: "pool")

is

"on":: "dog":: "was":: "a":: "denim":: "loop"

The function *gather* applies a function of two arguments "between" each element of a list and a terminal value. For the examples, let me use two functions *int-plus-int*, which is + for integers, and *int-times-int*, similarly. Thus *gather int-plus-int* 0 (1:: 2:: 3) evaluates to $(1 + (2 + (3 + 0)))$, and *gather int-times-int* 1 (1:: 2:: 3) to $(1 \times (2 \times (3 \times 1)))$. This function is also known as *reduce* or *fold* in other languages.

Now we can write an new version of *sum-squares*:

Let *sum-squares* $\triangleq$ List [Int] *ints* $\rightarrow$
gather int-plus-int 0
(*map square ints*);

Which just uses the *map* and *gather* examples together, so that *sum-squares* (1:: 2:: 3) is *gather int-plus-int* 0 *(map square* (1:: 2:: 3)), which in turn becomes *gather int-plus-int* 0 (1:: 4:: 9), and finally $1 + 4 + 9 + 0 = 14$.

7.1. Function Types

Functions, like all other objects in Ponder, have types. The type of a function is written with an arrow from the type of the parameter to the type of the result. *Square* has a parameter of type Int, and returns a result of type Int, so its type is written Int → Int. The compiler checks that every time a function is applied to an argument the type of the argument matches the type of the parameter of the function.

8. Printing Answers

So far, we have only seen how to compute values. We have seen a function that, when given a list of integers, will compute the sum of the squares of those integers, but not how to print it out. A Ponder programme is an expression, and it is required to have a particular type. A programme cannot just be a sequence of values to be printed. This is because it is impossible for the compiler to know how you want something to be printed—do you want your numbers in decimal, binary or Roman numerals? A programme is of type List [File-action].

For the moment we need not worry what a File-action is, merely note that there is a function called *print-to-terminal* that takes a List [Char] and converts it to a List [File--action]. The name is, perhaps, misleading. An expression such as *print-to-terminal* "Hello!" does not cause "Hello!" to be printed at the terminal. The value of *print-to-terminal* "Hello!" is a List [File-action] that, when returned as the value of a programme, will cause "Hello" to be printed at the terminal.

A programme that consists entirely of

print-to-terminal "Hello!"

will print "Hello!" on the terminal (without the quotation marks), and then stop. An object of type List [File-action] is a list, and so we can use the usual list operations to manipulate it:

print-to-terminal "Hello" @ *print-to-terminal* " world!"

will print "Hello world!" and

reverse (print-to-terminal "Hello" @ *print-to-terminal* " world!"*)*

will print " world!Hello".

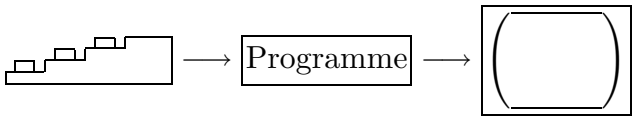
If we want to print out numbers, we must convert them first into String, and then into List [File-action]. There is a function called *print-int*, which is of type Int → String, which converts integers to their decimal character representation. Thus *print-int* 33 is "33". Now we can have a complete programme to calculate the sum of some squares:

```
Let sum-squares ≙ List [Int] ints →
                    gather int-plus-int 0
                           (map square ints);
print-to-terminal (print-int (sum-squares (1::2::3::4::5)))
```

9. Input

Input from the terminal is a list of characters called *terminal-input-list*. This list contains all the characters typed at the terminal, in the order in which they are typed. Thus the first character in the list is the first character typed, and so on. Of course, the computer can't predict what the person will type at the keyboard, so whenever the output of the programme depends on another character from the input list, the programme will have to wait for that character to be typed.

Note that the keyboard is considered to be separate from the screen, as in this picture:



Key depressions are recorded *only* in the input list—if anything is to happen on the screen the programme must make it happen.

I have not really described enough of the language to give much of an example of terminal input. The programme:

print-to-terminal ("Now type: "@*terminal-input-list*)

is very silly. When run, it will print "Now type: " followed by the terminal input list. You have not typed it in yet, so it will wait until you press a key. When you do, it will print the character corresponding to it on the terminal, and wait for another one. When you type that, it is printed, and so on *ad infinitum*. It depends on your particular computer as to what you have to do to stop it, and it may not be easy!

We can use the list function *map* to write an even sillier programme. The function Char $c \rightarrow$ 'x when applied to a character returns the character x. Thus the programme:

print-to-terminal (*map* (Char $c \rightarrow$ 'x) *terminal-input-list*)

will type an x for every character typed at it.

10. Conditionals

Clearly a programme must be able to test values, and perform different actions according to the result. We will eventually want to be able to write programmes that perform different actions according to their input.

In the familiar mathematical notation $x < 5$ is an expression that is true if x is less than 5, and false otherwise. Similarly in Ponder, $x < 5$ is an expression that is *true* or *false*; *true* and *false* are objects, so they must have a type, and their type is Bool. Bool stands for *Boolean*, after Boolean algebra. To test whether a Bool is *true* or *false*, there is a construct called a 'conditional expression'.

If condition
Then $thing_1$
Else $thing_2$
Fi

is an expression that has the value $thing_1$ if *condition* is *true* and $thing_2$ if it is *false*.

As an example, here is a function that returns the lesser of two arguments:

$$\begin{aligned}\textbf{Let}\ min \triangleq\ & \mathsf{Int}\ a \rightarrow \mathsf{Int}\ b \rightarrow \\ & \textbf{If}\ a < b \\ & \textbf{Then}\ a \\ & \textbf{Else}\ b \\ & \textbf{Fi};\end{aligned}$$

Since a conditional expression must have a type, and it can return either of two expressions, they must have the same type.

11. Pairs

A pair is an object with a left part and a right part. A pair is written with its left part separated from its right part by a comma, thus: $1, 2$. There are two functions to take pairs apart: *left* and *right*. They behave in the obvious manner: $left(1, 2)$ is 1 and $right(1, 2)$ is 2. To simplify the process of splitting pairs, functions and let-declarations both have special means of dealing with them. The declaration **Let** *one, two* $\triangleq 1, 2$*;* declares *one* to have the left part of the pair $1, 2$, and *two* to have the right part. In functions we must, as always, give the types of the parameters: Int *a,* Int $b \rightarrow a + b$ is a function that takes a pair of integers and adds them together.

Pairs are objects, so there can be pairs of pairs, as in $1, (2, 3)$ and $(3, 4), (5, 6)$. Comma associates to the right, so $1, 2, 3$ means $1, (2, 3)$. Thus **Let** *a, b, c, d* $\triangleq 1, 2, 3, 4$*;* declares a, b, c and d to have the values 1, 2, 3 and 4 respectively.

11.1. Pair Types

The type of a pair is *Pair*, which is parameterised on two arguments. $1, 2$ has the type Pair [Int, Int] and '6, "66" has the type Pair [Char, List [Char]]. For convenience, Pair may be written with the infix type operator $\times$, by analogy with the set product operation.

12. Splitting Lists

A list may come to an end: there are no characters after the 'o in "Hello". The list that has no elements is called *nil*.

As we have seen, lists may be constructed using ::, and as you may expect, it is possible to take them apart again. The possibility that a list may have no entries in it must always be taken into consideration. There is a special kind of conditional to do this.

If *the-list* **Is** *function*
Else *thing-to-return-if-it-isnt*
Fi

If the list *is*, (i.e. it isn't *nil*), it is transformed into a pair containing the first element of the list and the rest of the list, and *function* is applied to this pair. An example will make this clearer:

If *terminal-input-list* **Is** Char *first*, List [Char] *rest*
→ *first*:: *first*:: *rest*
Else *nil*
Fi

this expression is *nil* if the terminal input list is empty (a rare occurrence, but not impossible: the 'terminal' may be connected to a file), and the terminal input list with the first character duplicated otherwise.

13. Recursive Functions

It is often necessary for functions to be recursive, but it is inconvenient to declare recursive functions using let-declarations (with **Fix**). If a programme contains the sequence of declarations

Let $f \triangleq$ Int $a \rightarrow$ Int $b \rightarrow a \times a + b$;
Let $f \triangleq$ Int $a \rightarrow f\ a\ 6$;

the use of f in the body of the second definition of f simply refers to the first definition, so f will subsequently be a function of one parameter that returns the square of that parameter plus six. The second definition of f is not recursive.

To obtain a recursive definition, we can use a let*rec*-declaration.

Letrec *twiddle-pairs* $\triangleq$ List [Char] *list* → List [Char]:
If *list* **Is** Char c_1, List [Char] *rest*
→ **If** *rest* **Is** Char c_2, List [Char] *rest*
→ c_2 :: c_1 :: *twiddle-pairs rest*
Else c_1:: *nil*
Fi
Else *nil*
Fi

is a function from a list of characters to another list in which the characters are swapped in pairs. The List [Char]: specifies that the result of the function is to be of type List [Char]. This is necessary because the compiler cannot always discover the type of the body of a

recursive function, since it can depend on itself. In general, an expression of the form type: *expression* indicates that *expression* has type type.

It is instructive to run the programme

print-to-terminal (*twiddle-pairs terminal-input-list*)

When the first character is typed, nothing happens. This is because the first character to be printed is the second character typed, and of course this cannot appear before it is typed. Whenever an even numbered character is typed, a pair of characters appears on the screen.

14. Interaction with a Terminal

A further observation is that there is no way to rub out characters typed into the above programme; the rubout character is treated like any other and sent to the screen in its turn. It is possible to write a function that performs the correct operations for input of characters, but it is too complex to describe here. An example is given in appendix 3.

15. Extensibility

Ponder allows you to define your own syntaxes for your programmes. If you want to say *square* **All-of** (1:: 2:: 3) instead of *map square* (1:: 2:: 3), you can use an **Infix** declaration:

Infix All-of $\triangleq$ *map;*

this makes **All-of** into an infix operator.

If an operator is used in conjunction with another, as in $a \times b + c$, the *priority* of the operator is important. Operators with high priorities are considered first. $\times$ has a higher priority than $+$, so the above expression means the same as $(a \times b) + c$, and to get the other meaning you would have to write $a \times (b + c)$. Priorities are numbered in reverse order, as in common parlance; priority one is the highest priority. You may declare priorities for your own operators:

Priority 4 $\times$ **Associates Left***;*

the **Left** indicates that the left hand of two operators of the same priority is to take precedence: $a \times b \times c$ will mean $(a \times b) \times c$. Note that the priority applies to the operator rather than its associated values: the priority of the operator is the same regardless of the type of its arguments.

All you need to do to overload your own infix operators is to declare them again with a different value:

Infix $+ \triangleq$ String $a \rightarrow$ String $b \rightarrow a$ @ b*;*

defines that + for strings is concatenation, but leaves the meaning of + for integers unchanged.

Names of functions may not be overloaded, but prefix operators may be. A prefix operator is one which is placed in front of its (single) argument, such as − in $-(a+b)$. Notice that only one function can be bound to a name, but overloading allows the association of more than one function with an operator. A consequence of this is that an operator cannot be used as an argument to a function; only expressions can.

A prefix declaration is rather similar to the infix kind:

Prefix − ≙ String *a* → *reverse a;*

defines − of a string to be reverse of it, and again leaves − of integers as it was.

The only other kind of new syntax is the bracket operator. A bracket operator consists of two parts: the opening bracket and the closing bracket. After the definition

Bracket Begin End ≙ *function*;

an expression like **Begin** *thing* **End** means the same as *function thing*. Brackets may also be overloaded.

16. Polymorphism

You may have noticed that I said that *reverse* "is a list operation". I have used it on things of type List [Char] and List [Int]. What is the type of *reverse*? Its argument must be a List, but it matters not of what. Reverse works for all types of list. Its type is ∀ E. List [E] → List [E]. The ∀ is pronounced 'for all', as in logic.

To write a polymorphic function, just put in a ∀ to declare the name of the type of which it is independent:

```
Letrec twiddle-pairs ≙ ∀C. List [C] list → List [C]:
                       If list Is C c1, List [C] rest
                       → If rest Is C c2, List [C] rest
                          → c1:: c2:: twiddle-pairs rest
                          Else c1:: nil
                          Fi
                       Else nil
                       Fi;
```

declares *twiddle-pairs* again, but this time states that it works for any type of list.

Any expression may be preceded by '∀name.' this declares name for the expression and indicates that the expression would be valid whatever type replaced name. Moreover, all type variables must be bound either by '∀' or in declarations (see **17**). For convenience, a sequence of bindings such as '∀A. ... ∀Z.' may be abbreviated as '∀A, ..., Z.'.

17. Type Declarations

It is often convenient to give a type a name. For instance, if a programme manipulates a large number of personnel records, it would be tedious to write out the whole type of the record each time a function was defined.

Type Personnel-record $\triangleq$ Name $\times$ Birth-date $\times$ Salary $\times$ Tax-code;

declares Personnel-record as a type.

The declaration of parameterised types is similar, and a type may be made recursive:

Rectype List [T] $\triangleq$ Option [T $\times$ List [T]];

is the definition of List (the type Option is explained below). Note that a type name may only be applied to the same number of arguments as in its definition. In the case of $\forall$, the number is zero.

18. Options

The definition of *List*s uses *Option*. An *Option* is a thing that may or may not be there, for example the next element of a list may or may not be there. In fact, **If** ... **Is** ..., (which we used to test lists) tests Options. Options are also useful for the results of functions that may fail. For instance, if we have an association list (a list of pairs of keys and values), we might define *lookup* as follows:

```
Letrec lookup ≜
        ∀T. String key →
        List [String × T] association-list →
        Option [T]:
        If association-list Is (String × T) hd, List [String × T] rest
        → If key = left hd
            Then opt-in (right hd)
            Else lookup key rest
            Fi
        Else nil
        Fi;
```

The function *opt-in* has type $\forall$T. T $\rightarrow$ Option [T]; it converts an object to an optional object. The object *nil* has type $\forall$T. Option [T]: it is an optional anything, and represents the thing not being there. Thus *lookup* returns either an object if it finds one with the correct key, or *nil* if it does not. The result of *lookup* may be tested with **Is**:

```
If lookup "Date" books Is Book the-book
→ read the-book
Else buy-a-book-by "Date"
Fi
```

19. Capsules

Although the representations of two objects may be the same, it is often the case thatwe do not want to use them interchangeably. For instance, we may be using the integers 1, 2 and 3 to represent the colours red green and blue, and at the same time using them to represent the suits clubs, diamonds and hearts. It would merely be coincidence if clubs = red, and a programme that depended on this would almost certainly be wrong.

The mechanism that allows us to avoid this kind of mistake is the encapsulation of types. We can declare a type to be a capsule, declare some objects involving the type and then seal the capsule:

Capsule Type Colour $\triangleq$ Int;
Let *red*, *green*, *blue* $\triangleq$ Colour: 1, Colour: 2, Colour: 3;
Let *equal-colours* $\triangleq$ Colour $a \rightarrow$ Colour $b \rightarrow$ Bool: $a = b$;
Seal Colour;

Until the capsule is sealed, a Colour is just an Int, so = will work. After it is sealed, the only objects of type Colour are *red*, *green* and *blue*, and the only (non-polymorphic) function we can apply to them is *equal-colours*.

Of course, polymorphic functions (which work for every type of argument) can still be applied to colours, but expressions such as $red = 3$ are no longer valid.

20. Comments

Any text in a Ponder programme from a dash (–) to the end of a line is not processed by the compiler. This serves for the introduction of comments.

21. Useful Functions

A number of useful functions have been written already. Appendix 2 contains an annotated listing of the 'standard prelude', together with listings of functions from 'library' preludes.

Example Ponder Programmes

Examples 1–4 show the definitions of mundane constructs in terms of the basic language. This avoids the objection that built in constructs are *ad hoc* (see p49).

1. If

Boolean values may be represented as functions that take two arguments and return one of them. Thus

Type Bool $\triangleq \forall$ T. T $\rightarrow$ T $\rightarrow$ T;
Let *true* $\triangleq$ Bool: $\forall$ T. T $t \rightarrow$ T $f \rightarrow t$;
Let *false* $\triangleq$ Bool: $\forall$ T. T $t \rightarrow$ T $f \rightarrow f$;

Now *true a b* is *a*, and *false a b* is *b*.

To define the syntax **If-Then-Else-Fi**, we can arrange that the operators **Then** and **Else** bind like this: **If** b **Then** (t **Else** e) **Fi**. From this an implementation of **Else** is not difficult; (t **Else** e) is an expression that, when given a boolean argument, applies it to t and e:

Infix Else $\triangleq \forall$ T . T *thing-to-return-if-true* $\rightarrow$ T *thing-to-return-if-false* $\rightarrow$
Bool $b \rightarrow$
b thing-to-return-if-true thing-to-return-if-false;

and so **Then** merely applies the else-part to the boolean:

Infix Then $\triangleq$ Bool $b \rightarrow$
$\forall$ T . (Bool $\rightarrow$ T) *else-part* $\rightarrow$
else-part b;

Finally, we need to specify the associativity of the operators to get the right binding, and define the bracket:

Priority 9 **Else Associates Right**;
Priority 9 **Then Associates Right**;
Bracket If Fi $\triangleq$ *identity*;

Example

If *true*
Then *then-stuff*
Else *else-stuff*
Fi

The **If**–**Fi** bracket returns the expression unchanged, so expanding the **Then** first, we get:

(*then-stuff* **Else** *else-stuff*) *true*

which in turn becomes

true then-stuff else-stuff

and finally *then-stuff*.

A convenient improvement on this syntax is the **Elif** part, which reiterates the choice:

If *first-condition*
Then *first-answer*
Elif *second-condition*
Then *second-answer*
Elif ...
Else *last-answer*
Fi

If *first-condition* is satisfied, the value of this expression is *first-answer*, otherwise *second-condition* is tested, and if that is true, the value is *second-answer*, and so on until all the possible conditions have failed, in which case the value is *last-answer*.

If such an expression associates like this:

If c_1 **Then** (e_1 **Elif** (c_2 **Then** (e_2 **Else** e_e))) **Fi**

all that is needed is for **Elif** to produce an *else-part*:

Infix Elif $\triangleq\forall$ T. T *then-part* $\rightarrow$
T *conditional-expression* $\rightarrow$
then-part **Else** *conditional-expression*

This description differs slightly from the one in the standard prelude (Appendix 2): in the standard prelude capsules are used to prevent the misuse of **Elif** and **Else**; the above definitions make them interchangeable, whereas we would prefer that **Elif** was only used for reiteration of choices.

2. Pairs

In the current implementation Pairs are built in to the run-time system. There are two reasons for doing this. Both reasons are for the sake of getting an implementation going—neither is fundamental. The first is that it is easy to make a store-efficient implementation of pairs by using two words of store. The second is that the compiler and the run-time system must agree on how pairs are implemented.

It is possible, and would be worthwhile, to include the definition of pairs as part of the standard prelude.

Type Pair [L, R] $\triangleq$ $\forall$ U. (L $\rightarrow$ R $\rightarrow$ U) $\rightarrow$ U;
Let *left* $\triangleq$ $\forall$ L, R. Pair [L, R] *the-pair* $\rightarrow$
the-pair (L *left-element* $\rightarrow$ R *right-element* $\rightarrow$ *left-element*);
Let *right* $\triangleq$ $\forall$ L, R. Pair [L, R] *the-pair* $\rightarrow$
the-pair (L *left-element* $\rightarrow$ R *right-element* $\rightarrow$ *right-element*);
Infix , $\triangleq$ $\forall$ L, R. L *left-element* $\rightarrow$ R *right-element* $\rightarrow$ Pair [L, R]:
$\forall$ U. (L $\rightarrow$ R $\rightarrow$ U) *unpacking-function* $\rightarrow$
unpacking-function left-element right-element;

A pair is a function that takes an unpacking function and applies it to its left and right elements.

It would be possible for the implementation to detect objects of this type, and treat them specially.

3. Options

Like pairs, options are built in even though they can be defined within the language.

Type Option [T] $\triangleq$ $\forall$R.(T $\rightarrow$ R) $\rightarrow$ R $\rightarrow$ R

an option is thus an object that takes a function to apply if the object is there, and a default value to return if it is not.

