AUTHOR: **Gary Meehan** DEGREE: **Ph.D.**

TITLE: **Aspects of Functional Programming**

DATE OF DEPOSIT: ....

Aspects of Functional Programming

by

Gary Meehan

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Computer Science

August 1999

Contents

List of Tables

List of Figures

Acknowledgments

I would primarily like to thank my supervisor, Mike Joy, for his advice and help throughout the duration of my research, for getting me on the course in the first place, and not being too uptight about the size of my coffee bill. I'd also like to thank Steve Matthews for his encouragement, especially with regards to the fuzzy stuff (surely I'm allowed *one* colloquialism, Mike); Tom Axford, who along with Mike Joy, laid the groundwork for Aladin; and Ananda Amatya for various comments and papers over the years. In addition, I'd like to thank Mike and Steve for their comments and editorial advice on this thesis, and their corrections of my lousy typing.

Finally, I'd like to say to Richard Gault: 'Beat you!'

Declarations

The work on fuzzy logic and functional programming, described in Chapter 3, is based largely on that detailed in [77] published in the *Journal of Functional programming* and its preceding technical report [75]. The published paper was written in collaboration with my supervisor, Dr Mike Joy, who acted in an advisory and editorial capacity; the technical report was authored by me alone.

The Ginger compiler described in Chapter 4 has its origins in the technical report written by Dr Joy [53]. The compiler was written from scratch by me alone, apart from the parser for the language which is based on one developed by Dr Joy for a Ginger interpreter written in Java. The text of Chapter 4 is based on [78] published in *Software — Practice and Experience*, co-authored with Dr Joy who acted in an advisory and editorial capacity. An earlier version of the compiler is described in a technical report [74], written solely by me.

My work on Aladin detailed in Chapters 5 and 6 is based on that done by Dr Joy and Dr Tom Axford of Birmingham University [10, 11]. I developed the updating denotational semantics for Aladin, with and without partial evaluation, from which an operational semantics for Aladin was obtained. My implementation of the Aladin without partial evaluation, based on this semantics, has previously been described in a technical report [76]. Nothing has yet been published on my work on partial evaluation and Aladin.

Abstract

This thesis explores the application of functional programming in new areas and its implementation using new technologies. We show how functional languages can be used to implement solutions to problems in fuzzy logic using a number of languages: Haskell, Ginger and Aladin. A compiler for the weakly-typed, lazy language Ginger is developed using Java byte-code as its target code. This is used as the inspiration for an implementation of Aladin, a simple functional language which has two novel features: its primitives are designed to be written in any language, and evaluation is controlled by declaring the strictness of all functions. Efficient denotational and operational semantics are given for this machine and an implementation is developed using these semantics. We then show that by using the advantages of Aladin (simplicity and strictness control) we can employ partial evaluation to achieve considerable speed-ups in the running times of Aladin programs.

Chapter 1

Introduction

Functional programming [15, 26, 40, 93] is, as its name suggests, programming with functions, by defining them and applying them to arguments. Functions can be passed as arguments to other functions and returned as the result of a function. The definition of a function in functional programming is an expression rather than a sequence of commands. Functional programming is *declarative* in that we say *what* we want rather than *how* we get it. Functional languages have found applications in theorem proving, telephone controllers, database management, expert systems, control of distributed applications, workforce management and geometric modelling amongst other things [97, 116].

Functional programming has its roots in the work of Alonzo Church, Haskell Curry and Moses Schönfinkel into the λ-calculus and combinatory logic in the 1920s and 30s [13, 42]. The first functional language, Lisp (List Processing), was invented by John McCarthy in the early 60s [72, 73, 121] and found wide use in the field of Artificial Intelligence. John Backus [12] called for a functional style of programming to be used instead of the imperative (or von Neumann) style which he criticised in scathing terms for the bloated size of its languages, dependency on 'one word at a time' operations and historical state and the fact that its programs are not amenable to mathematical reasoning. He described a functional language called FP, similar to the purely functional subset of Lisp, and gave an algebra which could be used to reason about programs written in this language.

Robin Milner's work on strong polymorphic typing in the late seventies [80], following on from earlier work by J. Roger Hindley [43], led to the Hindley-Milner type-system, the basis for the type systems of most modern functional languages. The first language to use this type system was ML (Meta Language) [81, 112]. David Turner showed how a functional language could be implemented using a fixed set of basic functions known as *combinators* [109] and used this idea to implement Miranda [45, 106, 110, 111]. Other functional languages such as Hope [18] and Orwell [113] also appeared at this time.

The increasing proliferation of functional languages led in September 1987 to the start of development of the language Haskell [14, 85, 107] which aimed to concentrate the work being done in a number of disparate (lazy) languages into a single one. Haskell is a powerful language, with a sophisticated type and module system, yet is surprisingly clean with few of the idiosyncrasies found in other languages, both functional and imperative. This is due in part to its use of *monads* to cope with side-effects and I/O in a purely functional manner [114, 115] and its use of *type classes* to organise the overloading of function names in a systematic way [32, 88]. There are other functional languages in use today, notably Clean [91, 92] and Erlang [4, 5, 6], and ML and Lisp in their many variants continue to have a strong following.

It is hard to quantify precisely what facilities a functional language should offer. From the basic definition given above, an imperative language can be used functionally (using pointers to functions) [38], though the syntax of C does tend to fight against this. However, we shall attempt to enumerate the features and advantages [46, 64] one might expect of a functional language, drawing the reader's attention to the list of exceptions that follows the list.

- Functional languages are pure in that they:
  - are *referentially transparent*, that is the result of a function depends only on its arguments;
  - are side-effect free;
  - do not allow the destructive update of variables. This means that a variable in one part of a program can't have its value changed unexpectedly in another.

  This means that the various parts of a functional program can be executed in any order. This is a great advantage when it comes to program optimisation and makes it simpler to evaluate the various parts of a program in parallel. The purity of functional languages also makes them more amenable to mathematical reasoning.

- We can factor out commonly-used structures of function definitions into higher-order functions (that is functions which take functions as arguments) leading to greater modularity, abstraction and brevity of programs.
- Functional programs are polymorphic, that is functions can be defined to work over many types rather than just one. One single function can be used in place of many, essentially equivalent ones.
- It is commonly easier and neater to introduce new types into functional languages than their imperative equivalents, especially if we exploit polymorphism. Instances of such types can be cleanly and simply taken apart using pattern matching.
- Functional programs are elegant and far clearer than their imperative counterparts. They're usually far shorter, too.

- Functional programs can be evaluated lazily, that is we only evaluate those parts of a program that are actually needed and thus we only do the minimum amount of work necessary. This laziness is transparent to the user.

- Functional programs are strongly typed and can never 'go wrong' [43, 80] in the sense that any illegal operations can be caught at compile-time. Moreover, the user doesn't have to provide the types of functions explicitly: typing can be completely inferred by the implementation at compile-time.

- Functional languages offer automatic memory management. This is actually a necessity in a functional language since the programmer's lack of control means that they cannot insert manual memory (de)allocation operations into their programs.

Different functional languages do not conform with all the points above, either due to a philosophical viewpoint or due to reasons of efficiency. Standard ML, Erlang and Lisp are all strict (though there is a lazy variant of ML [8]) and non-pure, though in the former two the use of non-pure operations is discouraged and usually only used to facilitate Input/Output. Lisp is also completely type-less.

Why, given all the above advantages, are functional languages not more prevalent? Philip Wadler recently addressed this problem [117]. Some reasons he gave for the lack of adoption of functional languages are:

- It is hard to use functional languages in co-operation with other languages, in particular C/C++ libraries and components. However, there are projects currently being undertaken to solve this problem, such as Green Card [90] which interfaces Haskell with C libraries, and HaskellScript which interfaces Haskell with the COM/ActiveX framework [63]

- Libraries for functional languages (in particular GUI ones) are still not fully developed.

- Implementations of functional languages are not available on many machines. The ubiquitousness of C means that it is often preferred despite the fact that it is hardly ever the best language for the job.

- Functional language implementations tend to be provided by universities and are hard to install. They are also usually under active development and continually changing. Commercial users tend to prefer a stable system with plenty of support. There are some commercial offerings such as Miranda [110, 111] and implementations of ML [37] and Lisp [28, 36]. Ericsson also support Erlang [24] and efforts are being made to produce a stable version of Haskell — Haskell 98 [39].

- Stand alone applications implemented using a functional language tend to have an unacceptably large memory footprint caused by the need to incorporate the entire runtime package for the library in the program.

- There is a lack of tools for functional languages, in particular debuggers, profilers and integrated development environments.

- It is hard to train programmers used to imperative programming in some of the aspects of functional programming [54].

Wadler also gave a couple of non-reasons:

- Functional programs are not as slow as they are perceived: modern compilers can get performance that is as good as C in some cases and within a factor of two slower on average. Besides, the popularity of Visual Basic [108] and Java [7], neither of which has particularly fast implementations, shows that performance is not everything.

- Functional programming isn't hard for programmers used to other paradigms to understand — it may take time but once they understand it they are usually impressed with its clarity and elegance.

Much effort is being made to alleviate the disadvantages of functional programming given above, and it is comforting to know that obtaining solutions to these problems is probably only a matter of time. Indeed, given the fact that most implementations are the work of a handful of individuals compared to the hundreds or even thousands who work on the language products of Microsoft and Borland it is an achievement that functional languages have come so far.

The above provides the motivation for our research in that we aim to tackle some of the downsides of functional programming. First of all, we investigate a new application field for functional programming — fuzzy logic, sets and systems — and show how these concepts can be clearly and elegantly implemented in a functional language, primarily Haskell, but we also implement fuzzy concepts in Ginger and Aladin (Chapter 3).

We then address the problem of portability of functional programs in Chapter 4 and offer a solution in the form of a compiler for the functional language Ginger [53, 78] which produces code for the Java Virtual Machine (JVM) [7, 79]. The problem of interfacing functional languages with other languages is tackled in Chapter 5 by means of Aladin [10, 76]. This a stripped-down functional language which offers fine control over the strictness of functions. We develop an implementation of Aladin, which produces code for the JVM as in our Ginger compiler. Aladin has other applications too, and we show in Chapter 6 how it can be used to *partially* evaluate programs [51] and, coming full circle, how partially evaluating functional implementations of problems in fuzzy logic can yield substantial performance benefits.

Chapter 2

Background

In this chapter we aim to give a brief introduction to some of the key concepts of functional programming which we will encounter in the further chapters of this thesis. We do not aim to give an introduction to functional programming itself; instead the reader is directed to the excellent works by Bird and Wadler [14, 15] or Thompson [106, 107]. Note that all examples will be given using Haskell syntax unless stated otherwise.

2.1 Functions, The λ-calculus and Supercombinators

A functional program normally consists of a list of function definitions. What happens when the program is executed depends on the language and implementation. A functional language intepreter will allow the user to evaluate the application of functions to arguments on demand in the interpretive enivronment. A compiler for a functional language typically requires the definition of standard function which takes no arguments, usually named `main`, which will be evaluated when the program is executed.

The definition of a function is essentially a λ-expression [13, 42], though the syntax of the particular language usually disguises this. A λ-expression is either a basic expression, such as a variable, constant or a primitive function (for example addition), an application of one λ-expression to another, or a λ-abstraction. A λ-abstraction provides a way of defining a function without naming it. For example, the function `square` defined as:

```
square x = x * x
```

is equivalent to the λ-abstraction ($\lambda x.x$ `*` x). Here the variable x is said to be *bound* by the λ-abstraction; variables which aren't bound are said to be *free*.

We do not consider the λ-calculus in this thesis, however one particular class of λ-expression is important to us and that is the *supercombinator* [86]. A supercombinator is a λ-expression of the form $\lambda x_1 . \ldots . \lambda x_{n \geq 0}.E$ where E is not a λ-abstraction, no free variables occur in the expression, and any λ-abstractions in E are also supercombinators. For example, 2 `+` 5, $\lambda x.x$ `*` x and $\lambda f.f$ ($\lambda\ x.x$ `*` x) are all supercombinators, whereas $\lambda x.y$, $\lambda x.y$ `-` x and $\lambda f.f$ ($\lambda\ x.f\ x$) are not. Supercombinators are important since they can be compiled into efficent code [25, 86] and form the basis of our Ginger compiler (Chapter 4).

2.2 Strictness and Laziness

A function does not always need to know the value of some or all of its parameters for it to be able to return a result. For example, the function:

```
forty_two x = 42
```

always returns `42` no matter what the value of `x` is. In particular, we would expect the answer to be `42` even when `x` is undefined, for example, when it is the (undefined) result of `1 / 0`. Less trivially, in the case of boolean conjunction defined as:

```
x && y = if x then y else False
```

if we know that `x` has the value `False` then the result of the conjunction must also be `False` no matter what the value of `y` is, even if it is undefined. Also, if the value of `y` is the result of some, potentially expensive, computation then we would expect to be able to obtain the result of the conjunction without performing this computation when `x` is `False`.

Languages which do not evaluate their arguments before applying functions are said to be *lazy* and use lazy evaluation; languages which do evaluate their arguments are said to be *strict* and use strict evaluation. In a lazy language, we can expect the result of `forty_two (1 / 0)` to be 42, since lazy languages only evaluate their arguments only when needed. In a strict language, the result of `forty_two (1 / 0)` gives an error, since the language implementation would attempt to first evaluate `1 / 0` which is undefined. See the next section for further details of strict and lazy evaluation.

Formally, a function `f` is said to be strict in its argument if and only if:

$$\mathtt{f}\ \perp = \perp$$

It is said to be non-strict or lazy otherwise. All functions can be thought of as having 1 (or 0) arguments *via* currying. Here $\perp$ can be thought of as a non-terminating computation or an undefined or error value. So, `forty_two` is lazy in its argument since $\mathtt{forty\_two}\ \perp = \mathtt{42} \neq \perp$.

A lazy language can take full advantage of laziness without any intervention by the user; strict languages in general can't, though a particular language implementation might provide mechanisms where the user can simulate laziness. In addition, strict languages might provide primitives which are in effect lazy in some of their arguments. For example, the strict language Standard ML provides the operators `andalso` and `orelse` which implement logical conjunction and disjunction respectively, but which do not evaluate their second arguments unless necessary. This feature isn't restricted to functional languages: the `&&` and `||` operators of C and its derivatives have the same effect as Standard ML's `andalso` and `orelse`. In C parlance, the laziness is referred to as *short-circuiting*.

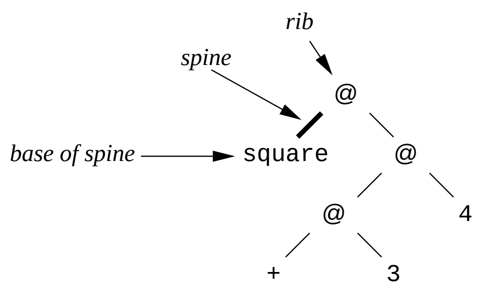
While lazy evaluation allows us to circumvent some evaluation and leads to more general programs, strict evaluation is generally more efficient as lazy evaluation can lead to the proliferation of unevaluated expressions (see Section 7.5 of [14], for example) and a clever compiler can exploit strictness to produce more efficient code. For this reason, *strictness analysis* (see Chapter 22 of [86], for example) is often used to determine when the arguments of a function can be safely treated as strict. This however is an undecidable problem, though good approximations can be obtained.

2.3 Graph Reduction

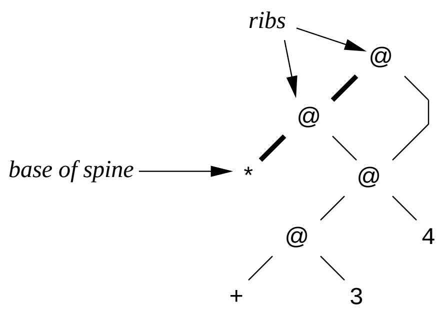
Graph reduction provides a way to implement functional languages [48, 86, 89]. The evaluation of an expression involves the application of reduction rules, that is, the definition of functions, until no more reduction rules are applicable, at which point the expression is said to be in *normal form*. For instance, suppose we have the definition:

```
square x = x * x
```

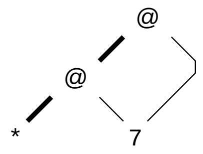
and we want to evaluate the expression `square (3 + 4)`. This is stored as the graph:



Note that we represent infix operators like `+` (which come between their arguments) as prefix ones (which come before their arguments). Since values in pure functional languages are immutable, we can share sub-expressions saving both time and memory. We can evaluate the expression by *reducing* the graph[14, 86]. For example, if we reduce `square (3 + 4)` we get the graph:



The first time `3 + 4` is evaluated the graph reduces to:



Hence we only need to evaluate `3 + 4` *once* as the result is shared between the nodes that point to it. Reduction of the multiplication yields the answer `49` which is the normal form of `square (3 + 4)`.

Usually when we are evaluating an expression there is more than one subexpression that can be reduced (each such subexpression is called a *redex*). Implementations of lazy functional languages choose to evaluate the outermost redex first; that is apply the reduction rule of the function at the base of the spine of the graph as in the example above. So we evaluate the function itself before its arguments. This is lazy evaluation, also known as outermost graph reduction. For instance, we reduce `square (3 + 4)` to `(3 + 4) * (3 + 4)` rather than `square 7`. Doing the latter is strict evaluation (also known as innnermost graph reduction). Of course, eventually we may need to evaluate the arguments of a function, but such an operation is only usually done explicitly by primitives of the language, such as `*` in our `square` example.

In a lazy functional language, a program is not usually evaluated to Normal Form but instead only to *Weak Head Normal Form* (WHNF). An expression is said to be in WHNF if it is of the form $f\ e_1\ \ldots\ e_n$ where either $n \geq 0$ and f is either a data object, such as an integer or a type constructor (see below) or is a function which requires more than n arguments.

Lazy and strict evaluation will always reduce an expression to the same normal form, however evaluating an expression using strict evaluation may not terminate in some cases where lazy evaluation does. For example, consider the program:

```
from x = x : from (x + 1)


main = head (from 0)
```

Here `:` (cons) is the list constructor and `head` selects the head of the list, that is, the expression to the left of the `:`. Note that the expression h : t is in normal form no matter what h and t are (since cons does not evaluate its arguments). The expression `from x` yields the list `[x, x + 1, x + 2,...]`, and hence evaluating the expression `head (from 0)` should give the answer `0`. With lazy evaluation, we have the reduction sequence:

$$
\begin{aligned}
\texttt{main} &\Longrightarrow \texttt{head (from 0)} \\
&\Longrightarrow \texttt{head (0 : from (0 + 1))} \\
&\Longrightarrow 0
\end{aligned}
$$

Like `*`, `head` evaluates its argument (to normal form). The result of the evaluation should be a cons, otherwise we have an error, at which point `head` selects the leftmost argument of the cons. Now suppose we evaluate the expression strictly:

```
main ⟹ head (from 0)
     ⟹ head (0 :  from (0 + 1))
     ⟹ head (0 :  from 1)
     ⟹ head (0 :  1 :  from (1 + 1))
     ⟹ head (0 :  1 :  from 2)
     ⟹ head (0 :  1 :  2 :  from (2 + 1))
     ...
```

The recursion would never terminate as the innermost expression can always be reduced.

2.4 Types, Polymorphism and Overloading

Every expression in a strongly-typed functional language, such as Haskell, ML or Miranda, has a type which can be inferred at run-time [43, 80]. Using Haskell's syntax, we can define a type as:

- A basic type, such as `Int`, `Char` or `Bool`. For instance, `1 + 3` has type `Int` and `1 <= 3` has type `Bool`.

- A type variable, for example, `a` or `b`. These variables stand for any type with repeated occurrences in a type standing for the same type.

- A function type, written as d `->` r where d is the domain of the function and r is the range. Functions are curried and `->` associates to the right. For instance, `toUpper` which converts characters into their upper case equivalents has type `Char -> Char`.

- A compound type, such as a list or a tuple. For instance, the list `[1, 2, 3]` has type `[Int]` while `(1, 'g')` has the type `(Int, Char)`.

A type which contains type variables is said to be *polymorphic*; otherwise it is said to be *monomorphic*. Functions which have a polymorphic type are themselves referred to as polymorphic. The simplest example of a polymorphic function is seen with the identity function:

```
ident x = x
```

Clearly it does not matter what the type of `x` is, be it an integer a string or a function, since all `ident` does is return it. We assign `x` the polymorphic type `a` and therefore, since `ident` returns a value with the same type of its argument, we can write the type of `ident` as `a -> a`. We can have more than one type variable in a type, for instance, the function `map` defined as:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

has type `(a -> b) -> [a] -> [b]`. So the expression `map (+ 1) [1, 2, 3]` has the value `[2, 3, 4]` of type `[Int]` and the expression `map toUpper "gary"` has the value `"GARY"` of type `String` which in Haskell is equivalent to `[Char]`.

The above form of polymorphism is known as *parametric* polymorphism. In parametric polymorphism, the same code for each function is used no matter what the type of the arguments are. The other form of polymorphism is known as *ad hoc* polymorphism. This form of polymorphism is used to enable function names to be reused for arguments of different types, for example, to allow `+` to represent integer and real addition, two operations which have fundamentally different implementations.

Haskell uses type classes [32] to organise the overloading in a systematic and powerful way which retains the ability of programs to be strongly typed. A type class is a collection of types all of which provide implementations for the functions defined by the class. For instance, consider the `Eq` class:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y       = not (x==y)
```

This states that any type, `a`, which is a member of this class must implement the functions `==` and `/=` of the given type. A default implementation, which the user is free to override, is provided for `/=` in terms of `not` and `==`. To make a type a member of the class, we declare it as an *instance* of the class and provide any necessary implementations. In the standard prelude (set of functions which are automatically imported into a user program) of the Haskell interpreter Hugs [50], the type `Char` is made an instance of the `Eq` class by the following declaration:

```
instance Eq Char where (==) = primEqChar
```

Here `primEqChar` is the Hugs primitive function which is used to compare two characters for equality. Note that the default implementation for `/=` is used. We are not restricted to overloading on basic types, either. For instance, for any `a` which is an instance of the `Eq` class we can make the type `[a]` an instance of the `Eq` class, *viz*:

```
instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x==y && xs==ys
    _      == _      =  False
```

At compile time, the compiler substitutes all overloaded functions with their non-overloaded definitions according to the types of the expressions in which they occur. This is known as *resolving* the overloading. Sometimes the compiler may not have enough type information to resolve the overloading, in which case explicit types have to be supplied.

Functions whose definitions contain overloaded functions have to have their types constrained. For instance, consider the function `member`:

```
member x []     = False
member x (y:ys) = x == y || member x ys
```

One would expect this function to have type `a -> [a] -> Bool`. However since we use the `==` operator to compare `x` and `y` the type `a` has to be an instance of the `Eq` class and hence the type of `member` is written as `Eq a => a -> [a] -> Bool`. Here the `Eq a` restricts the types to which `a` can range over to precisely those which are instances of `Eq`.

Haskell uses classes to overload a large number of functions. For instance, the `Ord` class is used to overload the comparison operators, the `Num` class to overload the arithmetic operators, and the `Show` class is used to group all those types whose members can be converted into strings (that is, shown on a terminal).

Other functional languages handle overloading in different ways. Miranda has a single numeric type, thus it does not need to overload the arithmetic operators, and treats the comparison operators as polymorphic operators, for example the operator `<=` has the Miranda type `* -> * -> bool` (equivalent to the Haskell type `a -> a -> Bool`). ML overloads its arithmetic and comparison operators but requires that the overloading be resolved at compile time, by the user giving explicit monotypes for the functions in which they are used if necessary. It however treats equality as a special case and uses an *equality type*. In ML, polymorphic types are given by preceeding the type variable by a single quote, for example `'a`, `'b`, etc., but if the type is expected to have equality defined over it then it preceeded by *two* quotes. For example, the `member` function described above would have the type