4. Unions

It is often necessary to manipulate objects that may be of any one of a number of types. We can define a binary disjoint union like this:

Type Union [L, R] $\triangleq$ $\forall$ X. (L $\rightarrow$ X) $\rightarrow$ (R $\rightarrow$ X) $\rightarrow$ X

so that an object that is in the union of L and R takes two functions as arguments and applies one of them to the value that it represents. We need injection functions to convert an object to a union:

Let in_l $\triangleq$ $\forall$ L, R, X. L *object* $\rightarrow$
(L $\rightarrow$ X) *function-to-apply* $\rightarrow$
(R $\rightarrow$ X) *function-not-to-apply* $\rightarrow$
function-to-apply object;
Let in_r $\triangleq$ $\forall$ L, R, X. R *object* $\rightarrow$
(L $\rightarrow$ X) *function-not-to-apply* $\rightarrow$
(R $\rightarrow$ X) *function-to-apply* $\rightarrow$
function-to-apply object;

For practical purposes, a different definition of Union is desirable (see the standard prelude appendix).

5. A Parser for λ-expressions

This section gives an example to illustrate the use of new syntaxes in specific problem areas. The example I have chosen is a parser for λ-expressions, but the techniques used are applicable to parsers in general. Here λ-expressions conform to the following grammar:

$$\lambda\text{-expression} = \begin{cases} \text{“}\lambda\text{”name-token“.”}\lambda\text{-expression} \\ \lambda\text{-expression solid-expression} \\ \text{solid-expression} \end{cases}$$

$$\text{solid-expression} = \begin{cases} \text{“(”}\lambda\text{-expression“)”} \\ \text{name-token} \end{cases}$$

The idea behind the programme is to make the grammar itself be the parser. A grammar of this kind is constructed using two basic operations: succession and alternation, together with recursion. Readers not interested in the details may prefer to skip to p34 to see the result.

I have omitted priority declarations etc., to make the description shorter.

A parser is a function that takes a list of tokens and converts it into a tree. It may also fail, if the list of tokens does not conform to the grammar, and if it succeeds, there will often be some more tokens in the list that it has not ‘read’:

Type Parser [Tokens, Tree] $\triangleq$ Tokens $\rightarrow$ Option [Tree $\times$ Token]

5.1. Succession

We can combine two grammars to make another simply by writing one after the other. For example

two-letter-word = letter letter

means that a two letter word is a letter followed by another letter. We want to do the same for parsers. *parse-letter* **THEN** *parse-letter* is to be a parser for the corresponding grammar *letter letter*.

```
Infix THEN ≜ ∀ Tokens, T1. Parser [Tokens, T1] parser1 →
             ∀ T2. Parser [Tokens, T2] parser2 →
             Tokens input →
             If parser1 input Is T1 t1, Tokens rest
             → If parser2 rest Is T2 t2, Tokens rest
                → opt-in ((t1, t2), rest)
                Else nil
                Fi
             Else nil
             Fi;
```

This may be read as “If $parser_1$ returns a tree t_1, try $parser_2$ on the remaining tokens and return the two trees if that succeeds, otherwise fail.”

5.2. Alternation

The method for alternation is similar. If the first parser succeeds, the result is the resulting tree, otherwise try the second parser:

Infix OR $\triangleq$ $\forall$ Tokens, T_1. Parser [Tokens, T_1] *parser*$_1$ $\rightarrow$
Parser [Tokens, T_1] *parser*$_2$ $\rightarrow$
Tokens *input* $\rightarrow$
If *parser*$_1$ *input* **Is** ($T_1\times$ Tokens) *p*
$\rightarrow$ *opt-in p*
Else *parser*$_2$ *input*
Fi;

For example, *parse-letter* **OR** *parse-digit* is a parser that succeeds if either *parse-letter* or *parse-digit* succeeds (and hence parses either a letter or digit).

Note that unlike true alternation, **OR** is not symmetrical. If the first parser succeeds, it will not try the second parser.

Example

The parser for the grammar

$$\text{exes} = \begin{cases} \text{x exes} \\ \text{x} \end{cases}$$

is

Letrec *parse-exes* $\triangleq$ Parser [Tokens, Tree]:
parse-x **THEN** *parse-exes* **OR**
parse-x

Notice that the *parse-x* alternative has to be second. If it came first (on the left of the **OR**), *parse-exes* would succeed on the first x, and never consider the rest.

5.3. Left Recursive Rules

The operators **THEN** and **OR** together with recursion allow the definition of grammars. Unfortunately, left-recursive rules are a problem. Our example grammar contains the production λ-expression = λ-expression solid-expression. When translated to parsers, this becomes

lambda-expression $\triangleq$ *lambda-expression* **THEN** *solid-expression*

The **THEN** will try to parse a *lambda-expression* first, which entails parsing a *lambda--expression* first. And so on.

I introduce the operator **Left-Repeat-Option** to solve this problem. **Left-Repeat-Option** is similar to the postfix operator '*' commonly added to grammars as an extension, except that it builds up from the left. A parser $p \triangleq a$ **Left-Repeat-Option** b corresponds to the left recursive rule

$$\text{p} = \begin{cases} \text{p b} \\ \text{a} \end{cases}$$

Letrec *build-up-from* $\triangleq$
∀ Tokens, T_1, T_2. T_1 t_1 →
Parser [Tokens, T_2] $parser_2$ →
((T_1× T_2) → T_1) *pair-to-t1* →
Tokens *input* → Option [T_1× Tokens]:
If $parser_2$ *input* **Is** T_2 t_2, Tokens *input*
→ *build-up-from* (*pair-to-t1* (t_1, t_2)) $parser_2$ *pair-to-t1 input*
Else *opt-in* (t_1, *input*)
Fi;

Infix Left-Repeat-Option $\triangleq$
∀ Tokens, T_1, T_2. Parser [Tokens, T_1] $parser_1$ →
Parser [Tokens, T_2] $parser_2$ →
((T_1× T_2) → T_1)*pair-to-t1* →
Tokens *input* →
If $parser_1$ *input* **Is** T_1 t_1, Tokens *input*
→ *build-up-from t1* $parser_2$ *pair-to-t1 input*
Else *nil*
Fi;

An expression of the form *a* **Left-Repeat-Option** *b conversion-function* is a parser that works as follows. First it tries *a*, and if it succeeds, it uses *build-up-from* together with *conversion-function* to build a tree with the result of *a* as the leftmost leaf.

5.4. Right Recursive Rules

A further problem with this kind of parser is that **OR** applies the rules from left to right, and gives up when the first one succeeds. This means that if the grammar contains

$$\mathrm{a} = \begin{cases} \mathrm{thing}_1 \\ \mathrm{thing}_1\mathrm{thing}_2 \end{cases}$$

we have to write the parser for *a* as

(*parse-thing*$_1$ **THEN** *parse-thing*$_2$)
OR *parse-thing*$_1$

otherwise the parse would always succeed on the first *thing*$_1$, and never consider a *thing*$_2$.

A side effect of writing things this way round is that whenever the parser looks for an *a*, it will attempt (*parse-thing*$_1$ **THEN** *parse-thing*$_2$), and when this fails it will do the *thing*$_1$ again. A more efficient way to deal with this is **THEN-Optionally**:

Infix THEN-Optionally $\triangleq$
$\forall$ Tokens, T_1, T_2.
Parser [Tokens, T_1] *parser*$_1$ $\rightarrow$
Parser [Tokens, T_2] *parser*$_2$ $\rightarrow$
Parser [Tokens, ($T_1\times$ Option [T_2])]:
Tokens *tokens* $\rightarrow$
If *parser*$_1$ *tokens* **Is** T_1 t_1, Tokens *rest*
$\rightarrow$ **If** *parser*$_2$ *rest* **Is** T_2 t_1, Tokens *rest*
$\rightarrow$ *opt-in* ((t_1, *opt-in* t_1), *rest*)
Else *opt-in* ((t_1, *nil*), *rest*)
Fi
Else *nil*
Fi;

5.5. Tree Conversion

When a parser is constructed using **THEN**, it produces a pair of trees from the results of the two parsers. It is then necessary to convert this pair to the appropriate type for the parse tree. If *parser* is a Parser [Tokens, T_1] and *conversion-function* converts T_1 to a T_2, the expression

parser **AS** *conversion-function*

gives a Parser [Tokens, T_2].

Infix AS $\triangleq$
$\forall$ Tokens, T_1. Parser [Tokens, T_1] *parser* $\rightarrow$
$\forall$ T_2. ($T_1 \rightarrow T_2$) *convert-to-t*$_2$ $\rightarrow$ Parser [Tokens, T_2]:
Tokens *input* $\rightarrow$ Option [$T_2\times$ Tokens]:
If *parser input* **Is** T_1 t_1, Tokens *rest*
$\rightarrow$ *opt-in (convert-to-t*$_2$ t_1*, rest)*
Else *nil*
Fi;

This version allows us to write **AS** after a **Left-Repeat-Option**, to make things look more regular:

Infix AS $\triangleq$
$\forall$ Tokens, T_1, T_2.
(($(T_1\times T_2) \rightarrow T_1$) $\rightarrow$ Parser [Tokens, T_1] *part-parser* $\rightarrow$
($(T_1\times T_2) \rightarrow T_1$) *convert-to-t*$_1$ $\rightarrow$
part-parser convert-to-t1;

5.6. Literals

We need some means of introducing terminal symbols into the grammar. The polymorphic function *literal* takes an equality function for Tokens and a Token *lit* and returns a parser that succeeds if the first token on the input equals *lit*:

```
Let literal ≙∀ Token. (Token → Token → Bool) equal →
                Token lit → Parser [List [Token], Token]:
                List [Token] input →
                If input Is Token first, List [Token] rest
                → If equal first lit
                   Then opt-in (first, rest)
                   Else nil
                   Fi
                Else nil
                Fi;
```

For convenience we can define a prefix operator **Literal** for the particular type of token we are going to use:

```
Prefix Literal ≙ literal token-equals-token;
```

where *token-equals-token* is defined appropriately.

5.7. The parser for λ-expressions

Now that we have had enough definitions, we can get on with the parser for λ-expressions. First define a briefer type for parsers, assuming that we know what the types of the tokens and tree are:

```
Type Parser ≙ Parser [Tokens, Tree];
```

A tree will be a union of the representations of abstractions, applications and names, so we will need some routines to convert the results of **THEN** to trees:

```
Let parenthesised-expression ≙ Token open, Tree expression, Token close →
                                 expression;
```

We can discard the parentheses when building a tree.

```
Let make-name ≙ Token c → Tree: in-name c;
Let application ≙ Tree t1, Tree t2 → Tree:
                  in-application (t1, t2);
Let abstraction ≙ Token lambda, Token name, Token dot, Tree expression → Tree:
                  in-abstraction (name, expression);
```

The functions *in-name*, *in-application* and *in-abstraction* are intended to put their arguments in the appropriate place in the Tree union.

Finally, the parser itself.

Letrec *expression*, *solid-expression* $\triangleq$
Parser $\times$ Parser:
– *expression*:
(**Literal** *lambda* **THEN Literal** *name-token* **THEN Literal** *dot*
THEN *expression*
AS *abstraction*) **OR**
(*solid-expression* **Left-Repeat-Option** *solid-expression*
AS *application*),
– *solid-expression*:
(**Literal** *left-parenthesis* **THEN** *expression* **THEN Literal** *right-parenthesis*
AS *parenthesised-expression*) **OR**
(**Literal** *name-token* **AS** *make-name*);

Thus *expression string* will either evaluate to a pair *tree, rest* if an initial segment of *string* can be parsed to a tree *tree* or to *nil* otherwise.

Although it has been known how to write parsers in functional languages for some time, I believe that the use of an extensible syntax has a considerable impact on the ease of writing and comprehensibility of such programmes.

6. *Fix*

A well-typed version of the fixed-point combinator may be defined as follows:

Type $\mathsf{T}_\Theta\ [\mathsf{T}] \triangleq \mathsf{T}_\Theta\ [\mathsf{T}] \to (\mathsf{T} \to \mathsf{T}) \to \mathsf{T}$;
Let $\Theta \triangleq \forall \mathsf{T}.\ \mathsf{T}_\Theta\ [\mathsf{T}]\ \theta \to (\mathsf{T} \to \mathsf{T})\ f \to \mathsf{T}\colon\ f\ (\theta\ \theta\ f)$;
Let *Fix* $\triangleq \Theta\ \Theta$;

Whereupon *Fix* has type $\forall \mathsf{T}.\ (\mathsf{T} \to \mathsf{T}) \to \mathsf{T}$.

II

Design

The Model

The first programming languages were designed around the computers that they were used to programme. Later language designs tended to be abstracted away from the machine, usually with the intention of making programmes portable. Even these languages were based on an abstract notion of computers, rather than of computation, with the result that attempts to give them formal semantics have often revealed that their meaning is obscure. Ashcroft and Wadge [Ashcroft 1982] put forward the view that programming languages should be built upon a mathematical semantics, rather than the semantics built for the language. Such an approach is beneficial, but it is important to remember that the intention is to make the language intelligible—it would solve no problems if one were to base a language on a formalism that, although thoroughly rigorous, was incomprehensible.

In this chapter I examine the requirements for a formalism on which to base a language. I will use the term 'model' to mean any abstract interpretation that may be placed upon a formalism. While this includes mathematical models, I do not intend to be that specific: the only models that people really use are the ones in their minds!

Richard Young [Young 1981] has examined the physical models underlying three different calculators in conjunction with an exploration of the mental models constructed by their users. The three mental models may be described as 'no model', the 'simple model' and the 'analogy model'. The first was a simple 'four function' calculator. This calculator has a simple but obscure physical model involving some registers. While it was possible to predict the effect of a sequence of key-strokes from the physical model, it was hard to use this model to decide what was needed to perform a desired calculation. The result was that the user could not construct his own model for it, and instead used rote-learned recipes to perform calculations. The second calculator used 'reverse-polish' notation. The physical model was a stack with some operations, each of which removed elements from the top of the stack and pushed the answer back. A user of such a calculator could easily understand this model, and use it to predict the behaviour of the calculator. Unfortunately it is quite hard to convert the conventional notation for an expression into a sequence of key presses for the calculator. The last of the calculators examined was 'algebraic'. In this case the physical model is very complicated, involving a large number of registers, but the effect was that the user could operate it by analogy. The physical model was an inexact implementation of the usual algebraic notation—the user could type in an expression in its original form. Up to a point.

On the face of it, the last model seems to be the best—the user can understand the language in terms of a model he already knows. There are disadvantages. The real model of the calculator allows only a finite number of parentheses. Calculations are performed to limited precision, so the normal rules for algebraic transformation no longer apply. Furthermore, the lack of knowledge of the real model makes it difficult to recover from a mistake; in the case of the reverse Polish calculator, only part of a calculation need be lost. Most programming languages are a combination of the first and third kind of model. This causes programmes to be written by a combination of recipes and analogies.

The perfect model for a language would be simple enough that all programmers would

have the same idea as to what it meant, and yet comprehensive enough to allow all possible programmes. I suspect that no such model exists, and so compromise is inevitable. I chose λ-calculus for Ponder because it is simple, computationally complete and has a pleasing comprehensibility.

In what follows I examine the requirements for a model in more detail, and present some alternative models and my reasons for rejecting them.

1. Requirements

1.1. Referential Transparency

An important property of mathematical notation is that the meaning of an expression does not change with time: if two expressions are equal, they are always equal. This means that any expression may be substituted for any other equivalent expression. This property is called referential transparency.

Referential transparency is desirable because it allows subexpressions of a programme to be understood with the minimum of knowledge about the rest of the programme. In many languages, the lack of referential transparency means that apparently equal expressions are not equivalent. Although in the programme fragment

If $x = 2$
Then *something*; $p\ (x)$
Else ...,

x is established to be equal to 2, we cannot necessarily substitute $p\ (2)$ for $p\ (x)$ because *something* may change x.

For a programming language to be referentially transparent there must be no side effects. In other words, an expression *only* has a value; whenever it is evaluated the value is the same. An expression may not cause something to be printed—if something is to be printed, the environment must be defined in such a way that (part of) the value of a programme is printed. Similarly there may be no transfers of control.

1.2. Implementability

It would be nice if the model was such that it could be implemented exactly on a real machine. Unfortunately real machines have finite limitations. I know of no model that expresses these limitations without becoming excessively complicated. As a compromise, it is useful to imagine that although real machines have finite speed, they have unbounded storage. This compromise is not too damaging, as in practice the bound on storage is very large: one can always add another disc.

While a model must be implementable, it is important not to make efficiency a prime consideration. A requirement of efficiency often means 'efficiency for a particular architecture'. The rate of development of computer hardware is sufficiently great that what may have been a tremendously slow process on yesterday's machine may be a single operation on tomorrow's (see next section).

2. Other models

2.1. An Abstract Machine

The underlying model of many programming languages is an 'abstract machine'. Such a model is intended to reflect the behaviour of real machines in a direct way. This has the effect that it is hard for implementations of the language to take advantages of new developments in hardware. The BCPL abstract machine has a linear store, but many of today's machines have paged or tagged memories. The operations on arrays in Fortran are restricted to those that may readily be applied to simple vectors. Several machines now have hardware operations on descriptors.

A more serious defect of the abstract machine model is that the language will inevitably be implemented for several different machines. In a low-level model such as BCPL or C, the addressing ability or word size may differ. Even in a high-level abstract machine such as that of Algol 68 the arithmetic operations may differ in precision. Unless the programme can specify the precision to which an operation is carried out, the same programme may produce different results on different machines.

The final objection to this kind of model is that it forces programming to be at a lower level. The programmer will inevitably consider the way a programme will run. Programmes will be written so that details such as the layout of store, how a loop fits into the cache memory or whether to use half- or full-words take precedence over the clarity of expression.

2.2. Computational Models

A mathematical model of computation is surely a better candidate for a language? As well as the λ-calculus, there are Markov Algorithms [Markov 1962], Turing machines [Turing 1937] register machines [Minsky 1967], recursive functions, and the game of Life [Berlekamp 1982]. All of these have been proved equivalent in expressive power: a programme written for a Turing machine may be translated to the notation of any of the other models and vice versa. Which one is the most appropriate?

Unlike most of the other models, Markov algorithms have been used as the basis of a programming language: Snobol [Griswold 1968]. In order to make Snobol a usable language it was necessary to add features that are not directly expressible in Markov algorithms. In particular it is possible to define new patterns in Snobol, whereas in a Markov algorithm the pattern would have to be written out each time it was used.

Turing machines, register machines and the game of Life all share this problem. In essence, what is missing is an abstraction mechanism: the ability to name an algorithm and express it in terms of its parameters.

The λ-calculus is nothing but an abstraction mechanism.

A common addition to functional programming languages is "pattern matching," in which a function is defined on the structure of its parameter. This requires an equality function on possible arguments, but equality is not computable on λ-expressions. This would mean that there was only a restricted set of types of argument on which pattern matching could be performed, which would conflict with the requirement for orthogonality. Pattern matching functions and syntaxes may of course be defined where necessary, as with the **Is** and **Case** constructs in appendix 2.

3. Non Computational Models

A language could be based on a model that is not computational.

3.1. Logic

Logic is quite an attractive model, because it allows programmes to be written almost directly as specifications. To write a sort function, it is necessary only to specify that the result of sorting a list contains the same elements as the original list, but in order. An implementation of a language that uses logic in this direct manner will need to use a decision procedure to compute answers. Unfortunately, if a logic is at all expressive, it is probable that it will not have a complete decision procedure. It will not be possible to implement with even moderate efficiency.

Prolog [Clocksin 1981] falls into this category. The intended model for Prolog is Horn clauses [Horn 1951]. The actual model for Prolog is its interpreter. This means that a Prolog programme looks like a collection of Horn clauses, but behaves differently. In particular the order in which the terms of a conjunction appear is significant, so that what seems to be a correct description of the problem may not terminate. It would be possible to write a Prolog interpreter that produced an answer whenever there was one. The standard interpreter searches the goal space depth first, but a complete version would have to use breadth first search. This kind of search requires workspace that grows exponentially with the depth of the tree. Although I have said that efficiency is not of paramount importance, I think that it is clear that such an evaluation scheme would be unacceptably costly.

3.2. Grammars

Another candidate for a non-computational basis is the idea of production rules. Van Wijngaarden has suggested [van Wijngaarden 1981] that 2-level grammars can provide a basis for language semantics, but again they are not properly implementable, for similar reasons.