```
''a -> ''a list -> bool
```

in ML, where `'a list` is ML's equivalent of Haskell's `[a]`.

2.5 Abstract Machines

The traditional concept of compilation involves taking a source program and producing a sequence of machine code instructions. However, since different processors have different instruction sets this machine code will only run on the machine it was compiled for; to run the program on another machine means we have to recompile the source program. This difference between processor instruction sets is not a trivial one, either [41, 122]. The CISC (Complex Instruction Set Computer) philosophy of the Intel 80x86 series (which includes the Pentium and its progeny) and the Motorola M680x0 series has a large complex set of instructions some of which can be quite large, measured in the number of bits they occupy on the machine, and specialised. The alternative, known as RISC (Reduced Instruction Set Computer) is to use a small, but fast, set of instructions and is used by Sun's SPARC, Acorn's ARM and the PowerPC of IBM and Motorola. Processors are constantly evolving, too: as a single example, in the past few years, the Pentium has been succeeded in turn by the Pentium MMX, the Pentium II and the Pentium III, each of which has more instructions than the last.

The above mitigates against producing a compiler which compiles a high-level language straight down to the machine code suitable for a particular processor. Instead, an intermediate *abstract machine* is typically used. An abstract machine (AM) can be viewed as a simplified, and more importantly portable, version of a processor, complete with its own instruction set and memory stores such as heaps, stacks and registers.

The first step of an implementation of a high-level language that uses an abstract

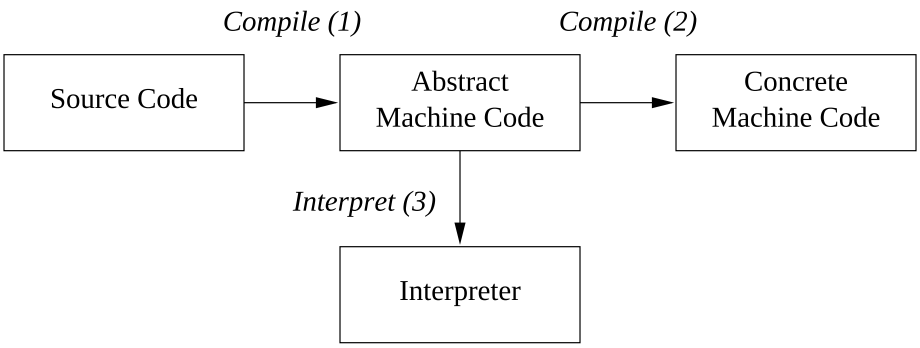


Figure 2.1: The steps taken by an Abstract Machine

machine is to compile the source code down to AM code (step 1 in Figure 2.1). This step should be completely portable. There are then two ways we can run this AM code: either using an interpreter (step 3) or by compiling the AM code to the machine code for a particular machine (step 2). Since the AM code is simpler and closer to the processor than the high-level source, this task is easier than producing a compiler which produces concrete machine code directly from the source code. The simplicity and platform-independence of abstract machines make them relatively easy to implement, optimise and port to different machines. The downside is that programs compiled using an abstract machine are generally slower than those which do not.

The first successful abstract machine for a functional language was Landin's SECD machine [60] which provided a platform for evaluating strict functional programs. This consisted of four components:

- The *Stack* used to hold intermediate results.
- The *Environment* used to map variable names to values.
- The *Control* where the instructions to be executed are stored.
- The *Dump* used to store previous states while other expressions are being evaluated, for example when a function calls another function.

This inspired variations such as Krivine's machine [34] and lazy machines such as the Categorical Abstract Machine [93] and the CASE machine [22].

Turner [109] produced an abstract machine based on combinatory logic named the SKI machine (the **S**, **K** and **I** combinators form the basis of combinatory logic). The G-Machine [8, 86], which forms the basis for our implementations of Ginger and Aladin (see Chapters 4 and 5), is an abstract machine based on graph reduction (see above). It is similar to the SECD machine except that it evaluates expressions lazily and the environment of the SECD is replaced by the graph of the expression being evaluated.

Other abstract machines for functional languages include the Three Instruction Machine (TIM) [25], the Spineless, Tagless G-Machine [87] and the lazy abstract machine derived from Launchbury's semantics for a lazy functional language [61] by Sestoft [99]. Abstract machines are not restricted to functional languages, either. The first implementations of Pascal, for example, used an abstract code called P-code, and today the Java Virtual Machine [79] provides a machine-independent platform for the object-oriented language Java.

Chapter 3

Fuzzy Functional Programming

Fuzzy logic, developed by Lotfi Zadeh [124, 125], is a form of multi-valued logic which has its grounds in Łukasiewicz's work on such logics [66, 67]. It finds many applications in expert systems (in particular control problems) [21, 68, 96, 120], neural nets [23], formal reasoning [82, 104], decision making [21, 82, 126], database enquiries [82] and many other areas. The use of fuzzy logic in such applications not only makes their solutions simpler and more readable but can also make them more efficient, stable and accurate (see, for example, Chapter 2 of [120], or Chapter 3 of [123]).

Fuzzy logic has been applied to many languages — both in extending standard languages such as Prolog [70], Fortran [44], APL [82] and Java [3], and in custom-designed languages such as Fuzzy CLIPS [83], FIL [1, 2], and FLINT [65]. However no one, to the author's knowledge, has combined fuzziness with a functional language.

In this chapter we aim to give an introduction to fuzzy logic using the language Haskell [85] to implement our solutions. Throughout the chapter we shall give examples of using the programs we develop using the Haskell interpreter Hugs [107]. We shall see how the high-level, declarative nature of a functional language allows us to implement easily and efficiently solutions to problems using fuzzy logic and, in particular, how the presence of functions as first-class values allows us to model the key concept of a *fuzzy subset* (see Section 3.2) in a natural way. We have chosen Haskell for the implementation language because of its strong typing and its facilities for overloading operators (in particular the logical ones). In later chapters, we will show how fuzzy concepts can be implemented in the weakly-typed language, Ginger (see Section 4.4.1), and how implementing fuzzy primitives in Aladin (see Sections 5.4.5 and 6.6.6) opens up the possibility for fuzzy programs to be partially evaluated and how, by doing so, significant performance benefits can be achieved.

3.1 Fuzzy Logic

In fuzzy logic, the two-valued truth set of boolean logic is replaced by a multi-valued one, usually the unit interval $[0,1]$. Truth sets taking values in this range are said to be *normalised.* In this set, 0 represents absolute falsehood and 1 absolute truth, with the values in between representing increasing degrees of truthness from 0 to 1. So we can say that 0.9 is 'nearly true', 0.5 is 'as true as it is false' and 0.05 is 'very nearly false'. The nearer a value is to 0 or 1 the *crisper* it is; the nearer it is to 0.5 (the middle value of the range) the *fuzzier* it is.

The standard connectives of boolean logic — $\wedge$, $\vee$ and $\neg$ — are adapted so that they work with the fuzzy truth set. There are many ways in which this can be done, but whatever definition we choose we expect the following to hold [27, 126]:

1. $\wedge$ and $\vee$ should be associative and commutative.

2. $\wedge$ and $\vee$ should be monotonic. That is, if $a, b, c \in [0,1]$ and $a \leq b$ then $a \wedge c \leq b \wedge c$ and similarly for $\vee$.

3. 1 and 0 are the identities of $\wedge$ and $\vee$ respectively. From this and monotonicity

we deduce that 1 and 0 are *annihilators* of $\vee$ and $\wedge$ respectively.

4. $\neg$ should be anti-monotonic. That is if $a, b \in [0, 1]$ and $a \leq b$ then $\neg b \leq \neg a$. In the majority of fuzzy logics this is strict monotonicity, that is if $a < b$ then $\neg b < \neg a$.

5. $\neg$ should be its own inverse, that is if $a \in [0, 1]$ then $\neg\neg a = a$.

6. If we restrict the truth set to just 0 and 1, then our logic should behave *exactly* as boolean logic.

Definitions of $\vee$ and $\wedge$ that satisfy the above are also known as *t-norms* and *t-conorms* (or *s-norms*) respectively. We would also expect the connectives to be continuous and to satisfy DeMorgan's laws:

$$\begin{aligned} \neg x \wedge \neg y &= \neg(x \vee y) \\ \neg x \vee \neg y &= \neg(x \wedge y) \end{aligned}$$

Two definitions which take values in the unit range $[0, 1]$ and which satisfy the above conditions are Zadeh's original definition [124, 125] using minimum and maximum operators:

$$\begin{aligned} x \wedge y &= \min(x, y) \\ x \vee y &= \max(x, y) \\ \neg x &= 1 - x \end{aligned}$$

and an alternative using sum and product definitions:

$$\begin{aligned} x \wedge y &= xy \\ x \vee y &= x + y - xy \\ \neg x &= 1 - x \end{aligned}$$

Note that $p \wedge \neg p = 0 \iff p \in \{0, 1\}$ in both these and most other definitions of fuzzy logic. For instance, $0.3 \wedge \neg 0.3 = 0.3 \wedge 0.7 = 0.3$ using Zadeh's definition, and 0.21 if we use the product definition of $\wedge$.

This is only an elementary introduction to fuzzy logic, and we have not mentioned more esoteric connectives such as *averaging operators*. For more information we refer the reader to [56], [126] and [27]. From now on we shall assume that all fuzzy truth values lie in $[0, 1]$.

We shall now set about implementing these ideas in Haskell. We shall place all our definitions in a module called `Fuzzy` which will redefine some of the functions defined in the Haskell prelude. This is done by *shadowing* the previous definitions (see Section 5.3.2 of the Haskell report [85]). Thus the `Fuzzy` module and any module which wishes to import it should contain the declaration:

```
import Prelude hiding ((&&), (||), not, and, or, any, all)
```

This forces an explicit import of the prelude (which is normally implicitly imported), but hides the functions which we want to redefine. An example of the importing procedure can be seen Section 3.2.5.

Fuzzy truth values are represented using the Haskell type `Double`. The connectives are implemented by overloading the operators `&&`, `||`, etc. so that they work on fuzzy values as well as boolean ones. This is done by shadowing the connectives (see above) and placing the connectives in a class (see Section 2.4):

```
class Logic a where
    true, false :: a
    (&&), (||)  :: a -> a -> a
    not         :: a -> a
```

The functions `and`, `or`, `any` and `all` are then also overloaded so that they now operate on instances of the `Logic` class, rather than just the `Bool` type as before:

```
and, or :: Logic a => [a] -> a
and      = foldr (&&) true
or       = foldr (||) false

any, all :: Logic b => (a -> b) -> [a] -> b
any p     = or . map p
all p     = and . map p
```

We can then declare instances of this class. `Bool` is declared in the obvious way, with `true` = `True`, `false` = `False`, etc. For fuzzy truth values (values of type `Double`) we have:

```
instance Logic Double where
    true        = 1
    false       = 0
    (&&)        = min
    (||)        = max
    not x       = 1 - x
```

Note that as with the `Bool` case, `true` is the identity of `&&` and `false` is the identity of `||` (provided we stick with values in $[0, 1]$, of course). So, for example, $0.5 \wedge (0.3 \vee \neg 0.8)$ can be evaluated in Hugs as:

```
Fuzzy> 0.5 && (0.3 || not 0.8) :: Double
0.3
```

where '`Fuzzy>`' is the Hugs prompt. The explicit typing is necessary to resolve the overloading.

3.2 Fuzzy Subsets

Given a set A and a subset of it, B say, we can define a characteristic (membership) function $\mu_B : A \rightarrow \{0, 1\}$ defined such that:

$$\begin{aligned} \mu_B(x) &= 1, \ \textbf{if } x \in B \\ &= 0, \ \textbf{otherwise} \end{aligned}$$

This characteristic function determines which elements of A are in B and which are not. Now suppose we replace the two-valued range of μ_B with the unit interval, just as we replaced the boolean truth set with this interval. Then membership of the subset B of A is no longer an absolute but rather something which takes varying degrees of truthness. For $x \in A$, the closer $\mu_B(x)$ is to 1, the more we can regard x as belonging to B, with $\mu_B(x) = 1$ holding if x definitely is in B. Conversely, the closer $\mu_B(x)$ is to 0, the more we can regard x as *not* belonging to B. The subset B is no longer a crisp set but a *fuzzy* one.

A fuzzy subset B of a set A is a set of pairs with each element of A associated with the degree to which it belongs to B (determined by μ_B). Formally, $B \subset A \times [0, 1]$ where $B = \{\langle x, \mu_B(x) \rangle \mid x \in A\}$

Given the set-theoretic definition of a function, that is a set of domain-range pairs, we note that *the definition of B and its characteristic function are synonymous.* This is the key fact that motivates our use of a functional language as an implementation language — by representing a fuzzy subset by its membership function, a functional language allows us to manipulate such sets/functions with ease. We shall thus use the notion of a fuzzy subset and that of a (fuzzy) characteristic function interchangeably. In particular, if we have a fuzzy subset F of a set X then we shall denote X as the *domain* of F.

To give a concrete example, consider the problem of determining whether a company is profitable based, say, on the profit expressed as a percentage of total costs.

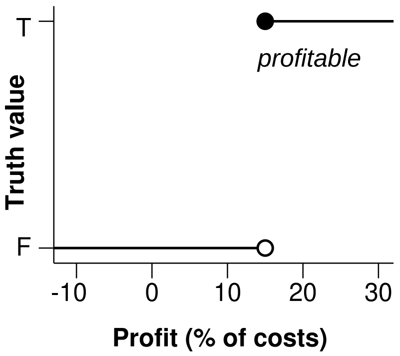


Figure 3.1: Crisp definition of Profit.

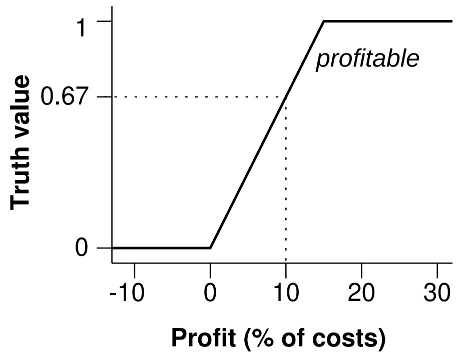


Figure 3.2: Fuzzy definition of Profit.

Using normal set theory, given a set of percentages, P, we would have to determine an arbitrary cut-off point at and above which we would consider profitable, 15% say (see Figure 3.1). So we can define $\mathit{profitable} \subseteq P$ as:

$$\mathit{profitable} = \{p \mid p \in P \wedge p \geq 15\}$$

This means however that a profit of 14.9% is *not* considered profitable, which is somewhat counter-intuitive considering its proximity to the cut-off point.

Contrast this with a fuzzy definition of *profitable* (see Figure 3.2). As before, profits above 15% are considered definitely profitable and those below 0% definitely *not* profitable; however between these two figures the degree of profitability increases linearly. For example, a profit of 10% can be regarded as profitable to a degree of 0.67 (that is, $\mu_{\mathit{profitable}} = 0.67$) and a profit of 14.9% is profitable to a degree of 0.993, in other words, almost definitely profitable.

As functions and fuzzy subsets are synonymous, we represent a fuzzy subset in Haskell as a function from some domain to the fuzzy truth value set. We define the following type synonym:

```
type Fuzzy a = a -> Double
```

A number of functions representing the shapes of common fuzzy subsets are provided (see Figure 3.3). For instance, `up` has the following definition:

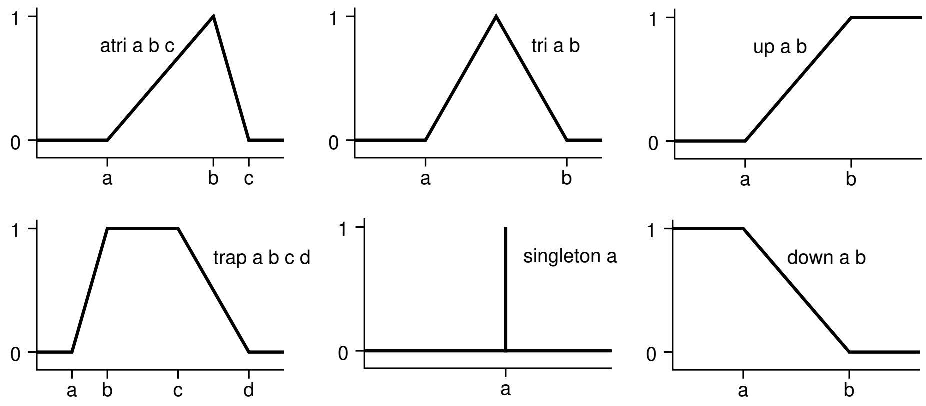


Figure 3.3: Standard fuzzy subset distributions

```
up :: Double -> Double -> Fuzzy Double
up a b x
    | x < a     = 0.0
    | x < b     = (x - a) / (b - a)
    | otherwise = 1.0
```

The other subsets in Figure 3.3 can be defined similarly. We can now define the fuzzy subset *profitable* as follows:

```
type Percentage = Double


profitable :: Fuzzy Percentage
profitable = up 0 15
```

Membership testing is then merely function application. For example:

```
Profit> profitable 10
0.666667
```

3.2.1 The Domain, Support and Fuzziness of a Fuzzy Subset

Knowing the domain of a fuzzy subset is necessary when defuzzifying it (see Section 3.2.4) and for evaluating its fuzziness (see below). We can also define fuzzy numbers in terms of their fuzziness (see Section 3.2.3) for which again we need to know the domain over which we are approximating.

Both discrete and continuous domains are represented using ordered lists (in the latter case we only have an approximation). We introduce the type synonym:

```
type Domain a = [a]
```

As the list is ordered, the upper and lower bounds of the domain are just the last and first elements of the domain list respectively:

```
lb, ub :: Domain a -> a

lb = head

ub = last
```

The 'dot-dot' method of defining lists can be used to define domains in a compact and easily-understandable way. So, for example, we can represent the domain of *profitable*, which is the range $[-10, 30]$ as the list `[-10..30]`.

The *support*, which we shall denote as $\sigma(B)$ (also written as $\text{supp}(B)$), of a fuzzy subset B is the set of those members of its domain, A say, which are in the fuzzy subset with non-zero truth value:

$$\sigma(B) = \{\mu_B(x) \neq 0 \mid x \in A\}$$

For example, if we take the domain of *profitable* as $[-10, 30]$ then its support is $(0, 30] = \{x \mid 0 < x \leq 30\}$. This has a simple translation into Haskell:

```
supp :: Domain a -> Fuzzy a -> [a]

supp dom f = filter (\x -> f x > 0) dom
```

For example, we can evaluate the support of `profitable` (defined above):

```
Profit> supp [-10..30] profitable
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0,
 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0,
 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0]
```

The *fuzziness*, ν, of a fuzzy subset is the degree to which the values of its membership function cluster around 0.5. It is defined in terms of the function δ measures the distance of a truth value to the nearest extreme, 0 or 1:

$$\begin{aligned}\delta(x) &= x, && \textbf{if } x < 0.5 \\ &= 1 - x, && \textbf{otherwise}\end{aligned}$$

For example $\delta(0.3) = 0.3$, $\delta(0.8) = 0.2$ and $\delta(0) = \delta(1) = 0$. If the domain of our fuzzy subset B is a continuous range, $[a, b]$ say, then we can define ν as:

$$\nu(B) = \frac{2}{b-a} \int_a^b \delta(\mu_B(x))\ dx$$

If the domain is a discrete set of points, $x_1, \ldots, x_n$ say, then the integral becomes a summation:

$$\nu(B) = \frac{2}{n} \sum_{i=1}^{n} \delta(\mu_B(x_i))$$

For example, the fuzziness of *profitable* (again over $[-10, 30]$) is 0.1875. Note that for any crisp set, A, in which the membership function returns only the values 0 or 1, $\nu(A) = 0$ as $\forall x \in A\ .\ \delta(\mu_A(x)) = 0$. The maximum possible fuzziness of a fuzzy subset is 1, which occurs when the membership function of a fuzzy subset always returns 0.5.

Translating the above into Haskell yields the following function:

```
fuzziness :: Domain a -> Fuzzy a -> Double
fuzziness dom f = (2.0 / size_dom) * sum (map (delta . f) dom)
```

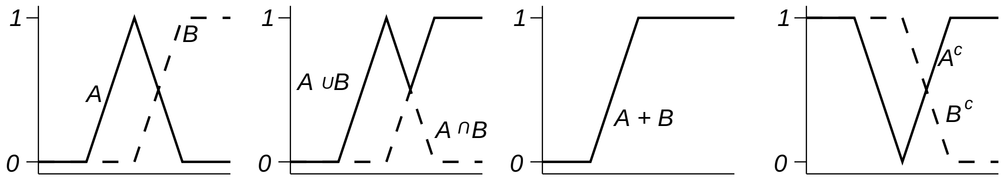


Figure 3.4: Operations on fuzzy subsets.

```
    where
        size_dom = fromInt (length dom)
        delta x
            | x < 0.5    = x
            | otherwise = 1 - x
```

For example, we can calculate the fuzziness of `profitable`, *viz*:

```
Profit> fuzziness [-10..30] profitable
0.182114
```

The value that Haskell returns is only an approximation, of course. A better approximation can be obtained by using a domain with more elements, for example:

```
Profit> fuzziness [-10,-9.75..30] profitable
0.186335
```

3.2.2 Fuzzy Subset Operations

Standard set operations — such as union, intersection and complement — can be used with fuzzy subsets. For fuzzy subsets, A, B of a set X, we have:

$$
\begin{aligned}
A \cup B &= \{\langle x, \mu_A(x) \vee \mu_B(x) \rangle \mid x \in X\} \\
A \cap B &= \{\langle x, \mu_A(x) \wedge \mu_B(x) \rangle \mid x \in X\} \\
A^c &= \{\langle x, \neg\mu_A(x) \rangle \mid x \in X\}
\end{aligned}
$$

This can be seen graphically in Figure 3.4, where the logical connectives are defined using Zadeh's method. A slightly unorthodox operation is addition defined as:

$$A + B = \{\langle x, \mu_A(x) + \mu_B(x)\rangle \mid x \in X\}$$

This leads to fuzzy subsets whose membership functions return values outside the range $[0, 1]$. This operation is generally only used in fuzzy systems (see below) where the resultant set is only used as an intermediate value and will be defuzzified (see Section 3.2.4) to yield a typical value.

If fuzzy subsets are Haskell functions, then the fuzzy subset operators are higher-order functions. If we look at the definition of intersection, for example, we see that we can regard it as a way of defining logical conjunction over sets. This concept holds for both fuzzy *and* crisp sets. Taking this to its logical conclusion we have:

```
instance (Logic b) => Logic (a -> b) where
    true        = \x -> true        -- everything
    false       = \x -> false       -- empty
    f && g      = \x -> f x && g x -- intersection
    f || g      = \x -> f x || g x -- union
    not f       = \x -> not (f x)  -- complement
```

This instance represents a generalised set, where `true` represents the set that everything is a member of and `false` is the empty set. If `true` is an identity for the `&&` over the type `b` then `true` it also an identity for `&&` over the type `a -> b`, and similarly for `false` and `||`.

In the context of fuzzy subsets, that is the type `Fuzzy a` (which in turn is the type `a -> Double`), true is the fuzzy subset, T say, with membership function $\mu_T(x) = 1$ and false is the fuzzy subset, F say, with membership function $\mu_F(x) = 0$. The function `true` remains the identity of `&&` and `false` the identity of `||`. We also need to be able to perform addition on fuzzy subsets. This is done by making the type `a -> b`, which remember is a generalisation of the type `Fuzzy a` a member of the `Num` class (which is used to overload the numeric operators `+`, `-`, etc.):

```
instance (Num b) => Num (a -> b) where
    f + g           = \x -> f x + g x
    f * g           = \x -> f x * g x
    abs f           = \x -> abs (f x)
    signum f        = \x -> signum (f x)
    negate f        = \x -> negate (f x)
    fromInteger i   = \x -> fromInteger i
```

We will also find it useful to use the operators of the `Logic` class over tuples, for instance in the shower controller described in Section 3.4.1 which groups its output variables in tuples. This is done pointwise, for example, for pairs we have:

```
instance (Logic a, Logic b) => Logic (a, b) where
    true                = (true, true)
    false               = (false, false)
    (a, b) && (a', b')  = (a && a', b && b')
    (a, b) || (a', b')  = (a || a', b || b')
    not (a, b)          = (not a, not b)
```

We also declare tuples to be instances of the `Num` class in a similar manner.

3.2.3 Hedges and Fuzzy Numbers

Just as adjectives such as profitable can be qualified by terms such as *very* and *somewhat*, so can fuzzy subsets. Terms such as these, known as *hedges*, alter the membership function by intensifying it (normally by raising it to a power greater than 1) in the case of *very* and similar terms such as *extremely*, or diluting it (normally by raising it to a power between 0 and 1) in the case of *somewhat*. Usually we have:

$$\begin{aligned} \mu_{very\ F}(x) &= \mu_F(x)^2 \\ \mu_{somewhat\ F}(x) &= \mu_F(x)^{1/2} \end{aligned}$$

The effect of *very* and *somewhat* on *profitable* can be seen in Figure 3.5. We see that a profit of 10% is *profitable* with truth value 0.67, *very profitable* by truth value 0.44, and *somewhat profitable* by degree 0.82.

In Haskell, we represent hedges as higher-order functions. We first define a generic hedge which will raise the value of a function to a specified power:

```
hedge :: Double -> Fuzzy a -> Fuzzy a
hedge p f x = if fx == 0 then 0 else fx ** p
    where fx = f x
```

Note that Hugs defines its power operator `**` in terms of logarithms and hence we need to check if `f x` is zero before attempting to raise it to the given power, otherwise we will attempt to take the logarithm of zero. We can now define more specific hedges as follows:

```
very, extremely, somewhat, slightly :: Fuzzy a -> Fuzzy a
```

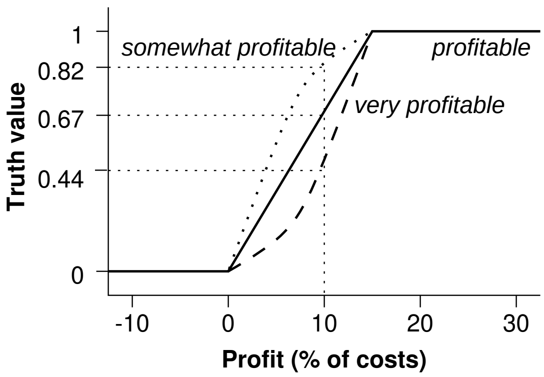


Figure 3.5: *Very profitable* and *Somewhat profitable.*

```
very        = hedge 2

extremely   = hedge 3

somewhat    = hedge 0.5

slightly    = hedge (1 / 3)
```

The user is free to redefine these functions with different numbers if they want, of course. An example of these in use, using the same sets and definitions in Figure 3.5:

```
Profit> very profitable 10

0.444444

Profit> somewhat profitable 10

0.816497
```

Hedges can also be used to approximate numbers by converting them into fuzzy subsets (also known as *fuzzy numbers* in this context) using such terms as *around* 20, *roughly* 20 and *nearly* 20. One typical way of defining these subsets is by symmetrical triangular fuzzy subsets, centred on the number, c say, that we are approximating and with base of width $2w$. The membership function of this set is thus:

$$\begin{aligned} \mu(x) &= 1 - \tfrac{|x-c|}{w}, \quad && \textbf{if } c - w \leq x \leq c + w \\ &= 0, && \textbf{otherwise} \end{aligned}$$

The tighter the approximation we want, the less fuzzy the fuzzy subset is, and hence the smaller the base of the triangular fuzzy subset is. In general, *roughly* is a looser approximation than *around* which in turn is looser than *nearly*.

For example, consider the fuzzy numbers in Figure 3.6, which approximate 20 over the domain $[0, 40]$ using triangular fuzzy subsets centred on 20. Here we see that *nearly 20* has a base of length 5 and a fuzziness of 0.125; *around 20* has a base of length 10 and a fuzziness of 0.25; and *roughly 20* has a base of length 15 and a

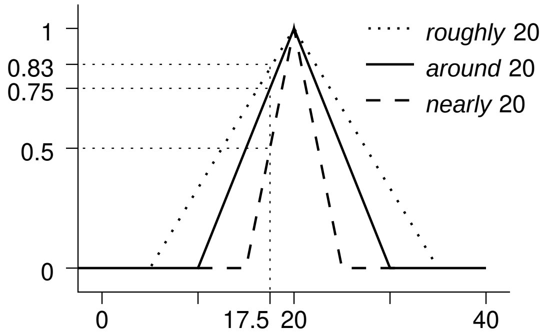


Figure 3.6: Fuzzy approximations to 20.

fuzziness of 0.375. So, for example, 17.5 is *nearly* 20 with truth value 0.5, *around* 20 with truth value 0.75 and *roughly* 20 with truth value 0.83.

As with hedges, to implement fuzzy numbers in Haskell we define a generic fuzzy number function, which approximates a number on a specific domain by a triangular fuzzy subset (see Figure 3.3) of specified fuzziness:

```
approximate :: Double -> Double -> Domain Double -> Fuzzy Double
approximate fuzziness n dom = tri (n - hw) (n + hw)
    where hw = fuzziness * (ub dom - lb dom)
```

We now define the fuzzy number generators `near`, `around` and `roughly` as:

```
near, around, roughly :: Double -> Domain Double -> Fuzzy Double
near    = approximate 0.125
around  = approximate 0.25
roughly = approximate 0.375
```

This leads to the same sets as in Figure 3.6 if we approximate 20 over the domain $[0, 40]$. For example:

```
Profit> near 20 [0..40] 17.5
0.5
Profit> roughly 20 [0..40] 17.5
```

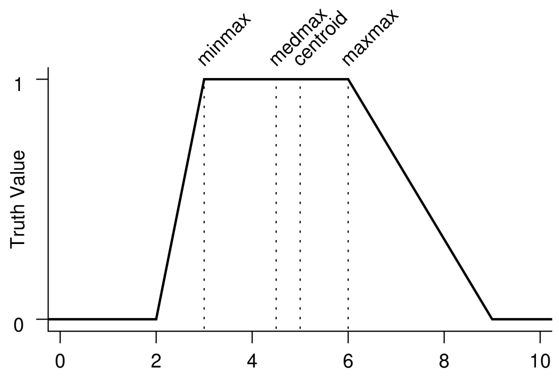


Figure 3.7: Defuzzifying a fuzzy subset

```
0.833333
Profit> around 20 [0..40] 17.5
0.75
```

3.2.4 Defuzzification

In a real-world situation, we often need a concrete value rather than a fuzzy subset. The process of extracting a typical value from a fuzzy subset is known as *defuzzification* and there are many methods for doing this. Two such methods are finding the *centroid* (or centre of gravity) of a fuzzy subset, or finding the *maxima* of a fuzzy subset and returning a member of this set.

If we have a fuzzy subset A with membership function μ_A over a domain X then the centroid of A is defined as:

$$\frac{\int_X x\mu_A(x)\ dx}{\int_X \mu_A(x)\ dx}$$

if X is a continuous domain. If X is discrete then the centroid is defined as:

$$\frac{\sum_X x\mu_A(x)}{\sum_X \mu_A(x)}$$

The latter is the definition we use in our implementation. We define the `centroid` function as:

```
centroid :: Domain Double -> Fuzzy Double -> Double
centroid dom f = (sum (zipWith (*) dom fdom)) / (sum fdom)
    where fdom = map f dom
```

For example, the centroid of the trapezoid fuzzy subset in Figure 3.7 can be evaluated *viz*

```
Profit> centroid [0, 0.5 .. 10] (trap 2 3 6 9)
5.06667
```

The maxima of a fuzzy subset A over a domain X are those elements $m \in X$ such that $\forall x \in X \,.\, \mu_A(m) > \mu_A(x)$. This can be implemented using the `compare` function from the Haskell prelude:

```
maxima :: Domain a -> Fuzzy a -> [a]
maxima dom f = foldl m [] dom
    where
    m [] x = [x]
    m ys x = case compare (f x) (f (head ys)) of
                 GT -> [x]
                 EQ -> x:ys
                 LT -> ys
```

We then typically defuzzify A by returning the minimum, the median or the maximum of the maxima of A:

```
minmax, medmax, maxmax :: Ord a => Domain a -> Fuzzy a -> a
minmax dom f = minimum (maxima dom f)
maxmax dom f = maximum (maxima dom f)
medmax dom f = median (maxima dom f)
    where
```

```
-- N.B. Domains are represented by *ordered* lists
median ms = head (drop (length ms `div` 2) ms)
```

Defuzzifying the fuzzy subset in Figure 3.7 using these three methods we get:

```
Profit> minmax [0, 0.5 .. 10] (trap 2 3 6 9)
3.0
Profit> medmax [0, 0.5 .. 10] (trap 2 3 6 9)
4.5
Profit> maxmax [0, 0.5 .. 10] (trap 2 3 6 9)
6.0
```

3.2.5 An Example — Fuzzy Database Queries

The linguistic nature of fuzzy subsets make them ideal in database enquiries. In a functional language this is akin to applying a filter to a list of information. We define a variant of the standard filter function, which takes a *fuzzy* predicate (that is, a function which returns a fuzzy truth value) and returns those members of the list that satisfy the predicate to a non-zero degree, along with the degree to which they satisfy the predicate:

```
ffilter :: Fuzzy a -> [a] -> [(a, Double)]
ffilter p xs = filter ((/=) 0 . snd) (map (\x -> (x, p x)) xs)
```

Referring back to our profit example, based originally on an example in [82], suppose we have the following module:

```
module Profit where

import Prelude hiding ((&&), (||), not, and, or, any, all)
import Fuzzy
```

```haskell
type Percentage = Double
type Sales      = Double -- thousands of pounds
type Company    = (String, Sales, Percentage)

sales :: Company -> Sales
sales (_, s, _)  = s

profit :: Company -> Percentage
profit (_, _, p) = p

percentages :: [Percentage]
percentages = [-10..30]

profitable :: Fuzzy Percentage
profitable = up 0 15

high :: Fuzzy Sales
high = up 600 1150

companies :: [Company]
companies = [("A", 500, 7),   ("B", 600, -9),  ("C", 800, 17),
             ("D", 850, 12),  ("E", 900, -11), ("F", 1000, 15),
             ("G", 1100, 14), ("H", 1200, 1),  ("I", 1300, -2),
             ("J", 1400, -6), ("K", 1500, 12)]
```

So, we have a list of companies, functions to extract their profit and sales, and fuzzy subsets `profitable` of `Percentage` (using the same definition as before) and `high` of `Sales`. To extract all the profitable companies from `companies`, we first define the fuzzy predicate `p1`:

```
p1 co = profitable (profit co)
```

and `ffilter` it over `companies`, *viz*:

```
Profit> ffilter p1 companies
[(("A",500.0,7.0),  0.466667), (("C",800.0,17.0),1.0),
 (("D",850.0,12.0), 0.8),      (("F",1000.0,15.0),1.0),
 (("G",1100.0,14.0),0.933333), (("H",1200.0,1.0),0.0666667),
 (("K",1500.0,12.0),0.8)]
```

So, of the original 11 companies, 7 are considered profitable with `C` and `F` being the most profitable. Profitability by itself might not be enough — we may also want high sales. Defining:

```
p2 co = profitable (profit co) && high (sales co)
```

we can then find all profitable companies with high sales:

```
Profit> ffilter p2 companies
[(("C",800.0,17.0),0.363636),  (("D",850.0,12.0),0.454545),
 (("F",1000.0,15.0),0.727273), (("G",1100.0,14.0),0.909091),
 (("H",1200.0,1.0),0.0666667), (("K",1500.0,12.0),0.8)]
```

Six companies satisfy the predicate, with `G` satisfying it the most. We can use hedges to tighten or loosen the conditions, for example, defining

```
p3 co = somewhat profitable (profit co) && very high (sales co)
```

we can find those companies which have very high sales and are somewhat profitable:

```
Profit> ffilter p3 companies
[(("C",800.0,17.0),0.132231),  (("D",850.0,12.0),0.206612),
 (("F",1000.0,15.0),0.528926), (("G",1100.0,14.0),0.826446),
 (("H",1200.0,1.0),0.258199),  (("K",1500.0,12.0),0.894427)]
```

Here the increased emphasis on sales, and decreased emphasis on profitability means that company `K` now satisfies the predicate we pass to `ffilter` to the highest degree.

3.3 Fuzzy Systems

Expert Systems [98] are used to model real-world situations in many areas of expertise. One common way of implementing these systems is as a set of *rules* and an *inference engine* which manages these rules. Rules are composed of two parts: an *antecedent*, which is a logical expression; and a *consequent* which is an action which is performed when the antecedent is true. When this happens we say that the rule *fires*.

As a simple example, consider predicting the shoe size, using British shoe sizes, of a man given his height in metres. In a standard expert system we might have rules like:

```
if 1.65 <= height & height <= 1.72 then shoe_size := 9
```

These rules are absolutes — if and only if the antecedent holds will the action be fired and fired completely.

In a rule-based fuzzy system, the antecedent is a fuzzy logic expression the value of which dictates the degree to which the action fires, the action being the assignment of a variable to a fuzzy subset. If we have a rule such as `if` p `then` a `:=` F then a is assigned to the fuzzy subset F' where F' is linearly weighted by the value of p and has

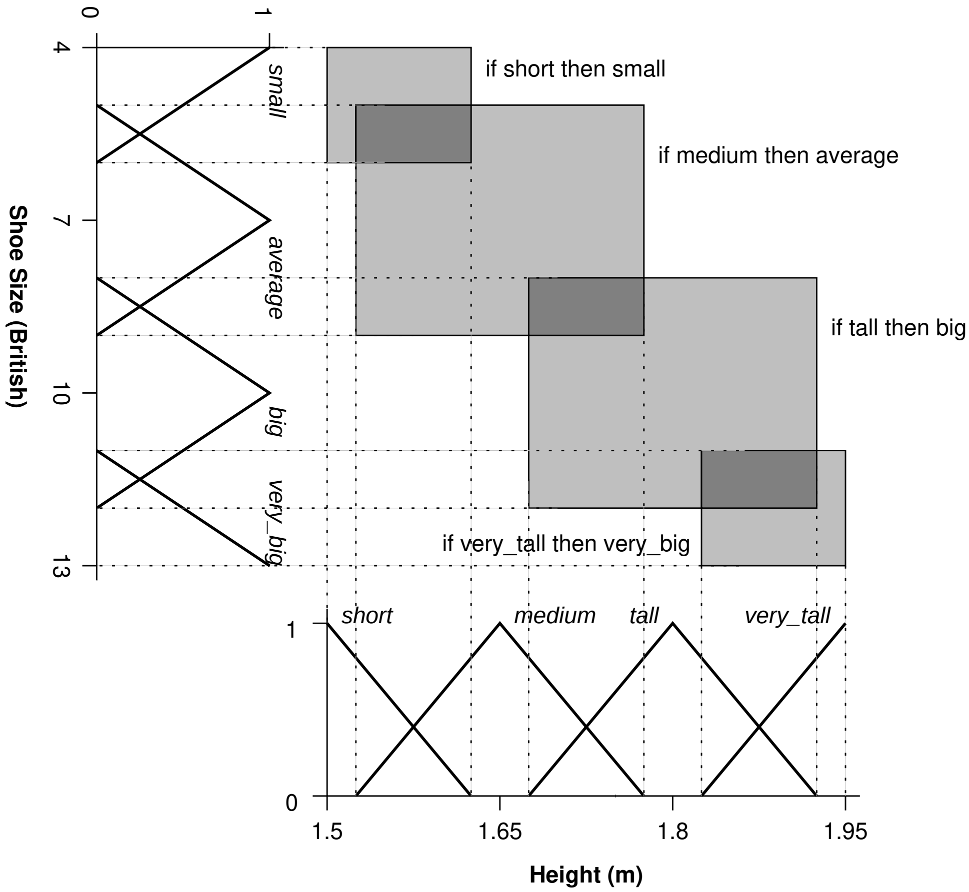


Figure 3.8: The fuzzy rule base for the height → shoe size expert system

membership function $\mu_{F'}(x) = p\mu_F(x)$. This can be extended to multiple variable assignments. Note that if the value of the antecedent is 0, then the membership function of the consequent fuzzy subset will be constantly 0 (the empty set) and we regard the rule as not having been fired. In our shoe size example, the rules are:

```
if height is short then shoe_size := small
if height is medium then shoe_size := average
if height is tall then shoe_size := tall
if height is very_tall then shoe_size := very_big
```

Here `is` serves as a membership test for `height`. These rules can be thought of as forming *patches* (see Figure 3.8) with the larger the patch the fuzzier the rule [58]. More input variables require more dimensions to the patches.

As can be seen, these patches overlap, which in practical terms means that more than one rule can fire, that is, we have more than one possible assignment to `shoe_size`. Rather than selecting one of the possible assignments to a we select them *all*, combining the subsets into one set using an operation such as union or addition. Addition has the property that, unlike union, when combining many sets the membership function of the result does not approach the constant function 1. Also all the sets that are part of the addition contribute to the final result, whereas in the case of union, large sets (measured by both their support and their height (truth values)) subsume smaller ones.

Once we have combined all the resultant sets, we then defuzzify them (see Section 3.2.4) to obtain a final result. For instance, if we have a height of 1.75m then this is tall to degree 0.6 and medium to degree 0.2. If we weight the relevant consequents, sum the sets and defuzzify using the centroid method we obtain an estimated shoe size of $9\frac{1}{4}$, while defuzzifying with any of the maxima methods yields a shoe size of 10, since 10 is the only element of the resultant fuzzy subset which yields the

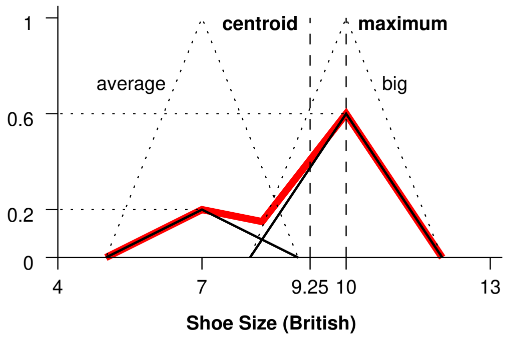


Figure 3.9: Weighting, adding and defuzzifying the rules for a height of 1.75m

largest truth value, in this case 0.6 (see Figure 3.9). Of course, this is a very simple example. More complex ones can be found in Section 3.4.

We introduce a new operator `==>`, which has the leastmost binding, to the `Logic` class:

```
infix 0 ==>


class Logic a where
    (==>) :: Double -> a -> a          -- other defs as before
```

This operator linearly weights its right-hand side by the value on its left-hand side. On fuzzy values, it is simply multiplication:

```
instance Logic Double where
    w ==> x     = if w == 0 then 0 else w * x
```

The conditional is there for efficiency purposes (we can use it to avoid evaluating `x`, see below). There are a number of definitions over `Bool`. One such definition is:

```
instance Logic Bool where
    w ==> False = False
    w ==> True  = w > 0.5              -- other defs as before
```

The `==>` function used over fuzzy truth values is useful in its own right as a fuzzy if-then function; an example of its use can be seen in Section 3.4.2. However its major use is to represent a rule in a fuzzy rulebase, where we normally expect the value on the RHS of the operator to be a fuzzy subset or a tuple of such sets. On fuzzy subsets, this operator has the definition:

```
instance (Logic b) => Logic (a -> b) where
    w ==> f = \x -> w ==> f x          -- other defs as before
```

Note that by the definition of `==>` on fuzzy values, and the fact that Haskell is a lazy language, if `w` is zero then we know the result is zero without having to evaluate `f x`. This can have a significant effect on run-time performance (roughly 20–25% fewer reductions in the case of our shower example below) if `f` is a complicated expression.

On tuples we weight each element of the tuple individually. For pairs we have:

```
instance (Logic b) => Logic (a -> b) where
    w ==> (a, b) = (w ==> a, w ==> b) -- other defs as before
```

The LHS of the `==>` is thus the antecedent of the rule and the RHS of the rule is the consequent. The result of the function is the consequent linearly weighted by the antecedent, which will usually be the result of evaluating fuzzy logic expression.

To combine the weighted subsets we define a function which takes a list of subsets and a function to combine (two of) them with, and returns the result of combining all the weighted subsets. We thus just have:

```
rulebase :: Logic a => (a -> a -> a) -> [a] -> a
rulebase = foldr1
```

Note that we can not apply `rulebase` to the empty list, but this would imply we had an empty set of rules. The resultant set can then be defuzzified using one the defuzzifying functions from Section 3.2.4.

Putting this all together, we have the following Haskell module which implements our shoe-size expert system from above:

```
module Shoe where

import Prelude hiding ((&&), (||), not, and, or, any, all)
import Fuzzy

type Height     = Double -- Metres
type ShoeSize   = Double -- British size

sizes :: Domain ShoeSize
sizes = [4, 4.5..13]

short, medium, tall, very_tall :: Fuzzy Height
short       = down 1.5   1.625
medium      = tri  1.525 1.775
tall        = tri  1.675 1.925
very_tall   = up   1.825 1.95

small, average, big, very_big :: Fuzzy ShoeSize
small       = down 4  6
average     = tri  5  9
big         = tri  8 12
very_big    = up  11 13

-- calculate the shoe size from a given height
```

```
shoe_size :: Height -> ShoeSize
shoe_size h = centroid sizes (
    rulebase (+) [
        short h     ==> small,
        medium h    ==> average,
        tall h      ==> big,
        very_tall h ==> very_big])
```

Consider the use of the `rulebase` function inside the `shoe_size` function. Its first argument is `+`, so we are using fuzzy subset addition to combine the weighted subsets. Its second argument is the set of rules, written using the `==>` operator. During evaluation of the `rulebase` function, each of these rules will be evaluated, giving the required weighted set, which will all then be combined, in this case using `+`. This set is then defuzzified using the `centroid` function over the domain `sizes`.

3.4 Further Examples

We now give two further examples of problems solved using fuzzy logic: another fuzzy system (more complex than our shoe size example), and a decision-making exercise.

3.4.1 Controlling a Shower

Consider the problem of controlling a shower [83]. We wish to get the temperature to between 34°C and 38°C and the flow of the water between 11 l/min and 13 l/min. To do this we have two taps, one hot and one cold, which take values between 0 (fully off) and 1 (fully on). We divide the temperature into the fuzzy subsets `hot`, `ok` and `cold`; the flow into the fuzzy subsets `weak`, `right` and `strong`; and the possible tap changes (ranging from -0.2 to 0.2) into seven fuzzy subsets: `pb` (big

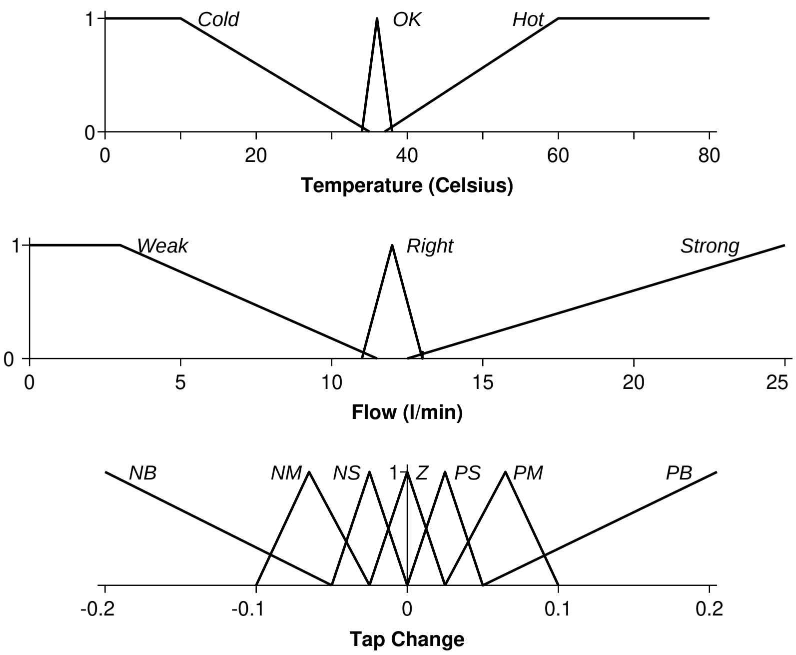


Figure 3.10: Fuzzy subsets of temperature, flow and tap change

positive change), `pm` (medium positive change), `ps` (small positive change), `z` (zero change), `ns` (small negative change), `nm` (medium negative change) and `nb` (big negative change). These fuzzy subsets can be seen in Figure 3.10.

Unlike our shoe size example, the shower is not meant to be a one-use function but rather to be continually iterated until the temperature and the flow are in the correct range. So we are continually making changes (with suitable gaps in between these changes to let the shower settle into its new settings) until the water becomes acceptable. We have the following system (note that these are not the original sets used in the Fuzzy CLIPS example, which used curved rather than polygonal fuzzy sets, and hence we have adjusted the numbers to get a better performance):

```
module Shower where


import Prelude hiding ((&&), (||), not, and, or, any, all)
import Fuzzy
```

```
type Temp = Double
type Flow = Double
type Change = Double

cold, ok, hot :: Fuzzy Temp
cold    = down 15 36
ok      = tri  32 40
hot     = up   36 75

weak, right, strong :: Fuzzy Flow
weak    = down  0 12
right   = tri   9 15
strong  = up   12 25

nb, nm, ns, z, ps, pm, pb :: Fuzzy Change
nb  = down (-0.2)  (-0.05)
nm  = tri  (-0.1)  (-0.025)
ns  = tri  (-0.05)   0.0
z   = tri  (-0.025)  0.025
ps  = tri    0.0     0.05
pm  = tri    0.025   0.1
pb  = up     0.05    0.2

change_valves :: (Temp, Flow) -> (Change, Change)
change_valves (temp, flow) = (defuz hv, defuz cv)
    where
```

```
    defuz = centroid [-0.2, -0.195..0.2]

    (hv, cv) = rulebase (+) [
                    cold temp && weak flow      ==> (pm, z),
                    cold temp && right flow     ==> (pm, z),
                    cold temp && strong flow    ==> (z, nb),
                    ok temp && weak flow        ==> (ps, ps),
                    ok temp && strong flow      ==> (ns, ns),
                    hot temp && weak flow       ==> (z, pb),
                    hot temp && right flow      ==> (nm, z),
                    hot temp && strong flow     ==> (nb, z)]
```

Normally such a system would be used in a real-word environment. However, for testing purposes, we provide a simple simulation in Haskell. The simulation creates a new shower, setting the temperatures of the hot and cold taps to a random value in the ranges $[67°\mathrm{C}, 77°\mathrm{C}]$ and $[10°\mathrm{C}, 20°\mathrm{C}]$ respectively and setting the flows of both taps are set to a random value in the range $[10\ \mathrm{l/min}, 14\ \mathrm{l/min}]$ using the `Random` package from the Haskell 1.4 standard library. Note the use of the IO Monad [114, 115] to allow printing as a side-effect and the generation of random numbers using a system-dependent seed number. The total temperature and flow is then calculated, assuming perfectly cylindrical pipes, and the changes to the valves required calculated using the fuzzy rulebase above. These changes are then made, the new temperature and flow calculated and further changes made if necessary:

```
type Valve = Double -- should be in the range [0..1]


-- Each tap contains the absolute temperature and flow of the water
-- it controls plus the tap setting
data Shower = Shower {hot_valve, cold_valve :: Valve,
```

```
                       hot_temp, cold_temp   :: Temp,
                       hot_flow, cold_flow   :: Flow}
             deriving Show

-- Creates a new shower using the given settings of the hot
-- and cold valves
mkShower :: Valve -> Valve -> IO Shower
mkShower hv cv =
    do rs <- randomIO (-100, 100)
       return (Shower {hot_valve  = hv,
                       cold_valve = cv,
                       hot_temp   = 72 + dht rs,
                       cold_temp  = 15 + dct rs,
                       hot_flow   = 12 + dhf rs,
                       cold_flow  = 12 + dcf rs})
       where
       dht rs = 5 * fromInteger (rs !! 0) / 100.0
       dct rs = 5 * fromInteger (rs !! 1) / 100.0
       dhf rs = 2 * fromInteger (rs !! 2) / 100.0
       dcf rs = 2 * fromInteger (rs !! 3) / 100.0

-- Calculates the temperature and flow of the given shower
getTempFlow :: Shower -> (Temp, Flow)
getTempFlow shower
    | flow > 0  = (temp, flow)
    | otherwise = error "Shower.getTempFlow: non-positive flow"
    where
```

```
    cf   = propOpen (cold_valve shower) * cold_flow shower
    hf   = propOpen (hot_valve shower) * hot_flow shower
    flow = hf + cf
    temp = (hf / flow) * hot_temp shower +
           (cf / flow) * cold_temp shower

    propOpen :: Valve -> Double
    propOpen x
        | x < 0         = 0
        | x > 1         = 1
        | x <= 0.5      = (two_theta - sin (two_theta)) / (2 * pi)
        | otherwise     = 1 - propOpen (1 - x)
        where two_theta = 2 * acos (1 - 2 * x)

-- applies the given changes to the shower
adjustValves :: (Change, Change) -> Shower -> IO Shower
adjustValves (dh, dc) shower =
    return (shower {cold_valve = restrict (cold_valve shower + dc),
                    hot_valve  = restrict (hot_valve shower + dh)})
    where
    restrict x
        | x < 0.0   = 0.0
        | x > 1.0   = 1.0
        | otherwise = x

-- repeatedly adjusts the given shower until its temp and flow are
-- satisfactory
```

```
adjustShower :: Shower -> IO Shower
adjustShower shower =
    do putStr ("Hot = " ++ show (hot_valve shower) ++
               ", Cold = " ++ show (cold_valve shower) ++
               ", Temp = " ++ show t ++
               ", Flow = " ++ show f ++
               "\nHot change = " ++ show dh ++
               ", Cold change = " ++ show dc ++ "\n\n")
       shower' <- adjustValves (dh, dc) shower
       if satisfactory then return shower' else adjustShower shower'
       where
       (dh, dc)     = change_valves (t, f)
       (t, f)       = getTempFlow shower
       satisfactory = 34.0 <= t && t <= 38.0 &&
                      11.0 <= f && f <= 13.0


shower :: Valve -> Valve -> IO ()
shower hv cv =
    do shower <- mkShower hv cv
       putStr ("Initial shower: " ++ show shower ++ "\n")
       final_shower <- adjustShower shower
       putStr ("Final shower: " ++ show final_shower ++ "\n")
```

So, for example, if we have an initial valve setting of 0.2 and 0.4 for the hot and cold valves respectively, then the adjustment sequence is:

```
Shower> shower 0.2 0.4
Initial shower: Shower{hot_valve=0.2, cold_valve=0.4,
```

```
                      hot_temp=72.8, cold_temp=19.65,
                      hot_flow=11.6, cold_flow=12.12}

Hot = 0.2, Cold = 0.4, Temp = 33.857, Flow = 6.17877
Hot change = 0.0342672, Cold change = 0.0204951

Hot = 0.234267, Cold = 0.420495, Temp = 35.5691, Flow = 6.90705
Hot change = 0.0275251, Cold change = 0.0238471

Hot = 0.261792, Cold = 0.444342, Temp = 36.5224, Flow = 7.62269
Hot change = 0.0241146, Cold change = 0.0368691

Hot = 0.285907, Cold = 0.481211, Temp = 36.7517, Flow = 8.50753
Hot change = 0.0234472, Cold change = 0.0448707

Hot = 0.309354, Cold = 0.526082, Temp = 36.7072, Flow = 9.51635
Hot change = 0.0138068, Cold change = 0.0467235

Hot = 0.323161, Cold = 0.572806, Temp = 36.1902, Flow = 10.4232
Hot change = 0.0197123, Cold change = 0.0357849

Hot = 0.342873, Cold = 0.60859, Temp = 36.285, Flow = 11.2405
Hot change = 0.0110444, Cold change = 0.0514842

Final shower: Shower{hot_valve=0.353918, cold_valve=0.660075,
                     hot_temp=72.8, cold_temp=19.65,
                     hot_flow=11.6, cold_flow=12.12}
```

3.4.2 Pricing Goods

The fact that fuzzy logic is inherently contradictory, that is we have truth values which are non-zero and whose negation is also non-zero, is useful in decision making processes where the decisions we have to make are based on conflicting demands or requirements. Fuzzy logic can be used to resolve these contradictions in a natural, simple and efficient way.

Consider the problem of pricing goods [21]. The price should be as high as possible to maximise takings but as low as possible to maximise sales. We also want to make a healthy profit, say a 100% mark-up on the cost price. Then we have to consider what the competition is charging. We can formalise these requirements as rules:

1. Our price must be high.
2. Our price must be low.
3. Our price must be around 2 × manufacturing costs, in other words, a 100% mark-up.
4. If the competition price is not very high then our price must be around the competition price (we do not want to indulge in a price war).

A boolean system may have difficulties trying to resolve the requirements that the price must be high *and* low, not to mention the other two requirements, but a fuzzy system has no such difficulties.

Suppose possible prices are in the range £15 to £35. We define fuzzy subsets `high` and `low` on this range, *viz*:

```
type Price = Double -- Pounds Sterling
```

```
prices :: Domain Price
prices = [15.00, 15.50 .. 35.00]


high, low :: Fuzzy Price
high = up 15.00 35.00
low  = not high
```

So if we want a price that is high and low (Rules 1 and 2) then we can calculate this by taking the intersection of `high` and `low` and defuzzifying the resultant set to get a typical value, *viz*:

```
our_price = centroid prices (high && low)
```

Evaluating `our_price` we get:

```
Prices> our_price
25.0
```

Rule 3 suggests that we can approximate the price by a fuzzy number centred on 2 × manufacturing costs. Taking the manufacturing costs as a parameter to `our_price` and combining this with what we have so far, we define

```
our_price' man_costs =
    centroid prices (high && low &&
                     around (2.0 * man_costs) prices)
```

Assuming manufacturing costs of £13.25, say, we have:

```
Prices> our_price' 13.25
26.252
```

Rule 4 is a conditional rule. The more that the competition price is not very high, the more it affects the calculation of our price. Using the `==>` operator and taking the competition price as another parameter, we get:

```
our_price'' man_costs comp_price =
    centroid prices (high && low &&
                    around (2.0 * man_costs) prices &&
                    ((not.very high) comp_price ==>
                         around comp_price prices))
```

Assuming the same manufacturing costs as before and a competition price of £29.99 we have:

```
Prices> our_price'' 13.25 29.99
28.5893
```

So our final retail price is £28.59.

3.5 Summary

We have introduced and explored the use of fuzzy logic in functional programming. The natural equivalence between fuzzy subsets and their membership functions motivates our idea to use a single function to model them both. We have shown how a functional language can be extended so that it provides facilities for the use of fuzzy logic and fuzzy subsets, achieved by overloading pre-existing operators and functions, and introducing new ones. We have also shown how fuzzy systems, used in a variety of control and decision making problems, can be implemented in a functional language in a natural and efficient way.

Chapter 4

Compiling Lazy Functional Programs to Java Byte-code

The Java Virtual Machine (JVM) [79] provides a machine-independent execution environment which executes Java *byte-code*, which is essentially a machine code for object-oriented programs. It was designed as the target of Java compilers, but there is no reason why compilers of other languages cannot produce code that will run on it. We are interested in using it to run functional programs, in particular pure lazy ones. This approach has several advantages:

- Java bytecode will run on any machine for which an interpreter is available.
- Java programs can be run as applets in web-browsers, or in embedded systems.
- Java has a built-in garbage-collector, hence any language which targets Java bytecode has no need to handle garbage collection itself.

Our aim is to create a compiler which will translate a functional program into byte-code, with each function being translated into a static method of the generated class file. If the functional program we are compiling is designed to be executed (as opposed to being a set of library functions, for example) then we also generate a `main` method which will, when the class file is executed, perform any initialisation necessary and evaluate the program which we have compiled. Our source language is Ginger [53], a simple, pure, lazy, weakly-typed functional language. We base our evaluation methods on those of the G-machine (see Section 2.5). We assume that the reader is familiar with programming in Java, and has some understanding of how Java classes are structured, and also how to program using a lazy functional language, but we assume no prior knowledge of the JVM (which we describe in Section 4.1) itself or of implementing functional languages. Reference is made back to Section 2.3 for an introduction to graph reduction.

4.1 The Java Virtual Machine

The JVM [79] provides an environment for executing object-oriented programs; that is, for creating objects, invoking their methods, manipulating their fields, as well as the usual basic operations, such as adding integers.

The format of the bytecode resembles that of Java programs [7]. For each new class, there is a header declaring the class, its superclass and its package and the declaration of the fields and methods. Each method provides a separate environment consisting of a stack, which is used as a working space, and a set of local variables. In the case of an instance method, register 0 holds a reference to the object that the method was invoked on (the `this` reference), and variables $1, \ldots, n$ hold the n parameters of the method. Static methods are slightly different in that since there is no object to invoke the method on, the n parameters of the method are stored in variables $0, \ldots, n-1$. Each class also has a *constant pool* where all the symbolic data used by the class — fields, methods, class, interfaces, etc. — is stored. For example, consider the following class definition:

```
public class ExampleClass {
  public int value;
  private static ExampleClass one = new ExampleClass(1);

  public ExampleClass(int v) {
    value = v;
  }

  public static ExampleClass getOne() {
    return one;
  }

  public void add(ExampleClass e) {
    value += e.value;
  }
}
```

This class can be compiled into a class file of bytecodes using a Java compiler, `javac` say. The class file can then be examined using a disassembler, such as `javap` [103]. If we examine the output of `javap`, first of all we have the header of the file which declares the class's super-class and its members:

```
Compiled from ExampleClass.java
public synchronized class ExampleClass extends java.lang.Object
{
    public int value;
    private static ExampleClass one;
    public ExampleClass(int);
```

```
    public static ExampleClass getOne();

    public void add(ExampleClass);

    static static {};

}
```

We then have the bytecode for the constructor function:

```
Method ExampleClass(int)

   0 aload_0

   1 invokespecial #3 <Method java.lang.Object()>

   4 aload_0

   5 iload_1

   6 putfield #6 <Field int value>

   9 return
```

A reference to the the object that is created by the constructor is held in register 0. This is placed on the stack by the `aload_0` instruction. The `invokespecial #3` pops the top object off the stack (`this`) and invokes on it the zero-argument constructor of its superclass which is `Object` (item number 3 in the constant pool). When this has completed, `this` is again loaded onto the stack and then the integer 1 is loaded onto the stack by the `iload_1` instruction. The `putfield #6` instruction pops the top two values of the stack and stores the value that was held at the top of the stack (the integer 1) in the `value` field (item number 6 in the constant pool) of the object that was the second topmost item in the stack (`this`). The work is now done, the stack is empty, and the constructor function returns to the method that called it, with a `void` result, using the `return` instruction.

We then have the two method declarations. The method `getOne` merely loads the static field `one` onto the stack and returns it using the `areturn` instruction:

```
Method ExampleClass getOne()
```

```
   0 getstatic #5 <Field ExampleClass one>
   3 areturn
```

The add method is a little more complicated:

```
Method void add(ExampleClass)
   0 aload_0
   1 dup
   2 getfield #6 <Field int value>
   5 aload_1
   6 getfield #6 <Field int value>
   9 iadd
  10 putfield #6 <Field int value>
  13 return
```

This first loads the this reference onto the stack and duplicates it using the dup instruction, leaving the copy on the top of the stack. The getfield #6 instruction pops the top of the stack and gets the value of its value field which it puts on top of the stack. We then load the argument of the method (named e in the original code) which is in register 1 and get the value of its value field. The top two items on the stack are now the two integers equal to the value fields of the object the method was invoked on and the argument of the method. These integers are popped off the stack and added together using the iadd instruction which places the result on the stack. This value is then stored in the value field of the object referenced by this — recall the dup instruction — and the method returns to its caller, returning a void value.

The final method is the class's static initialiser which is executed when the class is loaded. This has the job of creating a new ExampleClass whose value field is set to 1 and storing it in the static field one:

```
Method static {}
   0 new #1 <Class ExampleClass>
   3 dup
   4 iconst_1
   5 invokespecial #4 <Method ExampleClass(int)>
   8 putstatic #5 <Field ExampleClass one>
  11 return
```

4.2 The Ginger Language

The Ginger language [53] was developed at the University of Warwick as a means of investigating parallelism in functional languages (we do not consider the parallel features of the language in this thesis). It is a simple, weakly-typed, pure lazy functional language with no pattern-matching or user-defined types.

After parsing, our Ginger compiler transforms the source tree into a set of supercombinators (see Section 2.1) by lifting any λ-abstractions into separate function definitions [49]. Then dependency analysis [86] is performed, which transforms local variable definitions into blocks of simple, non-recursive `let` expressions and blocks of minimally mutually-recursive `letrec` blocks. The source at this stage is structured as in Figure 4.1.

We have an optional `package` declaration in which the class we eventually create will be placed. Then come a number of `import` declarations which deal with the importing of Ginger functions (stored in Java classes) from other sources followed by the definitions of the supercombinators, both those defined by the user and those created by lambda-lifting.

$$
\begin{array}{rcl}
\langle \mathit{program} \rangle & ::= & (\texttt{package}\ \langle \mathit{package} \rangle \texttt{;})? \\
 & & (\texttt{import}\ \langle \mathit{class} \rangle \texttt{;})* \\
 & & \langle \mathit{definition} \rangle * \\
 & & \\
\langle \mathit{definition} \rangle & ::= & \langle \mathit{identifier} \rangle\ \langle \mathit{identifier} \rangle *\ \texttt{=}\ \langle \mathit{expr} \rangle \texttt{;} \\
 & & \\
\langle \mathit{expr} \rangle & ::= & \langle \mathit{integer} \rangle \mid \langle \mathit{boolean} \rangle \mid \langle \mathit{float} \rangle \mid \langle \mathit{character} \rangle \mid \langle \mathit{string} \rangle \\
 & \mid & \langle \mathit{identifier} \rangle \\
 & \mid & \texttt{[]}\ \text{(empty list)} \\
 & \mid & (\langle \mathit{expr} \rangle \texttt{:} \langle \mathit{expr} \rangle)\ \text{(list constructor)} \\
 & \mid & (\langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle)\ \text{(application)} \\
 & \mid & \texttt{(}\langle \mathit{expr} \rangle \texttt{,} \ldots \texttt{,} \langle \mathit{expr} \rangle \texttt{)} \\
 & \mid & \texttt{if}\ \langle \mathit{expr} \rangle\ \texttt{then}\ \langle \mathit{expr} \rangle\ \texttt{else}\ \langle \mathit{expr} \rangle\ \texttt{endif} \\
 & \mid & \texttt{let}\ \langle \mathit{identifier} \rangle\ \texttt{=}\ \langle \mathit{expr} \rangle\ \texttt{in}\ \langle \mathit{expr} \rangle\ \texttt{endlet} \\
 & \mid & \texttt{letrec} \\
 & & \quad \langle \mathit{identifier} \rangle\ \texttt{=}\ \langle \mathit{expr} \rangle \\
 & & \quad \ldots \\
 & & \quad \langle \mathit{identifier} \rangle\ \texttt{=}\ \langle \mathit{expr} \rangle \\
 & & \texttt{in}\ \langle \mathit{expr} \rangle\ \texttt{endletrec}
\end{array}
$$

Figure 4.1: The EBNF of the Ginger language after lambda-lifting and dependency analysis.

4.3 Representation of Graph Nodes

Since we are creating a Java class file, it makes sense to represent graph nodes by Java objects. The five simple types — integers, floats, booleans, characters and strings — are represented using the Java classes `Long`, `Double`, `Character`, `Boolean` and `String` found in the `java.lang` package. The first four are the object equivalent of the primitive Java types `long`, `double`, `char` and `boolean` respectively. Note that we use the largest size possible for integers and floats and we do *not* represent strings as list of characters.

Lists are represented using the `List` class, which is just an empty class which its two subclasses `Cons` (list constructor) and `EmptyList` subclass. The `Cons` class has the following skeleton definition:

```
public final class Cons extends List {
  public Object head;
  public Object tail;

  // ...
}
```

The representation of functions utilises the `java.lang.reflect` package which provides classes that 'reflect' the members of the class, in particular there is a `Method` class the instances of which reflect a particular method of a particular class. This class has an instance method `invoke`:

```
public Object invoke(Object obj, Object[] args)
```

This invokes the the method reflected by the instance on the object `obj` with the arguments in the array `args`. If the method being reflected is static then `obj` is ignored and can be `null`.

Our compiler will translate all the supercombinators in a file into static methods of the generated class file (see Section 4.5). This gives us a maximum of 255 arguments and local variables per function. We use static methods as these only operate on their arguments whereas instance methods also require an instance of the class of which they were defined in.

When we import a class file we thus store each of its methods (functions of the original program) as `Method` objects which are held inside `Func` objects:

```
public class Func {
  public final Method method;
```

```
  public final int arity;
  public final boolean isCAF;

  public Func(Method m) {
    method = m;
    arity = m.getParameterTypes().length;
    isCAF = arity == 0;
  }

  // ...
}
```

Our representation of applications is guided by the type of the `invoke` method of the `Method` class, which will be called every time we apply a function. Since this method expects its arguments to be packed into an array, it makes sense to store the arguments of an application in an array, rather than in some intermediate data structure from which we make an array. In particular, we use multiple-argument applications. The `App` class has the following definition:

```
public final class App {
  public Object functor;
  public Object[] args;

  public boolean in_nf = false;
  public boolean total_app = false;

  // ...
}
```

This represents the application `functor args[0] ... args[args.length-1]`. The field `in_nf` is set when the `App` is in normal form, that is when the `functor` is a function and there are not enough arguments present, or the `App` is acting as an indirection to a non-application (see Section 4.3.1). If `functor` is a function and it is applied to exactly the right number of arguments then we set the `total_app` field. The use of the `in_nf` and `total_app` fields prevents unnecessary work being done.

4.3.1 Updating

As we saw in the Section 2.3, we need to update the original application with the result of the application. This is to prevent the unnecessary re-evaluation. For instance, the application `square ((+) 3 4)` becomes `(*) ((+) 3 4) ((+) 3 4)` (with the instances of `(+) 3 4` being shared) and `(+) 3 4` becomes 7.

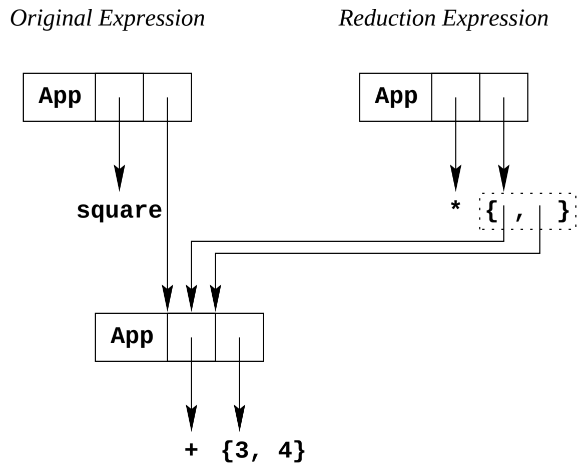
In the case where we update one application with another, we simply copy the result field-by-field onto the original application. However, if the result is *not* an application, as in the second case, then things are not so simple as we cannot copy an object of one type onto an object of a different type. Instead, we turn the `App` into an indirection by setting its `args` field to the `null` reference and setting its `functor` field to the result in question. We can view any `App` with a `null args` field as serving as an indirection or a wrapper to its actual argument. Note that once the result of an application becomes a non-application it is in normal form and thus no further evaluation is necessary and so we do not get chains of indirections. Updating is done in the method `App.update`:

```
private void update(Object o) {
  if (o instanceof App) {
    // copy o onto this App
    App a = (App) o;
```

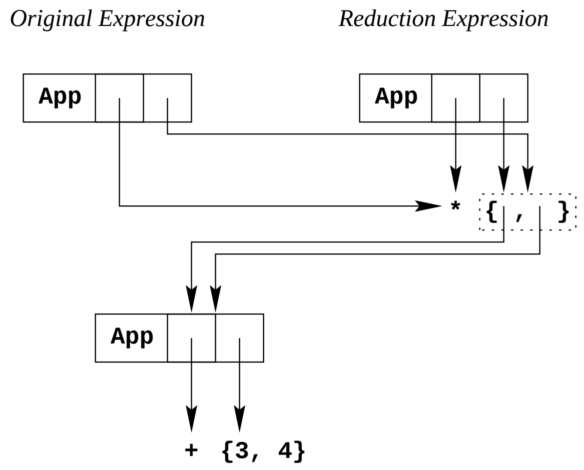
```
    functor = a.functor; args = a.args;
    in_nf = a.in_nf; total_app = a.total_app;
  }
  else if (o instanceof Func) {
    functor = o; args = empty;
    if (((Func) o).isCAF) {
      total_app = true; in_nf = false;
    }
    else {
      total_app = false; in_nf = true;
    }
  }
  else { // we have an indirection
    functor = o; args = null;
    in_nf = true; total_app = false;
  }
}
```

Here `empty` is a field of `App` which is set to b an empty array of `Object`s.

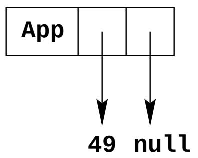
Recalling our original example, `square ((+) 3 4)` reduces to the expression `(*) ((+) 3 4) ((+) 3 4)`. Before updating we have:



Updating the original expression involves reassigning its `functor` and `args` fields (ignoring the other two `boolean` fields for the moment):



After the reduction of `(*) ((+) 3 4) ((+) 3 4)` our `App` becomes an indirection to `49`:



Applications are not the only graph nodes that can be updated, there is one other case, namely CAFs, that is functions taking zero arguments. These can be treated as applications of the CAF in question to zero arguments and we do just this by storing all CAFs as `App`s whose `functor` is the actual CAF in question and whose `args` is the empty array (note that if we wish to invoke a method that takes no arguments then we need to pass an empty array to `Method.invoke`).

4.3.2 Evaluation

The evaluation of graph nodes is controlled by the method `eval` in the class `Node` which contains various static methods used for the evaluation and printing of graph nodes. The method only has to do something when it is called to evaluate an `App`, otherwise it simply returns its argument.

The instance method `App.eval` repeatedly evaluates and updates itself until it is in normal form. Once it becomes so it returns its `functor` if it is an indirection, or itself otherwise:

```
public Object eval() {
  while (! in_nf) {
    if (total_app)
      // ...
    else if (functor instanceof Func)
      // ...
    else
      // ...
  }
  // if we have an indirection, return the functor,
  // else return the whole App
  return (args == null) ? functor : this;
}
```

The `App` being evaluated forms the spine of the graph, with the `functor` being the base of the spine and the `args` forming the ribs. If the `App` is not in normal form then we have one of three cases, corresponding to the three branches of the `if-then-else` ladder inside the `while` loop.

If we have a function applied to the exact number of arguments (that is when `total_app` is true) we simply need to apply the function to its arguments and update the `App`:

```
if (total_app)
  update(((Func) functor).apply(args));
```

The method `Func.apply` is where all the work is done. In this method we unpack any indirections from `App`s and then invoke the function on these unpacked arguments:

```
public Object apply(Object[] as) {
  for (int i = 0; i < as.length; i++)
    if (as[i] instanceof App) {
      App a = (App) as[i];
      if (a.args == null) // N.B. we keep CAFS inside Apps
        as[i] = a.functor;
    }

  return method.invoke(null, as);
}
```

If we have a function applied to too many arguments, that is, the `functor` is a function and both `total_app` and `in_nf` are false, we split the `args` array into two, apply the `functor` to the number of arguments it needs and update the `functor` and `args` fields appropriately:

```
else if (functor instanceof Func) {
  // we must have more arguments than the function takes
```

```
  Func f = (Func) functor;

  // split this array into two parts.
  int unused = args.length - f.arity;

  Object[] first = new Object[f.arity];
  Object[] rest = new Object[unused];
  split(args, f.arity, first, rest);

  // the functor becomes the result of apply f to the first
  // arity arguments
  functor = f.apply(first);

  // and the args become the rest of the args
  args = rest;

  setType();
}
```

The method `App.setType` examines the `functor` and `args` fields of the `App` and sets `total_app` and `in_nf` appropriately.

The last case is when the functor is another `App` in which case we need to unwind the `App` at the functor onto this one. If the `functor` is a function applied to the correct number of arguments, then we do the application before unwinding:

```
else { // functor instanceof App
  // need to unwind
  App a = (App) functor;
```

```
  if (a.total_app)
    // the functor contains a function applied to the correct
    // number of arguments, so we apply it and continue unwinding
    a.update(((Func) a.functor).apply(a.args));
  else {
    functor = a.functor;
    args = cat(a.args, args);
  }

  setType();
}
```

The static method `App.cat` joins together two arrays in a manner similar to list concatenation in functional languages.

4.3.3 Printing

The evaluation of graph nodes is initially triggered by the `Node.print` method which evaluates a node and prints it on standard output:

```
public final static void print(Object node) {
  node = eval(node);

  if (node instanceof Cons) {
    System.out.print("[");
    boolean not_at_end = true;
    do {
      Cons c = (Cons) node;
```

```
      c.head = eval(c.head); // evaluate and update head
      c.tail = eval(c.tail); // and tail
      printNoEval(c.head);
      if (c.tail instanceof Cons) {
        System.out.print(", ");
        node = c.tail;
      }
      else
        not_at_end = false;
    } while (not_at_end);
    System.out.print("]");
  }
  else if (node instanceof Evaluatable)
    ((Evaluatable) node).print();
  else
    System.out.print(node);
}
```

The method `Node.printNoEval` is similar to `print` except that it presumes that its argument is in normal form and thus does not need to evaluate the object before printing it.

If the node to be printed is a `Cons` then we iteratively print out each element of the list. It is done this way, rather than farming it out to some method of `Cons`, which would be the standard object-oriented way, as we do not wish to keep an unnecessary reference to the head of the list (note that we repeatedly overwrite the head in `Node.print`), which could lead us to keeping the whole of the list that is being printed in memory when in reality it could be garbage and the memory it occupies could be freed. We also reassign `c.head` and `c.tail` after evaluating them; this enables us to bypass any indirection introduced by any evaluation which needed to be done since the `eval` method will return the evaluated object unpacked from the application which is serving as an indirection.

The `Evaluatable` class is implemented by all Ginger classes which have components which need to be evaluated before being printed, in particular it is implemented by the various tuple classes. It specifies a `print` method which evaluates the components of an object and prints them out on standard output. Note that the `toString` method on these objects does *not* evaluate the components before printing (this is used for debugging purposes). So, `node` is an instance of this interface, we print it out using the `print` method.

Finally, if `node` is neither a `Cons` object or implements the `Evaluatable` interface, we simply print out the node using the `toString` method (which is implicitly called by the `System.out.print` method).

4.4 Primitives

Like user-defined functions we store primitives, such as arithmetic operators, comparison operators, list constructors and deconstructors, in Java class files. The primitives are spread over a number of classes, which are all subclasses of the `Node` class which contains the top-level evaluation and print routines. The strict primitives are kept in different classes to the lazy ones, thus giving us a simple, if restricted, way of determining if a primitive is strict. We also ensure that the result of all strict primitives is in normal form.

We allow overloading on primitives, for example, we use `-` for integer and real subtraction, but this is done in an *ad hoc* way inside the primitive itself making use of the Java `instanceof` operator rather than in any systematic kind of way such as the use of type classes in Haskell (see Section 2.4). For example, subtraction is performed using the `_minus` method in the class `StrictPrimitives`:

```
public class StrictPrimitives extends Node {
  public final static Class TYPE = StrictPrimitives.class;

  public static Object _minus(Object lhs, Object rhs) {
    lhs = eval(lhs);
    rhs = eval(rhs);
    if (lhs instanceof Long && rhs instanceof Long)
      return new Long(((Long) lhs).longValue() -
                      ((Long) rhs).longValue());
    else
      return new Double(((Number) lhs).doubleValue() -
                        ((Number) rhs).doubleValue());
  }
  public final static Object _minus =
    Function.make(TYPE, "_minus", 2);
  // ...
}
```

The `StrictPrimitives` class has a `TYPE` field which stores the `Class` representation of itself which is used to construct the `Object` equivalent of each method.

The `_minus` method first evaluates its two arguments and reassigns them (remember that the `eval` method unpacks any indirections). It then determines whether it is integer or real subtraction is required by means of the `instanceof` operator (integers are cast to reals if necessary), does the necessary subtraction and returns a new object containing the result of the subtraction.

The structure of the primitive classes and the classes created by the compiler are identical and hence we can treat primitives exactly the same way as we do user-defined functions except in the case of a tail recursion (see Section 4.5.5).

4.4.1 Fuzzy Primitives

As with our Haskell implementation, we overload the logical operators so that they work on fuzzy values as well as boolean ones, and also so that they act as set operators when used on functions. This is supplied as an optional extra with the compiler since the increased overloading leads to a small performance penalty.

Since Ginger has no systematic method of overloading, the overloading is done inside the primitive definition and overloading resolution done at run-time. For instance, logical conjunction (set intersection) is defined in the method `_and`:

```
public static Object _and(Object lhs, Object rhs)  {
  lhs = eval(lhs);

  if (lhs instanceof Boolean)
    return (FALSE.equals(lhs)) ? FALSE : rhs;
  else if (lhs instanceof Number)
    return new Double(Math.min(((Number) lhs).doubleValue(),
                               ((Number) eval(rhs)).doubleValue()));
  else if (lhs instanceof Tuple)
    return ((Tuple) lhs).apply(_and, (Tuple) eval(rhs));
  else
    return new App(new Object[] {rhs, lhs, _and}, combine);
}
public static final Object _and = Function.make(TYPE, "_and", 2);
```

The first two branches of the conditional should be fairly self-explanatory: they implement boolean and fuzzy conjunction respectively. The next branch deals with conjunction over tuples. The method `Tuple.apply` is used to combine two tuples of the same size into a new one, combining corresponding elements with the supplied function, here `_and`. The final case assumes we have a function, that is, a fuzzy subset. The two sets and the primitive `_and` are joined together with the function `combine` which is equivalent to the $\mathbf{S}'$ combinator defined as:

$$\mathbf{S}' \ op \ f \ g \ x = op \ (f \ x) \ (g \ x)$$

We can define the other primitives, disjunction, negation and addition, similarly. The rest of the definitions in Chapter 3 have the obvious translations from Haskell into Ginger.

4.5 Compilation

In this section we shall deal with the creation of Java class files from our Ginger source which has been lambda-lifted and had dependency analysis performed on it (see Figure 4.1). Rather than creating the class files directly, or using the Java language itself as a source (which would complicate matters *viz* local variables), we target the Jasmin assembly language [79]. This language is very similar to the byte-code used by the Java Virtual Machine, but is easier to program in as it deals with such things as the Java constant pool (where all constants and object references as such) and calculating offsets for jumps automatically. Our Ginger program, in the file *prog*`.g`, is compiled into an intermediate Jasmin file *prog*`.j` (which may be discarded after use), which is assembled into a Java class file *prog*`.class` by Jasmin.

Each supercombinator definition of our Ginger program is compiled into a static method of the class file we are creating. Functions are not compiled by creating code to create a new instance each time one occurs, but rather a single instance is created and is stored it as a static field of the class we are creating. These fields will be set up in a static initialiser of the class we are creating. Therefore, whenever we want a function we just access the relevant field. This method also applies when we want to access a function defined in another class. The job of the various `import` declarations is thus just to tell us in which class to find each function that we use. There is a limit on the number of fields (in our case, functions) in a class (65,535) but if this limit is reached then program can be split up into smaller segments (a program which hit this limit must have been fairly big and unwieldy anyhow).

Our compilation schemes are based on those presented in [86, 89]. We view our source as a triple $\langle cl, fs, ss \rangle$ where cl is the class we are to create, fs is the set of *all* functions defined or imported and ss is the set of supercombinator definitions. Note that although the JVM has shorter, more optimal versions of some instructions (the instruction `iconst_0` is a more efficient way of loading the integer zero onto the stack than `ldc2_w 0`, for example) for clarity we use the most general instruction in our description of the compilation schemes, though we do use the most efficient instruction in our actual implementation. Our primary compilation scheme, $\mathcal{P}$, starts off as:

$\mathcal{P}\langle cl, fs, \{s_1, \ldots, s_n\}\rangle =$

> `.class public` cl
>
> `.super` $Object$

This declares our class and its superclass. Note that Jasmin requires the full name of all classes and members but for brevity we have omitted the package name where this is obvious, indicating the omission by using italics for object names rather than teletype. We then proceed by declaring the fields corresponding to each function defined in the file:

> `.field public static` $\phi(s_1)$ `L`$Object$`;`

⋮

`.field public static` $\phi(s_n)$ `L`*Object*`;`

The function ϕ returns the name of the field that holds the supercombinator, which is just its name of said supercombinator. Note that when an object name, *obj* say, is used as a type it is written as `L`*obj*`;`. Functions imported will be declared and defined in the class that they are imported from. $\mathcal{P}$ progresses by setting each of these fields to its appropriate value inside a static initialiser which is a method called `clinit` which takes no arguments and returns `void` (indicated by the `V`):

`.method <clinit>()V`

`new` *cl*

`dup`

`invokespecial` *cl*`/<init>()V`

`invokevirtual` *cl*`/getClass()L`*Class*`;`

`astore_0`

$\mathcal{D}\ s_1$

⋮

$\mathcal{D}\ s_n$

`return`

`.end method`

The method first gets the `Class` object reflecting the class we are creating and stores it in register 0. This `Class` object is used when creating the `Func` object representing each supercombinator (or `Apps` in the case of CAFs). $\mathcal{D}$ creates the code necessary to create a new instance of its argument and store it in the relevant field. If we have a supercombinator, s say, then we have:

$$
\begin{aligned}
\mathcal{D}\, s \;=\;& \texttt{aload\_0} \\
& \texttt{ldc}\ n(s) \\
& \texttt{ldc}\ a(s) \\
& \texttt{invokestatic}\ \mathit{Function}\texttt{/make(L}\mathit{Class}\texttt{;L}\mathit{String}\texttt{;I)L}\mathit{Object}\texttt{;} \\
& \texttt{putstatic}\ \phi(s)\ \texttt{L}\mathit{Object}\texttt{;}
\end{aligned}
$$

This loads the `Class` object reflecting cl onto the stack, then the name of the supercombinator (using the function n) and its arity (using a). This information is then used by the method `Function.make` to create a `Func` object (non-CAFs) or an `App` (CAFs) which is then stored in the appropriate field.

Returning to the $\mathcal{P}$ scheme, after declaring and defining our constants, we now need to create the code for each of the supercombinators. This is done using the $\mathcal{F}$ scheme:

$$
\begin{aligned}
&\mathcal{P}\langle cl, fs, \{c_1, \ldots, c_m\}, \{s_1, \ldots, s_n\}\rangle = \\
&\quad \vdots \\
&\quad \mathcal{F}\ fs\ s_1 \\
&\quad \vdots \\
&\quad \mathcal{F}\ fs\ s_n
\end{aligned}
$$

The $\mathcal{F}$ scheme is defined below. Finally, we need to determine if we need to create a `main` method which will make the class file executable, using the `java` interpreter, say. This is so if we have defined a function called `main`. The `main` method will print the result of evaluating the Ginger function `main` which we rename `_main`, so as to separate the reduction rule from the code which does the evaluation and printing. The code for this is:

```
.method public static main([LString;)V
```

```
getstatic φ(_main) LObject;
invokestatic Node/print(LObject;)V
return
.end method
```

Here `[L`*String*`;` denotes an array of strings (the closing brace is not used) which in the case of the arguments to the `main` method represent the command-line arguments passed by the Java interpreter. If we do not create a `main` method then we can view the generated class as a library of functions.

4.5.1 Compiling Supercombinators

We now need to give the definition of $\mathcal{F}$ which creates a method from a supercombinator. This method takes n arguments of type `Object`, where n is the arity of the supercombinator in question, and returns an object of type `Object`. The return object will be the object left on the top of the stack by the code generated by the $\mathcal{R}$ scheme, which compiles the expression on the right-hand side of a supercombinator definition.

$\mathcal{F}[f\ x_1\ \ldots\ x_n = E]\ \mathit{fs} =$

$$\texttt{.method public static } f\,\texttt{(}\underbrace{\texttt{L}\mathit{Object}\texttt{;}\ldots\texttt{L}\mathit{Object}\texttt{;}}_{n\text{ times}}\texttt{)L}\mathit{Object}\texttt{;}$$

$\mathcal{R}\ E\ [x_1 = 0, \ldots, x_n = n-1]\ \mathit{fs}\ n$

```
areturn
.end method
```

The scheme $\mathcal{R}$ takes as arguments the expression to compile, an environment detailing which register each variable is in, the set of functions defined and imported, and the next free variable register (used to store local variables).

4.5.2 The $\mathcal{R}$ Compilation Scheme

The purpose of the $\mathcal{R}$ scheme is to take an expression which forms the right-hand side of a definition (the return expression) and compile it to code that will, when executed, create the graph of the expression and leave a reference to it on top of the stack.

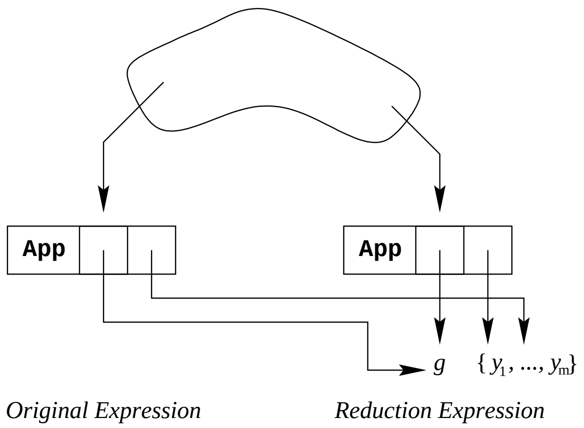
If we have an integer, i say, then we have to create a new `Long` object to store it in:

$$\mathcal{R}\ i\ \rho\ fs\ v \quad = \quad \texttt{new}\ \mathit{Long}$$
$$\texttt{dup}$$
$$\texttt{ldc2\_w}\ i$$
$$\texttt{invokespecial}\ \mathit{Long}\texttt{/<init>(J)V}$$

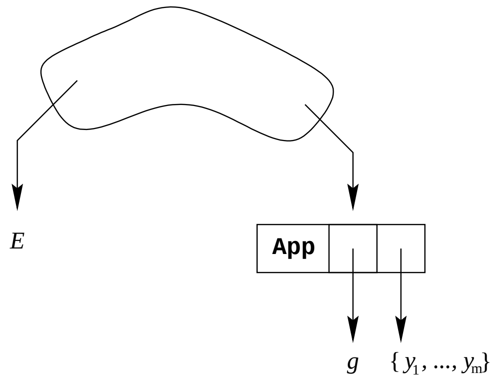
The instruction `ldc2_w` is used to load a long or a double onto the stack (each stack cell is 32 bits in size, but long integers and doubles take up 64 bits so they have to be spread across two cells). A similar method is used to define the other types of constants.

If the right-hand side of a super-combinator consists of a single variable then, unless we evaluate it before returning, we risk doing extra work because some loss of sharing occurs. Suppose we have an expression $f\ x_1\ \ldots\ x_n$ which reduces to an expression $g\ y_1\ \ldots\ y_m$ and further suppose that this latter expression is reducible. These are represented as two `App`s and recall that updating the original expression involves copying the fields of the `App` representing the value of the reduction onto the fields of the `App` representing the original expression.

If the `App` $g\ y_1\ \ldots\ y_m$ was created during the evaluation of the reduction rule for f then the `App` object $g\ y_1\ \ldots\ y_m$ is never used again (in other words, it becomes garbage), though its fields become the fields of the original expression. However, if the `App` was *not* created by reduction rule then it must be referenced by some other part of the graph and hence we have two copies of the same application:



Suppose that $g\ y_1\ \ldots\ y_m$ evaluates to some expression E, say. Then if we continue our evaluation we have:



Note that we have only reduced one of the `App`s: by copying we have lost not only the sharing of nodes but the sharing of work.

This situation occurs whenever we have a function whose return value is a single variable or function (or, more specifically, a CAF). We can prevent the replication of work by making sure that whenever we have such a function we first evaluate it (to normal form) before returning it. Thus if such a function still returns an `App` it will be in normal form and although we may still make an unnecessary copy of it we cannot waste time by duplicating evaluation as there is no evaluation to do.

The $\mathcal{R}$ scheme for variables and CAFs is thus:

$$\mathcal{R}\ id\ \rho\ fs\ v \quad = \quad \mathcal{C}\ id\ \rho\ fs\ v$$

$$\qquad\qquad\qquad\qquad \texttt{invokestatic}\ \mathit{Node}\texttt{/eval(L}\mathit{Object}\texttt{;)L}\mathit{Object}\texttt{;}$$

If id is a function of arity greater than zero, we have:

$$\mathcal{R}\ id\ \rho\ fs\ v = \texttt{getstatic}\ \phi(id)\ \texttt{L}\mathit{Object}\texttt{;}$$

Before compiling applications we first unwind them, using the left-associativity of function application, so they are of the form $f\ e_1\ \ldots\ e_n$ where f is an identifier (anything else would be a type error). If f refers to a variable then we just compile the arguments and functor of the application, packing the arguments into an array, and construct the `App` object:

$$\mathcal{R}\ (f\ e_1\ \ldots\ e_n)\ \rho\ fs\ v =$$

$$\begin{array}{l}
\texttt{new}\ \mathit{App} \\
\texttt{dup} \\
\left.\begin{array}{l} \texttt{ldc}\ n \\ \texttt{anewarray}\ \mathit{Object} \end{array}\right\}\text{Create an array of } n \text{ objects} \\
\left.\begin{array}{l} \texttt{dup} \\ \texttt{ldc 0} \\ \mathcal{C}\ e_1\ \rho\ fs\ v \\ \texttt{aastore} \end{array}\right\}\text{Set the } 0^{\text{th}} \text{ element of the argument array} \\
\vdots \\
\left.\begin{array}{l} \texttt{dup} \\ \texttt{ldc}\ (n-1) \\ \mathcal{C}\ e_n\ \rho\ fs\ v \\ \texttt{aastore} \end{array}\right\}\text{Set the } (n-1)^{\text{th}} \text{ element of the argument array} \\
\mathcal{C}\ f\ \rho\ fs\ v
\end{array}$$

invokespecial *App*/<init>([L*Object*;L*Object*;)V

Here $\mathcal{C}$ is the generic scheme used to compile expressions (see Section 4.5.3).

If f is a function then we can provide a bit more information to the constructor. If there are not enough arguments present then we can tell the App constructor to set the in_nf field. If there are exactly enough arguments present then we tell the constructor to set the total_app field. This is done by using a three-constructor of App which as well as taking the arguments and functor of the application takes an additional boolean which if set to true sets the in_nf and the total_app to false and *vice versa*. For these two cases we have:

$\mathcal{R}\ (f\ e_1\ \ldots\ e_n)\ \rho\ fs\ v =$

new *App*

dup

nf

⋮

compile arguments and functor as before

⋮

invokespecial *App*/<init>(Z[L*Object*;L*Object*;)V

where

$$\begin{aligned} nf &= \texttt{iconst\_1}, \quad \textbf{if } n < \text{ arity of } f \\ &= \texttt{iconst\_0}, \quad \textbf{if } n = \text{ arity of } f \end{aligned}$$

The JVM uses integers to represent booleans (the type of which is denoted by a Z). The most efficient way to load these onto the stack are using the instructions iconst_1 for true and iconst_0 for false.

If there are too many arguments present then we split the application into two: if f is of arity m and is applied to n arguments where $n > m$ then we create the application of f applied to the first m arguments applied to the other $n - m$ arguments.

$\mathcal{R}$ $(f\ e_1\ \ldots\ e_m\ e_{m+1}\ \ldots\ e_n)\ \rho\ fs\ v =$

```
new App
dup
ldc (n − m)
anewarray Object
dup
ldc 0
𝒞 e_m ρ fs v
aastore
⋮
dup
ldc (n − m − 1)
𝒞 e_n ρ fs v
aastore
𝒞 (f e_1 ... e_m) ρ fs v
invokespecial App/<init>([LObject;LObject;)V
```

Compiling `if` statements requires us to evaluate the antecedent and jump accordingly.

$\mathcal{R}$ (`if` a `then` t `else` f `endif`) $\rho\ fs\ v =$

```
ℰ a ρ fs v
checkcast Boolean
invokevirtual Boolean/booleanValue()Z
```

$\texttt{ifeq}\ \mathit{FALSE}$

$\mathcal{R}\ t\ \rho\ \mathit{fs}\ v$

$\texttt{goto}\ \mathit{ENDIF}$

$\mathit{FALSE}:$

$\mathcal{R}\ f\ \rho\ \mathit{fs}\ v$

$\mathit{ENDIF}:$

where *FALSE* and *ENDIF* are unique labels. The `checkcast` instruction makes sure that we have a `Boolean` object after evaluating the antecedent. The scheme $\mathcal{E}$ is used to compile an expression whose result is known to be needed (see Section 4.5.4). Note that the $\mathcal{R}$ scheme is used to compile the two branches of the conditional.

Compiling a simple `let` requires us to compile the definition, store it in the next free local variable, updating the environment accordingly, and compiling the body with respect to this new environment.

$\mathcal{R}\ (\texttt{let}\ x\ \texttt{=}\ d\ \texttt{in}\ b\ \texttt{endlet})\ \rho\ \mathit{fs}\ v =$

$\mathcal{C}\ d\ \rho\ \mathit{fs}\ v$

$\texttt{astore}\ v$

$\mathcal{R}\ b\ \rho[v = n]\ \mathit{fs}\ (v + 1)$

Compiling a `letrec` is more complex as each definition in the block will refer to at least one other one and hence if we are not careful we could end up loading objects from registers that have not yet been filled. We thus need to first of all put place-holders in each of the registers that are to be defined by the `letrec` and update them when each appropriate definition is compiled. These place-holders are `Apps` whose `functor` and `arg` fields are null and they are updated using the same `update` method used to update `Apps` that have been evaluated (see Section 4.3.1).

$\mathcal{R}\ (\texttt{letrec}\ \mathit{ds}\ \texttt{in}\ b\ \texttt{endletrec})\ \rho\ \mathit{fs}\ v =$

$A\ ds\ v$

$CL\ ds\ \rho'\ fs\ v'$

$\mathcal{R}\ b\ \rho'\ fs\ v'$

where $(\rho', v') = X\ ds\ \rho\ v$

Here A allocates the place-holders, CL compiles the definitions and updates the registers and X updates the environment and the next free variable. A is defined as:

$A\ (x_1 = e_1, \ldots, x_k = e_k)\ v =$

`new` App

`dup`

`invokespecial` App`/<init>()V`

`astore` v

$\vdots$

`new` App

`dup`

`invokespecial` App`/<init>()V`

`astore` $(v + k)$

CL is defined as:

$CL\ (x_1 = e_1, \ldots, x_k = e_k)\ \rho\ fs\ v =$

`aload` $(v - k)$

$\mathcal{C}\ e_1\ \rho\ fs\ v$

```
dup
invokevirtual App/update(LObject;)V
⋮
aload v
C e_k ρ fs v
dup
invokevirtual App/update(LObject;)V
```

Finally, X is defined as

$$X\ [x_1 = e_1, \ldots, x_k = e_k]\ \rho\ v = (\rho[x_1 = v, \ldots, x_k = v + k - 1], v + k)$$

4.5.3 The $\mathcal{C}$ Compilation Scheme

The $\mathcal{C}$ scheme, for the most part, is similar to the $\mathcal{R}$ scheme. The major difference is in the handling of variables. Since we do not have to worry about any loss of sharing, the definition of $\mathcal{C}$ when compiling a single variable is:

$$\begin{array}{lcll} \mathcal{C}\ id\ \rho\ fs\ v & = & \texttt{getstatic}\ \phi(id)\ \texttt{L}Object\texttt{;}, & \textbf{if}\ id \in fs \\ & = & \texttt{aload}\ \rho(id), & \textbf{otherwise} \end{array}$$

When compiling conditionals with the $\mathcal{C}$ scheme, the branches of the conditional are also compiled with $\mathcal{C}$ scheme. Similarly, the body of local variable declarations are compiled with the $\mathcal{C}$ scheme when `let letrec`s are compiled with the $\mathcal{C}$ scheme.

4.5.4 The $\mathcal{E}$ Compilation Scheme

The $\mathcal{E}$ scheme is used when the we know that an expression is to be evaluated and is *not* a tail call (this is handled by $\mathcal{R}$). Later on, we shall see how we can optimise the code produced by this scheme, but for now we shall just add a call to `Node.eval`:

$$\mathcal{E}\ e\ \rho\ fs\ v = \quad \mathcal{C}\ e\ \rho\ fs\ v$$
$$\texttt{invokestatic}\ Node\texttt{/eval(L}Object\texttt{;)L}Object\texttt{;}$$

We can also use the $\mathcal{E}$ scheme to compile the branches of conditionals and the body of local variable declarations.

4.5.5 Tail Recursion

A tail recursion occurs when the result of one function is the result of applying another function to the correct number of arguments. Any tail-recursive calls in our program will be compiled using the $\mathcal{R}$ scheme. It is a property of graph reduction that tail-recursive calls can be run in constant space no matter how deep the recursion is [109], and our implementation preserves this property. This is because if we have an expression $f\ x_1 \ldots\ x_n$ that reduces to $g\ y_1\ \ldots\ y_m$ then the call to g is not executed inside the code of f but the application is passed back to the `eval` method which then calls g. Thus, if we have a chain of tail-recursive calls $g_1, \ldots, g_n$ with g_i calling g_{i+1} then instead of recursing down the chain of g_is and back again, and having a recursion that is $O(n)$ levels deep, we instead 'bounce' between `eval` and each g_i and the recursion only $O(1)$ levels deep. Hence our implementation executes tail-recursive functions in constant space.

4.5.6 Initial Results

Table 4.1 shows the running times of some programs compiled using our Ginger compiler. Because the running times of Java programs can vary significantly, we have averaged our times over three runs. All times include the time needed to inialise the JVM, which is roughly 0.5–1.0 seconds. The programs were run on a

Program	Time (s)
`take 500 primes`	65.4
`nfib 30`	357.4
`soda`	11.5
`cal`	34.1
`edigits 250`	82.8
`queens 8`	107.2

Table 4.1: Initial running times of programs produced by the Ginger compiler

Sun Enterpise 3000 with two 168MHz processors and 512MB of memory running Solaris 2.6 and using the Sun JDK 1.1.5. Descriptions of the programs are as follows:

- `take 500 primes` outputs the first 500 prime numbers using the sieve of Eratosthenes method.
- `nfib 30` calculates the number of reductions used to calculate the 30th Fibonacci number using the naïve doubly-recursive method.
- `soda` performs a serial word-search on a 10×15 grid.
- `cal` outputs calendars for the years 1990–99.
- `edigits 250` evaluates the first 250 digits of e (2.7182...).
- `queens 8` prints all 92 solutions to the eight queens problem.

4.6 Optimisations

In this section we shall detail two optimisations that can increase the performance of the code produced by our compiler: the direct invocation of some function applications and the use of a single instance to represent each constant declared by a program.

4.6.1 Direct Function Invocations

Implementations of lazy functional languages use application nodes to store 'suspended' function calls, that is function calls whose result may or may not be needed. However, in some cases it can be predicted that the result of the function call will be needed, and we can avoid having to build the application representing the function call and instead invoke the function directly.

The first place we can use this optimisation is in the $\mathcal{E}$ scheme, as we know that we always need the result of an expression compiled using this scheme. If we have a function, f, of arity n applied to the correct number of arguments, we have:

$$
\begin{array}{l}
\mathcal{E}\ (f\ e_1\ \ldots\ e_n)\ \rho\ \mathit{fs}\ v = \\
\quad \mathcal{C}\ e_1\ \rho\ \mathit{fs}\ v \\
\quad \vdots \\
\quad \mathcal{C}\ e_n\ \rho\ \mathit{fs}\ v \\
\quad \texttt{invokestatic}\ \ f\texttt{(}\underbrace{\texttt{L}\mathit{Object}\texttt{;}\ldots\texttt{L}\mathit{Object}\texttt{;}}_{n\ \text{times}}\texttt{)L}\mathit{Object}\texttt{;} \\
\quad \texttt{invokestatic}\ \ \mathit{Node}\texttt{/eval(L}\mathit{Object}\texttt{;)L}\mathit{Object}\texttt{;}
\end{array}
$$

If f is known to be strict then we can compile each of the arguments using the $\mathcal{E}$ scheme rather than the $\mathcal{C}$ one.

We can also use this technique with the $\mathcal{R}$ scheme, with one major caveat. It is safe to evaluate all tail calls, which will be compiled using the $\mathcal{R}$ scheme, since their result will eventually be needed. However if the method we invoke is itself tail-recursive then we could have a long chain of recursions and evaluation will no longer occur in constant space and we could be in danger of overflowing the stack on which the JVM stores the return address for each method call. Although it is possible to optimise tail calls by replacing a method call with a jump, many JVM implementations do not do so. We cannot provide this optimisation manually either,

Program	Non-optimised	Direct invocation	% decrease
`take 500 primes`	65.4	38.2	42
`nfib 30`	357.4	140.2	61
`soda`	11.5	8.4	27
`cal`	34.1	25.6	25
`edigits 250`	82.8	49.1	41
`queens 8`	107.2	73.0	32

Table 4.2: Running times (s) of programs produced by the Ginger compiler with and without direct function invocation.

as the JVM does not allow jumps between methods.

We thus only directly invoke tail calls involving functions that are not tail-recursive, which for simplicity's sake we assume that is just our set of primitives, none of which are tail-recursive. If we have a primitive p applied to the correct number of arguments, we have:

$$
\begin{aligned}
&\mathcal{R}\ (p\ e_1\ \ldots\ e_n)\ \rho\ \mathit{fs}\ v = \\
&\qquad \mathcal{C}\ e_1\ \rho\ \mathit{fs}\ v \\
&\qquad \vdots \\
&\qquad \mathcal{C}\ e_n\ \rho\ \mathit{fs}\ v \\
&\qquad \texttt{invokestatic}\ \ p\texttt{(}\underbrace{\texttt{L}\mathit{Object}\texttt{;}\ldots\texttt{L}\mathit{Object}\texttt{;}}_{n \text{ times}}\texttt{)L}\mathit{Object}\texttt{;}
\end{aligned}
$$

If p is known to be strict then we can compile each of the arguments using the $\mathcal{E}$ scheme rather than the $\mathcal{C}$ one. In all other cases, $\mathcal{R}$ remains as before. As can be seen from Table 4.2, direct invocation is a very worthwhile optimisation.

4.6.2 Single-instance Constants

Since constants are immutable in a pure functional language, we can represent each constant used by a pure functional program by a single instance, saving both the time and space needed to create a new instance of a constant each time one is encountered. We shall store each constant as a static field of the class that our functional program is compiled to (cf. how we store functions) and thus each time we need an instance of a constant we just need to access the relevant field. This is similar to a technique used by most Java implementations to implement strings (which are immutable in Java).

It is required that the $\mathcal{P}$ scheme is modified to handle constants. Suppose that $c_1, \ldots, c_n$ form the set of constants that are program uses. Then $\mathcal{P}$ becomes:

$$\mathcal{P}\langle cl, fs, \{c_1, \ldots, c_m\}, \{s_1, \ldots, s_n\}\rangle =$$

```
.class public cl
.super Object
.field public static φ(s_1) LObject;
⋮
.field public static φ(s_n) LObject;
.field public static φ(c_1) LObject;
⋮
.field public static φ(c_m) LObject;


.method <clinit>()V
new cl
dup
invokespecial cl/<init>()V
invokevirtual cl/getClass()LClass;
astore_0
D s_1
```

$\vdots$

$\mathcal{D}\ s_n$

$\mathcal{D}\ c_1$

$\vdots$

$\mathcal{D}\ c_m$

```
return
.end method
```

where the other names are as in the original definition of $\mathcal{P}$. The function ϕ has been extended to return the field name of a constant as well as that of a supercombinator. The scheme $\mathcal{D}$ is also extended to take constants as arguments. For example, if we have an integer, i, we have:

$\mathcal{D}\ i \;=\;$ `new` *Long*

`dup`

`ldc2_w` i

`invokespecial` *Long*`/<init>(J)V`

`putstatic` $\phi(i)$ `L`*Object*`;`

A similar method is used to define the other types of constants. As we can see from Table 4.3, this results in a modest, but significant speed-up in the running times of our programs.

4.7 Results and Other Work

The only other work we are aware of in this area is that of Wakeling, one based on the G-Machine [119], which translates the G-code produced by HBC (the Haskell compiler developed at Chalmers) into Java bytecode; and one based on the $\langle \nu, \mathrm{G} \rangle$

Program	Non-optimised	Single-instance constants	% decrease
`take 500 primes`	65.4	47.1	28
`nfib 30`	357.4	326.0	9
`soda`	11.5	11.4	1
`cal`	34.1	33.4	2
`edigits 250`	82.8	67.2	19
`queens 8`	107.2	103.1	4

Table 4.3: Running times (s) of programs produced by the Ginger compiler with and without single-instance constants.

machine [118] which compiles a core language into a set of $\langle \nu, \mathrm{G} \rangle$ instructions which are then transformed in Java byte-code, again using HBC. Both versions use a separate class, and hence a separate file, for each *function*, rather than each *program* as with our compiler. There is also a compiler for Standard ML from Persimmon (now being supported at Edinburgh Univesity at http://www.dcs.ed.ac.uk/home/mlj/) which compiles stand-alone SML programs to Java bytecode [47], but as this is for a strict language we do not include it in our comparisons. A different approach is taken by Pizza [84]. Instead of using the JVM as the target of functional code in instead introduces functional features — namely parametric polymorphism, higher-order functions and algebraic data types — into Pizza which is a superset of Java. This language is then transformed into pure Java which is then compiled to produce byte-code. A related issue is tackled by Claus Reinke [94] who investigated the possibility of using Java components, in particular graphical ones, in Haskell programs, with some success. Reinke's work utilised the Java Native Interface [30], which we use to integrate C/C++ functions into our Aladin implementation (see Section 5.4.2).

Table 4.4 gives the running times for several programs using our compiler with both optimisations switched on, and using both Sun's JVM and the Kaffe Open VM [105] to run the generated class files; Wakeling's compiler (the $\langle \nu, \mathrm{G} \rangle$-Machine

Program	Gingerc		Wakeling's	Hugs	GHC
	Sun JDK	Kaffe			
`take 500 primes`	30.5	266.2	50.5	6.3	0.8
`nfib 30`	118.0	638.4	56.6	114.6	6.7
`soda`	8.7	43.9	4.2	0.9	0.1
`cal`	25.0	140.3	19.4	5.0	0.3
`edigits 250`	46.2	174.4	10.0	4.0	0.5
`queens 8`	72.6	369.6	46.9	16.2	0.9

Table 4.4: Comparisons of running times of various lazy functional language implementations

version [118]) using Sun's JDK; the Haskell interpreter Hugs (version 1.4); and the Glasgow Haskell Compiler, GHC (version 2.10). The Haskell and Ginger sources were made as close as possible, but all the Haskell programs compiled using Wakeling's compiler have been explicitly mono-typed where appropriate. This is because if the overloading used in the programs is not resolved at compile-time then the running times can slow down by as much as a factor of 10 in extreme cases, because of the need to pass around dictionaries to resolve the overloading at run-time.

Both the JVM-targeting compilers perform poorly when compared to Hugs, with our compiler (using the JDK to execute the class files) being some 4–11 time slower than Hugs, except in the case of `nfib` (which is a somewhat artificial benchmark anyway) when performance matched that of Hugs. Why then is our compiler so much slower than Hugs, when both are either interpreters or produce code that is interpreted? First of all there are the deficiencies in our source language, Ginger, when compared to the much more complex Haskell. In particular, the lack of static type-checking and user-defined (algebraic) types and all but the crudest form of strictness analysis will have an appreciable effect on run-time performance, but Wakeling's compiler has these and it is not much faster than ours. There is also the cost incurred by not being able to resolve overloading until runtime; if we could, at least partially, then we could replace function calls to methods implementing basic primitives such as arithmetic and comparison operators to uses of basic JVM instructions (see Section 7.2). There is also the cost of using a high-level language (Java) as opposed to the lower-level C used to implement Hugs. For instance, Java does bounds checking on array accesses and checks that casts are legal whereas C does not, and both these operations occur frequently in our compiler.

Wakeling [119] ascribed the poor performance of his compiler when compared to Hugs to the poor memory handling in the JVM, hypothesising that memory-allocation in Java is an order of magnitude more expensive than in Hugs. Functional programs certainly will create and destroy objects on a more frequent basis than an imperative object-oriented one — both `primes` and `edigits` create something in the order of 500,000 `App` nodes and 130,000 `Cons` nodes, for example. Unfortunately, using the `-profile` of the `java` interpreter to look at the cost of these allocations is not useful as we can only look at the time taken by the code in the constructor (around 2.5 seconds for all 500,000 `App` objects used by `primes`) and not the time taken to allocate the memory, which is done before the constructor is invoked.

It is the allocation of objects which is probably the reason why running our programs using the Kaffe VM is some 3–9 times slower than running them using the Sun JVM, despite the fact that in some cases Kaffe can be around 2–3 times faster than the Sun JVM. Consider the following segment of code, which we compile using both the Kaffe and the Sun Java compiler:

```
Long l;
for (int i = 0; i < 1000000; i++)
  l = new Long(i);
```

The Kaffe VM takes 44 seconds to run the code produced by both compilers, while the Sun JVM takes only 3 seconds (both Virtual Machines take only half a second to run the same loop with an empty body).

Program	1MB	4MB	% decrease
`take 500 primes`	30.5	28.0	8
`nfib 30`	118.0	119.2	-1
`soda`	8.7	7.8	10
`cal`	25.0	20.7	17
`edigits 250`	46.2	23.2	50
`queens 8`	72.6	64.2	12

Table 4.5: Running times (s) of programs produced by the Ginger compiler with initial heap sizes of 1 and 4 megabytes

The large amount of mutual recursion on our programs means that we can't use the Java profiler to determine how long the JVM spends executing each method call. We can however examine the overhead imposed by the garbage collector, using the `-profile` or the `-verbosegc` options of the Java interpreter. This shows that using the default initial heap size of 1 megabyte garbage collection (as in Table 4.4) takes between from 10% of the total running time to 50% in the extreme cases like `edigits`. If we increase the initial heap size to 4 megabytes then we get the running times in Figure 4.5. The speed-up in running time is due to garbage collection being run less frequently, but freeing more memory when it does.

4.8 Summary

We have succeeded in producing a compiler for a functional language which creates Java class files as its object code, with a performance comparable to that of an approach which used a fully-fledged compiler, with various optimisations not present in our compiler, as its front end. However the performance of our compiler is poor when compared to a conventional lazy functional language interpreter (Hugs). This leads us to suspect that we may have reached a point where we cannot achieve any significant speed-ups no matter how we optimise our run-time architecture of the graph reduction on the JVM, and that any major leaps in performance can only come from optimising the JVM itself.

Chapter 5

The Aladin Abstract Machine

The Aladin Abstract Machine (AAM) [10] provides an abstract definition of a functional language. There are no primitives built into Aladin, instead primitives are intended to be programmed in *any* language, functional or imperative, and imported into the AAM. These primitives could be simple functions like addition; more complex higher-order ones like *map* or *fold*; or even complete programs such as *grep* or *wc*.

As well as the lack of primitives, Aladin has two more major features not found in the majority of functional languages: the ability to specify the strictness of a function's arguments *and* results; and the use of streams for ordered I/O and real-time operations (which we shall not consider further in this thesis).

Unlike the majority of other abstract machines for functional languages (see Section 2.5), the AAM is concerned only with the evaluation of programs and not their construction. Since Aladin is designed to import its primitives from *any* language, it would be inappropriate to tie the abstract machine down to one particular method of constructing programs.

It is this simplicity and the requirement for the user to specify the strictness of functions that gives Aladin its advantages. In the next chapter, we shall see how these advantages can be used to enable Aladin programs to be partially evaluated with only a small number of modifications to the machine described in this chapter. The purity and simplicity of Aladin could also be used to investigate other areas, such as parallel evaluation of functional programs and the effects of strictness on the time/space requirements of functional programs.

In this chapter we shall develop an efficient operational semantics for the AAM, starting from the original denotational semantics and going *via* a denotational semantics which includes explicit sharing and updating. We then use this semantics to develop an implementation of Aladin, using the Java language as with our Ginger implementation. Our compiler lets the user write Aladin programs in an Aladin scripting language that we develop, and to use primitives written in C [57], C++ [102], Java [7] and Ginger (see the previous chapter).

5.1 The Semantics of the AAM

An Aladin program is either a data object, a function or an application of one program to another. Denoting our set of data objects as D and our set of functions as F, our set of programs, P, is:

$$P ::= D \mid F \mid P\ P$$

Multiple-arguments to functions are curried and applications are left-associative. Hence for $p_1, p_2, p_3 \in P$ we have:

$$p_1\ p_2\ p_3 \equiv (p_1\ p_2)\ p_3$$

Functions may be written in any language of the user's choice. A function f of arity m is denoted as f^m. The meta-function @ is used to primitively apply f to its arguments by executing the code associated with f. The expression $f^m@(p_1, \ldots, p_m)$ denotes the result of the primitive application.

The user is required to specify the strictness of each of its argument (see Section 2.2). Aladin uses this information to control evaluation: strict arguments are evaluated before the function is applied; lazy ones are not. Expanding the definition given in Section 2.2, we extend the concept of strictness to functions of any arity (that is, without having to curry arguments). An Aladin function `f` of arity m is strict in its ith argument, where $1 \leq i \leq m$, if:

$$\mathtt{f}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_{i-1} \perp \mathtt{x}_{i+1}\ \ldots\ \mathtt{x}_m = \perp$$

for all possible values of $\mathtt{x}_j, 1 \leq j \leq m, j \neq i$ and non-strict or lazy otherwise. In a slight abuse of notation, we also apply the terms strict and lazy to the result of a function: a strict result does not need further evaluation; a lazy one does. For a function of arity m we use the notation:

$$f :: \sigma_1 \times \ldots \times \sigma_m \rightarrow \rho, \quad \sigma_i, \rho \in \{s, l\}$$

to denote a function which is strict in its ith argument if $\sigma_i = s$ and lazy if $\sigma_i = l$, and strict in its result if $\rho = s$ and lazy if $\rho = l$.

5.1.1 The Original Denotational Semantics

The original evaluation rules used a meta-function, **Eval**, to return the result of evaluating a program. If we have a data object applied to any number of programs, then **Eval** simply returns the original program:

$$\mathbf{Eval}[\![\ (d \in D)\ p_1\ \ldots\ p_n\ ]\!] = d\ p_1\ \ldots\ p_n \tag{5.1}$$

If we have a function applied to too few arguments, then we return the original expression, but we evaluate any strict arguments:

$$\begin{aligned} &\mathbf{Eval}[\![\ (f :: \sigma_1 \times \ldots \times \sigma_m \rightarrow \rho)\ p_1\ \ldots\ p_{n<m}\ ]\!] = f\ q_1\ \ldots\ q_n \\ &\quad \mathbf{where} \\ &\quad\quad q_i \quad = \quad \mathbf{Eval}[\![\ p_i\ ]\!], \quad \mathbf{if}\ \sigma_i = s \\ &\quad\quad\quad\quad = \quad p_i, \quad\quad\quad\quad \mathbf{otherwise} \end{aligned} \tag{5.2}$$

If we have a function applied to too many arguments, then we evaluate the inner application, that is the function applied to the exact number of arguments it needs, then apply the result of this to the remaining arguments.

$$\begin{aligned}&\textbf{Eval}[\![\ f^m\ p_1\ \ldots\ p_{n>m}\ ]\!] = \textbf{Eval}[\![\ r\ p_{m+1}\ \ldots\ p_n\ ]\!]\\ &\quad\textbf{where}\ r = \textbf{Eval}[\![\ f^m\ p_1\ \ldots\ p_m\ ]\!]\end{aligned} \tag{5.3}$$

Finally we have the case when we have a function applied to exactly the right number of arguments. We have to do three things: evaluate any strict arguments; apply the function using @; and evaluate the result if necessary.

$$\begin{aligned}&\textbf{Eval}[\![\ (f :: \sigma_1 \times \ldots \times \sigma_m \to \rho)\ p_1\ \ldots\ p_m\ ]\!] = r\quad \textbf{where}\\ &\begin{array}{rcll} r &=& e, & \textbf{if}\ \rho = s\\ &=& \textbf{Eval}[\![\ e\ ]\!], & \textbf{otherwise}\\ e &=& f@(q_1,\ldots,q_m) & \\ q_i &=& \textbf{Eval}[\![\ p_i\ ]\!], & \textbf{if}\ \sigma_i = s\\ &=& p_i, & \textbf{otherwise}\end{array}\end{aligned} \tag{5.4}$$

5.1.2 The Denotational Semantics with Explicit Updates

Implementations of functional languages use sharing and updating to avoid doing repeated work (see Section 2.3). Our first step towards an efficient operational semantics for the AAM is thus a denotational semantics of the AAM which makes explicit sharing and updating.

We associate each program with a variable $v \in V$, changing our syntax of Aladin programs to accommodate this:

$$P ::= D \mid F \mid V \mid V\ V \tag{5.5}$$

Applications now involve the application of one variable to another and a program can also be a variable, that is, an indirection to the actual value. The explicit naming of all parts of a program, in particular functions, means we can implement recursion directly.

These associations are defined in a mapping of variables to programs which we shall call the *heap.* A heap provides the context for the evaluation of a program, and thus a program evaluated with respect to one heap can yield a different result to the same program evaluated with respect to a different heap. The expression

$$\Gamma[x_0 \mapsto p_0, x_1 \mapsto p_1, \ldots, x_n \mapsto p_n]$$

denotes that in the heap Γ variable x_i maps to program p_i where $i \in 0, \ldots, n$. We may occasionally use the heap as a lookup-function, that is:

$$\begin{aligned} \Gamma\, x &= p, && \textbf{if } \Gamma[x \mapsto p] \\ &= \bot, && \textbf{otherwise} \end{aligned}$$

Since a program could be a variable we could have a chain of indirections:

$$\Gamma[x_0 \mapsto x_1, x_1 \mapsto x_2, \ldots, x_n \mapsto p \notin V]$$

In such cases we allow ourselves to 'short-circuit' the chain and write $\Gamma[x_0 \mapsto p]$. Note that the chain could in fact be a cycle:

$$\Gamma[x_0 \mapsto x_1, x_1 \mapsto x_2, \ldots, x_n \mapsto x_0]$$

Which is akin to writing something like:

```
foo = let
          x = y; y = z; z = x;
      in
          x
      endlet;
```

in Ginger. The result of evaluating any program which tries to access a variable in such a cycle will be $\bot$.

We will allow ourselves to abuse notation and let $\Gamma \cup \Gamma'$ denote the heap Γ updated with the mappings in heap Γ' with any clashes being resolved in the favour of those defined in Γ':

$$
\begin{array}{rcll}
(\Gamma \cup \Gamma')\ x & = & p, & \textbf{if } \Gamma'[x \mapsto p] \\
 & = & q, & \textbf{if } \Gamma[x \mapsto q] \\
 & = & \bot, & \textbf{otherwise}
\end{array}
$$

In particular, $\Gamma \cup \{x \mapsto p\}$ denotes a heap where x is associated with p and all other variables are associated to the programs that they were in Γ. The design of the semantics allows the heap to be represented as a global entity, which can be destructively updated, in an actual implementation, and this is the approach we use (see Section 5.3).

Since all programs are evaluated with respect to a heap, it follows that our primitive application meta-function must also execute with respect to a heap. The informal type of @ changes from:

$$P \times \ldots \times P \rightarrow P$$

to:

$$\mathit{Heap} \times V \times \ldots \times V \rightarrow (P \times \mathit{Heap}) \tag{5.6}$$

Note that while @ takes a heap and variables as arguments it returns a *program* and a heap (we need to return a heap as the function may want to create new objects in the heap).

Our evaluation philosophy changes from that used in the original semantics. Whereas before we returned a new expression which represented the evaluation, we now evaluate a program by updating the heap which provides the context for the program. The meta-function $\mathbf{E}$ provides the top-level interface to this procedure:

$$\mathbf{E}\ p\ \Gamma[x \mapsto p] = (\mathbf{U}\ \Gamma\ x)\ x \tag{5.7}$$

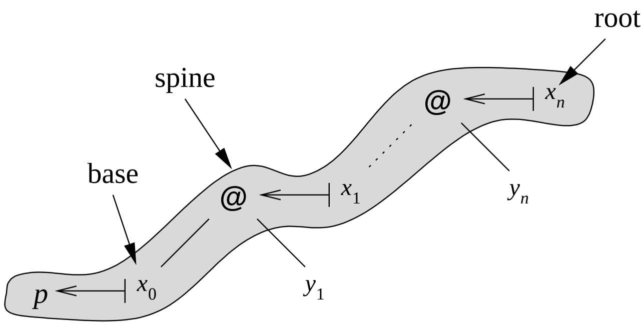


Figure 5.1: An generic unwound application

First we need to find a variable associated to the program we want to evaluate. Since a heap is not injective, there could be any number of variables associated with the same program but it doesn't matter which one we pick. Then we update the heap with respect to this variable, and finally look up the value of the variable in the updated heap.

The updating of the heap is done by the meta-function **U** which updates a given heap by evaluating the program referred to by a given variable, which we shall refer to as the *root* of the program. This evaluation is done by graph reduction as described in Section 2.3. If the root refers to an application then we need to unwind the spine of the application graph until we reach a non-application. In the heap

$$\Gamma[x_0 \mapsto p, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]$$

if x_n forms the root of the program then the $x_i, i \in 0, \ldots, n$ form the spine and x_0 forms the base (see Figure 5.1).

The four rules for **U** (5.8–5.11) directly reflect those of **Eval** (5.1–5.4). Suppose we have a data object applied to a number of arguments. Then since no evaluation needs to be done, no updating of the heap has to be done either:

$$\mathbf{U}\ \Gamma[x_0 \mapsto d \in D, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]\ x_n = \Gamma \tag{5.8}$$

If we have a function applied to too few arguments then we just need to evaluate the

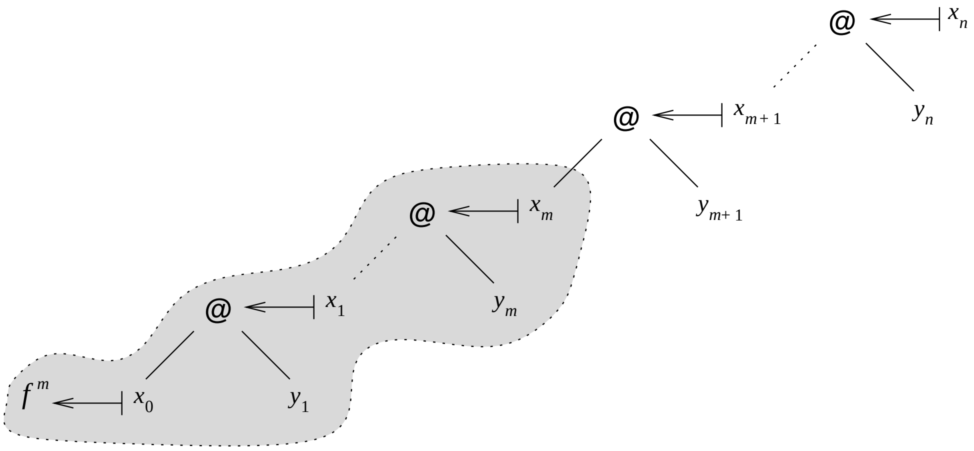


Figure 5.2: Application of a function to too many arguments

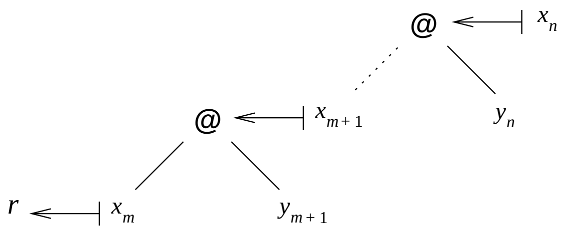


Figure 5.3: Application of a function to too many arguments after evaluation of inner application

strict ones. The **A** meta-function (see Rule 5.13) returns the heap resulting from evaluating the strict arguments of a function.

$$\mathbf{U}\ \Gamma[x_0 \mapsto f^m, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]\ x_{n<m} = \mathbf{A}\ \Gamma\ x_n \tag{5.9}$$

If we have a function f^m applied to too many arguments, $x_1, \ldots, x_m$ say, where $n > m$ (see Figure 5.2), then we first obtain the heap resulting from evaluating the inner application (the shaded part of the figure) the root of which is x_m. In this new heap, x_m will refer to the evaluated version of $f^m\ x_1\ \ldots\ x^m$, r say (see Figure 5.3). This result becomes the new base of our program (note that we may need to do some more unwinding if r is an application) and we continue updating.

$$\begin{aligned}&\mathbf{U}\ \Gamma[x_0 \mapsto f^m, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]\ x_{n>m} = \mathbf{U}\ \Gamma'\ x_n \\ &\quad \textbf{where } \Gamma' = \mathbf{U}\ \Gamma\ x_m\end{aligned} \tag{5.10}$$

If we have a function f applied to the exact number of arguments, we first evaluate any strict ones using $\mathbf{A}$ and primitively apply the function using @. The root of the application then needs to be updated with the result of @ and finally we may need to continue evaluation if f returns a lazy result.

$$
\begin{aligned}
&\mathbf{U}\,\Gamma \begin{bmatrix} x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho \\ x_1 \mapsto x_0\ y_1, \ldots, x_m \mapsto x_{m-1}\ y_m \end{bmatrix} x_m = \Gamma_4 \\
&\quad \textbf{where} \\
&\qquad \begin{aligned}
\Gamma_4 &= \Gamma_3, & &\textbf{if } \rho = s \\
&= \mathbf{U}\,\Gamma_3\ x_m, & &\textbf{otherwise} \\
\Gamma_3 &= \mathit{update}\ \Gamma_2\ x_m\ r \\
(r, \Gamma_2) &= f@(\Gamma_1, y_1, \ldots, y_m) \\
\Gamma_1 &= \mathbf{A}\,\Gamma\ x_m
\end{aligned}
\end{aligned}
\tag{5.11}
$$

For each of the applications that form the arguments of f, the argument component is extracted and passed as the corresponding argument to @. The function *update* takes care of the updating of a variable with a new value. First we define a couple of auxiliary functions:

$$
\mathit{reducible}\ \Gamma\ x_m = \left(\bigwedge_{i=1}^{m} \Gamma\ x_i = x_{i-1}\ yi \right) \wedge \Gamma\ x_0 = f^m
$$

and

$$
\begin{aligned}
\mathit{value}\ \Gamma\ p &= p, & &\textbf{if } p \notin V \\
&= \Gamma\ p, & &\textbf{otherwise}
\end{aligned}
$$

We now define *update* as:

$$
\begin{aligned}
&\mathit{update}\ \Gamma[x \mapsto x_1, x_1 \mapsto x_2, \ldots, x_n \mapsto p \notin V]\ x\ r = \Gamma \cup \{x_n \mapsto \mathit{val}\} \\
&\quad \textbf{where} \\
&\qquad \begin{aligned}
\mathit{val} &= r, & &\textbf{if } \mathit{reducible}\ (\mathit{value}\ r) \\
&= \mathit{value}\ r, & &\textbf{otherwise}
\end{aligned}
\end{aligned}
\tag{5.12}
$$

Note that it is the last variable in the indirection chain, x_n, that is updated, not the first one, x. This allows the result of the update to be propagated to any other variables which are referring to elements in this chain. Also, if r, the value we are updating with, is also a variable and the ultimate value of r is not reducible, then it is this value we use to update x_n, not r itself. The update value is also r if r is not a variable. This allows us to circumvent any indirection chains, possibly freeing the variables in this chain for garbage collection in an implementation. It is safe to do this if the the value of r is not reducible since r cannot be further evaluated and thus we cannot lose any shared work since there is no work to be done.

Finally we need to define the **A** meta-function which evaluates each argument. This proceeds by updating each argument in turn and passing the heap obtained by each evaluation into the recursive call to **U** used to evaluate the next argument. Although the definition here implies that arguments are evaluated left to right, this ordering is arbitrary and only adopted for syntactic convenience — any ordering could be adopted in practice. Arguments could even be evaluated concurrently provided appropriate care was taken.

$$
\begin{array}{l}
\mathbf{A}\,\Gamma \left[\begin{array}{l} x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho \\ x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n \end{array} \right] x_{n \leq m} = \Gamma_n \\
\quad \textbf{where} \\
\quad \begin{array}{rcll} \Gamma_i & = & \mathbf{U}\,\Gamma_{i-1}\ y_i, & \textbf{if } \sigma_i = s \\ & = & \Gamma_{i-1}, & \textbf{otherwise} \end{array}
\end{array}
\tag{5.13}
$$

5.1.3 The Operational Semantics

We represent the operational semantics as transition rules of the state of the AAM. This state is a quadruple:

$$(\mathit{Control}, \mathit{Stack}, \mathit{Heap}, \mathit{Dump})$$

where

- *Control* is a stack of instructions. These instructions are `EVAL`, `EVALARGS`, `EVALITH` and `APPLY`.

- *Stack* is a stack of variables, used as a working space to store the spine of an application when we are unwinding, with the head of the stack forming the base and the last element in the stack the root.

- *Heap* is a mapping from variables to programs as in the denotational case.

- *Dump* is a stack of *Control-Stack* pairs, used to hold previous states while we are evaluating a different part of the program.

Our machine is similar to the SECD machine and the G-Machine (see Section 2.5). The Stack, Control and Dump used by Aladin serve similar functions as their counterparts in the SECD and G machines. The Aladin Heap is more like the G-Machine heap, which like Aladin's heap is a global object where the graph being evaluated is held, than the SECD machine's Environment which serves a local mapping between values and variables dependent on the expression being evaluated. The key difference between Aladin and the SECD machine and the G-Machine is that arguments and the results of Aladin functions may be strict or lazy. In the SECD machine all arguments are treated as strict and in the G-machine all arguments are lazy, and neither of these machines has the concept of a strict or lazy result and must evaluate the results of all functions.

The meta-function **F** evaluates a program using the state-transition rules (see below) with respect to a given heap (cf. **E**):

$$\begin{aligned} &\mathbf{F}\ p\ \Gamma[x \mapsto p] = \Gamma' x \\ &\quad \textbf{where } (\langle\rangle, S, \Gamma', \langle\rangle) = \mathbf{T}\ (\langle \mathtt{EVAL} \rangle, \langle x \rangle, \Gamma, \langle\rangle) \end{aligned} \tag{5.14}$$

The meta-function **T** repeatedly applies the state-transition rules (Rules 5.15–5.23 below) to a state until no more apply, returning the final state. So, if we have a transition sequence:

$$s_1 \Longrightarrow s_2 \Longrightarrow \ldots \Longrightarrow s_n$$

and no rules apply to s_n then $\mathbf{T}\ s_i = s_n, i \in 1, \ldots, n$.

We now have to give the transition rules for a state, starting from the initial state used as the argument to $\mathbf{T}$. If none of these rules applies then the machine terminates.

The first case is when the head of the stack references a data object, that is, when we have a data object applied to a number of arguments, cf. Rules 5.1 and 5.8. We need to make no changes and no further work can be done in this state. If the dump is non-empty we restore it (by virtue of Rule 5.23), else we terminate as no more rules apply.

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle\rangle \\ \langle x_0, x_1, \ldots, x_n \rangle & & \langle\rangle \\ & \Longrightarrow & \\ \Gamma[x_0 \mapsto d \in D] & & \Gamma \\ \Delta & & \Delta \end{array} \tag{5.15}$$

N.B. `EVAL` can only occur in the control stack as the sole element.

If the head of the stack is an application, we need to unwind and carry on evaluating:

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle \texttt{EVAL} \rangle \\ x_1 : S & & x_0 : x_1 : S \\ & \Longrightarrow & \\ \Gamma[x_1 \mapsto x_0\ y_1] & & \Gamma \\ \Delta & & \Delta \end{array} \tag{5.16}$$

If the head of the stack refers to a function of arity m and there are less than m other elements on the stack, we have the case of a function applied to too few arguments, cf. Rules 5.2 and 5.9. In this case, we need to trigger the evaluation of the strict arguments which is done using the `EVALARGS` n instruction (Rule 5.20), where n is the number of other elements on the stack.

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle \texttt{EVALARGS}\ \ n \rangle \\ \langle x_0, x_1, \ldots, x_{n<m} \rangle & \Longrightarrow & \langle x_0, y_1, \ldots, y_n \rangle \\ \Gamma[x_0 \mapsto f^m, x_i \mapsto x_{i-1}\ y_i] & & \Gamma \\ \Delta & & \Delta \end{array} \tag{5.17}$$

Note that the arguments of the function are extracted from the applications in which they reside.

If the head of the stack refers to a function of arity m and there are strictly more than m other elements on the stack, we have the case of a function applied to too many arguments, cf. Rules 5.3 and 5.10. We need to evaluate the inner application, which is done in a new state. After this, we evaluate the result of evaluating the inner application applied to the rest of the arguments. This is done by saving the control-stack pair that represents this application on the dump. They will be restored from the dump when evaluation of the outer application has completed.

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle \texttt{EVAL} \rangle \\ \langle x_0, x_1, \ldots, x_{n>m} \rangle & \Longrightarrow & \langle x_0, x_1, \ldots, x_m \rangle \\ \Gamma[x_0 \mapsto f^m] & & \Gamma \\ \Delta & & (\langle \texttt{EVAL} \rangle, \langle x_m, \ldots, x_n \rangle) : \Delta \end{array} \tag{5.18}$$

The final case for `EVAL` is when we have a function applied to exactly the right number of arguments (this, with Rule 5.22, reflects Rules 5.4 and 5.11 in the denotational cases). We need to evaluate any strict arguments and apply the function.

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle \texttt{EVALARGS}\ \ m, \texttt{APPLY} \rangle \\ \langle x_0, x_1, \ldots, x_m \rangle & \Longrightarrow & \langle x_0, y_1, \ldots, y_m, x_m \rangle \\ \Gamma[x_0 \mapsto f^m, x_i \mapsto x_{i-1}\ y_i] & & \Gamma \\ \Delta & & \Delta \end{array} \tag{5.19}$$

Note that as well as unpacking the arguments to the function, we keep x_m, the root of the program, as the last element of the stack. It is this variable that will be updated with the result of applying f to its arguments. In particular, if $m = 0$, that is f is a Constant Applicative Form (CAF), then it is the variable referring to the function itself that will be updated with the result.

We now need to give the state transition rules for the other instructions. First we have `EVALARGS` which triggers the evaluation of any strict arguments, cf. the **A** meta-function (Rule 5.13).

$$
\begin{array}{lcl}
\texttt{EVALARGS}\ n : C & & e_1 \mathbin{++} \ldots \mathbin{++} e_n \mathbin{++} C \\
x_0 : y_1 : \ldots : y_{n \leq m} : S & \Longrightarrow & x_0 : y_1 : \ldots : y_n : S \\
\Gamma[x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho] & & \Gamma \\
\Delta & & \Delta
\end{array}
\tag{5.20}
$$

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\textbf{where}} \\
e_i & = & \langle \texttt{EVALITH}\ i \rangle, & \textbf{if}\ \sigma_i = s \\
 & = & \langle \rangle, & \textbf{otherwise}
\end{array}
$$

The `EVALITH` i triggers the evaluation of the ith element of the stack (where the head of the stack is the 0th element) in a new state:

$$
\begin{array}{lcl}
\texttt{EVALITH}\ i : C & & \langle \texttt{EVAL} \rangle \\
x_0 : x_1 : \ldots : x_i : S & \Longrightarrow & \langle x_i \rangle \\
\Gamma & & \Gamma \\
\Delta & & (C, x_0 : x_1 : \ldots : x_i : S) : \Delta
\end{array}
\tag{5.21}
$$

By evaluating each argument in its own state we open up the possibility of evaluating all the arguments that need evaluating concurrently, a process which would be more difficult if we used the same state as the original application.

The APPLY instruction initiates the primitive application of a function to its argument, updating the root of the application and evaluating the result if necessary.

$$
\begin{array}{lcl}
\langle \texttt{APPLY} \rangle & & C \\
\langle x_0, y_1, \ldots, y_m, root \rangle & \Longrightarrow & \langle root \rangle \\
\Gamma[x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho] & & update\ \Gamma'\ root\ r \\
\Delta & & \Delta
\end{array}
\tag{5.22}
$$

$$
\begin{array}{rcll}
\multicolumn{4}{l}{\textbf{where}} \\
(r, \Gamma') & = & f@(\Gamma, y_1, \ldots, y_m) & \\
C & = & \langle\rangle, & \textbf{if } \rho = s \\
 & = & \langle \texttt{EVAL} \rangle, & \textbf{otherwise}
\end{array}
$$

where the function *update* is the one defined in equation 5.12.

The final rule concerns the case when the control stack is empty but we have previous states on the dump. In this case, we restore the first previous state and continue. This is similar to returning from a procedure call in an imperative language.

$$
\begin{array}{lcl}
\langle\rangle & & C \\
S' & \Longrightarrow & S \\
\Gamma & & \Gamma \\
(C, S) : \Delta & & \Delta
\end{array}
\tag{5.23}
$$

Note that by throwing away the old stack we do not in any way return a value.

5.1.4 An Example of the Operational Semantics

Consider again the expression *square* $(3 + 4)$ first encountered in Section 2.3. Representing addition and multiplication as the prefix functions *plus* and *times* respectively, both of strictness $s \times s \to s$, we can define *square* as:

$$
\begin{aligned}
& square :: l \to l \\
& square@(\Gamma, x) = (r, \Gamma') \\
& \quad \textbf{where} \\
& \quad \Gamma' \;=\; \Gamma \cup \{v \mapsto t\ x, r \mapsto v\ x\} \\
& \quad\ \ t \;=\; \Gamma\ times
\end{aligned} \tag{5.24}
$$

where v and r are variables undefined in Γ. Note that we have made *square* lazy in its argument for illustration purposes — it could have safely been made strict — and that the result of *square* is the application $t\ x\ x$ and not the result of this application. Assume that we have an initial heap Γ:

$$\Gamma = \left\{ \begin{array}{l} a \mapsto times, b \mapsto square, c \mapsto plus, d \mapsto 3, \\ e \mapsto 4, f \mapsto c\ d, g \mapsto f\ e, h \mapsto b\ g \end{array} \right\}$$

then we need to update this Γ by evaluating h. Our initial machine state (the one passed to $\mathbf{T}$) is: $(\langle \texttt{EVAL} \rangle, \langle h \rangle, \Gamma, \langle\rangle)$ giving us the transition sequence:

Control	**Stack**	**Heap**	**Dump**
`EVAL`	$\langle h \rangle$	Γ	$\langle\rangle$

$\Longrightarrow$ (by Rule 5.16)

`EVAL`	$\langle b, h \rangle$	Γ	$\langle\rangle$

$\Longrightarrow$ (by Rule 5.19)

`EVALARGS` 1	$\langle b, g, h \rangle$	Γ	$\langle\rangle$
`APPLY`			

$\Longrightarrow$ (by Rule 5.20 and strictness of $\Gamma\ b = square$)

`APPLY`	$\langle b, g, h \rangle$	Γ	$\langle\rangle$

$\Longrightarrow$ (by Rule 5.22 and definition of *square*)

`EVAL`	$\langle h \rangle$	Γ_1	$\langle\rangle$

where $\Gamma_1 = \Gamma \cup \{v \mapsto a\ g, r \mapsto v\ g, h \mapsto r\}$

$\Longrightarrow$ (by Rule 5.16)

EVAL	$\langle r, h\rangle$	Γ_1	$\langle\rangle$

$\Longrightarrow$ (by Rule 5.16)

EVAL	$\langle v, r, h\rangle$	Γ_1	$\langle\rangle$

$\Longrightarrow$ (by Rule 5.16)

EVAL	$\langle a, v, r, h\rangle$	Γ_1	$\langle\rangle$

$\Longrightarrow$ (by Rule 5.19)

EVALARGS 2	$\langle a, g, g, h\rangle$	Γ_1	$\langle\rangle$
APPLY			

$\Longrightarrow$ (by Rule 5.20 and strictness of $\Gamma\ a = \mathit{times}$)

EVALITH 1	$\langle a, g, g, h\rangle$	Γ_1	$\langle\rangle$
EVALITH 2			
APPLY			

$\Longrightarrow$ (by Rule 5.21)

EVAL	$\langle g\rangle$	Γ_1	$\langle\delta\rangle$

where $\delta = (\langle \texttt{EVALITH 2}, \texttt{APPLY}\rangle, \langle a, g, g, h\rangle)$

$\Longrightarrow$ (by Rule 5.16)

EVAL	$\langle f, g\rangle$	Γ_1	$\langle\delta\rangle$

$\Longrightarrow$ (by Rule 5.16)

EVAL	$\langle c, f, g\rangle$	Γ_1	$\langle\delta\rangle$

$\Longrightarrow$ (by Rule 5.19)

EVALARGS 2	$\langle c, d, e, g\rangle$	Γ_1	$\langle\delta\rangle$
APPLY			

$\Longrightarrow$ (by Rule 5.20 and strictness of $\Gamma\ c = plus$)

`EVALITH 1`	$\langle c, d, e, g\rangle$	Γ_1	$\langle\delta\rangle$
`EVALITH 2`			
`APPLY`			

$\Longrightarrow$ (by Rule 5.21)

`EVAL`	$\langle d\rangle$	Γ_1	$\langle\delta_1, \delta\rangle$

where $\delta_1 = (\langle \texttt{EVALITH 2}, \texttt{APPLY}\rangle, \langle c, d, e, g\rangle)$

$\Longrightarrow$ (by Rule 5.15)

$\langle\rangle$	$\langle\rangle$	Γ_1	$\langle\delta_1, \delta\rangle$

$\Longrightarrow$ (by Rule 5.23)

`EVALITH 2`	$\langle c, d, e, g\rangle$	Γ_1	$\langle\delta\rangle$
`APPLY`			

$\Longrightarrow$ (by Rule 5.21)

`EVAL`	$\langle e\rangle$	Γ_1	$\langle\delta_2, \delta\rangle$

where $\delta_2 = (\langle \texttt{APPLY}\rangle, \langle c, d, e, g\rangle)$

$\Longrightarrow$ (by Rule 5.15)

$\langle\rangle$	$\langle\rangle$	Γ_1	$\langle\delta_1, \delta\rangle$

$\Longrightarrow$ (by Rule 5.23)

`APPLY`	$\langle c, d, e, g\rangle$	Γ_1	$\langle\delta\rangle$

$\Longrightarrow$ (by Rule 5.22 and definition of $plus$)

$\langle\rangle$	$\langle g\rangle$	Γ_2	$\langle\delta\rangle$

where $\Gamma_2 = \Gamma_1 \cup \{g \mapsto 7\}$

$\Longrightarrow$ (by Rule 5.23)

`EVALITH 2`	$\langle a, g, g, h\rangle$	Γ_2	$\langle\rangle$

APPLY

$\Longrightarrow$ (by Rule 5.21)

EVAL	$\langle g \rangle$	Γ_2	$\langle \delta_3 \rangle$

where $\delta_3 = (\texttt{APPLY}, \langle a, g, g, h \rangle)$

$\Longrightarrow$ (by Rule 5.15)

$\langle\rangle$	$\langle g \rangle$	Γ_2	$\langle \delta_3 \rangle$

$\Longrightarrow$ (by Rule 5.23)

APPLY	$\langle a, g, g, h \rangle$	Γ_2	$\langle\rangle$

$\Longrightarrow$ (by Rule 5.22 and definition of *times*)

$\langle\rangle$	$\langle h \rangle$	Γ_3	$\langle\rangle$

where $\Gamma_3 = \Gamma_2 \cup \{h \mapsto 49\}$

In the heap component of the final state, the variable which referred to the root of the original application, h, now refers to 49, which is the result of *square* (*plus* 3 4).

5.2 A Scripting Language for Aladin

To enable us to write programs, we need a top-level language which we can program in. This language should let us import primitives from a variety of sources and combine them into programs. The syntax of this scripting language is closely tied to our implementation of the AAM and choices in one will be reflected in each other. We necessarily lose some of the purity of the abstract machine, but we shall keep this to a minimum.

We choose to implement a version of the AAM, using the above semantic rules, using Java. The approach is similar to the one we used for our Ginger compiler in the previous chapter. Each Aladin script will be converted into a single Java class. Choosing Java as our implementation language for Aladin has the same advantages as choosing it for Ginger with the additional benefit that Java can interface with other languages, in particular C and C++, using the JNI [30].

For the syntax of the script, we adopt a subset of Ginger, omitting local function and lambda expressions, with added constructs for specifying the strictness of functions and an enhanced syntax for importing primitives. The extended BNF of this language is as follows:

$$\begin{array}{rcl}
\langle \textit{script} \rangle & ::= & \langle \textit{packagedec} \rangle ?\ \langle \textit{decl} \rangle * \\
\\
\langle \textit{packagedec} \rangle & ::= & \texttt{package}\ \langle \textit{package} \rangle \texttt{;} \\
\\
\langle \textit{decl} \rangle & ::= & \langle \textit{def} \rangle \mid \langle \textit{import} \rangle \mid \langle \textit{id} \rangle\ \langle \textit{strictsig} \rangle \\
\\
\langle \textit{import} \rangle & ::= & \langle \textit{im} \rangle\ (\langle \textit{id} \rangle\ \langle \textit{strictsig} \rangle ?)*\ \texttt{;} \mid \texttt{import?}\ \langle \textit{id} \rangle *\ \texttt{;} \\
\langle \textit{im} \rangle & ::= & \texttt{import}\ \langle \textit{class} \rangle \mid \texttt{importc}\ \texttt{"}\langle \textit{filename} \rangle\texttt{"} \mid \texttt{importg}\ \langle \textit{class} \rangle \\
\\
\langle \textit{strictsig} \rangle & ::= & \texttt{::}\ \langle \textit{argstrict} \rangle ?\ \texttt{->}\ \langle \textit{strict} \rangle \\
\langle \textit{argstrict} \rangle & ::= & \langle \textit{strict} \rangle\ (\texttt{*}\ \langle \textit{strict} \rangle)* \\
\langle \textit{strict} \rangle & ::= & \texttt{s} \mid \texttt{l} \\
\\
\langle \textit{def} \rangle & ::= & \langle \textit{id} \rangle\ \langle \textit{id} \rangle *\ \texttt{=}\ \langle \textit{prog} \rangle \texttt{;} \\
\langle \textit{prog} \rangle & ::= & \langle \textit{let} \rangle \mid \langle \textit{simpleprog} \rangle + \\
\langle \textit{let} \rangle & ::= & \texttt{let}\ \langle \textit{id} \rangle\ \texttt{=}\ \langle \textit{prog} \rangle\ \texttt{in}\ \langle \textit{prog} \rangle\ \texttt{endlet} \\
\langle \textit{simpleprog} \rangle & ::= & \langle \textit{javaid} \rangle \mid \langle \textit{data} \rangle \mid \texttt{(}\langle \textit{prog} \rangle\texttt{)} \\
\langle \textit{data} \rangle & ::= & \langle \textit{int} \rangle \mid \langle \textit{float} \rangle \mid \langle \textit{bool} \rangle \mid \langle \textit{char} \rangle \mid \langle \textit{string} \rangle
\end{array}$$

$$\langle \mathit{javaid} \rangle \ ::= \ (\langle \mathit{id} \rangle \ \ .)* \ \langle \mathit{id} \rangle$$

$$\langle \mathit{package} \rangle \ ::= \ \langle \mathit{javaid} \rangle$$

$$\langle \mathit{class} \rangle \ ::= \ (\langle \mathit{package} \rangle \ .)? \ \langle \mathit{id} \rangle$$

where *id* is any valid Java identifier containing *no* period (‘`.`’) separators, and *filename* is a legal file name, the syntax of which is dependent on the operating system we are running under. An Aladin script is thus an optional package declaration followed by a number of declarations. These declarations are:

Import declarations used to import primitives from a class.

Strictness declarations used to specify the strictness of a primitive.

Definitions used to specify a primitive function by giving a definition which constructs a graph from its arguments and other primitives.

Each script file will be appended with the suffix ‘`.as`’. As with our Ginger compiler, we compile scripts written in this language into a Java class file, translating each definition into a static Java method, and generating Java code to deal with the importing of functions and the setting of their strictnesses.

5.2.1 Package Declarations

A package declaration, which must be the first statement of an Aladin script, follows the same syntax as package declarations in Java and has the same meaning. All primitives defined in this file will be placed inside a class with will be in the package declared by the user. If no package declaration is made then the empty package is used. For instance the declaration:

```
package fp.aladin;
```

places all the following definitions in the package `fp.aladin`.

5.2.2 Import Declarations

The various import declarations allow the user to import primitives from various classes or files. These declarations have an optional list of primitives we wish to import from that source, along with optional strictness signatures (see Section 5.2.3 for an example). The `import` declaration is used to import primitives from actual Java classes and Java classes that were obtained from compiling Aladin scripts. The `importg` declaration is used to import Java classes that resulted from compiling Ginger programs and `importc` is used to import primitives from library files generated by compiling C and C++ programs. Finally, the `import?` declaration is used to declare primitives that the script expects the script importing this one to import at some stage. This declaration is only used in library files which can use more than one set of basic primitives, for example, either boolean or fuzzy primitives, or partial or non-partial primitives. The mechanics of importing are discussed in Section 5.4.6.

5.2.3 Strictness Signatures

A strictness declaration declares the strictness of a primitive function's arguments and result. If the primitive in question is defined in the script, then this signature can appear anywhere in the script but *must* be given; if the primitive is imported then a strictness signature can be given in the import declaration in which case it overrides the original signature (if there was one).

An argument or result is either strict, denoted by an `s`, or lazy, denoted by an `l`. The arguments of a primitive defined in a strictness signature are separated using the symbol `*`, and the arguments are separated from the result by the symbol `->`, which is present even if the primitive in question takes no arguments. It is an error to give a strictness signature to a non-function, or to a function which is neither imported nor has no definition. As an example, we can give the strictness of the primitives imported from the class `fp.aladin.lib.List`, our standard representation of lists:

```
import fp.aladin.lib.List
    _op_list_cons   :: l * l -> s  // :
    _op_list_empty  :: -> s        // []
    _op_list_index  :: s * s -> s  // !
    _op_list_length :: s -> s      // #
    isEmpty         :: s -> s
    hd              :: s -> l
    tl              :: s -> l
```

Thus `_op_list_cons` takes two lazy arguments and returns a strict result; the function `_op_list_empty` takes no arguments and returns a strict result; `_op_list_index` takes two strict arguments and returns a strict result; `_op_list_length` and `isEmpty` take a strict result and return a strict result; and finally `hd` and `tl` take a strict argument and return a lazy result.

The number of arguments referred to in a strictness signature refer to the number of *explicit* arguments given, not to any implied by η-conversion. For instance, if we define:

```
hd1 = hd;
```

then the strictness signature of `hd1` is `-> s` (since it takes no arguments and returns a function) and not that of `hd` (`s -> l`).

5.2.4 Definitions

Aladin definitions follow the normal syntactic style found in functional languages, in particular the style of Ginger. The five basic data types — integers, reals, strings, characters and booleans — follow the normal syntax. These types are the only ones that are built into our compiler that are not in the Aladin semantics.

Applications are written using juxtaposition (and associate to the left) and we allow the use of infix operators, standard functional list notation (see below) and simple local variable definitions *via* the `let` declaration. The use of binary prefix functions as infix ones can be achieved by surrounding the function in back-quotes as in Haskell. We use the Ginger `if-then-elsif-else-endif` notation for conditionals (again see below for more details). For example, the Fibonacci function can be defined in Aladin as:

```
fib :: s -> l
fib n =
    if n == 0 then 1
    else
        let
            f1 = fib (n - 1);
            f2 = fib (n - 2);
        in
            f1 + f2;
        endlet
    endif;
```

See Section 5.2.3 for more details on strictness signatures.

This definition merely takes its arguments and produces the graph described in its left-hand side, hence we have to return a lazy result as we want the graph

<table>
<tr><th>Prec.</th><th>Infix form</th><th>Prefix form</th><th>Method</th></tr>
<tr><td>0</td><td><code>++</code></td><td><code>_op_list_cat</code></td><td>N/A</td></tr>
<tr><td></td><td><code>:</code></td><td><code>_op_list_cons</code></td><td>N/A</td></tr>
<tr><td>1</td><td><code>|</code></td><td><code>_op_or</code></td><td><code>Logical.or</code></td></tr>
<tr><td>2</td><td><code>&</code></td><td><code>_op_and</code></td><td><code>Logical.and</code></td></tr>
<tr><td>3</td><td><code>==</code></td><td><code>_op_eq</code></td><td><code>Object.equals</code></td></tr>
<tr><td></td><td><code>~=</code></td><td><code>_op_ne</code></td><td><code>Object.equals</code></td></tr>
<tr><td></td><td><code><</code></td><td><code>_op_lt</code></td><td><code>Orderable.lt</code></td></tr>
<tr><td></td><td><code><=</code></td><td><code>_op_le</code></td><td><code>Orderable.le</code></td></tr>
<tr><td></td><td><code>></code></td><td><code>_op_gt</code></td><td><code>Orderable.gt</code></td></tr>
<tr><td></td><td><code>>=</code></td><td><code>_op_ge</code></td><td><code>Orderable.ge</code></td></tr>
<tr><td>4</td><td><code>+</code></td><td><code>_op_plus</code></td><td><code>Num.plus</code></td></tr>
<tr><td></td><td><code>-</code></td><td><code>_op_minus</code></td><td><code>Num.minus</code></td></tr>
<tr><td>5</td><td><code>*</code></td><td><code>_op_times</code></td><td><code>Num.times</code></td></tr>
<tr><td></td><td><code>/</code></td><td><code>_op_divide</code></td><td><code>Num.div</code></td></tr>
<tr><td></td><td><code>%</code></td><td><code>_op_modulus</code></td><td><code>Num.mod</code></td></tr>
<tr><td>6</td><td><code>^</code></td><td><code>_op_exp</code></td><td><code>Num.exp</code></td></tr>
<tr><td>7</td><td><code>.</code></td><td><code>_op_compose</code></td><td>N/A</td></tr>
<tr><td>8</td><td><code>!</code></td><td><code>_op_list_index</code></td><td>N/A</td></tr>
<tr><td>9</td><td><code>~</code></td><td><code>_op_not</code></td><td><code>Logical.not</code></td></tr>
</table>

Table 5.1: Aladin infix operators and their prefix form

to be evaluated. No evaluation or compiler optimisations are performed, these being handled by the evaluation mechanism defined in the previous section. The construction and evaluation of programs are completely divorced in the scripting language.

We allow the use of infix operators, though these are converted to prefix functions during the parsing stage according to the precedence rules defined in Table 5.1 (we include the already prefix not, `~`, for completion). Note that in the case of '`.`', `foo . bar`, with spaces around the period, is function composition but `foo.bar`, with no spaces, is a single identifier. Also, `-` is used for both binary minus and to indicate a negative *constant*, but not for general unary minus: `-3` is legal but `-x` is not (we provide the primitive `neg` to negate a general program).

Some of these operators are overloaded, the resolution being done at runtime.

In these cases, the prefix form only exists to unwrap the arguments and call the interface method. Any user-defined class which implements the method specified in the fourth column adds another overloading to the prefix form. For instance, if we have a class `Foo` which implements the `Orderable` interface then we can use the comparison operators on objects of that type. Overloading is explained in more detail in Section 5.4.4.

As with infix operators, the Ginger style `if` syntax, which is adopted to reduce excessive bracketing, is converted to a prefix function application. The expression:

$$\texttt{if}\ a_1\ \texttt{then}\ c_1\ \texttt{elsif}\ a_2\ \texttt{then}\ c_2 \ldots\ \texttt{elsif}\ a_n\ \texttt{then}\ c_n\ \texttt{else}\ d\ \texttt{endif}$$

is converted into the program:

$$\texttt{\_op\_if}\ a_1\ c_1\ (\texttt{\_op\_if}\ a_2\ c_2\ (\ldots(\texttt{\_op\_if}\ a_n\ c_n\ d)\ldots))$$

Here `_op_if` is the boolean conditional. This isn't built into Aladin, instead the user provides their own implementation. One such implementation can be seen in Section 5.4.1.

We also allow the use of the standard square-bracket notation to denote lists, with the list $\texttt{[}x_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{]}$ being translated into multiple applications of the list constructor function `_op_list_cons`:

$$\texttt{\_op\_list\_cons}\ x_1\ (\ldots(\texttt{\_op\_list\_cons}\ x_n\ \texttt{\_op\_list\_empty})\ldots)$$

We also allow the use of the 'dot-dot' notation, again converting to a prefix function application defined in Table 5.2. The user can provide any implementation of lists that they want, as long as they provide implementations of the list constructor and deconstructor functions. We describe an implementation similar to the one normally used in functional languages such as Ginger and Haskell in Section 5.3.2.

As with lists, tuples are converted into applications of a constructor function to the components of the tuple. The tuple $\texttt{(}x_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{)}$ is converted to the appli-

Dot-dot form	Prefix form
`[]`	`_op_list_empty`
`[` a `.. ]`	`from` a
`[` a `..` b `]`	`fromTo` a b
`[` a `,` b `.. ]`	`fromThen` a b
`[` a `,` b `..` c `]`	`fromThenTo` a b c

Table 5.2: Aladin 'dot-dot' lists and their prefix form

cation `mkTuple_`n x_1 $\ldots$ x_n. Again, the user can provide their own implementation and one such implementation is presented in in Section 5.3.2.

5.3 Representation and Evaluation of Aladin Programs

We first need a Java representation of the four components of the AAM defined in the semantics. The Control is simply the code structure of the evaluation mechanism. The Stack is implemented as an array which can grow on demand (we could use a Java `Stack` or `Vector` but we use an array as it enables us to perform certain operations more efficiently and avoids excessive casting). The Heap is a combination of the object heap of the JVM plus a hash table in which we can look up functions by their name. Finally, the Dump, which in the semantics is used to simulate function calls and recursion, is implemented by actual functional calls and recursion in our implementation.

The definition of programs as given in Section 5.5 suggests that we implement programs as a class hierarchy as seen in Figure 5.4. Some of these classes are used only by the compiler: `Aladin` is the class containing the parser; `AladinCompiler` contains the top-level compilation methods; the classes `CompileTimeFunction` and `CompileTimeApp` are compile-time representations of functions and applications which need to contain more information than their run-time brethren; `Ident` is a compile-time identifier (function or variable name); and `Let` and `LocalBlock` are

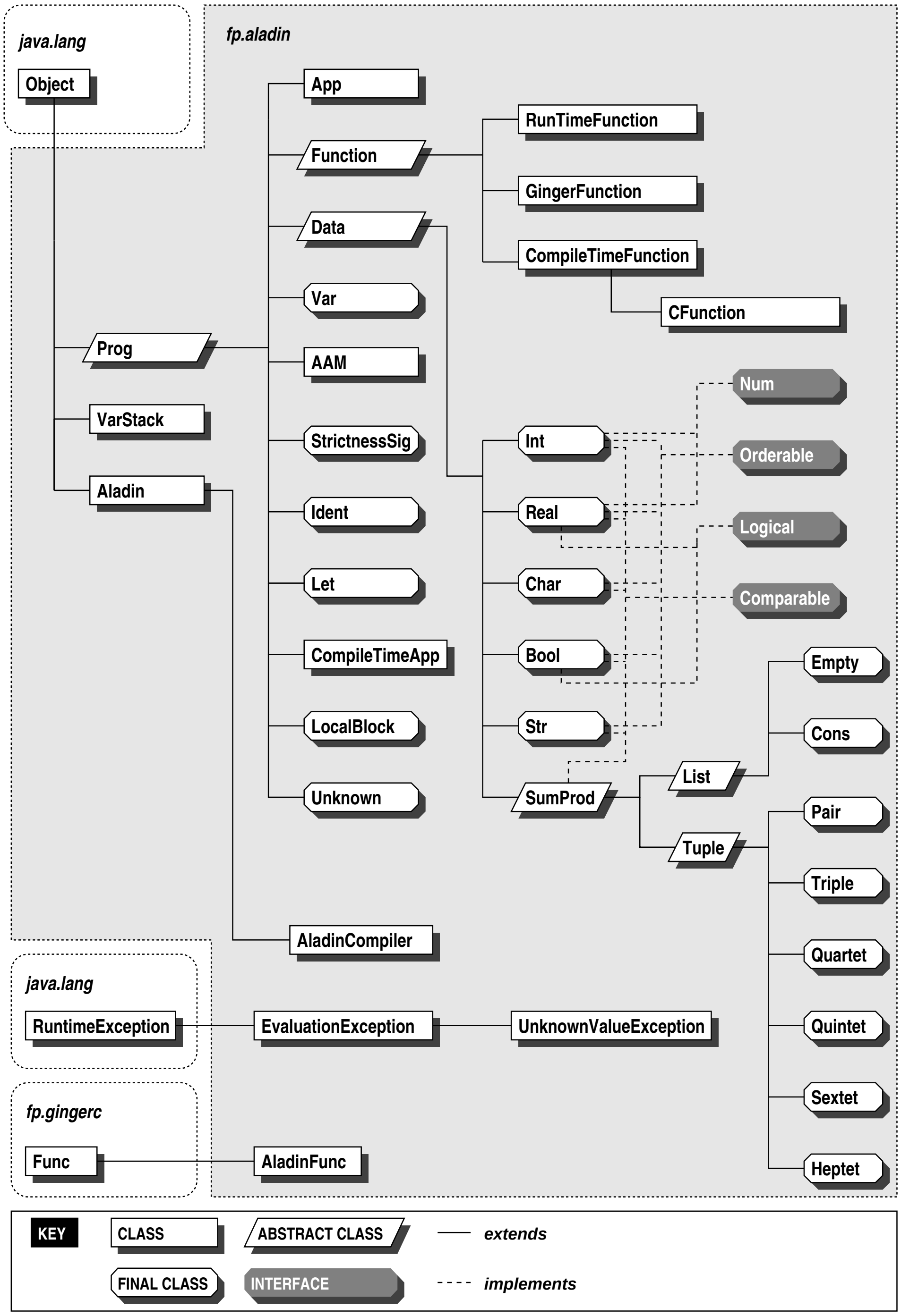


Figure 5.4: The `fp.aladin` package

used to store local variable declarations and will be disappear after compilation. In addition, the class Unknown and the exception class UnknownValueException are only used when partially evaluating programs (Chapter 6).

5.3.1 Aladin Programs

All programs are represented as subclasses of the abstract class Prog. Variables are represented using the Var class:

```
public final class Var extends Prog {
  public Prog value;

  // ...
}
```

where the field value holds the value of the variable. Data objects are represented by instances of subclasses of the abstract class Data. In particular, the five basic types — integers, reals, characters, booleans and strings — are represented by the classes Int, Real, Char, Bool and Str respectively. These classes have a field value containing the actual value of the data object of type long, double, char, boolean or String respectively. Unlike in our Ginger compiler we do not use the wrapper classes provided in the package java.lang. This is slightly inefficient (especially in the case of strings) but leads to a more elegant and object-oriented design. Functions are represented by instances of subclasses of the Function class:

```
public abstract class Function extends Prog {
  public String pack = "";
  public String cl = "";
  public String short_name = "";
  public int arity = -1;
```

```
  public StrictnessSig strictness;

  public final boolean strictIn(int i) {
    // ...
  }

  public final boolean hasStrictResult() {
    // ...
  }

  public abstract Prog primApply(Var[] args);

  // ...
}
```

Java naming conventions are used in naming Aladin functions: each function has a short name (`short_name`), class (`cl`) and package (`pack`). In use, the short name refers to the last such function with that short name that was loaded. This can be qualified with the class and maybe the package name if necessary.

The other fields and methods of the `Function` class are fairly self-explanatory. The field `arity` represents the arity arity of the function and `strictness` its strictness signature. The method `strictIn` returns whether the function is strict in a certain argument (indexing starting at 1) and `hasStrictResult` returns whether the function has a strict result. Finally, `primApply` primitively applies the function to the given arguments (cf. @).

Finally, applications are represented as instances of the `App class`:

```
public class App extends Prog {
```

```
  public Var functor;
  public Var arg;

  // ...
}
```

5.3.2 Data Structures

In our implementation of Aladin, data structures are based on the sum-of-product type used in ML, Miranda and Haskell and described in [86]. We introduce a class, `SumProd`, which will be the superclass of our list, tuple and other similar classes:

```
public abstract class SumProd extends Data implements Comparable {
  protected Var[] fields;

  public abstract String getConstructorName();

  // ...
}
```

The `Comparable` interface, used by Aladin to overload the equals and not equals functions, is described in Section 5.4.4.

The array `fields` holds the fields of the structure, for example, in a cons object the array would have length 2 with the first element holding the head and the second the tail. The method `getConstructorName` returns the full name of the function used to construct instances of that class. For instance, we can implement lists using the following classes:

```
public abstract class List extends SumProd {
  public static final Empty EMPTY = new Empty();
```

```
  public static Prog _op_list_empty() {
    return EMPTY;
  }

  public static Prog _op_list_cons(Var h, Var t) {
    return new Cons(h, t);
  }

  public static Var hd(Var l) {
    List list = (List) l.get();

    if (list instanceof Empty)
      throw new EvaluationException("Can't take hd of empty list");
    else
      return list.fields[0];
  }

  // ...
}

final class Empty extends List {
  public Empty() {
    fields = empty_fields;
  }

  public String getConstructorName() {
```

```
    return "fp.aladin.lib.List._op_list_empty";
  }

  // ...
}

final class Cons extends List {
  public Cons(Var head, Var tail) {
    fields = new Var[] {head, tail};
  }

  public String getConstructorName() {
    return "fp.aladin.lib.List._op_list_cons";
  }

  // ...
}
```

Here `empty_fields` is a final static member of the `SumProd` class which is an array of zero `Var` objects.

As well as acting as the superclass of `Empty` and `Cons`, the `List` class contains the definition of primitives acting on lists. The method `_op_list_empty` is the alphanumeric form of `[]` and `_op_list_empty` is the alphanumeric form of `:` (see Section 5.2.4). These methods call the relevant constructor of the `Empty` and `Cons` classes respectively. The method `hd` returns the head of a list; after casting the value of its argument to a `List` it then determines whether the value is an instance of the `Empty` class, in which case the head does not exist, or the only other alternative, the

`Cons` class, in which case we just return the first component of the `fields` array. Other list primitives can be defined similarly.

For tuples, we have a superclass `Tuple` from which we subclass specific tuples, for example the `Pair` class represents pairs. The user is free to subclass the `SumProd` class to create his or her own data structures, for example, trees and other container types. Note that the creation and manipulation of the these types is done entirely by function application: there is no equivalent of Haskell's `data` declaration and pattern matching, for instance.

Relating this implementation back to the sum-of-product concept, we have the `fields` array representing the 'product' part, the class itself as the constructor tag, and subclassing used to sum types.

5.3.3 The Evaluation Mechanism

The main evaluation mechanism is contained in the `VarStack` class which is used to represent the Aladin stack. The function-lookup table is held as a static field of the `AAM` class which, for convenience, will be subclassed by the classes where we define our primitives. Each of these primitives is represented by a static method as in our Ginger compiler. Again we use the `java.lang.reflect` package to reflect these methods as as `Method` objects which are stored as a member of the `Function` class (see above). These methods take a number of `Var` objects as arguments and return a value of type `Prog`. Unlike in the semantics given in Section 5.1 the heap is not passed or returned as an explicit parameter since it is a global object which can be destructively updated.

The evaluation of a program is triggered by a call to the `eval` method of the `Var` object that refers to the program we wish to evaluate. This method creates a new stack and triggers the transformation of the stack (cf. the meta-function **F** defined in equation 5.14) and is defined as follows:

```
public Prog eval() throws UnknownValueException {
  Prog p = get();

  if (p instanceof App ||
      p instanceof Function && ((Function) p).arity == 0) {
    // only need to bother evaluating if this variable refers to an
    // App or a CAF.
    (new VarStack(this)).transform();
    return get();
  }
  else
    return p;
}
```

If it is possible to evaluate the value of the variable, that is if it is an application or a CAF, then a new `VarStack` is created and we trigger the transformation of the stack. Otherwise, no evaluation needs to be done. For convenience, the method returns the final value of the variable using the method `get` (which short-circuits chains of `Var` objects).

The method `transform` in the `VarStack` class repeatedly transforms the stack using Rules 5.15–5.23 as a guide. This method starts as follows:

```
public void transform() throws EvaluationException {
  for (;;) {
    Var v = head();

    Object head = v.get();
```

Transformation of the stack will terminated by a `return` statement. The method `head` returns, without popping, the `Var` at the top of the stack. If the value of this object is an `App` then we push its functor onto the stack and continue (cf. Rule 5.16):

```
    if (head instanceof App)
      push(((App) head).functor);
```

If the value is a function we have to determine its arity. If we have enough arguments we can trigger the application of the function to its arguments (cf. Rules 5.19 and 5.22):

```
      else if (no_args < f.arity) {
        // Not enough arguments, so just evaluate those we can
        evalArgs(f);

        return;
      }
```

The final case if the head is a function is when we have too many arguments. In this case (described in Rule 5.18) we evaluate the inner application by popping the appropriate elements from the stack (the function itself and the correct number of arguments) and forming these into a new stack which we then transform. Once this new stack has been fully evaluated, we continue transforming the original one, the head of which will be the result of evaluating the new one.