4. Examination of λ-calculus

The λ-calculus satisfies all my stated requirements: it is simple, mathematically precise and (as I will show later) implementable. It could be asked whether applicative order evaluation would be more appropriate in a programming language. My answer is that the requirement for simplicity makes normal order semantics essential. For example, the definition of the conditional (**Examples** above) would be impossible in an applicative order language without the introduction of a new primitive.

4.1. Deficiencies

All of the models I have considered are 'answer orientated'. A computation is assumed to begin with some data and compute an answer. In practice, the data are not always all available until the programme has begun to run (consider interactive programmes), and sometimes one would like the behaviour of a programme to depend on the time between data items.

The problem of the production of part of an answer before all the data arrives is largely solved by normal order semantics. In some cases the behaviour of a programme is not what one would wish (see the chapter **Analysis** below).

The requirement that a programme may depend on the timing of its input data is not modelled at all. One approach, taken by Henderson [Darlington 1982] is to add a new primitive, *non-deterministic-merge* on top of a λ-calculus model. There are models that do take time into account (Temporal logic, CCS etc.), but none is as attractively simple or has been so extensively explored as the λ-calculus. It is possible to deal with this problem by modifying the environment in which a programme runs, rather than its model.

Two other approaches include the one used in SKIM [Stoye 1984], and the concept of *hiatons* [Park 1982], which are objects inserted into the input to indicate the passing of time. A programme on the SKIM machine consists of a number of 'tasks' communicating via a non-deterministic micro-coded kernel. A task is a pure function, taking a list of messages from other tasks and returning a list of messages to the kernel. The kernel decodes the messages and passes them on to the appropriate task. Although I am inclined to prefer this kind of approach over the use of a different model I have not given the problem much consideration.

The use of λ-calculus as a model in some ways suffers from the problems caused by the use of an analogy-model. It is inevitable that a programme will contain expressions that are meant to reflect operations that are symmetrical in mathematics. For instance, in mathematics, both $\mathit{false} \wedge a = \mathit{false}$ and $a \wedge \mathit{false} = \mathit{false}$. No definition of $\wedge$ in the λ-calculus has precisely this property; if $\perp$ stands for a non-terminating computation either $\perp \wedge \mathit{false} = \perp$ or $\mathit{false} \wedge \perp = \perp$ because $\wedge$ must 'look at' one of its arguments first [Scott 1975]. Similar problems will arise in any model, and λ-calculus has no more problems in this respect than any other.

4.2. Pragmatics

Finally, I shall reconsider the question of the finite limitations of real machines. I

have said that it is reasonable to pretend that the machine has infinite memory. In many cases it is worthwhile to make the behaviour of a programme depend on how much free memory is left. The most justifiable case is when a programme cannot perform a required operation in the remaining amount of memory. It is reasonable to expect the programme to report this, and go on to tidy up, rather than to 'crash' and lose the effect of the work done so far. The requirement for referential transparency makes it very difficult to take care of this kind of eventuality; anything that says how much memory is available will vary with time.

The Type-System

Should a language have a type system at all? Type checking can detect mistaken applications of functions. A programme that contained an attempt to compute the square of a banana would be rejected. This is clearly a Good Thing [Sellar 1930], since the alternative method of detecting a mistake (namely running the programme) can consume a great deal of time. Indeed there are numerous cases of programmes in which the silly mistake was not detected until after it was put into service.

It has been argued that type checking itself is unnecessary; the programmer can work within a type discipline without the compiler checking it. While I accept that this is feasible for small programmes, and that large programmes can be broken down into small fragments, it is when the fragments are joined together that most of the type-errors crop up. Almost all useful programmes are written by more than one programmer: even the shortest programmes make some use of library subroutines. When someone makes use of a routine written by someone else it is quite a common mistake to attempt to use it with the parameters in the wrong order. A type checker reduces the amount of cross checking that the programmer must do.

1. The Scope of Type Checking

The design of a type-system inevitably involves compromise. At one extreme, a specification of a programme can be regarded as a type; type-checking in this regime corresponds to proof-checking [Martin-Löf 1975]. At the other, types could be restricted to the arities of functions. The checker would merely have to count numbers of arguments. The former requires the inclusion of excessive amounts of information in the programme and the checking algorithm is slow (or even non-existent). The latter would be barely helpful. In what follows I describe what I regard as the most reasonable interpretation of the idea of type-checking.

If the purpose of type checking is to detect errors before the programme is run, it is pointless to call anything that happens in a run a type error. To a certain extent, it is a matter of choice whether an error is a type error or not. If we regard a function for computing reciprocals as being from integers to integers, division by zero is a run-time error. The reciprocal function could be given a type that indicated that its domain does not include zero—in this case division by zero is prevented by the type system.

This argument can be carried further. To a large extent it is the choice of the programmer that determines the degree to which the type-system prevents errors. If the reciprocal function has a type that prevents it from being applied to zero the programme will always have to include instructions to test whether an integer is zero before its reciprocal may be taken. This may be seen as an unbearable imposition. Conversely, if a programme is dealing with a limited subset of the integers it may be a great deal of help to define them as a different type to ensure the integrity of the programme. Essentially type checking can

assure us that no function is ever applied to an object outside its *intended* domain. The type-system must allow this freedom of choice.

2. Choosing a Type System

I began my search for a polymorphic type system with the observation that although strong typing is helpful it has its drawbacks. Strong type-systems prevent useful abstractions. In a language such as Pascal or Algol 68 it is often necessary to write what is essentially the same routine more than once. The routine to append lists of integers is the same as the one for lists of characters in all but type. It would be better if one could abstract the type of the append function in some way, so that it did not include the type of the elements of the list. What is required is that the type checker should not prevent the application of functions to different types of argument when the difference is irrelevant. A polymorphic type-system allows one to state that the workings of a function do not depend on every aspect of its parameters.

One kind of polymorphism is found in Russell [Demers 1980] and Poly [Matthews 1982]. A type is regarded as being identified with its set of operators. Thus integer may be defined as having $+, -, \times, \div, =, 1$ and 0. This would be very useful if there were some way of deciding whether two operators were the same or not. The limitation is that in both of these languages an operator is regarded as the same as another if and only if it has the same symbol. Thus + for concatenating strings is be regarded as the same kind of operation as the one on integers, for the purposes of checking. A more severe limitation is that although types may involve expressions, the type checker only considers textual equivalence. Suppose $\mathsf{Array}\ (m, n)$ generates a type for arrays with lower bound m and upper bound n; although $2+4 = 4+2$, $\mathsf{Array}\ (1, 2+4)$ is regarded as being different from $\mathsf{Array}\ (1, 4+2)$.

The type system of ML (see **Background** above) seemed more promising, but still had some undesirable aspects. In ML, polymorphism is expressed in terms of *type-variables*. Thus $\lambda x.x$ has type $\alpha \to \alpha$. The arrow is used to indicate the type of functions, and may be viewed as an alternative to the more common set notation: $\mathsf{A} \to \mathsf{B}$ corresponds to B^{A}. α is a type-variable, and may take on any type as a value. Thus $\alpha \to \alpha$ stands for the set of types $\{\mathsf{Int} \to \mathsf{Int}, \mathsf{Bool} \to \mathsf{Bool}, \ldots\}$. My intuitions led me to regard the free type-variables as a little unpleasant. Indeed the freeness of the type variables makes it impossible to express the type of function that relies on its argument being polymorphic. $\lambda f.f(3,4), f(true, false)$ is not well typed in ML, even though it is clear that it could be applied to an object of type $\alpha \times \alpha \to \beta$.

Among the types that cannot be expressed in ML are pairs and unions. This means that in ML these must be built in, as must their constructor functions. For the purposes of making the basic language as simple as possible, this is undesirable.

The absence of declarations of type variables has an obvious solution. Type variables should be declared by quantifiers over type expressions. Now the type of $\lambda x.x$ is $\forall\alpha.\alpha \to \alpha$. Happily, this quantification of types also solves the problem of expressive power. The type of the function indicated above can be expressed simply by using the type of f:

$\forall\beta.(\forall\alpha.\alpha \times \alpha \to \beta) \to \beta \times \beta$. This expresses that the parameter f must be a polymorphic function taking a pair of arguments and returning a result of type β. We can say that the declaration of α is *local* to the parameter type.

Interestingly, this is not the only type that this function has. Another type is $(\forall\alpha.\alpha \times \alpha \to \alpha) \to \mathsf{Int} \times \mathsf{Bool}$. Neither of these types subsumes the other (see $\geq$, below). It is therefore impossible to infer a most general type within this new type system. I do not regard this as a problem, since I prefer to exchange the convenience of type-inference for the convenience of overloaded operators, which of themselves preclude inference.

The presence of locally quantified types allows types such as pairs to be expressed, so the type system need only include $\to$ for functions and quantification. This, however would be clumsy. It is desirable that the programmer should not have to write out the definition of pairs every time he writes a function that manipulates them. Evidently what is needed is a means of giving permanent names to types. Naming is not enough to solve the problem. A pair type involves two other types—the types of the left and right halves of the pair. The solution is to allow type declarations with parameters, called generators.

The ability to define named types leads to the question of whether they may be recursive. Experience has shown that the most natural forms of many data-structures are recursive in type. Examples are lists and trees. It is possible to define a list-like type that is not recursive ($\mathsf{List}\ [\mathsf{T}] \triangleq \forall\mathsf{A}.\ \mathsf{A} \to (\mathsf{T} \to \mathsf{A} \to \mathsf{A}) \to \mathsf{A}$), but a recursive definition is more natural.

Finally, a desirable facility is to be able to hide certain properties of a type. Although we may define a type called Colours that has objects called *red*, *green* and *blue*, which are represented with integers, we do not want to be able to perform operations such as $\times$ on Colours, nor do we want things like $1, 2$ and 3 to be counted as Colours. The mechanism I have adopted for this is similar to abstract types in ML. If a named type is declared to be a **Capsule**, and is subsequently sealed, type checking will then be by name equivalence (as in ML).

2.1. Summary of the Type System

The type-system to which this chain of reasoning brought me closely resembles that of MacQueen and Sethi [MacQueen 1982, 1984], but has fewer constructors.

$$\mathbf{Type} = \begin{cases} \mathbf{Type} \to \mathbf{Type} & \text{Function type} \\ \forall \mathsf{V}.\mathbf{Type} & \text{Quantified type} \\ \mathsf{Generator}[\mathbf{Type}_1, \ldots, \mathbf{Type}_n] & \text{Generated type} \\ \mathsf{V} & \text{Type variable} \end{cases}$$

Generated types may be sealed.

3. Type Validity

It is reasonably easy to see that an object of type $\forall\mathsf{T}.\mathsf{T} \to \mathsf{T}$ may be used anywhere that an object of type $\mathsf{Int} \to \mathsf{Int}$ or $\mathsf{Bool} \to \mathsf{Bool}$ may be used. In general, objects with

types quantified at the outer level may be used in place of objects having the type variable instantiated. This section defines the relationships between types and gives the type rules to which Ponder programmes must conform.

The type $\forall \mathsf{T}.\mathsf{T} \rightarrow \mathsf{T}$ is said to be 'more general than' the type $\mathsf{Int} \rightarrow \mathsf{Int}$. An object of type $\forall \mathsf{T}.\mathsf{T} \rightarrow \mathsf{T}$ may be used in more circumstances than an object of type $\mathsf{Int} \rightarrow \mathsf{Int}$. Before I can describe the rules that make sure that a Ponder programme contains no applications of functions to arguments for which they are insufficiently general, I must give the rules that define $\geq$, 'at least as general as'.

3.1. The Relation of generality between types

Rules $R0$ to $R8$ below define the relation $\geq$. V_n are type variables, T_n are arbitrary types (possibly with free variables), G_n are generators, Γ stands for a set of assumptions each of which is of the form $\mathsf{T}_1 \geq \mathsf{T}_2$ or $\mathsf{G}[\mathsf{V}_1, \ldots, \mathsf{V}_2] \triangleq \mathsf{T}$ and $\geq$ is as above.

Assumption

$$\Gamma \cup \{\mathsf{T}_1 \geq \mathsf{T}_2\} \vdash \mathsf{T}_1 \geq \mathsf{T}_2 \qquad R0$$

Reflexivity

$$\Gamma \vdash \mathsf{T} \geq \mathsf{T} \qquad R1$$

This means that from any set of assumptions Γ and type T we can deduce that $\mathsf{T} \geq \mathsf{T}$; i.e. any type is at least as general as itself.

Transitivity

$$\frac{\Gamma_1 \vdash \mathsf{T}_1 \geq \mathsf{T}_2, \quad \Gamma_2 \vdash \mathsf{T}_2 \geq \mathsf{T}_3}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{T}_1 \geq \mathsf{T}_3} \qquad R2$$

In rules such as this, assumptions are written above the line and conclusions below it. This rule may be read as 'If we can prove from Γ_1 that $\mathsf{T}_1 \geq \mathsf{T}_2$ and from Γ_2 that $\mathsf{T}_2 \geq \mathsf{T}_3$, then we can prove from the union of Γ_1 and Γ_2 that $\mathsf{T}_1 \geq \mathsf{T}_3$.'

Instantiation

$$\Gamma \vdash \forall \mathsf{V}.\mathsf{T}_1 \geq \mathsf{T}_1[\mathsf{T}_2/\mathsf{V}] \qquad R3$$

(Expressions of the form $\mathsf{T}_1[\mathsf{T}_2/\mathsf{V}]$ mean " T_1 with every free occurrence of V replaced by T_2.") An object that works for all types is more general than an instance of it.

Generalisation

$$\frac{\Gamma \vdash \mathsf{T}_1 \geq \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \geq \forall \mathsf{V}.\mathsf{T}_2} \ \mathsf{V} \text{ not free in } \mathsf{T}_1 \text{ or } \Gamma \qquad R4$$

If a type T_1 is more general than a type parameterised on V, regardless of the value of V, then T_1 is also more general than the generalised version of that type.

Function

$$\frac{\Gamma_1 \vdash \mathsf{T}_3 \geq \mathsf{T}_1, \quad \Gamma_2 \vdash \mathsf{T}_2 \geq \mathsf{T}_4}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{T}_1 \to \mathsf{T}_2 \geq \mathsf{T}_3 \to \mathsf{T}_4} \quad R5$$

A function that requires a less general argument is more general. This is explained by the following analogy. If a person is giving you something you want, the more he gives, the more generous he is. Conversely, if a person is taking something you owe him, the *less* he takes the more generous he is.

Result

$$\Gamma \vdash (\forall \mathsf{V}.\mathsf{T}_1 \to \mathsf{T}_2) \ \geq \ \mathsf{T}_1 \to \forall \mathsf{V}.\mathsf{T}_2 \qquad \mathsf{V} \text{ not free in } \mathsf{T}_1 \quad R6$$

A quantifier that does not appear in the parameter specifier of a function can be moved to the result.

Recursion

$$\frac{\Gamma \cup \{\mathsf{G}[\mathsf{V}_1, \ldots \mathsf{V}_n] \triangleq \mathsf{T}, \mathsf{G}[\mathsf{T}_1, \ldots \mathsf{T}_n] \geq \mathsf{T}_0\} \vdash \mathsf{T}[\mathsf{T}_1, \ldots \mathsf{T}_n / \mathsf{V}_1, \ldots \mathsf{V}_n] \geq \mathsf{T}_0}{\Gamma \cup \{\mathsf{G}[\mathsf{V}_1, \ldots \mathsf{V}_n] \triangleq \mathsf{T}\} \vdash \mathsf{G}[\mathsf{T}_1, \ldots \mathsf{T}_n] \geq \mathsf{T}_0} \quad R7$$

this rule allows the comparison of recursive types.

Expansion

$$\Gamma \cup \{\mathsf{G}[\mathsf{V}_1, \ldots \mathsf{V}_n] \triangleq \mathsf{T}\} \vdash \mathsf{T}[\mathsf{T}_1, \ldots \mathsf{T}_n / \mathsf{V}_1, \ldots \mathsf{V}_n] \geq \mathsf{G}[\mathsf{T}_1, \ldots \mathsf{T}_n] \quad R8$$

this gives the meaning of definition (the case with the generator on the left hand side of $\geq$ is covered by $R7$).

4. Type Validity of Expressions

This section presents the rules to which valid Ponder programmes must conform. In general a programme will consist of a 'casted' expression, the type of which denotes the environment in which the programme is intended to run. Most programmes will be required to have type List [File-action] and are therefore treated as if they were preceded by a cast: the type checker will be given an expression like List [File-action]: *programme*.

A programme p is type-valid if a statement of the form $\mathsf{T} : p$ for some T may be proved within the following rules.

The notation $\mathsf{T} : e$ means that e has the type T, and Γ is a set of assumptions as before but may also include assumptions of the form $\mathsf{T} : v$.

Variable Assumption

$$\Gamma \cup \{\mathsf{T} : v\} \vdash \mathsf{T} : v \quad V0$$

Application

$$\frac{\Gamma_1 \vdash (\mathsf{T}_1 \to \mathsf{T}_2) : e_1,\ \Gamma_2 \vdash \mathsf{T}_1 : e_2}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{T}_2 : e_1\ e_2} \quad V1$$

Function

$$\frac{\Gamma \cup \{\mathsf{T}_1 : v\} \vdash \mathsf{T}_2 : e}{\Gamma_1 \vdash \mathsf{T}_1 \to \mathsf{T}_2 : (\mathsf{T}_1 v \to e)} \text{ where } \Gamma = \Gamma_1 - \{\mathsf{T} : v \mid \mathsf{T} \text{ is a type}\} \quad V2$$

Cast

$$\frac{\Gamma \vdash \mathsf{T}_1 : e}{\Gamma \vdash \mathsf{T}_1 : (\mathsf{T}_1 : e)} \quad V3$$

Generalisation

$$\frac{\Gamma \vdash \mathsf{T} : e}{\Gamma \vdash \forall \mathsf{V}.\mathsf{T} : \forall \mathsf{V}.e} \ \mathsf{V} \text{ not free in } \Gamma \quad V4$$

Restriction

$$\frac{\Gamma_1 \vdash \mathsf{T}_1 : e,\ \ \Gamma_2 \vdash \mathsf{T}_1 \geq \mathsf{T}_2}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{T}_2 : e} \quad V5$$

Apart from overloaded operators, all other constructs may be dealt with by expanding them out. An overloaded operator is expanded into a function application of the most recent version of the operator and this application is checked. If the check fails, the next most recent version is tried, and so on until one is found which works (otherwise the programme is invalid).

Syntax

The syntax of a programming language is what the reader sees. It is the vehicle by which the meaning of the programme is transmitted. The ease with which a programme is understood by the reader is something that is difficult to measure, and difficult to predict. Without performing extensive psychological tests it is impossible to determine the best syntax for a given concept. Inevitably I have had to design the syntax of Ponder without recourse to such tests, but Weinberg's book [Weinberg 1971] contains some useful information. Some aspects of the readability of the programme appear to be self evident. Different constructs should be sufficiently different that they are easily distinguished. The syntax of Lisp, while being admirably simple suffers greatly from this problem: all Lisp programmes look much the same. If a notation for the problem exists already, it would be useful if that notation could be mimicked in the programme. Why should one have to write

```
CALL F01CKF (C, A, B, 8, 8, 8, Z, 1, 1, IFAIL)
```

to multiply two matrices when one would normally use $A \times B$? The syntax should allow expressions to be concise, without their becoming over dense—a programme should include names of objects or functions to help the reader, rather than be a sequence of symbols unrelieved by words.

It would be impossible to include all possible notations as part of a language. If the syntax is to resemble that of the problem, it is inevitable that new syntaxes must be defined. The syntax must therefore be extensible to at least some extent.

Previous attempts at extensibility have produced rather unwieldy languages. If the programmer is allowed to define arbitrary syntaxes the task of parsing a programme becomes an exceedingly slow one. Furthermore, a completely flexible syntax includes so many new ways of making mistakes that it is impossible for the compiler to generate sensible error messages. To avoid these problems I have placed severe limitations on the way in which the language may be extended, but attempted to make the extension mechanism sufficiently powerful to allow the representation of most plausible notations.

The first level of the syntax of a language is lexical syntax: how the string of characters representing the programme is broken into symbols.

1. Lexical Syntax

To make symbols readily recognisable by both the compiler and the human reader, they are divided into distinct classes. The kind of a symbol is determined exclusively by its font. This restriction makes it much easier to construe a sequence of symbols. Without it, it would be more difficult to see which symbols were operators and which were names. It makes it easier to learn to recognise certain constructions by eye.

1.1. Names

Names are needed for the parameters of functions and for types. A name must describe the meaning of the object that it represents. This cannot often be done in one word. If application is to be represented by juxtaposition (with spaces), spaces clearly cannot be used to separate the words of a name. Hyphens are used to separate the words in names, as in *long-name*.

The names of types are distinguished from the names of variables by the font. A parameter name is written in *italic*, operator names and keywords in **bold**, and types in sans serif.

Layout has the usual purpose of separating symbols—*name name* means the same as *name name*, but is different from *namename*.

The remaining classes of symbols are the single character symbols, which are {, },(,),[,], comma and semicolon, and the 'special symbols', which are all the characters that are in none of the other classes, and are used to form new operators.

2. Language Syntax

2.1. Function Application

In many programming languages, the application of a function (or procedure) is written with parentheses round the argument, as in $f(x)$. In Lisp, the parentheses surround the whole application: $(f\ x)$. In a functional language, the most common syntactic object is the application, so either of the above syntaxes would result in programmes containing large numbers of parentheses. In Ponder, as in λ-calculus, the application of a function f to an argument g is written $f\ g$. Application associates to the left, so $f\ g\ h$ means the same as $(f\ g)\ h$.

2.2. Function Representation

The λ-notation for functions is syntactically already rather clumsy. For the representation of functions in this kind of typed language we need to specify the types of the parameters. The addition of types to λ-notation would result in an ugly syntax.

Functions are represented as Type *name* $\rightarrow$ *body*, which means the same as λ*name* : Type.*body* in the notation of the typed λ-calculus, but provides all the information in one compact form. I chose '$\rightarrow$' rather than '.' to separate the parameter from the body, to reflect the notation for the types of functions.

2.3. Types

The syntax of types is described in the previous chapter. Types may be declared just as types, or as recursive types, and either of these may be qualified as capsules:

Type type-name [parameters] $\triangleq$ type;
Rectype type-name [...] $\triangleq$ ...;
Capsule Type type-name [...] $\triangleq$ ...;
Capsule Rectype ...;

A capsule must be sealed:

Seal type-name;

2.3.1. Casts

It is sometimes necessary to give the type of an object explicitly, for example in **Letrec** declarations (see below). This is written with a *cast*, which is the type, then a colon and then the object. This is the opposite way round to the notation commonly used. The reason for this is that the parameters come before the body of a function with their types. With casts this way round, the type of a function with a casted body may readily be read off: Int $i \rightarrow$ Int : *potentially-large-expression* clearly has type Int $\rightarrow$ Int.

2.4. Extensibility

As was mentioned in the preliminary part of this chapter, previous attempts at extensibility have made languages difficult to use.

Algol 68 allows the definition of new infix and prefix operators, so is in a sense extensible. This extensibility is sufficiently restricted that an Algol 68 compiler can still parse a programme in reasonable time and give comprehensible diagnostics. This prompted me to have a similarly limited extensibility, but just a little more was needed.

The extension mechanism in Algol 68 was sufficiently limited that most of the basic constructs had to be built in. This meant that the predefined syntax of Algol 68 was quite large, and that it was impossible for new constructs to be added. I have found that the introduction of only one more mechanism allows the definition of analogues of most syntaxes.

An examination of popular language constructs suggested to me that they could be divided into two classes: declarations and expressions. Expressions such as 'If ... Then ... Else ...' can be broken down into infix operators (**Then** and **Else**), and a prefix operator (**If**). The alternative syntax for conditionals, 'If Then Else Fi' cannot be treated in this way. What is needed is the ability to define new brackets. 'If Then Else Fi' is now just a bracket—**If** ... **Fi**— and two infix operators (see **Examples** above).

2.4.1. Distfixes

As an alternative to brackets, infixes and prefixes I considered having one more general mechanism of 'distfix' operators, as in HOPE [Burstall 80]. Distfix operators are defined by means of patterns. This requires that the definition should include indications of where the arguments are to go, which in turn necessitates the inclusion of their types.

I finally rejected this idea when I realised how complex a mechanism would be needed to allow the definition of a syntax with optional, repeatable parts such as 'Elif' in 'If Then

Elif Then ... Else Fi'. The easiest thing to do is to define Then, Else and Elif as infix operators after all.

2.4.2. Precedence

The implied grouping of a bracket operator is always obvious. Infix operators are more ambiguous: $a\ Op_1\ b\ Op_2\ c$ can be parsed in two ways: $(a\ Op_1\ b)\ Op_2\ c$ and $a\ Op_1\ (b\ Op_2\ c)$. The simplest solution is to require that all infix expressions are made explicit by the inclusion of parentheses. This would defeat the object. It would be unacceptable if the syntax for a conditional had to be cluttered with parentheses.

The prototype version of Ponder uses a numeric priority mechanism similar to that of Algol 68, with the addition of an indication of the associativity of the operator. This is unsatisfactory in a number of ways, and I consider alternatives in the concluding chapters.

2.4.3. Declarations

Bracket and infix operators provide a means to define most expression syntaxes. Declarations are still a problem. A language in which the only means of introduction of new names was as the parameters of functions would be very tedious to use. As with expressions it would have been preferable to allow the definition of new declarative constructs. A declaration necessarily involves the manipulation of a name that has not yet been declared and is of unknown type, and such manipulations are beyond the scope of λ-calculus. I have been unable to find a simple way of treating this.

In the prototype of Ponder, I have therefore been forced to include a number of predefined declaration mechanisms. There are declarations for each of the syntactic extension mechanisms. There are declarations for types, and there are declarations for names. As the simplest kind of declaration I chose the 'Let' declaration [Landin 1966].

Let *name* $\triangleq$ *value*;
expression

declares *name* to have the value *value* in the expression *expression*. It is thus equivalent to

$$(\lambda name.expression)\ value.$$

Worse still, the definition of recursive functions using **Let** with **Fix** was so cumbersome that I was forced to introduce a special **Letrec** declaration.

$$\textbf{Letrec}\ name \triangleq value;\ expression$$

is equivalent to

$$(\lambda name.expression)\ (\textbf{Fix}(\lambda name.value))$$
$$\text{where } \textbf{Fix} \triangleq ((\lambda y.\lambda f.f\ (y\ y\ f))(\lambda y.\lambda f.f\ (y\ y\ f)))\dagger$$

In order that the body may be type-checked, it is required that the *expression* have a cast—because the type of the expression may depend on itself.

---

† for a typed version of **Fix**, see **Examples 6**

A further deviation from the rule of simplicity is the simultaneous declaration. Declarations of the form $a_1, \ldots, a_n \triangleq$ *expression* (with **Let** or **Letrec**) where *expression* must have a type $\mathsf{T}_1 \times \ldots \times \mathsf{T}_n$ declare the names $a_1 \ldots a_n$ to be the corresponding values extracted from *expression*. This constitutes quite a severe breach of the principle of simplicity. The problem of new declarations is discussed further in the concluding chapters.

3. Summary of syntax

Below is an informal summary of the syntax.

$$
\begin{aligned}
\text{expression} &= \begin{cases} \text{declaration; expression} \\ \text{function-representation} \\ \text{application} \\ \text{infix-application} \\ \text{prefix-application} \\ \text{bracket-application} \\ \text{casted-expression} \end{cases} \\
\text{declaration} &= \begin{cases} \text{type-declaration} \\ \text{type-seal} \\ \text{type-infix-declaration} \\ \text{infix-declaration} \\ \text{prefix-declaration} \\ \text{bracket-declaration} \\ \text{let-declaration} \\ \text{letrec-declaration} \end{cases} \\
\text{function-representation} &= \text{Type name} \rightarrow \text{expression} \\
\text{application} &= \text{expression expression} \\
\text{infix-application} &= \text{expression op expression} \\
\text{prefix-application} &= \text{op expression} \\
\text{bracket-application} &= \text{left-bracket expression right-bracket} \\
\text{casted-expression} &= \text{type : expression}
\end{aligned}
$$

III

Implementation

Implementing the Model

This chapter is a description of the techniques used in the implementation of the model of Ponder. The approach adopted combines supercombinators with strictness detection.

1. History

In **Background** I pointed out that normal order evaluation is preferable to applicative order because it is guaranteed to find a terminating result, if one exists. Implementations of Ponder must have normal order semantics.

The most obvious way to implement normal order semantics is to perform the reduction of the programme in exactly the way that normal order reduction is described. Each application of a function would involve the substitution of the unevaluated argument value for the bound variable. Such a reduction scheme is very slow because of the time taken to locate each instance of the bound variable, and in copying both the body of the function and each instance of the argument.

A major improvement on this naïve technique is lazy evaluation. Rather than copy an argument for each instance of the parameter, lazy evaluation takes advantage of the referential transparency of functional programmes and uses a pointer to the argument. When any instance of the argument is later evaluated, it is overwritten with its result, so that all other occurrences benefit from the work done. I call this property (the overwriting of expressions to achieve single evaluation) 'laziness'. Lazy evaluation on its own does not remove the necessity of copying the function as it is applied. Other techniques are needed for this.

One method is to use 'closures' to represent the application of a function to its arguments. A closure consists of a pointer to the function and a vector containing the arguments. When evaluating the body of the function, the bound variables are looked up in the vector. An alternative is to use combinators in the manner suggested by Turner (see **Background** above). This has the advantage that all variables are removed from the programme when it is run, removing the need for substitution completely. An early implementation of Ponder used this technique, but was found to be rather slow. The inefficiency arises from the small amount of work that was done in each combinator: experiments showed that only 1% of the time was spent reducing combinators; the rest was spent by the evaluator looking for the next combinator to reduce.

Hughes [Hughes 1982] describes a solution to this problem (see **Background** again), which he calls super-combinators. Supercombinators are combinators chosen individually for the compilation of a particular programme. Rather than transform programmes into combinations of a small set of pre-defined combinators, Hughes suggested that the compiler should determine the largest possible combinators for each programme. Lazy evaluation can only occur when an expression is passed as an argument to a combinator. Lazy evaluation is only needed when an expression *may* be evaluated. Supercombinators are an attempt to pass expressions as arguments only when necessary.

The algorithm developed by Hughes ignores some of the pragmatic properties of evaluation, and as a result some expressions are converted to smaller combinators than necessary. This produces a programme that allows for the possibility of the lazy evaluation of expressions where no useful work can be done. I term the presence of such expressions 'redundant laziness'. The techniques described below go some way toward alleviating this.

Another applicable technique is programme transformation. I have not used general programme transformation because the benefits it produces are largely distinct from the problem of evaluating functional programmes directly—we could advantageously transform one programme to another, but we need to evaluate the resulting one efficiently too. Programmes could be transformed into imperative ones, but this then precludes the use of evaluation techniques that rely on referential transparency.

2. On Not Building Trees

In a combinator evaluator (whether for pre-defined- or super-combinators) unevaluated expressions are generally represented as trees. Each node of such a tree is an 'application cell' (F, A), which represents the application of a function F to an argument A, each of which may be either another application cell or an instance of a combinator. A secondary use of these cells is to record the value of the expression when it is evaluated.

The construction of such a tree necessarily involves the consumption of space that will have to be released when the application is no longer in use. This necessitates the use of a heap, together with a garbage collector. The allocation of a new cell takes time when it occurs, and gives rise to the expectation of more time wasted because of garbage collections. Another overhead is the time taken to overwrite a cell with its evaluated value; something that is clearly a waste if the cell is never again examined.

The optimisations below all go towards the elimination of unnecessary tree building.

3. Conditional Expressions

In an expression of the form **If** a **Then** b **Else** c **Fi**, either b is needed or c is needed according to the value of a. To evaluate this expression, both b and c would be constructed as trees in order to instantiate any variables they need, but upon the evaluation of a, one would be immediately discarded.

Here the type system comes to the rescue. All boolean values have the same type ($\forall\ \mathsf{T}\ .\ \mathsf{T} \to \mathsf{T} \to \mathsf{T}$), and so the application of a boolean can be detected by the compiler, and more sensible code can be generated. The resulting programme will contain code for the evaluation of a, the value of which will then be tested, and either b or c will be evaluated. In the compiler this is generalised to all types of the form $\forall\mathsf{T}.\mathsf{T} \to \ldots \to \mathsf{T}$, the *selector* functions.

Many functions test conditions, and so this optimisation is very useful.

4. Improvements to Hughes' Algorithm

As mentioned above, Hughes' algorithm divides programmes into combinators that preserve laziness. Whenever an expression may be lazily evaluated it is 'lifted out' as a parameter to a combinator. Not all expressions can be reduced. Hughes' algorithm does not take this into account, and can therefore be improved. I shall describe examples of this, and the means to avoid it.

4.1. Combinators with Too Few Arguments

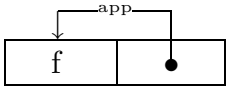
In the expression $\lambda h.\mathbf{S}\ \mathbf{B}\ h\ h$, **S B** is free of the parameter h, and therefore is regarded by Hughes' algorithm as being a candidate for lazy evaluation, because its value can be computed regardless of the value of h. This λ-expression would be transformed into the combinator application $\alpha(\mathbf{S}\ \mathbf{B})$ where $\alpha \triangleq \lambda\beta.\lambda h.\beta\ h\ h$. This application requires a tree for **S B**, and α must form a tree. However, we know from the definition of **S** that it needs three arguments before it can do any work. The expression **S B** $h\ h$ can be evaluated directly by calling the subroutine that performs **S**, with those arguments. We can therefore generate better code for this if **S B** is not lifted out.

In the general case, the modification to Hughes' algorithm is that expressions should not be lifted out if they consist of a combinator applied to too few arguments. Of course, each of the arguments of such combinator must be considered for lifting, as they may be evaluable.

4.2. Recursion

In his paper, Hughes suggests that it may be possible to modify the algorithm to deal with recursion in a more efficient way. Here I consider such improvements.

Recursion is introduced by use of the **Fix** combinator. **Fix** finds a fixed-point of a function: **Fix** $f \triangleq f$ (**Fix** f). Although **Fix** may be expressed in terms of λ-calculus (see **Background**, above), the use of such a function is not as efficient as possible. A direct implementation of **Fix** can reduce **Fix** f to an application cell like this:



which collapses the new **Fix** f into a pointer to the original. This, together with lazy evaluation allows the construction of cyclic structures. The evaluation of f can result in a structure that involves the first argument directly, and this structure will be overwritten on the original cell.

The Ponder compiler could potentially detect occurrences of versions **Fix** textually and replace them with applications of a pre-defined version. It does not do this, because most applications of **Fix** are introduced by the compiler itself, for letrec declarations, so it merely introduces the pre-defined one.

The important property of the pre-defined **Fix** is that it does not re-compute **Fix** f every time. Better results can be achieved if **Fix** f is replaced by explicit recursion in the

code for the combinator. If F is a combinator $\lambda yf.\lambda arg_1.body$, we can replace instances of **Fix** F with a combinator F' recursively defined as in $F' \triangleq F\ F'$. This transformation preserves laziness because **Fix** F is a constant—it contains no free variables.

It is often the case that the argument of **Fix** is not a combinator. This most frequently occurs for local letrec declarations. If a function is defined recursively and has free variables $a_1 \ldots a_n$, the resulting combinator expression will be of the form **Fix** $(F\ a_1 \ldots a_n)$, where $F \triangleq \lambda a_1 \ldots a_n.\lambda yf.body$. In this case, the analogous transformation is to replace F with $F' \triangleq \lambda a_1 \ldots a_n.F\ a_1 \ldots a_n\ (F'\ a_1 \ldots a_n)$. This could mean, however, that $F'\ a_1 \ldots a_n$ is evaluated many times, and so laziness would not be preserved.

A subset of this class of transformations is valid. If F is a combinator requiring more than $n+1$ arguments, **Fix** $(F\ a_1 \ldots a_n)$ can only evaluate to an application of F to too few arguments, and so no useful work may be done. In such a case, the replacement of F with F' loses no laziness.

4.3. Single Occurrences of Lazy Expressions

If an expression is used exactly once, there is obviously no advantage to be got from evaluating it lazily. Some such cases can be detected at compile-time. Commonly an expression that returns a function is immediately applied to another argument. If the expression is never applied to any other argument, it is effectively used once. For example, Hughes' algorithm will convert $M \triangleq \lambda a.\lambda b.a \times a + b$ to $M \triangleq \lambda a.\alpha(a \times a)$ where $\alpha \triangleq \lambda a_2.\lambda b.a_2 + b$. This transformation is justifiable if M occurs in a programme like this: $(\lambda f.f\ 3 + f\ 4)(M\ 8)$, because $M\ 8$ involves a computation of 8×8 that would be repeated if not bound to a parameter. Conversely, if all the applications of M are to two arguments, no benefit can be had from the binding, and the overhead of remembering $a \times a$ is wasted.

The algorithm to deal with this may be approximately summarised as "count the number of arguments given to each λ-expression, and take this information into account when extracting the combinators". There is more to it than this. If M is as before, and occurs in $N \triangleq \lambda a.\lambda b.1 + M\ a\ b$ and nowhere else, M would be seen as always having two arguments. If N occurs with one argument, $M\ a$ will be lifted out of N when extraction occurs, invalidating the information about M. The correct solution is analogous to the use of 'levels' in the original algorithm. In the original algorithm, the level of a bound variable increases by one for each λ binding. In the modified algorithm the level increases by one up to the point where all applications have at least that number of arguments at the same level.

With M as before, if the only applications of M have two arguments, both a and b have the same level. If M and N both occur, the level of the parameters of M depends on those of N.

Unfortunately this optimisation can only be applied when *all* occurrences of a function have been found, so it precludes separate compilation.

5. Compile Time Reduction

In traditional compilers, constant expressions are 'folded'. An expression such as $2+2$ is replaced by an evaluated version (4). An analogous concept for functional languages is the reduction of certain combinators at compile time.

If an expression is to be reduced at compile time, it must be certain to terminate, because otherwise the compilation may not terminate. Laziness must also be preserved. Furthermore, the reduction must not result in significant growth in the size of compiled code. The requirement for termination can be met by restricting the class of combinators that may be applied, and laziness is preserved by more restrictions. The requirement about code growth is harder to quantify.

Laziness is only lost if an evaluable expression is duplicated. Thus an application may not be applied if an evaluable expression is bound to a parameter that occurs more than once in the body of the candidate function. For termination, a sufficient restriction is that an argument must either be a parameter to an outer function or the corresponding parameter must occur in the body less than twice.

6. Applicative Order Evaluation

The overheads of normal order evaluation are significantly greater than those for applicative order. If an argument is evaluated before a function call, there is no need to build a tree to represent it in an unevaluated form. If applicative order can be used without changing the meaning of the programme, it clearly should be.

Mycroft [Mycroft 1981] describes an algorithm to detect whether a function is 'strict on' any of its parameters. A function is strict on a parameter if, when it is passed a non terminating argument for that parameter, it will not terminate. More directly, it is strict on a parameter if all applications of the function will result in the evaluation of the corresponding argument. The Ponder compiler uses a modified version of this algorithm to determine when applicative order evaluation may be used.

Wray [Wray 1984] implemented the algorithm for Ponder and included some improvements to do with the passing of functions as arguments. If a parameter is used as a function, the application must always be built as a tree because it is impossible to determine in advance whether the argument is going to be a combinator or not. In such cases there is no advantage in forcing the evaluation of the argument.

7. Optimisations at the Machine Code Level

The combinators are compiled into a simple abstract machine code (see Appendix 1). Although obvious benefits would accrue from the use of peephole optimisation, the techniques are already well understood, so none have been included. A version of the Ponder compiler has been constructed for the IBM 3081 that does do register optimisation [Tillotson 1984], resulting in code that runs about five times faster.

One significant machine level optimisation is the removal of tail recursion. Since functional programming relies heavily on recursion, the consumption of stack in tail recursive functions is undesirable. The code-generator includes a very simple tail recursion removal mechanism in which the entire stack frame is moved back along the stack at tail recursive calls.

8. Performance Comparisons

This section presents measurements of the performance of the system. For benchmarking, I have used the two functions *nfib* and *tak*, and a programme to sort numbers. Peter Henderson suggested *nfib* as a benchmark because its value is the number of calls that it takes to evaluate, so *nfib n* divided by the time taken for *nfib n* is a number of calls per second and *tak* has been popularly used as a benchmark for Lisp.

```
Letrec nfib ≙ Int n → Int:
              If n ≤ 1
              Then 1
              Else 1 + nfib (n − 1) + nfib (n − 2)
              Fi
```

```
Letrec tak ≙ Int x → Int y → Int z → Int:
             If not (y < x)
             Then z
             Else tak (tak (x − 1) y z)
                      (tak (y − 1) z x)
                      (tak (z − 1) x y)
             Fi
```

```
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) rest)

                 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;
```

For the timings, *quick-sort* was applied to a list of 41 random numbers. The following timings were obtained on a Cambridge 8MHz 68000 system.

Optimisation		*time for*	*time for*	*time for*	
Mycroft	Improve	*quicksort*	*tak* 18 12 6	*nfib* 20	*nfib* calls/s
		6.88s	157.5s	12.69s	1700
on		6.80s	130.5s	12.69s	1700
	on	3.36s	36.6s	8.79s	2500
on	on	2.94s	7.4s	3.02s	7250

An 'on' in the column 'Mycroft' means that strictness detection was performed for those tests, an on in the 'Improve' column means that the improved version of Hughes' algorithm was used.

For comparison with more conventional languages on the 68000, *tak* 18 12 6 takes 9.26s in compiled Cambridge Lisp and *nfib* written in compiled Cambridge Lisp runs at 7400 calls/s, in BCPL at 14 000 and in Algol 68C at 30 000.

Notice the interaction between the two sets of optimisations. Without the improvements to Hughes' algorithm, Mycroft's makes relatively little difference. The reason is that the unimproved Hughes' algorithm produces small combinators with function arguments, which prevent the strictness detection from having any effect. In the case of quicksort, the strictness detection is ineffective even with the improved abstraction algorithm. Mycroft's algorithm does not have any means of dealing with programmes that manipulate lists, so all the arguments to list constructing operations are evaluated lazily.

The Type Checking Algorithm

This chapter describes an algorithm that checks whether a Ponder programme conforms to the rules laid out in section 3 of **The Type-System** above. The type checker is presented as a function *type-check*, which is defined recursively on the structure of Ponder programmes. For simplicity, the mechanism to deal with recursive types is presented separately. Without recursive types, all generators except capsules may be replaced with their expansions, so the case analyses of types are written under the assumption that this has been done. Capsules with no parameters may be treated as if bound by $\forall$. In comparisons involving capsules with parameters, the generator names are first compared for equality and only if they are equal are then expanded out.

In what follows, things represented by V are type variables, T are arbitrary types and e are Ponder expressions. Whenever $\forall \mathsf{V}.\mathsf{T}$ occurs, it is assumed that V is distinct from all other variables encountered (in practice this is implemented by renaming).

1. *type-check*

The type checker *type-check* takes a triple $(\mathbf{A}, \tau, e)$ where e is a Ponder expression, $\mathbf{A}$ is a set of assumptions about type variables and τ is a set of typings of the form $\mathsf{T} : e$. Each element of $\mathbf{A}$ is one of

- Fixed V,
- $\mathsf{T} \geq \mathsf{V}$,
- $\mathsf{V} \geq \mathsf{T}$.

The result of *type-check* $(\mathbf{A}, \tau, e)$ is a pair $(\mathbf{A}', \mathsf{T})$ where T represents the type of e under the new assumptions $\mathbf{A}'$, if e is valid, and Fail otherwise. In other words, *type-check* $(\mathbf{A}, \tau, e) = (\mathbf{A}', \mathsf{T})$ implies that $\mathbf{A}' \cup \tau \vdash \mathsf{T} : e$.

The type checker requires two subsidiary functions *valid*, and $\boxed{\geq}$; *valid* checks a set of constraints and either returns the same set of constraints or fails, and $\mathsf{T}_1 \boxed{\geq} \mathsf{T}_2$ is the set of assumptions needed to show that $\mathsf{T}_1 \geq \mathsf{T}_2$.

The definition of *type-check* is given as a case analysis of possible expressions, together with a short description of the idea underlying each particular case.

Variables

The type of a variable is given by its environment:

$$\textit{type-check}\ (\mathbf{A}, \tau, v) = (\mathbf{A}, \mathsf{T}),\ \text{if}\ (\mathsf{T} : v) \in \tau;\ \text{otherwise Fail}$$

Application

To check an application, check the function and argument, and then calculate the result type:

$$\begin{aligned} \textit{type-check}\ (\mathbf{A}, \tau, e_1\ e_2) = (\textit{valid}\ (\mathbf{A}_1 \cup \mathbf{A}_2\ \cup\ \mathsf{T}_1 \boxed{\geq} (\mathsf{T}_2 \to \mathsf{V}_r)), \mathsf{V}_r), \\ \text{where } (\mathbf{A}_1, \mathsf{T}_1) = \textit{type-check}\ (\mathbf{A}, \tau, e_1) \\ \text{and } (\mathbf{A}_2, \mathsf{T}_2) = \textit{type-check}\ (\mathbf{A}, \tau, e_2) \\ \text{and } \mathsf{V}_r \text{ is not free in } \mathbf{A}_1,\ \mathbf{A}_2,\ \mathsf{T}_1,\ \mathsf{T}_2 \text{ or } \tau \end{aligned}$$

Function

Type-check the body of the function given that the parameter has the stated type; the result type is a function from the parameter type to the type of the body:

$$\begin{aligned} \textit{type-check}\ (\mathbf{A}, \tau, \mathsf{T}_v v \to e) = (\mathbf{A}', \mathsf{T}_v \to \mathsf{T}_e), \\ \text{where } (\mathbf{A}', \mathsf{T}_e) = \textit{type-check}\ (\mathbf{A}, \tau \cup \{\mathsf{T}_v : v\}, e) \end{aligned}$$

Casts

The type of a cast expression is the type given in the cast, but we must check to see that the type of the expression $\geq$ that type:

$$\begin{aligned} \textit{type-check}\ (\mathbf{A}, \tau, \mathsf{T} : e) = (A_2, \mathsf{T}), \text{ where } \mathbf{A}_2 = \textit{valid}\ (\mathbf{A}_1 \cup (\mathsf{T}_1 \boxed{\geq} \mathsf{T})) \\ \text{where } (\mathbf{A}_1, \mathsf{T}_1) = \textit{type-check}\ (\mathbf{A}, \tau, e) \end{aligned}$$

Quantified Expressions

The body of a quantified expression is checked given that the type variable is fixed:

$$\begin{aligned} \textit{type-check}\ (\mathbf{A}, \tau, \forall \mathsf{V}.e) = (\mathbf{A}' - \{\text{Fixed } \mathsf{V}\}, \forall \mathsf{V}.\mathsf{T}_e), \\ \text{where } (\mathbf{A}', \mathsf{T}_e) = \textit{type-check}\ (\mathbf{A} \cup \{\text{Fixed } \mathsf{V}\}, \tau, e) \end{aligned}$$

1.1. $\boxed{\geq}$

$\boxed{\geq}$ is an infix operation between two types. $\mathsf{T}_1 \boxed{\geq} \mathsf{T}_2$ is a set of assumptions $\mathbf{A}$, such that $\mathbf{A} \vdash \mathsf{T}_1 \geq \mathsf{T}_2$. The assumptions may not be consistent; see *valid* below.

The following rewriting rules define $\boxed{\geq}$ case by case ($a \Rightarrow b, c$ means reduce a to b, otherwise try c):

$$\begin{aligned} \mathsf{V} \boxed{\geq} \forall \mathsf{V}_1.\mathsf{T} &\Rightarrow \{\mathsf{V} \geq \mathsf{T},\ \text{Fixed } \mathsf{V}_1\}, & C0 \\ \mathsf{V} \boxed{\geq} \mathsf{T} &\Rightarrow \{\mathsf{V} \geq \mathsf{T}\}, & C1 \\ \forall \mathsf{V}.\mathsf{T}_1 \boxed{\geq} \mathsf{T}_2 &\Rightarrow \mathsf{T}_1 \boxed{\geq} \mathsf{T}_2, & C2 \end{aligned}$$

This corresponds to $R3$

$$\begin{aligned} \mathsf{T}_1 \to \mathsf{T}_2 \boxed{\geq} \mathsf{V} &\Rightarrow \{\mathsf{T}_1 \to \mathsf{T}_2 \geq \mathsf{V}\}, & C3 \\ \mathsf{T}_1 \to \mathsf{T}_2 \boxed{\geq} \forall \mathsf{V}.\mathsf{T}_3 &\Rightarrow (\mathsf{T}_1 \to \mathsf{T}_2 \boxed{\geq} \mathsf{T}_3) \cup \{\text{Fixed } \mathsf{V}\}, & C4 \end{aligned}$$

the 'Fixed' represents the fact that V must be free in T_3 as in $R4$

$$\mathsf{T}_1 \to \mathsf{T}_2 \boxed{\geq} \mathsf{T}_3 \to \mathsf{T}_4 \;\Rightarrow\; (\mathsf{T}_3 \boxed{\geq} \mathsf{T}_1) \cup (\mathsf{T}_2 \boxed{\geq} \mathsf{T}_4) \qquad C5$$

2. Checking Assumption Sets

I now give the rules for *valid*; *valid* first forms the closure of the assumption set and then checks it for consistency. **A** is a set of assumptions as for *type-check*, and $\boxed{\veebar}$ is defined below.

$$\mathit{valid} \;= \mathit{check}\ (\mathit{closure}\ \mathbf{A}) \qquad S1$$

where *closure* **A** is the least set **C** such that

$$\mathbf{A} \subseteq \mathbf{C} \qquad S2$$

$$\{\mathsf{T}_1 \geq \mathsf{V}, \mathsf{V} \geq \mathsf{T}_2\} \subseteq \mathbf{C} \Longrightarrow \mathsf{T}_1 \boxed{\geq} \mathsf{T}_2 \subseteq \mathbf{C} \qquad S3$$

$$\{\mathsf{T}_1 \geq \mathsf{V}, \mathsf{T}_2 \geq \mathsf{V}\} \subseteq \mathbf{C} \Longrightarrow \mathsf{T}_1 \boxed{\veebar} \mathsf{T}_2 \subseteq \mathbf{C} \qquad S4$$

Here $\Longrightarrow$ stands for logical implication. For non-recursive types this set is clearly finite, because all types introduced to **C** are subtypes of types in **A**. This permits an algorithmic implementation, but care must be taken not to compute the same comparisons repeatedly. The easiest solution to this is to use the same memory mechanism as for recursive types (see **4** below).

$$\begin{aligned} \mathit{check}\ \mathbf{A} \;= \text{Fail}, &\quad \text{if } \{\text{Fixed } \mathsf{V}, \mathsf{T} \geq \mathsf{V}\} \subseteq \mathbf{A} && S5 \\ \text{Fail}, &\quad \text{if } \{\text{Fixed } \mathsf{V}, \mathsf{V} \geq \mathsf{T}\} \subseteq \mathbf{A} && S6 \\ \mathbf{A} &\quad \text{Otherwise} && S7 \end{aligned}$$

In $S5$ and $S6$ T is understood to be distinct from V.

2.1. $\boxed{\veebar}$

This operation is reminiscent of unification [Robinson 1965]. $\mathsf{T}_1 \boxed{\veebar} \mathsf{T}_2$ is the set of restrictions on variables in T_1 and T_2 needed to show that there is a T_3 such that $\mathsf{T}_1 \geq \mathsf{T}_3$ and $\mathsf{T}_2 \geq \mathsf{T}_3$.

$$\begin{aligned} \mathsf{V} \boxed{\veebar} \mathsf{T} &\;\Rightarrow\; (\mathsf{V} \boxed{\geq} \mathsf{T}) \cup (\mathsf{T} \boxed{\geq} \mathsf{V}), && U1 \\ \forall \mathsf{V}.\mathsf{T}_1 \boxed{\veebar} \mathsf{T}_2 &\;\Rightarrow\; \mathsf{T}_1 \boxed{\veebar} \mathsf{T}_2, && U2 \\ \mathsf{T}_1 \to \mathsf{T}_2 \boxed{\veebar} \mathsf{V} &\;\Rightarrow\; (\mathsf{T}_1 \to \mathsf{T}_2 \boxed{\geq} \mathsf{V}) \cup (\mathsf{V} \boxed{\geq} \mathsf{T}_1 \to \mathsf{T}_2), && U3 \\ \mathsf{T}_1 \to \mathsf{T}_2 \boxed{\veebar} \forall \mathsf{V}.\mathsf{T}_3 &\;\Rightarrow\; \mathsf{T}_1 \to \mathsf{T}_2 \boxed{\veebar} \mathsf{T}_3, && U4 \\ \mathsf{T}_1 \to \mathsf{T}_2 \boxed{\veebar} \mathsf{T}_3 \to \mathsf{T}_4 &\;\Rightarrow\; \mathsf{T}_2 \boxed{\veebar} \mathsf{T}_4 && U5 \end{aligned}$$

Notice that in $U5$ no account is taken of T_1 or T_3 because the type $(\forall \mathsf{V}.\mathsf{V}) \to \mathsf{T}$ is always less general than both $\mathsf{T}_1 \to \mathsf{T}$ and $\mathsf{T}_3 \to \mathsf{T}$.

3. Examples

If f and x have been declared with types $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Bool}$ and $(\forall \mathsf{V}.\mathsf{V} \to \mathsf{V})$ respectively (where Bool and Int are capsules), then *type-check* $(\{\}, \{(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Bool} : f, (\forall \mathsf{V}.\mathsf{V} \to \mathsf{V}) : x\}, f\ x)$ results in checking f and x, which in turn results in

$$(\mathit{valid}\ ((\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Bool} \boxed{\geq} (\forall \mathsf{V}.\mathsf{V} \to \mathsf{V}) \to \mathsf{V}_r), \mathsf{V}_r)$$

taking

$$(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Bool} \boxed{\geq} (\forall \mathsf{V}.\mathsf{V} \to \mathsf{V}) \to \mathsf{V}_r$$

we get

$$((\forall \mathsf{V}.\mathsf{V} \to \mathsf{V}) \boxed{\geq} \mathsf{Int} \to \mathsf{Int}) \cup (\mathsf{Bool} \boxed{\geq} \mathsf{V}_r) \qquad \text{By } C5$$

$$(\forall \mathsf{V}.\mathsf{V} \to \mathsf{V} \boxed{\geq} \mathsf{Int} \to \mathsf{Int}) \cup \{\mathsf{Bool} \geq \mathsf{V}_r\} \qquad \text{By } C1$$

$$((\mathsf{V} \to \mathsf{V}) \boxed{\geq} (\mathsf{Int} \to \mathsf{Int})) \cup \{\mathsf{Bool} \geq \mathsf{V}_r\} \qquad \text{By } C2$$

$$(\mathsf{Int} \boxed{\geq} \mathsf{V}) \cup (\mathsf{V} \boxed{\geq} \mathsf{Int}) \cup \{\mathsf{Bool} \geq \mathsf{V}_r\} \qquad \text{By } C5$$

$$\{\mathsf{Bool} \geq \mathsf{V}_r, \mathsf{V} \geq \mathsf{Int}, \mathsf{Int} \geq \mathsf{V}\} \qquad \text{By } C1 \text{ (twice)}$$

valid checks $\mathsf{Int} \geq \mathsf{Int}$ (which is true), so the answer is

$$(\{\mathsf{Bool} \geq \mathsf{V}_r, \mathsf{V} \geq \mathsf{Int}, \mathsf{Int} \geq \mathsf{V}\}, \mathsf{V}_r)$$

Which means that $f\ x$ has type V_r, provided that $\mathsf{Bool} \geq \mathsf{V}_r$.

Suppose that the argument and parameter types had been the other way round. The initial tests would follow the same course, but

$$(\forall \mathsf{V}.\mathsf{V} \to \mathsf{V}) \to \mathsf{Bool} \boxed{\geq} (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{V}_r$$

would reduce like this:

$$(\mathsf{Int} \to \mathsf{Int} \boxed{\geq} \forall \mathsf{V}.\mathsf{V} \to \mathsf{V}) \cup \mathsf{Bool} \boxed{\geq} \mathsf{V}_r \qquad \text{By } C5$$

$$(\mathsf{Int} \to \mathsf{Int} \boxed{\geq} \forall \mathsf{V}.\mathsf{V} \to \mathsf{V}) \cup \{\mathsf{Bool} \geq \mathsf{V}_r\} \qquad \text{By } C1$$

$$(\mathsf{Int} \to \mathsf{Int} \boxed{\geq} \mathsf{V} \to \mathsf{V}) \cup \{\mathsf{Bool} \geq \mathsf{V}_r, \mathsf{V}\ \mathrm{Fixed}\} \qquad \text{By } C4$$

$$(\mathsf{V} \boxed{\geq} \mathsf{Int}) \cup (\mathsf{Int} \boxed{\geq} \mathsf{V}) \cup \{\mathsf{Bool} \geq \mathsf{V}_r, \mathsf{V}\ \mathrm{Fixed}\} \qquad \text{By } C5$$

$$\{\mathsf{Bool} \geq \mathsf{V}_r, \mathsf{V}\ \mathrm{Fixed}, \mathsf{Int} \geq \mathsf{V}, \mathsf{V} \geq \mathsf{Int}\}$$

but this time *valid* would produce Fail (By $S5$).

4. Recursive generators

In the absence of recursive type generators it is clear that rules $C1$–5 will produce a finite set of assumptions, since each rule involves a reduction in size of the comparands. Recursive generators have the effect that types are no longer finite, and so recursion on their structure may not terminate. The restrictions Ponder imposes on recursive generators ensure that the expansions of recursive calls to type generators are equivalent to the original application;

$$\mathsf{G}[\mathsf{V}_1, \ldots, \mathsf{V}_n] \triangleq \ldots \mathsf{G}[\mathsf{V}_1, \ldots, \mathsf{V}_n] \ldots$$
$$\Uparrow$$

so the application of G marked with $\Uparrow$ will be equivalent to the initial application of G.

Since the texts of recursive generators are finite, it is clear that any infinite series of comparisons that could arise must be a cycle. Hence I believe that it suffices to introduce a memory into the algorithm, and not perform any comparison twice (the assumptions generated must be equivalent to some that have occurred already).

This same mechanism is used in *check* to avoid checking the same thing twice.

5. Optimisations

The description of the algorithm above is simplified. In fact, it is desirable that a type error should be detected as soon as possible (the overloading mechanism of Ponder relies on type-checks that fail).

A number of optimisations to the algorithm are therefore desirable. Instead of computing the sets of assumptions and then checking for consistency at the end it is useful to pass the set as computed so far to succeeding comparisons, and use a special function to insert new elements in the set. This means that inconsistencies of the form $\{\mathsf{V} \leq \mathsf{T}, \mathsf{V}\ \text{Fixed}\}$ are discovered immediately, and the comparison may stop. To speed the process of insertion, it is useful to sort the assumptions by variable, and to duplicate cases where two variables are compared, so that the relationship is keyed by both variables.

A further optimisation arises from the observation that it is not necessary to perform an expansion when comparing type generators that are the same. A more efficient approach is to compare only the arguments of the two generators. However, it is necessary to pre-calculate the directions that comparisons will take. For example if $\mathsf{F}\ [\mathsf{A},\ \mathsf{B}] \triangleq \mathsf{A} \rightarrow \mathsf{B}$, then for $\mathsf{F}\ [\mathsf{A}_1,\ \mathsf{B}_1] \geq \mathsf{F}\ [\mathsf{A}_2,\ \mathsf{B}_2]$ we need $\mathsf{B}_1 \geq \mathsf{B}_2$, but $\mathsf{A}_2 \geq \mathsf{A}_1$. In a comparison $\mathsf{G}[\mathsf{T}_{1L}, \ldots \mathsf{T}_{nL}]\ \boxed{\geq}\ \mathsf{G}[\mathsf{T}_{1R}, \ldots \mathsf{T}_{nR}]$, each T_{iL} will be compared with the corresponding parameter T_{iR}, either by $\mathsf{T}_{iL}\ \boxed{\geq}\ \mathsf{T}_{iR}$, or by $\mathsf{T}_{iR}\ \boxed{\geq}\ \mathsf{T}_{iL}$ or both. It is a simple matter to pass over the definition of a generator and classify each parameter according to which way it will be compared.

IV

Conclusions

Analysis

The work on Ponder has led to number of solutions to old problems, but inevitably has uncovered new ones. Here I analyse some of the flaws in Ponder and suggest possible avenues of research that might lead to solutions.

1. The Conceptual Model

The λ-calculus has some inherent peculiarities.

1.1. Interaction

Interactive programmes are still rather difficult to write. If a programme reads lines from the terminal, it must send out the reflections of the characters typed before it "reads" from the line. In a language with normal order semantics this all works correctly, because answers (the reflected characters) are computed as soon as it is possible.

Unfortunately, the same mechanism can cause answers to come out *before* they are wanted. Suppose the programme in question is manipulating graphics at the terminal and movement information on the input is represented by sequences of characters. Such a programme should not reflect the characters it receives because it must translate them into graphics operations. If a command is of the form 'start move: point to source; point to destination', and the resulting operation is to be 'remove at source; re-draw at destination', it is quite likely that the 'remove at source' will be performed as soon as the 'point to source' has been received. What is wanted is that the 'remove ... re-draw' sequence does not start until the destination is received.

1.2. Efficiency

A consequence of the implementation technique is that functions passed as arguments will always be built into trees. One alternative would be to introduce a new instruction to the abstract machine, which applied the function to a number of arguments, but this would often still require tree-building. There is a possibility that the strictness transforming properties of higher order functions can be detected. If $C \triangleq \lambda f.\lambda g.\lambda h.f\ h\ g$, $C\ F$ is strict on its arguments if F is. Wray has been working on this [Wray 1984].

Another source of inefficiency is that the list constructor operation is lazy on both its arguments. In practice, if the constructor is ever called, one of the arguments will be evaluated. Is it possible to detect that a particular list object has the property that if any cell is looked at, its left hand side is also looked at? If such lists can be detected, they could be constructed with an operator that is strict on its first argument.

A common behaviour in programmes is that one function produces a list while another consumes it. In the present implementation the first function will use a cell from the heap and the second will discard it. Wadler [Wadler 1984] describes an implementation

technique that avoids this problem, but which only works on a certain class of programmes. Is there an algorithm that detects only some cases, in a manner analogous to the strictness detection algorithm?

At present only half of the information that Mycroft's strictness algorithm can produce is used. As well as the detection of strict arguments, the algorithm can detect expressions that *necessarily* terminate. This is potentially useful because such expressions can be evaluated in applicative order. The present Ponder compiler does not do this because a function that necessarily terminates does not necessarily take a short time. Consider

$$(\mathsf{Bool}\ \mathit{safe} \rightarrow \mathsf{Int}\ x \rightarrow \textbf{If}\ \mathit{safe}\ \textbf{Then}\ x\ \textbf{Else}\ 0\ \textbf{Fi})\ \mathit{false}\ (\mathit{factorial}\ 100\,000\,000)$$

Although it may be possible to prove that *factorial* 100 000 000 terminates, it is not necessarily a good idea to evaluate it!

Again an imprecise algorithm would help. Perhaps all that is needed is an estimate of the time taken to construct the tree of the expression and of the number of reductions *likely* in its evaluation.

2. Types

2.1. Type Syntaxes

An objection against Ponder types is that there are no record types or discriminated unions. I believe that this is really a problem with syntax. At present, 'records' may be defined by use of pairs and operators. The distraction is that for every new record type one must define all of the field selectors and update functions. Union types have a similar property. I suspect that a sufficiently sophisticated answer to the module problem mentioned below would also solve this.

2.2. Capsules

When a capsule is sealed, it ceases to have any relationship with any other type. If we have the definitions

$$\begin{aligned}
\textbf{Type}\ \mathsf{True} &\triangleq \forall \mathsf{T}_1, \mathsf{T}_2.\mathsf{T}_1 \rightarrow \mathsf{T}_2 \rightarrow \mathsf{T}_1 \\
\textbf{Type}\ \mathsf{Bool} &\triangleq \forall \mathsf{T}.\mathsf{T} \rightarrow \mathsf{T} \rightarrow \mathsf{T} \\
\textbf{Type}\ \mathsf{Positive\text{-}integer} &\triangleq \mathsf{True} \times \mathsf{Natural} \\
\textbf{Type}\ \mathsf{Integer} &\triangleq \mathsf{Bool} \times \mathsf{Natural}
\end{aligned}$$

then all Positive-integers are Integers, i.e. Positive-integer $\geq$ Integer (The Bool is for the sign of the number; True is a type containing only *true*, indicating a positive sign). If any of

these types are sealed, the relationship is lost. It might be useful to allow the programmer to specify that certain relationships are to be retained even after sealing.

2.3. Coercions

One solution to the previous problem is to allow the programmer to specify coercions. A coercion is a function that transforms an object of one type to another, but which can be inserted automatically by the compiler. Mitchell [Mitchell 1982] describes a method for doing this in Milner's type system. I suspect that it would be rather more complicated to do this in the presence of overloading. Even if it can be done, is it possible for the compiler to create a coercion function from $\mathsf{List}\ [\mathsf{T}_1]$ to $\mathsf{List}\ [\mathsf{T}_2]$ given only a function from $\mathsf{T}_1 \to \mathsf{T}_2$?

2.4. Overloading

The mechanism for overloading operators in Ponder is useful. It allows one to use the same symbol for the addition of integers, rationals and reals. Arbitrary function names cannot be overloaded, so the meaning of an expression (with regard to overloading) is always unambiguous. This does mean that to *gather* a function down a list it must be named explicitly: the infix operator cannot be used.

The overloading mechanism in HOPE is more promising at first sight. All function symbols may be overloaded, and resolution of overloading depends on the whole expression, rather than just the arguments of the overloaded function. For instance *gather plus* 0 is a function that adds together the members of an integer list. HOPE does not allow overloading to be inherited. Thus *gather plus* is not an overloaded function; it must be applied to an argument before the ambiguity can be resolved. This means that one cannot declare an overloaded object by using overloaded parts. In turn, this means that abstraction is restricted: **Let** *sum* $\triangleq$ *gather plus*; *sum* 0 is not equivalent to *gather plus* 0.

A partial solution would be to provide some concise notation that restricted the type of the operator.

2.5. Undefinable Objects

The type-system is closely linked with the λ-calculus. Although I have not defined a semantic model of the type system, it is easy to prove certain properties of some types. For example, there are no useful functions of type $\forall\mathsf{T}.\ \mathsf{T} \to \mathsf{T} \to \mathsf{Bool}$. Intuitively this is because in the body of such a function, T is fixed, so there are no non-polymorphic operations that can be applied to either of the arguments. The only way the function can return a Bool is always to return *true*, always return *false* or never return at all.

2.6. A Mathematical Model

Although I have defined the Ponder type-system rigorously, I have not provided a semantic model for it. While such a model would be worthwhile, I have not the necessary

mathematical background to do this; it should not be a difficult task for anyone experienced in this area.

3. Syntax

3.1. Priority Mechanism

I feel that the current method of specifying the relative binding powers of infix operators is inadequate:

- It is impossible to specify the relationship between a new operator an old one without knowing the number associated with it.
- It is impossible to specify a relationship between an infix operator and a built in construct.
- It is impossible to have a pair of operators for which no relationship is specified.

In particular, the first two make it impossible to get the most desirable scope rules for clauses like **If** ... **Fi**, because the semicolon in a **Let** declaration binds less tightly than all operators except priority nine. In

If Let $i \triangleq 1$;
  $i = 1$
Then e_1
Else e_2
Fi

it would make sense for the scope of i to include both e_1 and e_2. At present, the priority of **Then** is such that a declaration like this is only in scope for the boolean part of the clause.

The solution may be to treat semicolon *etc.* as infix operators for the purposes of defining precedence, and to have a precedence declaration that relates operators with other operators rather than with numbers. For example

Precedence [$\times$ / + $-$];
Precedence [; **Then Else**];

Which gives the relative precedences of the mathematical operators, and then the relationship between semicolon, **Then** and **Else**. This solution has several drawbacks. The order of operators within the brackets is ambiguous: which is the most tightly binding? There is no obvious way to include the direction of association of the operators. There is still no means of leaving the relationship between two operators unspecified.

3.2. Definitions

The **Let** and **Letrec** declarations are somewhat ad-hoc. It could be argued that a **Where** declaration would be just as good, or perhaps better. The pre-defined use of commas to unpick declarations with pairs is arbitrary: this pre-definition requires the

definition of pairs to be fixed, and there is no way of introducing similar declarations for new constructors.

I feel that what is lacking is a means by which the programmer can introduce new declaration mechanisms. For instance, the familiar $\sum_{i=0}^{n}$ notation introduces i as a variable and so cannot be defined for Ponder. The trouble with declarations is that they manipulate unbound variables—how can this be done safely, and where does the type of the variable come from?

3.3. Parameterised Modules

At present the only parameter passing mechanism is for functions. It would often be useful to parameterise a whole package. For example, a floating point package can only work to a limited precision, but there is no *a priori* reason for choosing a limit. It would be possible to write the package in such a way that the precision had to be specified at each operation. This would sometimes be useful, but on the whole would be terribly cumbersome.

I include this problem under the heading of syntax because I believe that the problem is mainly syntactic. The module could even now be written as a function that, when given a limit for precision, returned all the floating point functions as a tuple. This is a poor solution. The user of such a package would have to write new definitions for all the types and operators.

What is needed is a parameterised module mechanism, which primarily allows the export of definitions of operators and so-on. Such a facility must be simple, and not give rise to confusion. The hardest thing seems to be to justify the assertion that modules are not first-class objects. Modules clearly cannot be first-class, because if passed as arguments to functions, they would give rise to dynamic binding properties.

The type system in Poly [Matthews 1982] goes some way towards solving this problem. Because a type is identified with its operators, the instantiation of a precision into a floating-point package effectively results in the declaration of a new set of operators. Unfortunately the use of operators is not as simple as one would hope. Either every instance of an operator must be labelled with the type from which it is to be drawn, or the operators must be declared separately from the type (leaving us with the original objection).

While on the subject of Modules it is perhaps worth mentioning separate compilation. The reason I have not provided a separate compilation mechanism as part of the language is that I believe it to be an extra-lingual feature. Separate compilation is only needed to save time when compiling a programme—even if a programme was compiled from three different files it should still be understood as one programme. Indeed, there is no reason why Ponder should be compiled 'in batch'—separate compilation is redundant if the user sees the Ponder system as a repository for functions. The requirement for separation into modules within a programme should not be confused with separation for compilation.

3.4. Icons

The types Int, Char and String are only built in because of the need for icons for them. It is not necessary to build them in because of communication with the outside world—all

that is needed is a mechanism by which the compiler and the run-time-system agree how to construct and take apart lists. Characters could just be names for objects declared in the standard prelude.

It would be nice if the language didn't need all of the different kinds of icon, though. I considered the possibility of using a conversion function on strings to provide integer icons. The difficulty is that an error such as *decimal* "12B" would not be detected until the programme was run. Is it feasible to arrange that characters are collected into groups with different types, all of which are coercible (see above) to Character? Thus "128" would have type Digits whereas "12B" would have type Letters-and-digits.

Despite this mechanism the notation for numbers would still be cumbersome: a slight improvement would be to declare # as a prefix operator for the function *decimal*, but even so # "128" is uglier than 128. This is quite a strong argument for integer icons, but the new mechanism would be useful in other circumstances, for example when dealing with hexadecimal constants.

Summary

I have demonstrated that in many cases a more powerful, smaller system is preferable to a less powerful but larger one. In particular the addition of two constructs to the ML type-discipline removes the need for many others and results in a more expressive system. I believe that I have demonstrated that minimal languages are not just theoretical curiosities; a spreadsheet system has been written by Stuart Wray in about 2500 lines of Ponder.

The application of this approach to syntax has been somewhat less successful. Constructs such as if- and case- clauses may be defined, but the resulting syntax is sometimes less elegant than would be possible in a built-in version. For example the case-clause for unions requires the type of the object to be specified for every case, even though it is implicitly known already.

The use of a simple model for the semantics of the language is successful in that it allows powerful tools to be built, particularly in conjunction with the syntax-defining mechanism. The implementation of the model still leaves much to be desired. A randomly chosen Ponder programme may be more than ten times slower than its imperative counterpart.

Bibliography

[Ashcroft 1982]:
E.A. Ashcroft, W.W. Wadge,
R*x* *for Semantics,*
ACM Transactions on Programming Languages and Systems Number 4, 1982

[Backus 1978]:
J. Backus,
Can Programming be Liberated from the von Neumann Style?,
Communications of the Association for Computing Machinery Volume 21 Number 8

[Barendregt 1980]:
H.P. Barendregt,
The λ-calculus, its Syntax and Semantics,
North Holland

[Berlekamp 1982]:
E.R. Berlekamp, J.H. Conway and R.K. Guy,
Winning Ways Volume 2: Games in Particular,
Academic Press

[Burstall 1977]:
R.M. Burstall, J. Darlington,
A transformation System for Developing Recursive Programs,
Journal of the ACM 24

[Burstall 1980]:
R.M. Burstall, D.B. Macqueen, D.T. Sanella,
Hope: An Experimental Applicative Language,
Edinburgh Department of Computer Science Technical Report CSR 62-80

[Church 1941]:
Alonzo Church,
The Calculi of Lambda-Conversion,
Princeton University Press

[Clarke 1980]:
T.J.W. Clarke, P.J.S. Gladstone, P.D. Maclain, A.C. Norman,
SKIM—The S, K, I Reduction Machine,
Proceedings of the ACM Lisp Conference 1980

[Clocksin 1981]:
W.F. Clocksin and C.S. Mellish,
Programming in Prolog,
Springer Verlag

[Curry 1958]:
H.B. Curry and R. Feys,
Combinatory Logic,
North Holland, Amsterdam

[Darlington 1981]:
J. Darlington, M. Reeve,
Alice—A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages,
Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire 1981

[Darlington 1982]:
J. Darlington, P. Henderson and D. Turner (editors),
Functional Programming and its Applications,
Cambridge University Press

[Darlington 1983]:
J. Darlington,
Program Transformation in the Alice Project,
Department of Computing, Imperial College London, November 1983

[Davis 1965]:
Martin Davis (Editor),
The Undecidable,
Raven Press

[Demers 1980]:
A.J. Demers, J.E. Donahue,
Data Types, Parameters and Type Checking,
Proceedings of the 7^{th} Symposium on the Principles of Programming Languages, ACM January 1980

[Frege 1960]:
Gottlob Frege,
Function and Concept,
In [Geach 1960]

[Geach 1960]:
D. Geach and R.W. Black,
Translations from the Philosophical writings of Gottlob Frege,
Oxford University Press

[Gordon 1979]:
M.J.C. Gordon, R. Milner, C.P. Wadsworth,
Edinburgh LCF,
Springer Lecture notes in Computer Science Number 78, Springer Verlag

[Griswold 1968]:
R.E. Griswold, J.F. Poage and I.P. Polonsky,
The Snobol4 Programming Language,
Prentice Hall

[Henderson 1976]:
P. Henderson, J.H. Morris,
A Lazy Evaluator,
Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages, January 1976

[Horn 1951]:
A. Horn,
On Sentences which are True of Direct Unions of Algebras,
Journal of Symbolic Logic 16, March 1951

[Hughes 1982]:
R.J.M. Hughes,
Graph Reduction with Super-combinators,
Oxford University Programming Research Group Technical Monograph PRG-28

[Johnsson 1984]:
T. Johnsson,
Efficient Compilation of Lazy Evaluation,
Proceedings of the 1984 Symposium on Compiler Construction, Montreal

[Landin 1966]:
P.J. Landin,
The Next 700 Programming Languages,
Communications of the ACM, Volume 9 Number 3, March 1966

[MacQueen 1982]:
D.B. MacQueen and Ravi Sethi,
A Semantic Model of Types for Applicative Languages,
Symposium on Lisp and Functional Programming, ACM

[MacQueen 1984]:
D.B. MacQueen, Ravi Sethi and G. Plotkin,
An Ideal Model for Recursive Polymorphic Types,
Eleventh Annual ACM Symposium on the Principles of Programming Languages

[Magó 1980]:
G.A. Magó,
A Network of Processors to Execute Reduction Languages,
International Journal of Computer and Information Science, Volume 8 Numbers 5 & 6

[Markov 1962]:
A.A. Markov,
Theory of Algorithms,
Israel Program for Scientific Translations, Jerusalem

[Martin-Löf 1975]:
Per Martin-Löf,
Intuitionistic Type-Theory,
Logic Colloquium 1973, Rose and Shepherdson (Editors), North Holland

[Matthews 1982]:
D.C.J. Matthews,
Introduction to Poly,
Cambridge University Computer Laboratory Technical Report Number 29

[Milner 1978]:
R. Milner,
A Theory of Polymorphism in Programming,
Journal of Computer and Systems Science Number 17

[Minsky 1967]:
M. Minsky,
Computation: Finite and Infinite Machines,
Prentice Hall

[Mitchell 1982]:
J.C. Mitchell,
Coercion and Type Inference,
ACM Symposium on Lisp and Functional Programming

[Mycroft 1981]:
A. Mycroft,
The Theory and Practice of Transforming Call by Need Into Call by Value,
4th International Symposium on Programming, Springer Lecture Notes in Computer Science 83, Springer Verlag

[OANAR 1949]:
United States Office of Naval Research
The EDSAC Computing Machine, Cambridge University,
Technical Report OANAR-43-49

[Park 1982]:
D. Park,
The Fairness Problem and Determinism in Computing Networks,
Proceedings of the 4th Advanced Conference on the Theory of Computer Science, Mathematisch Centrum, Amsterdam

[Robinson 1965]:
J.A. Robinson,
A Machine Orientated Logic Based on the Resolution Principle,
Journal of the ACM 12 pp 23–31

[Rosser 1982]:
J. Berkely Rosser,
Highlights of the History of the λ-calculus,
In Conference Record of the ACM Symposium on Lisp and Functional Programming

[Scott 1975]:
D. Scott,
Some Philosophical Issues Concerning the Theory of Combinators,
Proceedings of the 1975 Rome Symposium on λ-calculus and Computer Science Theory, Springer Lecture Notes in Computer Science Volume 73, B. Lercher (Editor)

[Sellar 1930]:
W.C. Sellar and R.J. Yeatman,
1066 And all that,
Methuen

[Stoye 1984]:
W.R. Stoye,
A New Scheme for Writing Functional Operating Systems,
Cambridge University Computer Laboratory Technical Report Number 65

[Tillotson 1984]:
M. Tillotson,
Implementing Ponder on the 3081,
University of Cambridge Computer Science Tripos Dissertation

[Turing 1937]:
A.M. Turing,
On Computable Numbers with an Application to the Entscheidungsproblem,
Proceedings of the London Mathematical Society. (Reprinted in [Davis 65])

[Turner 1979]:
D.A. Turner,
A New Implementation Technique for Applicative Languages,
Software Practice and Experience Number 9, January 1979

[Turner 1979′]:
D.A. Turner,
Another Algorithm for Bracket Abstraction,
Journal of Symbolic Logic Volume 44 Number 2, Association for Symbolic Logic

[Turner 1982]:
D.A. Turner,
Recursion Equations as a Programming Language,
In [Darlington 1982]

[van Wijngaarden 1981]:
A. van Wijngaarden,
Languageless Programming,
Mathematisch Centrum Department of Computer Science 1W 181/81

[Wadler 1984]:
P.L. Wadler,
Listlessness is Better than Laziness,
Proceedings of the 3rd Symposium on Lisp and Functional Programming

[Wadsworth 1971]:
C.P. Wadsworth,
A Graph Evaluation Technique of the λ-calculus,
Chapter 4 of his PhD. Thesis *Semantics and Pragmatics of the Lambda-Calculus*, University of Oxford

[Weinberg 1971]:
G.M. Weinberg,
The Psychology of Computer Programming,
van Nostrand Reinhold

[Wray 1984]:
S.C. Wray,
Strictness Detection in the Ponder Compiler,
Cambridge University Computer Laboratory Technical Report *in preparation*

[Young 1981]:
R. Young,
Mental Models,
International Journal of Man-Machine Studies 15, July 1981, D.R. Hill and B.R. Gaines (Editors), Academic Press

Appendix 1: The Ponder Abstract Machine

1. Overall description

The machine consists of a stack and a heap, and some registers. The stack is used mainly to store arguments to functions and information about how to return to the previous stack frame. The heap is used for all other information: the cells of lists, applications of functions etc. The heap consists entirely of two word cells, as in Lisp.

The present interface with the outside world is that input is read from files named within the programme, and the only means of output is to return a list of 'file actions'. For the purposes of implementing the abstract machine, this is adequate, allowing sufficiently many test programmes to be run. Because of this, the programme loaded is driven from a main evaluation loop that calls Eval on the programme to get a list cell, the head of which must then be evaluated (using Eval) to get a file action.

The description assumes a syntax for machine instructions, but in practice this may depend on the particular assembler used, and may in fact never be used, as it is intended that the code-generator should be transported by rewriting the segments of the code-generator that output code, rather than by writing a separate programme to convert from abstract to concrete code.

2. Registers

The abstract instructions are stack based, and hence mention no registers explicitly, but the configuration of store implies the existence of certain address registers. To simplify description we assume that the stack is laid out in a particular way, but an implementation may do something different if the effect is the same;

Programme-Counter points to the next instruction to be evaluated.

Stack-top points to the top element of the stack.

Stack-base points to the base of stack so that the garbage collector can find all pointers to active store.

Arg-base points to the first argument of the current supercombinator, and (hence) to the return link and its own previous value.

Free-list points to the list of cells that are known not to be in use, and from which new cells will be allocated.

First-heap-block points to the base of the first block of store, so that the garbage collector can know where everything is.

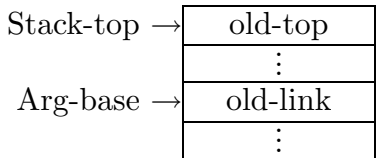
Of these, it would seem sensible to keep only some in real registers, since the others will be accessed very rarely. Stack-base and First-heap-block will only be accessed at a garbage collection. Programme-Counter will normally be in the corresponding real register.

3. Calling sequence

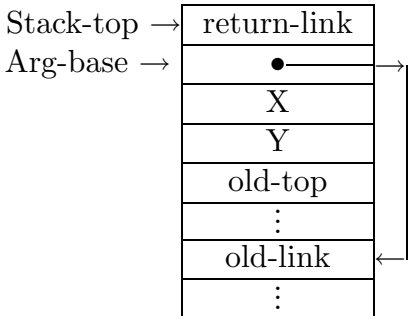
The arguments to the combinator to be called are pushed onto the stack; eg: the code generated for (Three-argument-combinator X Y Z) would be Push Z; Push Y; then Push X; Call Three-argument-combinator.

The Call instruction would enter the compiled code for the combinator, pushing the return address and the previous argument base onto the stack.

Hence the sequence Push Y; Push X; Call S would take the stack from



to



4. Machine Representable Values

May be any one of the following:

Object	Use
Integer	integer values
Character	character values
Return address	address of code to return to
Combinator	point to the code of combinators
Application	point to an application on the heap
Pair	point to a pair on the heap
Selector	encode selector function
Bottom	represents "Undefined"

The objects on the stack, and the elements of each side of an application will be of one of these types.

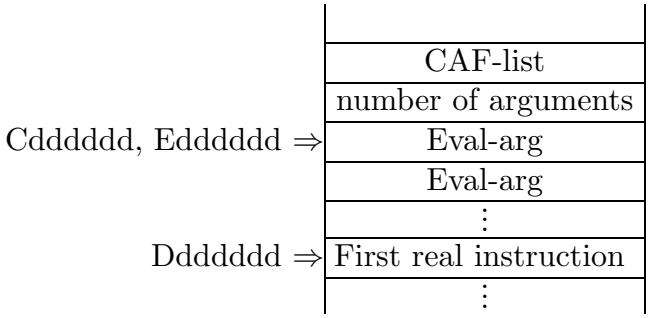
Objects must be distinguishable by the machine, so that Eval can work—one method of achieving this is to put tag bits in the top three bits of each word. (Another two bits will be used by the garbage collector).

Integers and characters normally both have tag bits of zero, so that arithmetic operations are easily done.

Return addresses and other code and stack pointers normally have tag bits of zero, since the garbage collector does not need to follow them.

Combinators are represented as pointers into the code, and have three external labels, for various uses. The first is Cdddddd (where d is a decimal digit), and is a pointer to the entry point of the combinator, and has combinator tag bits, so that it may be passed around as an object recognisable to the garbage collector. The second is Eddddd and is known as the 'eval args label'. This is the entry point of the combinator for use when the arguments have not yet been evaluated, and is the same as the Cdddddd label except that it has no tag bits. The third is Ddddddd, and is for use when the arguments have been evaluated, and is know as the 'do not eval args label'. Neither Eddddd nor Ddddddd have tag bits, because they are used as machine addresses for calls and jumps, and we don't want to get tag bits on Programme-Counter.

The code pointer also gives access to two other fields, so that a combinator looks like this:



The number of arguments field is used by Eval to see if there are sufficient arguments for the combinator. CAF-list is a list of expressions that are pointed to directly by the combinator, and that the garbage collector must mark. CAF stands for Constant Applicative Form. CAFs are put into a heap block to be loaded with the programme (see below).

Applications and pairs are represented as pointers into the heap with appropriate tag bits.

Selectors are compact representations of functions that return exactly one of their arguments. They are represented as two numbers packed into one machine word (with appropriate tag bits). In this appendix they are indicated as '[U nargs which]' where 'nargs' is the number of arguments that the selector takes, and 'which' is the number of the one that the selector will return.

5. Machine instructions

Within the code-generator, each of these instructions is represented by a different type of structure. An implementation could simply output them as macro calls in the assembler language of the target machine. This would be a sensible course while debugging Eval and the garbage collector, but of course precludes useful register and store optimisations.

Jump

The next instruction executed is the one specified by the parameter to this instruction.

Jump-label <label>

Call

The next instruction executed is the one specified by the parameter to this instruction. The return address and arg-base are pushed onto the stack, as described in 'calling sequence' above.

Call-label <label>

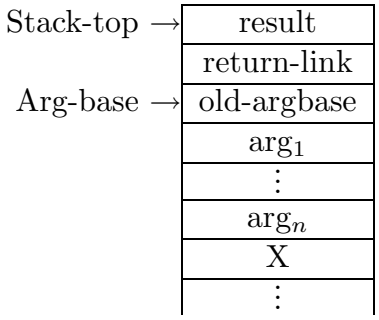
Switch-into

Pop a value from the stack. It will be a selector of the form [U nargs which]. The next instruction executed is the which$^{\text{th}}$ one in the label list that is a parameter to this instruction.

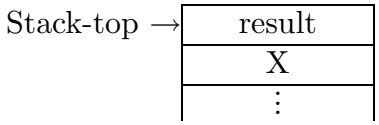
Switch-into<nargs> <label>[,<label>]*

Return

When it is time to return from a combinator, the stack will look like this:



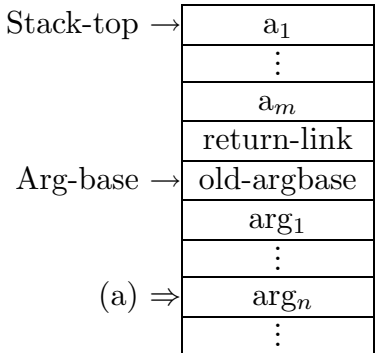
And after returning it will look like this:



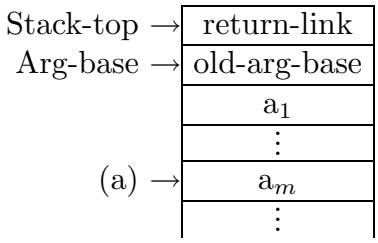
with Arg-base = old-argbase (and we will be executing the code indicated by the return-link.)

Tail-to

Takes execution from this combinator to another, without creating a new stack frame, so the stack goes from:



to



Tail-to <name>,m

Push

The parameter to this instruction is pushed onto the stack.

Constant data

Push-constant <constant>
Push-label <label>
Push-application <application>
Push-sel <nargs>,<which>
Push-bottom
Push-pair <pair>

Each of the constant data instructions set the tag bits of their argument appropriately.

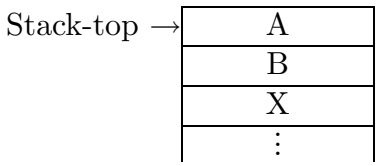
Argument

Push-arg <argument number>

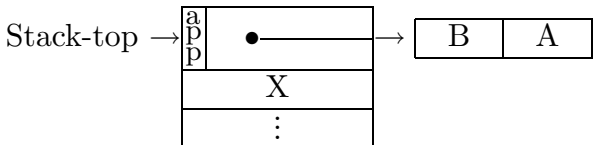
Note that argument n is at offset n from Arg-base. (or $-n$ depending on the direction that the stack grows)

Fap

Form an application on the stack, by popping the top two items on the stack, building an application on the heap from them, and pushing a pointer to this application back onto the stack.



After Fap becomes



Eval

Ensures that the parameter to this instruction has been evaluated—causes a new stack frame to be created, with only the parameter on it. When evaluation of this need proceed no further, the value left on the stack is placed back in the stack location whence it came,

and the recently created stack frame destroyed. Thus Eval must save such information as the old stack base and size, since combinators that it calls may in turn call Eval.

Eval-arg <argument number>

Eval-top

Eval works by testing first to see whether the object is an application, and if it is not, it simply returns it. Otherwise it traverses the left spine of the application, pushing pointers to the applications on the stack, until it reaches either a combinator or a selector at the head. Ponder's type checking ensures that this is the case, i.e. that no applications will be built in which the left hand side is not a function.

If it is a combinator, the appropriate number of arguments is copied from the right hand sides of the cells pushed in the first phase up to the top of the stack. The combinator is then called (by a simulated Call instruction), and returns its result at the top of the stack. This result is then overwritten into the application cell that was the sub-expression just evaluated (and therefore may be identified as the cell that contained the last argument to the combinator). Because the whole cell must be overwritten with something that is potentially not an application, the result is written into the right hand side, and an I combinator (represented as the selector [U 1 1]) in the left.

In the case of a selector the action is slightly more complicated, since it is possible to replace many of the applications involved with simpler ones:

$$[\text{U } 1\ 1]\ \text{x} \Rightarrow \text{x}$$
$$[\text{U nargs } 1]\ \text{x y} \Rightarrow [\text{U (nargs-1) } 1]\ \text{x}$$
$$[\text{U nargs which}]\ \text{x y} \Rightarrow [\text{U (nargs-1) (which-1)}]\ \text{y}$$

however, in the case of selectors there is no need to copy the arguments up the stack, as the result may as well be selected directly from the application that contained it.

Tail-Eval

In order to save stack, the sequence

Eval-top

Return

is always converted to

Tail-eval

This means that if Eval is implemented as a subroutine call, the stack frame can be cleared up first. This is important in that it can correspond to tail recursion in some combinators.

6. Pseudo-operations

The operations in this section are only used when initialising the programme.

Fill-cell

Takes the top application on the stack and overwrites them into the named location in the data block.

Fill-cell <label>

7. Directives

Start-combinator

Indicates the name of the next combinator and how many arguments it takes. This constructs code containing the CAF list of the combinator, and its number of arguments. It should also remember the number of arguments so that instructions like Return can work.

Start-combinator <name>,<number of arguments>

Start-data-block

Indicates the start of the heap block that contains initial data (such as strings and CAFs).

End-list

Fills in the last element of a list in the data block with nil (see 'List-cell' below).

List-cell

Reserves a cell in the data block, and initialises the left hand half to a constant value:
List-cell-pair 'a: make a pair with 'a in the LHS
List-cell-application f: make an application with f in the LHS

If one List-cell follows another, the right hand half of the former is filled with an application or pair pointer to the latter, so that

List-cell-pair 'h
List-cell-pair 'e
List-cell-pair 'l
List-cell-pair 'l
List-cell-pair 'o
End-list

will build the string "hello" in the data block.

Reserve-cell

Reserves a cell in the data block to be filled in by 'Fill-cell'.

End-data-block

Indicates the end of the data block.

Appendix 2: The Standard Prelude

This appendix contains an annotated copy of the 'standard prelude' of common functions, followed by a 'library prelude' of list operations. The preludes have been developed concurrently with the language, so the functions are written in various styles.

1. The Standard Prelude

Certain functions are built into the run-time-system. The definitions of these functions are included here for their type information.

The types Bool and File-action must be defined now because they are involved in the types of the functions. All the operations on File-action belong in the 'outside world', so it is sealed without any definitions.

Capsule Type Bool $\triangleq$ ∀T. T → T → T;
Capsule Type File-action $\triangleq$ Pair [Int, List [Char]];
Seal File-action;

Two functions that convert between characters and their integer equivalents:

Let *int-to-char* $\triangleq$ Int → Char: ...;
Let *char-to-int* $\triangleq$ Char → Int: ...;

Operations on lists and file actions:

Let *null* $\triangleq$ ∀T. Option [T] → Bool: ...;
Let *nil* $\triangleq$ ∀T. Option [T]: ...;
Let *opt-in* $\triangleq$ ∀T. T → Option [T]: ...;
Let *opt-out* $\triangleq$ ∀T. Option [T] → T: ...;
Let *get-file* $\triangleq$ List [Char] → Option [List [Char]]: ...;
Let *append-to-file* $\triangleq$ List [Char] → Option [(List [Char] → File-action)]: ...;
Let *make-file* $\triangleq$ List [Char] → Option [(List [Char] → File-action)]: ...;
Let *delete-file* $\triangleq$ List [Char] → Option [File-action]: ...;
Let *head* $\triangleq$ ∀T. List [T] → T: ...;
Let *tail* $\triangleq$ ∀T. List [T] → List [T]: ...;
Let *list* $\triangleq$ ∀T. T → List [T] → List [T]: ...;

Character comparison functions:

Let *char-eq-char* $\triangleq$ Char → Char → Bool: ...;
Let *char-lt-char* $\triangleq$ Char → Char → Bool: ...;
Let *char-le-char* $\triangleq$ Char → Char → Bool: ...;
Let *char-gt-char* $\triangleq$ Char → Char → Bool: ...;
Let *char-ge-char* $\triangleq$ Char → Char → Bool: ...;

Integer comparisons

Let *int-eq-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Bool: ...;
Let *int-lt-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Bool: ...;
Let *int-gt-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Bool: ...;
Let *int-le-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Bool: ...;
Let *int-ge-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Bool: ...;
Let *int-ne-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Bool: ...;

Other integer operations:

Let *int-plus-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *int-minus-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *int-times-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *int-over-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *int-rem-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *int-and-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *int-or-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *minus-int* $\triangleq$ Int $\rightarrow$ Int: ...;
Let *int-shift-left-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *int-shift-right-int* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Int: ...;
Let *not-int* $\triangleq$ Int $\rightarrow$ Int: ...;
Let *int-times-int-giving-pair* $\triangleq$ Int $\rightarrow$ Int $\rightarrow$ Pair [Int, Int]: ...;

Now the real definitions. First we have some simple functions:

Letrec *abort* $\triangleq \forall$T. T: *abort*;
Let $i \triangleq \forall$T. T $t \rightarrow t$;

Now define the relative binding powers of the common operators.

Priority 5 $\times$ **Associates Right**;
Priority 5 **Long-times Associates Right**;
Priority 5 $\div$ **Associates Left**;
Priority 5 **Rem Associates Left**;
Priority 6 $+$ **Associates Left**;
Priority 6 $-$ **Associates Left**;
Priority 7 $\wedge$ **Associates Left**;
Priority 7 $\vee$ **Associates Left**;
Priority 7 **Shift-left Associates Right**;
Priority 7 **Shift-right Associates Right**;
Priority 8 $=$ **Associates Left**;
Priority 8 $<$ **Associates Left**;
Priority 8 $>$ **Associates Left**;
Priority 8 $\leq$ **Associates Left**;
Priority 8 $\geq$ **Associates Left**;
Priority 8 $\neq$ **Associates Left**;
Priority 8 **Elem Associates Left**;
Priority 8 :: **Associates Right**;
Priority 8 @ **Associates Right**;
Priority 8 **And Associates Right**;
Priority 8 **Or Associates Right**;

The Cartesian product operator:

Typeinfix $\times \triangleq$ Pair;

Functional composition:

Priority 5 $\circ$ **Associates Left**;
Let *compose* $\triangleq \forall$A, B, C. (B $\rightarrow$ A) *ba* $\rightarrow$ (C $\rightarrow$ B) *cb* $\rightarrow$ C *c* $\rightarrow$
ba (*cb c*);
Infix $\circ \triangleq$ *compose*;

The C combinator is useful for the operator definitions, because $a < b$ means *int-lt-int a b*, but the prefix version $< a\ b$, means *int-lt-int b a*, which is *c int-lt-int a b*.

Let $C \triangleq \forall$Tx, Ty, Trf. (Ty $\rightarrow$ Tx $\rightarrow$ Trf) $f \rightarrow$ Tx $x \rightarrow$ Ty $y \rightarrow$ Trf: $f\ y\ x$;

The following are defined in terms of built in functions: predicates are given prefix versions for use with Case clauses (see below)

Infix $=$ $\triangleq$ Char $\to$ Char $\to$ Bool: *char-eq-char*;
Prefix $=$ $\triangleq$ Char $\to$ Char $\to$ Bool: *char-eq-char*;
Infix $<$ $\triangleq$ Char $\to$ Char $\to$ Bool: *char-lt-char*;
Prefix $<$ $\triangleq$ Char $\to$ Char $\to$ Bool: C *char-lt-char*;
Infix $\leq$ $\triangleq$ Char $\to$ Char $\to$ Bool: *char-le-char*;
Prefix $\leq$ $\triangleq$ Char $\to$ Char $\to$ Bool: C *char-le-char*;
Infix $>$ $\triangleq$ Char $\to$ Char $\to$ Bool: *char-gt-char*;
Prefix $>$ $\triangleq$ Char $\to$ Char $\to$ Bool: C *char-gt-char*;
Infix $\geq$ $\triangleq$ Char $\to$ Char $\to$ Bool: *char-ge-char*;
Prefix $\geq$ $\triangleq$ Char $\to$ Char $\to$ Bool: C *char-ge-char*;

Infix $=$ $\triangleq$ Int $\to$ Int $\to$ Bool: *int-eq-int*;
Prefix $=$ $\triangleq$ Int $\to$ Int $\to$ Bool: *int-eq-int*;
Infix $<$ $\triangleq$ Int $\to$ Int $\to$ Bool: *int-lt-int*;
Prefix $<$ $\triangleq$ Int $\to$ Int $\to$ Bool: C *int-lt-int*;
Infix $>$ $\triangleq$ Int $\to$ Int $\to$ Bool: *int-gt-int*;
Prefix $>$ $\triangleq$ Int $\to$ Int $\to$ Bool: C *int-gt-int*;
Infix $\leq$ $\triangleq$ Int $\to$ Int $\to$ Bool: *int-le-int*;
Prefix $\leq$ $\triangleq$ Int $\to$ Int $\to$ Bool: C *int-le-int*;
Infix $\geq$ $\triangleq$ Int $\to$ Int $\to$ Bool: *int-ge-int*;
Prefix $\geq$ $\triangleq$ Int $\to$ Int $\to$ Bool: C *int-ge-int*;
Infix $=$ $\triangleq$ Int $\to$ Int $\to$ Bool: *int-ne-int*;
Prefix $=$ $\triangleq$ Int $\to$ Int $\to$ Bool: C *int-ne-int*;

There are no prefix versions for the other integer operators except minus:

Infix $+$ $\triangleq$ Int $\to$ Int $\to$ Int: *int-plus-int*;
Infix $-$ $\triangleq$ Int $\to$ Int $\to$ Int: *int-minus-int*;
Infix $\times$ $\triangleq$ Int $\to$ Int $\to$ Int: *int-times-int*;
Infix Long-times $\triangleq$ Int $\to$ Int $\to$ (Int $\times$ Int): *int-times-int-giving-pair*;
Infix $\div$ $\triangleq$ Int $\to$ Int $\to$ Int: *int-over-int*;
Infix Rem $\triangleq$ Int $\to$ Int $\to$ Int: *int-rem-int*;
Infix $\wedge$ $\triangleq$ Int $\to$ Int $\to$ Int: *int-and-int*;
Infix $\vee$ $\triangleq$ Int $\to$ Int $\to$ Int: *int-or-int*;
Infix Shift-left $\triangleq$ Int $\to$ Int $\to$ Int: *int-shift-left-int*;
Infix Shift-right$\triangleq$ Int $\to$ Int $\to$ Int: *int-shift-right-int*;
Prefix $-$ $\triangleq$ Int $\to$ Int: *minus-int*;
Prefix Not $\triangleq$ Int $\to$ Int: *not-int*;

Boolean values; *true a b* becomes *a* and *false a b* becomes *b*:

Let *true* $\triangleq$ Bool: $\forall$T. T $t \to$ T $f \to t$;
Let *false* $\triangleq$ Bool: $\forall$T. T $t \to$ T $f \to f$;

Similar functions for pairs:

Let *left* $\triangleq$ $\forall$L, R. L l, R $r \to l$;
Let *right* $\triangleq$ $\forall$L, R. L l, R $r \to r$;

Now the syntax definitions for If clauses. We want to be able to write

If condition
Then expression to return if condition is *true*
Else expression to return if condition is *false*
Fi;

and also extend this to allow a simpler writing of

If condition1
Then expression to return if condition1 is *true*
Else If condition2
    Then thing2
    Else thing3
    Fi
Fi;

as

If condition1
Then expression to return if condition1 is *true*
Elif condition2
Then thing2
Else thing3
Fi

Unfortunately the type checker in the present compiler is too pessimistic about result types, so that we have to make this use a pair to 'balance' the alternatives of the condition, otherwise an expression like

If condition
Then *abort*
Else thing
Fi

would not work.

The capsule If-clause is used solely to make sure that **If** ... **Fi** is only put around the correct kind of expression:

Capsule Type If-clause [T] ≙ T;
Bracket If Fi ≙ ∀T. If-clause [T] *it* → T: *it*;
Priority 9 **Then Associates Right**;
Priority 9 **Else Associates Right**;
Capsule Type Else-part [T_1, T_2] ≙ Pair [T_1, T_2];
– **Else** just forms a pair from its two arguments:
Infix Else ≙ ∀T_1. T_1 *then-part* → ∀T_2. T_2 *else-part* → Else-part [T_1, T_2]:
then-part, *else-part*;
– **Then** 'balances' the types of the two alternatives, and applies the Boolean:
Infix Then ≙ Bool *b* → ∀T. Else-part [T, T] → If-clause [T]:
∀T. T *then*, T *else* → *b then else*;
– **Elif** reiterates the choice:
Priority 9 **Elif Associates Right**;
Infix Elif ≙ ∀T_1. T_1 *previous-then* → ∀T_2. If-clause [T_2] *if-clause* →
Else-part [T_1, T_2]:
(*previous-then* **Else If** *if-clause* **Fi**);

It is also convenient to have a special kind of choice clause for options:

If *option-value* **Is** *function*
Else ...
Fi;

That looks a bit odd, but it makes more sense when *function* is written out with its arguments like this:

If *option* **Is** Sometype *something*
→ thing to do if it is
Else thing to do if it isn't
Fi

Let is-fn ≙ ∀T. Option [T] *obj* → ∀R. Else-part [(T → R), R] *ep* →
If-clause [R]:
If *null obj*
Then *right ep*
Else *left ep* (*opt-out obj*)
Fi;

Priority 9 **Is Associates Right**;
Infix Is ≙ *is-fn*;
Seal If-clause;
Seal Else-part;
Seal Bool;

some familiar operations on Booleans:

```
Let not ≙ Bool a → If a
                   Then false
                   Else true
                   Fi;
Prefix Not ≙ not;
Infix And ≙ Bool a → Bool b → If a
                              Then b
                              Else false
                              Fi;
Infix Or ≙ Bool a → Bool b → If a
                             Then true
                             Else b
                             Fi;
```

Syntax definitions for case clauses: Unions and 'switchons'
'Switchons': Aim for a syntax:

Case value
In predicate ⇒ Thing to return if (predicate value).
▌ Other predicate ⇒ other thing
Out thing to return if value satisfied none of the predicates
Esac;

The definitions of the **In** and **Out** operators for switchons appear after their definitions as operators. *Switchon-choice* is needed for ⇒.

```
Let switchon-choice ≙∀T. (T → Bool) predicate →
                         ∀R. R thing₁, (T → R) thing₂ →
                         T x →
                         If predicate x
                         Then thing₁
                         Else thing₂ x
                         Fi;
```

Binary unions. Unfortunately the cases must be referenced by number.

```
Capsule Type Union [A, B] ≙ ∀R. (A → R) → (B → R) → R;
Priority 5 ⊎ Associates Right;
Typeinfix ⊎ ≙ Union;
Type Dot ≙ ∀T. T;
```

Dot marks the end of a sequence of united values. Without it, we would need different injection functions etc for every number of united types. Always use A ⊎ B ⊎ ... ⊎ Dot.

Injection Functions

```
Let in₁ ≙∀A. A a → (A ⊎ Dot):
        ∀R. (A → R) ar → (Dot → R) br → ar a;
```

with the aid of an injection function for the right half,

Let $in_r \triangleq \forall$A, B. B $b \rightarrow$ (A $\uplus$ B):
$\forall$R. (A $\rightarrow$ R) $ar \rightarrow$ (B $\rightarrow$ R) $br \rightarrow br\ b$;

we can build a function to make new injection functions from old:

Let *next-in* $\triangleq \forall$A, B, T. (T $\rightarrow$ (A $\uplus$ B)) *in-n* $\rightarrow$ $in_r \circ$ *in-n*;

The *is* functions project from unions. if A $\uplus$ B $\uplus$ Dot: x, $is_1\ x$ will return an Option [A].

Type Is-tag [U, Which] $\triangleq$ U $\rightarrow$ Option [Which];
Let $is_1 \triangleq \forall$A, B. Is-tag [(A $\uplus$ B), A]:
$\forall$A, B. (A $\uplus$ B) *uab* $\rightarrow$
uab opt-in ($\forall$T. T $t \rightarrow$ *nil*);

Internally we need to be able to test A $\uplus$ B for B:

Let $is_r \triangleq \forall$A, B.
(A $\uplus$ B) *uab* $\rightarrow$
uab ($\forall$T. T $t \rightarrow$ Option [B]: *nil*) *opt-in*;

given an is_n produce is_{n+1}

Let *next-is* $\triangleq \forall$New, U, Which. Is-tag [U, Which] *previous-is* $\rightarrow$
Is-tag [(New $\uplus$ U), Which]:
(New $\uplus$ U) *unewu* $\rightarrow$
unewu ($\forall$T. T $t \rightarrow$ Option [Which]: *nil*) *previous-is*;
Seal Union;

Function for use with the Case syntax:

Let *option-choice* $\triangleq \forall$U, W, R. Is-tag [U, W] $is_n \rightarrow$
(W $\rightarrow$ R) *which-res*, (U $\rightarrow$ R) *if-not* $\rightarrow$ U $u \rightarrow$
If $is_n\ u$ **Is** W *which*
$\rightarrow$ *which-res which*
Else *if-not u*
Fi;

now define some frequently used union operators

Let $in_2 \triangleq$ *next-in* in_1; **Let** $is_2 \triangleq$ *next-is* is_1;
Let $in_3 \triangleq$ *next-in* in_2; **Let** $is_3 \triangleq$ *next-is* is_2;
Let $in_4 \triangleq$ *next-in* in_3; **Let** $is_4 \triangleq$ *next-is* is_3;

define some operators to get a reasonable syntax

Priority 9 **In Associates Right**;
Priority 9 **Out Associates Right**;
Priority 9 $\Rightarrow$ **Associates Right**;
Priority 9 ▌ **Associates Right**;

Let *out-part* $\triangleq$ ∀T. T $t \to$ ∀R. R $r \to t$;
Let *in* $\triangleq$ ∀A, B. A $a \to$ (A $\to$ B) $ab \to ab\ a$;

Infix In $\triangleq$ *in*;
Infix Out $\triangleq$ ∀A, B. A $a \to$ B $b \to a$, *out-part* b;
Infix $\Rightarrow \triangleq$ *option-choice*;
Infix $\Rightarrow \triangleq$ *switchon-choice*;
Infix ▌$\triangleq$ ∀A, B. A $a \to$ B $b \to a$, b;
Bracket Case Esac $\triangleq i$;

so that we can go

Case union-typed-expression
In $is_1 \Rightarrow$ **Type**-1 t $\to$ thing
▌ $is_3 \Rightarrow$ **Type**-3 t $\to$ thing
Out something else
Esac

and

Case expression
In predicate-giving-option $\Rightarrow$ thing
...
Esac

Two alternative brackets:

Bracket Begin End $\triangleq i$;
Bracket $\triangleleft\ \triangleright \triangleq i$;

Option operations (other than If and Case)

Priority 4 $ **Associates Right**;
Let *default* $\triangleq$ ∀T. Option [T] *opt* $\to$ T *default* $\to$
If *opt* **Is**
i
Else *default*
Fi;

Infix $ $\triangleq$ *default*;

Thus *e1* $ *e2* is *opt-out e1* if *e1* Is, and *e2* otherwise.

List **operations:**

This is a collection of polymorphic functions that almost every programme needs:

Constructing a List

$a :: b :: c$ means the same as $a :: b :: c :: nil$ if a, b and c are of the same type, which is achieved by overloading ::

Infix :: $\triangleq \forall \mathsf{T}.\ \mathsf{T}\ h \to \mathsf{T}\ t \to \mathit{list}\ h\ (\mathit{list}\ t\ \mathit{nil})$;
Infix :: $\triangleq \mathit{list}$;

Letrec *append* $\triangleq \forall \mathsf{T}.\ \mathsf{List}\ [\mathsf{T}]\ l_1 \to \mathsf{List}\ [\mathsf{T}]\ l_2 \to \mathsf{List}\ [\mathsf{T}]$:
If *null* l_1
Then l_2
Else *head* l_1 :: *append* (*tail* l_1) l_2
Fi;

convenient shorthand: $(a{::}\ b{::}\ c)$ @ $(d{::}\ e)$ becomes $(a{::}\ b{::}\ c{::}\ d{::}\ e)$

Infix @ $\triangleq$ *append*;

map f $(a :: b :: c)$ becomes $f\ a :: f\ b :: f\ c$

Let *map* $\triangleq \forall \mathsf{T}_1, \mathsf{T}_2.\ (\mathsf{T}_1 \to \mathsf{T}_2)\ f \to$
Begin Letrec *mapf* $\triangleq$ $\mathsf{List}\ [\mathsf{T}_1]\ l \to \mathsf{List}\ [\mathsf{T}_2]$:
If *null* l
Then *nil*
Else f (*head* l) :: *mapf* (*tail* l)
Fi;
mapf
End;

Left-gather and *right-gather*, functions for infixing functions in lists. Both need an initial element to get them off the ground. *right-gather* $f\ e\ (a :: b :: c)$ becomes $f\ a\ (f\ b\ (f\ c\ e))$ Written in infix notation: $a\ \boxed{f}\ (b\ \boxed{f}\ (c\ \boxed{f}\ e))$ as if $\boxed{f}$ associated to the right.

Let *right-gather* $\triangleq \forall \mathsf{T}_1, \mathsf{T}_2.\ (\mathsf{T}_2 \to \mathsf{T}_1 \to \mathsf{T}_1)\ f \to \mathsf{T}_1\ e \to$
Begin Letrec *gather* $\triangleq$ $\mathsf{List}\ [\mathsf{T}_2]\ l \to \mathsf{T}_1$:
If *null* l
Then e
Else f (*head* l) (*gather* (*tail* l))
Fi;
gather
End;

left-gather $f\ e\ (a :: b :: c)$ becomes $f\ (f\ (f\ e\ a)\ b)\ c$

Let *left-gather* $\triangleq \forall \mathsf{TI}, \mathsf{Ta}.\ (\mathsf{Ta} \to \mathsf{TI} \to \mathsf{Ta})\ f \to$
Begin Letrec *gather* $\triangleq \mathsf{Ta}\ e \to \mathsf{List}\ [\mathsf{TI}]\ l \to \mathsf{Ta}$:
If *null* l
Then e
Else *gather* (f e (*head* l)) (*tail* l)
Fi;
gather
End;

Let *gather* ≜ *right-gather*; – because it is the most popular of the two.

Another powerful one: *filter predicate l* is a list of the members of *l* that satisfy *predicate*.

```
Letrec filter ≜ ∀T. (T → Bool) predicate → List [T] l → List [T]:
                  If null l
                  Then nil
                  Elif predicate (head l)
                  Then head l :: filter predicate (tail l)
                  Else filter predicate (tail l)
                  Fi;
```

reverse a list:

```
Letrec reverse ≜ ∀T. List [T] reversed → List [T] reversee → List [T]:
                   If null reversee
                   Then reversed
                   Else reverse (head reversee :: reversed) (tail reversee)
                   Fi;
Let reverse ≜ reverse nil;
```

The length of a list:

```
Letrec length ≜ Int len → ∀T . List [T] l → Int:
                  If null l
                  Then len
                  Else length (len + 1) (tail l)
                  Fi;

Let length ≜ length 0;
```

take the first (up to) *n* elements of a list:

```
Letrec first ≜ Int n → ∀T. List [T] l → List [T]:
                 If null l
                 Then nil
                 Elif n = 0
                 Then nil
                 Else head l :: first (n − 1) (tail l)
                 Fi;
```

pick the n^{th} member of a list:

```
Letrec elem ≜ Int → (∀T.List [T] → T):
                Int n → ∀T. List [T] l →
                If n = 1
                Then head l
                Else elem (n − 1) (tail l)
                Fi;
Infix Elem ≜ elem;
Type String ≜ List [Char];
```

Functions for printing and reading strings:

```
Let digits ≙ "0123456789";


Let print-int ≙ Int n →
                Begin Let char-from-dig ≙ Int n →
                                              (n + 1) Elem digits;
                      Letrec string-from-int ≙ Int n → String so-far → String:
                              If n = 0
                              Then so-far
                              Else string-from-int (n ÷ 10)
                                   ((char-from-dig (n Rem 10)) :: so-far)
                              Fi;
                      If n = 0
                      Then '0 :: nil
                      Elif n < 0
                      Then '− :: string-from-int (− n) nil
                      Else string-from-int n nil
                      Fi
                End;

Let position-of-char-in ≙ String s → Char ch → Int:
    BeginLetrec pos-in ≙ String s → Int p → Int:
                        If null s
                        Then 0
                        Elif head s = ch
                        Then p
                        Else pos-in (tail s) (p + 1)
                        Fi;
        pos-in s 1
    End;
```

Functions for interaction with the terminal:

```
Let terminal-input-list ≙ String: opt-out (get-file "*"); – always works.
Let string-to-terminal ≙ String s → opt-out (make-file "*") s;
Let print-to-terminal ≙ String s → string-to-terminal s :: nil;

HOLE
```

The **HOLE** indicates to the compiler that the preceding text has been a prelude; programmes which use a particular prelude (in this case all programmes) are compiled as if they were inserted in place of the **HOLE**.

2. List operations

Join two lists with a filler between them, provided that the second is not *nil*. This is useful for printing things with commas between the elements:

```
Let join-with ≙ ∀T. List [T] filler → List [T] l1 → List [T] l2 →
                  l1 @ If null l2
                       Then nil
                       Else filler @ l2
                       Fi;

Letrec is-in ≙ ∀T. (T → T → Bool) eq → T s → List [T] list → Bool:
               If list Is T head, List [T] tail
               → If eq head s
                 Then true
                 Else is-in eq s tail
                 Fi
               Else false
               Fi;

Letrec skip-first ≙ Int n → ∀T. List [T] l → List [T]:
                    If null l
                    Then nil
                    Elif n = 0
                    Then l
                    Else skip-first (n − 1) (tail l)
                    Fi;

Priority 5 FROM Associates Right;
Infix FROM ≙ c skip-first;

Letrec upto ≙ ∀T. List [T] l → Int n → List [T]:
              If n = 0
              Then nil
              Elif l Is T h, List [T] t
              → h :: upto t (n − 1)
              Else nil
              Fi;

Priority 5 UPTO Associates Right;
Infix UPTO ≙ upto;
```

Functions for splitting lists. *Split predicate list* splits *list* into a pair of lists. The first is an initial sublist of list for which none of the members satisfy *predicate*, and the second is the remainder.

```
Letrec split ≙ ∀ T. (T → Bool) predicate → List [T] l →
                 List [T] × List [T]:
                 If l Is T first, List [T] rest
                 → If predicate first
                    Then nil, l
                    Else Let before, after ≙ split predicate rest;
                         (first:: before), after
                    Fi
                 Else nil, nil
                 Fi;

Let until ≙ ∀ T. (T → Bool) predicate → left ∘ split predicate;
– This one should be somewhere else:
Let not-predicate ≙ ∀T. (T → Bool) predicate → T x → Not predicate x;
Let while ≙ until ∘ not-predicate;


Let associated ≙
    ∀ T1, T2, T3. (T3 → T1 → Bool) equal →
    T3 thing →
    List [(T1 × T2)] association-list →
    Option [T2]:
    Begin Type Assoc ≙ T1 × T2;
          If filter (equal thing ∘ left) association-list Is Assoc a, List [Assoc] rest
          → opt-in (right a)
          Else nil
          Fi
    End
HOLE
```

In this case, the prelude is called "list-operations". To access the definitions contained above, a programme can begin with a comment like this:

– **PARENT** "list-operations"

which indicates that the definitions in the file "list-operations" should be included.

Appendix 3: Reading From a Terminal

```
– PARENT "string-operations"
– read an item, reflecting characters as we go.
– uses valid-characters and terminators to decide whether or not to
– accept a character.
–
– some useful characters:
Let rubout ≜ int-to-char 127;
Let backspace ≜ int-to-char 8;
Let bell ≜ int-to-char 7;

Letrec read-item ≜
        (Char → Bool) is-a-valid-char → (Char → Bool) is-a-terminator →
        String so-far → String input →
        String × String × String:
–       (Item × remainder of input × reflections)
        If input Is Char ch, String rest
        → Begin Let read-item ≜ read-item is-a-valid-char is-a-terminator;
                Let invalid ≜ Begin Let new, rest, ref ≜ read-item so-far rest;
                                    new, rest, (bell :: ref)
                              End;
                Case ch
                In = rubout
                   ⇒ If Not null so-far
                     Then Let new, rest, ref ≜ read-item (tail so-far) rest;
                                new, rest, (backspace :: ' ' :: backspace :: ref)
                     Else invalid
                     Fi
                ▌ is-a-terminator
                  ⇒ If is-a-valid-char ch
                    Then (ch :: so-far), rest, (ch :: nil)
                    Else so-far, rest, nil
                    Fi
                ▌ is-a-valid-char
                  ⇒ Let new, rest, ref ≜ read-item (ch :: so-far) rest;
                    new, rest, (ch :: ref)
                Out invalid
                Esac
           End;
        Else so-far, input, nil
        Fi;
```

– But that produces the *item* backwards, and involves a variable
– that the user needn't know about, so redefine it:

```
Let read-item ≜
    (Char → Bool) is-a-valid-char →
    (Char → Bool) is-a-terminator →
    String input →
    Begin Let item, rest, ref ≜ read-item is-a-valid-char is-a-terminator nil input;
          reverse item, rest, ref
    End;

Let read-line ≜ read-item (Char x → true) (Char c → c = ''r   – carriage return
                                                 Or c = ''l   – line feed
                                                 Or c = ''n   – new line
                                                 Or c = ''e); – escape
    HOLE
```

Now *read-line terminal-input-list* will return three strings. The first will be the first line read from the terminal, the second will be the remainder of the input and the third will be the characters which should be printed at the terminal in response to the user's key-strokes.

The following programme illustrates the use of *read-line*:

```
Let line, rest, reflection ≜ read-item terminal-input-list;
"Please type a line terminated with carriage return, line feed or escape'n" @
print-to-terminal reflection @ "'nThis line backwards is:'n" @
reverse line @ "'nFinished'n"
```

when the programme is run the user is prompted for a line. When a key other than rubout is pressed, it is reflected to the terminal. When rubout is pressed, the programme responds with backspace, space, backspace to erase the last character on the line, or a bell if there are no characters on the line. When one of the terminator characters is typed, it is reflected, but the programme then goes on to print a message, followed by the text of the line backwards and then "Finished", after which it stops.

Read-item behaves in a similar way to *read-line*, but requires two predicate arguments. The first should return true for every character that is to be accepted, and the second should return true for every character that is to indicate the end of an item.