```
     else { // too many arguments
        VarStack inner = partition(f.arity);
        inner.transform();
      }
    }
  }
```

The method `partition` does the job of splitting the stack up. Its definition is given below. If the head of the stack is a data object then we can just return from the `transform` method:

```
    }
    else {
      // head must be a data object or an unknown.
      return;
    }
  }
}
```

The method `evalArgs` extracts the arguments from the stack and evaluates them with respect to the strictness of the passed function:

```
private Var[] evalArgs(Function f) {
  // we need to preserve the ordering -- the stack is held reversed
  // for efficiency when pushing
  Var[] args = new Var[count - 1];
  UnknownValueException unknown = null;

  for (int i = 0; i < count - 1; i++) {
    // get the argument part
    args[i] = ((App) elements[count - (i + 2)].get()).arg;

    if (f.strictIn(i + 1))
      args[i].eval();
  }
```

```
  return args;
}
```

The method `partition` splits the current stack in two, returning the stack representing the inner application and keeping the stack representing the outer application on the original stack. Note that the root of the inner application is kept as the head of the stack used in the outer application (thus, strictly speaking, we have not got a partition since one element is in both halves):

```
private VarStack partition(int num_args) {
  VarStack inner = new VarStack(num_args + 1);
  int root_index = count - (num_args + 1);

  inner.count = num_args + 1;
  System.arraycopy(elements, root_index,
                   inner.elements, 0, inner.count);
  count = root_index + 1;

  return inner;
}
```

The method `update` of `Var` deals with the updating of variable with a new value as in Equation 5.12. Note that for simplicity we assume that `p` is reducible if it is an `App` or a CAF, rather than a function applied to at least the correct number of arguments as in Equation 5.12.

```
public final void update(Prog p) {
  // find the end of the indirection chain
  Var v = this;
  while (v.value instanceof Var)
```

```
    v = (Var) v.value;

  if (p instanceof Var) {
    // might be able to short-circuit any chains
    Prog p_value = ((Var) p).get();

    if (p_value instanceof App)
      v.value = p_value;
    else if (p_value instanceof Function)
      v.value = (((Function) p_value).arity == 0) ? p : p_value;
    else
      v.value = p_value;
  }
  else
    v.value = p;
}
```

5.4 Primitives

As mentioned above, our basic mechanism for implementing primitives is the static Java method. In this section we shall only deal with how the user writes such primitives and how primitives in written in different languages are handled by Java. The mechanism of importing primitives is handled in Section 5.4.6. Since Aladin is essentially a pure functional language, it is expected that the user makes the primitives defined referentially transparent, that is, depend only on the value of their arguments.

5.4.1 Java and Aladin Primitives

Each primitive that is written in Java (and also Aladin since we compile Aladin definitions to Java methods — see Section 5.5) is represented by an instance of the `RunTimeFunction` class a subclass of `Function` (see Section 5.3). This has a field of type `java.lang.reflect.Method` which reflects the (static) method where the actual code is stored. To primitively apply such a function to a number of arguments merely requires applying the `invoke` method from the `Method` class to the arguments.

```
public class RunTimeFunction extends Function {
  public Method app_rule;

  public Prog primApply(Var[] args) {
    return (Prog) app_rule.invoke(null, args);
  }
}
```

Note that `primApply` only returns the result rather than the updated heap as in Rule 5.6 as any updates to the heap will be done globally. The actual writing of the Java code can be split into three parts:

1. Extract some or all of the values from the `Var` objects.
2. Do the actual work involved.
3. Wrap the result in a `Prog` object if necessary and return.

For example, consider the method `_op_if` of strictness $s \times l \times l \to l$ which does the work of the conditional `if`:

```
public static Prog _op_if(Var a, Var t, Var f) {
```

```
  return (a.getBool()) ? t : f;
}
```

We first extract the value of `a` using the method `Var.getBool` which presumes that the ultimate value of the variable is a boolean and returns it as a Java `boolean`. This method saves us doing the type cast and field access in the code of the primitive. There are also methods `getInt`, `getReal`, `getChar` and `getString` for getting the Java `long`, `double`, `char` and `String` values of a variable. If evaluation needs to be done before the value is extracted than the related functions, `evalBool`, `evalInt`, etc. may be used.

Depending on the value of `a`, we either return the true branch of the condition, `t`, or the false branch, `f`. Note that the evaluation of the result is left to Aladin's evaluation mechanism since `_op_if` declares that it returns a lazy result. We could declare `_op_if` to have a strict result, in which case the evaluation would have to be done inside the primitive:

```
public static Prog _op_if(Var a, Var t, Var f) {
  return (a.getBool()) ? t.eval() : f.eval();
}
```

Note that such an approach would have to create a new state (more specifically, a `VarStack` object) in which to evaluate either `t` or `f`; the version which returns a lazy result will evaluate `t` or `f` in the original state. The latter approach means that not only do we avoid the work needed to create a new state, but we do not waste memory by having two states in memory at the same time.

Of course, primitives can be more complex than this one-liner. Consider the following function to compute factorials (of strictness $s \to s$):

```
public static Prog fac(Var v) {
  long value = v.getInt();
```

```
  // do the actual work
  long f = 1;

  for (long i = 2; i <= value; i++)
    f *= i;

  /* wrap the answer in an Int and return */
  return new Int(f);
}
```

We first extract the integer which we want to take the factorial of. The answer is then generated in an iterative loop and packed into an Aladin `Int` before returning.

5.4.2 C Primitives

Implementing primitives in C relies heavily on the Java Native Interface [30]. Each C function, which originates from some shared dynamic library, has a corresponding Java method declaration which is used by Aladin to interface to that function, as with the standard use of native methods in Java. Thus, to Aladin, a C primitive is the same as a Java or Aladin one (there is a `CFunction` class for representing primitives written in C, but this is only used at compile time).

The actual code for the primitives follows the pattern for writing primitives in Java. For example, we can define the factorial function in C, analogous to the Java one in Section 5.4.1, as follows:

```
JNIEXPORT jobject JNICALL Java_update_fac
    (JNIEnv *env, jclass cl, jobject var)
{
```

```
  jlong value = getInt(env, var);

  /* do the actual work */
  jlong f = 1;
  jlong i;

  for (i = 2; i <= value; i++)
    f *= i;

  /* wrap the answer in an Int */
  return makeInt(env, f);
}
```

The somewhat involved-looking prototype for each function is generated by Aladin from its strictness signature using the `javah` utility and placed in a C header file. This can then be copied by cut and paste into the `.c` file where the code is to maintain consistency and save typing a complicated expression.

The first argument of the function is a pointer to an environment representing the JVM. This allows us to call Java methods, create Java objects, etc., from C (similar to the necessity of passing the heap as the first parameter of primitives in the semantics). The second argument is the class of the Java method which this function provides the implementation for; it is not used by the function. The rest of the arguments are the actual arguments of the function, in this case we have just one.

The function `getInt` is the C equivalent of the Java `getInt` method (see Section 5.4.1), in fact the former calls the latter during execution:

```
jlong getInt(JNIEnv *env, jobject v) {
```

```
  jobject Var = getVar(env);
  jmethodID I = (*env)->GetMethodID(env, Var, "getInt", "()J");

  /* get the value of var  */
  return (*env)->CallLongMethod(env, v, I);
}
```

This code has to find the `Var` class, get an ID for its `getInt` method and finally call it. It is defined in a separate shared library (`libAladin.so` in UNIX) which must be linked in when the user compiles their code. The other related methods, `getBool`, `evalInt` etc., have similar definitions in C.

After computing the factorial we need to wrap the result (a `jlong`) in an Aladin `Int` object. This is done using the `makeInt` function:

```
jobject makeInt(JNIEnv *env, jlong val) {
  jobject Var = getVar(env);
  jmethodID ConsInt = (*env)->GetMethodID(env, Var,
                                          "<init>", "(J)V");

  /* Invoke and return */
  return (*env)->NewObject(env, Var, ConsInt, val);
}
```

The steps take are similar to the `getInt` function. We have a whole family of functions to make Aladin programs from each primitive Java/C type. For example, `makeReal` and `makeApp` construct reals and applications respectively.

5.4.3 Primitives in Ginger

Each primitive written in Ginger is represented by a `GingerFunction` object:

```
public class GingerFunction extends Function {
  private fp.gingerc.Func ginger_function;

  // ...
}
```

Instead of the app_rule field found in the RunTimeFunction class we have an object of the Func class (we explicitly qualify each class from the fp.gingerc package to make it clear where methods and classes originate and because there are some name clashes between packages) reflecting the Java method that the Ginger code has been converted to (see Section 4.3). The primApply method is implemented to use this field:

```
public Prog primApply(Var[] args) throws EvaluationException {
  Object[] gargs = new Object[args.length];
  for (int i = 0; i < gargs.length; i++)
    gargs[i] = args[i].toGinger();

  Object g_res = ginger_function.apply(gargs);
  g_res = fp.gingerc.Node.eval(g_res);

  return fromGinger(g_res);
}
```

Before calling the function with the given arguments, the arguments in question must be converted to a format Ginger will recognise, and similarly we must convert the result back into Aladin. Since Aladin and Ginger are both functional languages this conversion is fairly straightforward, with a couple of caveats.

As Ginger is lazy, it is possible that an argument, or a part of it, may pass through a call to a Ginger function completely unaltered. Converting an Aladin program to a Ginger object involves the creation of a new and distinct object, similarly with the reverse process. Suppose we have an Aladin program a which is passed to a Ginger function f; a will be converted to a Ginger function g, say. Suppose now that g is passed back as part of the result of the Ginger function (for instance as part of a list); it will be converted to an Aladin object a', say. So, a and a' are really the same object, but during the conversion process they have become divorced from each other. In particular, if we evaluate a then this evaluation is *not* reflected in a' and *vice versa*. This can have a serious effect on performance. For instance consider the following Ginger program.

```
lists x = [x..] : lists (x + 1);


hdlists n =
    let
        ns = map hd (lists 0);
    in
        if n == 0 then
            ns
        else
            take n ns
        endif
    endlet;
```

Although `hdlists` normally runs in linear time with respect to the parameter `n`, if we import it into Aladin then the above conversion problem slows it down to exponential time.

The solution is to cache the Aladin-Ginger conversions so that when we wish to convert an unchanged object back from Ginger we can retrieve the original Aladin program, and *vice versa*, rather than creating a new one. This cache stores only the most recent conversions created or referenced to prevent the machine being clogged up with long-irrelevant conversions and to enable Aladin and Ginger objects to be released for possible garbage collection. To minimise the conversion work done, we make sure that the Ginger object is in WHNF, by calling the `Node.eval` method, before conversion is done.

Although the conversion of Aladin objects to and from Ginger is for the most part straightforward, converting Aladin functions to Ginger ones is slightly more complex. To solve this problem we represent Aladin functions in Ginger using the class `AladinFunc`, a subclass of the Ginger function class, `Func`.

```
public class AladinFunc extends Func {
  protected Var al_func;

  public Object apply(Object[] as) {
    Var v = al_func;

    for (int i = 0; i < as.length; i++)
      v = new Var(v, Prog.fromGinger(as[i]));

    v.eval();

    return v.toGinger();
  }
}
```

This class stores as a member the original Aladin function (or rather, a `Var` object which refers to it). When Ginger applies such a function, by calling its `apply` method, we form a new Aladin application by converting the passed arguments from Ginger to Aladin which are given as arguments to the original Aladin function. We then evaluate this function using Aladin, convert the result back to Ginger and then return. Again these conversions are cached.

5.4.4 Overloading

Our implementation of Aladin allows primitives to be overloaded in a manner that is more systematic than Ginger's approach, though not as expressive and powerful as Haskell's. This overloading is only known to the primitives and is in no way built into our implementation.

We use Java interfaces to classify classes whose instances can be combined using the methods of the interface. For instance, instances of the `Comparable` interface must implement the `eq` (equals) and `ne` (not equals) methods (cf. Haskell's `Eq` class described in Section 2.4) defined as:

```
public interface Comparable {
  public Prog eq(Prog p);
  public Prog ne(Prog p);
}
```

It is these methods that are called by the `_op_eq` (prefix variant of `==`) and `_op_ne` (prefix variant of `~=`) methods (see Section 5.2.4). For instance, `_op_eq` is defined as:

```
public static Prog _op_eq(Var x, Var y) {
  Prog a = x.get(), b = y.get();
```

```
  if (a instanceof Comparable)
    return ((Comparable) a).eq(b);
  else
    return (a.equals(b)) ? Bool.TRUE : Bool.FALSE;
}
```

This primitive (which is strict in both its arguments) first tests of it can compare its two arguments using the `Comparable.eq` method; if not then it uses the `equals` method which all Java classes implement, either through inheritance or by overriding. Note that `Comparable.eq` is preferred as it returns a general Aladin program rather than a `boolean`. This is preferred when partially evaluating programs as we might not be able to calculate all of the result (see Section 6.5.1). So, the user can overload the `==` and `~=` operators on any type they wish by implementing the `Comparable` type in the type's class definition.

The interface `Orderable` is implemented by classes whose instances can be ordered (cf. Haskell's `Ord` class), while the `Num` interface is implemented by classes whose instances can be used in arithmetic expressions (cf. Haskell's `Num` class and its subclasses). Since we wish to use Aladin to implement solutions to problems in fuzzy logic, we also introduce the `Logical` interface (cf. the `Logic` class introduced in Section 3.1) which is implemented by classes whose instances can be combined in logical expressions. See Section 5.4.5 for further details.

5.4.5 Fuzzy Primitives

As mentioned in Section 5.4.4 we overload the logical primitives so that they implement both boolean and fuzzy logic. For fuzzy logic, we have to implement the `Logical` interface in the `Real` class. This interface defines the methods `and`, `or` and `not` which are called by our implementation of `_op_and` (`&`), `_op_or` (`|`) and `_op_not`

(~) respectively. Fuzzy conjunction, using Zadeh's definition, is implemented using the following method:

```
public Prog and(Var v) {
  if (value == 0.0)
    return ZERO;
  else if (value == 1.0)
    return v;
  else
    return (value < v.evalReal()) ? (Prog) this : (Prog) v;
}
```

The instance field `value` is a double containing the value of the `Real`. `ZERO` is a static field of `Real` containing the `Real` with the value 0. Note that this primitive is still lazy in its second argument as in the second case, since 0 and 1 are the annihilator and identity of fuzzy conjunction and we test for these values first. The methods `_op_or` and `_op_not` are implemented similarly.

Unlike in Ginger and Haskell, we do not overload the logical primitives so that they also act as set operators since, because Aladin is untyped, this can lead to complications when we partially evaluate fuzzy Aladin programs. Instead we introduce the functions `union`, `inter`, `add` and `comp` to perform operations of union, intersection, addition and complement of sets. These are defined using the `s_dash` function (the $\mathbf{S}'$ combinator) and function composition:

```
s_dash :: l * l * l * l -> l
s_dash op f g x = f x `op` g x;

union :: l * l -> l
union x y = s_dash (|) x y;
```

```
inter :: l * l -> l

inter x y = s_dash (&) x y;


add :: l * l -> l

add x y = s_dash (+) x y;


comp :: l -> l

comp x = (~) . x;
```

We also need to define fuzzy set operations point-wise over tuples for use in fuzzy systems which have more than one output variable. Again, because of complications due to partial evaluation, this is done using separate functions rather than overloading. The functions `applyUn` and `applyBin` map a unary function over a tuple and combine two tuples with a binary function respectively (cf. the list functions `map` and `zipWith`). These have the following behaviour:

$$\texttt{applyUn}\ f\ (x_1, \ldots, x_n) = (f\ x_1, \ldots, f\ x_n)$$

and

$$\texttt{applyBin}\ \oplus\ (x_1, \ldots, x_n)\ (y_1, \ldots, y_n) = (x_1 \oplus y_1, \ldots, x_n \oplus y_n)$$

The weighting operator (`==>` in Haskell and Ginger) becomes the function `when` in Aladin:

```
when :: l * l * l -> l

when w f x = if isZero w | isZero (f x) then 0 else w * f x endif;
```

The function `isZero` returns whether a program has the value 0; it is especially useful if we are partially evaluating (see Section 6.6.6). All other fuzzy functions have obvious translations from their Haskell counterparts.

We can now implement our shower controller from Section 3.4.1 in Aladin. Given the domain `changedom = [-0.2, -0.175 ..  0.2]` over which we defuzzify the changes to the hot and cold taps, and the fuzzy subsets of temperature, flow and change, we can define the fuzzy rule base function, `change_valves` as:

```
change_valves :: s * s -> l
change_valves temp flow =
    let
        defuz = centroid changedom;
        changes = rulebase (applyBin add) [
            applyUn (when (cold temp & weak flow))   (pm,  z),
            applyUn (when (cold temp & right flow))  (pm,  z),
            applyUn (when (cold temp & strong flow)) ( z, nb),
            applyUn (when (ok temp & weak flow))     (ps, ps),
            applyUn (when (ok temp & strong flow))   (ns, ns),
            applyUn (when (hot temp & weak flow))    ( z, pb),
            applyUn (when (hot temp & right flow))   (nm,  z),
            applyUn (when (hot temp & strong flow))  (nb,  z)];
    in
         (defuz (fst changes), defuz (snd changes))
    endlet;
```

While this may not be as elegant as the Haskell version it is very amenable to partial evaluation (see Sections 6.6.6 and 6.6.7) and, as we shall see in the next chapter, partially evaluating it leads to a residual program which is an order of magnitude faster than the original.

5.4.6 Importing Primitives

As we shall see in Section 5.5, we shall compile Aladin definitions into static Java methods, so the import procedure for primitives defined in Aladin and Java is the same. The import declarations are converted into method calls and placed in the static method `__initialise__class__` that is created for each compiled Aladin script. This method is called when the class is loaded, either by the class's `main` method or by the import mechanism depending on whether the class is the one being run to evaluate programs or has been imported respectively (see Section 5.5.4).

The first section of code in `__initialise__class__` creates the function, setting its strictness signature and inserting it into the heap, but for the moment leaving its `app_rule` (see Section 5.4.1) null for the moment. As an example, we have the following from `stdlib.java` — the code generated from `stdlib.as` where the arithmetic functions, amongst others, are defined:

```
public static void __initialise__class__() throws Exception {
  put("fp.aladin.lib", "Operators", "_op_plus", CONST_0);
  put("fp.aladin.lib", "Operators", "_op_minus", CONST_0);
  put("fp.aladin.lib", "Operators", "_op_times", CONST_0);
  put("fp.aladin.lib", "Operators", "_op_divide", CONST_0);

  // ...
}
```

The static `put` method (inherited from the `AAM` class which is the superclass of all classes created by the compiler) creates a new function from its arguments and inserts it into the heap. The first three arguments represent the package, class and name of the function to be created. The last argument is the strictness signature, which is created only once and stored as a private field of the generated class (see

Section 5.5.2). Here `CONST_0` represents the strictness signature $s \times s \rightarrow s$.

After setting the functions up, we have to fill in their `app_rule` field. This is done by going through each class, including the class that is doing its importing since any functions it defines itself are held in that class, using the `getDeclaredMethods` method from the `java.lang.Class` class and filling in each function with its appropriate `Method` object. If we end up with any functions which we have a strictness signature for but no corresponding `Method` then an error has occurred.

Each primitive that is written in C and directly imported will have had a corresponding Java method header created in the generated class file, hence these primitives will be imported by the same mechanism that imports Java and Aladin primitives. However, we also need to load in the libraries where the actual object code can be found. This is done by loading each library using the `System.loadLibrary` method.

Importing Ginger primitives is done similarly to importing Aladin and Java primitives, except that as each Ginger function is stored as a `fp.gingerc.Func` field of the Ginger class generated by the Ginger compiler from each Ginger script (see Section 4.5.2) we have to examine the fields of each Ginger class rather than the methods.

5.5 Compilation

The aim of the compiler is to take an Aladin script and translate it into a Java program which will then be compiled by a Java compiler into a Java class file. Our compiler has three jobs:

1. To provide a static Java method for each primitive defined in the script.

2. To set the strictness of each function defined in the script and any imported function whose strictness is redefined in the script.

3. To import all the functions specified in the script.

The latter two jobs will be accomplished by putting the code that achieves their object in the method `__initialise_class__` of the created class, which will be called by Aladin when that class is loaded. The whole import procedure is triggered when a class is run using the Java interpreter by placing a call to `__initialise_class__` in the `main` method of each created class file.

5.5.1 Compiling Scripts

To compile a script we need the following parameters:

- c, the class to create (derived from the name of the script);
- p, the package to place the created class in, declared using `package` (if no such declaration the p is set to the empty string);
- ss, the set of *distinct* strictness signatures used in a script;
- cs, the set of distinct constants used in a script;
- fs, the set of functions defined in the script with those imported into the script who have their strictness signature set or over-ridden;
- cls, the classes containing primitives defined in Java and Aladin *directly* imported into a script;
- gs, the classes containing primitives defined in Ginger directly imported into a script;
- ls, the shared object libraries containing primitives defined in C/C++ directly imported into a script;
- ds, the functions defined in the script (with their definitions) plus any primitives defined in C which are directly imported.

The compilation scheme $\mathcal{P}$ creates a Java program using these parameters

$$\mathcal{P}\ p\ c\ ss\ cs\ fs\ cls\ gs\ ls\ ds$$

```
= package p;
  import fp.aladin.*;
  public final class c extends AAM {
    S ss cs fs cls gs ls ds


    public static void main(String[] args) throws Exception {
      __initialise_class__();
      parseAndEval(args);
    }
  }
```

If no package has been given then we omit the `package` declaration. The $\mathcal{S}$ scheme creates the code used to initialise the class and creates the method definitions for each function defined in the script.

The `main` method will be called if we run the created class using the `java` interpreter or some other way of executing Java classes. It first initialises the class, by importing all the required classes and functions and setting any required strictness signatures. It then parses the command line arguments. Some of these are used to set options, for instance, whether to pretty print, whether to partially evaluate and how many decimal places to use when printing real numbers. The rest are assumed to form an expression to be evaluated. If there is no such expression then Aladin attempts to evaluate the `main` CAF defined in the class (if there is one). Note that Java allows us to overload the `main` method, so there is no conflict between `main(String[])`, the method called by the Java interpeter when the class is run, and `main()`, the compiled form of the `main` CAF.

5.5.2 Compiling Constants and Strictness Signatures

We do not create a separate object representing all the constants and strictness signatures in a class. Rather, for each distinct constant we create just one instance and store it as a private field of the class we are creating and read this field whenever we want an instance of the particular constant or strictness signature.

$\mathcal{S}\ (s_1, \ldots, s_m)\ (c_1, \ldots, c_n)\ fs\ cls\ gs\ ls\ ds$

$=\quad cc\ 1\ c_1$

$\ldots$

$cc\ n\ c_n$

$cs\ (n+1)\ s_1$

$\ldots$

$cs\ (n+m)\ s_m$

`public static void __initialise_class__() throws Exception` $\{$

$\mathcal{D} fs\ cls\ gs\ ls\ ds\ \rho$

where

$$\rho = \left[\begin{array}{l} c_1 \mapsto \texttt{CONST\_}i, \ldots, c_n \mapsto \texttt{CONST\_}n, \\ s_1 \mapsto \texttt{CONST\_}(n+1), \ldots, s_m \mapsto \texttt{CONST\_}(n+m) \end{array}\right]$$

The function *cc* compiles a constant and places it in a static private field of the class being created:

$cc\ i\ c =$ `private static Var CONST_`$i =$ `new Var(`c`);`

The `Var` class has a constructors to construct variables which point to the appropriate program for each basic type. The function *cs* is similar to *cc* but it compiles a strictness signature instead.

$$
\begin{aligned}
& cs\ i\ (\sigma_1 \times \ldots \times \sigma_n \to \rho) \\
& \quad = \texttt{private static StrictnessSig CONST\_}i = \\
& \qquad \texttt{new StrictnessSig(}as\texttt{, }r\texttt{);} \\
& \qquad \textbf{where} \\
& \qquad\quad \begin{array}{rcll}
as & = & \texttt{new boolean[] } \{a_1\texttt{, } \ldots\texttt{, } a_n\} & \\
a_i & = & \texttt{true}, & \textbf{if } \sigma_i = s \\
 & = & \texttt{false}, & \textbf{otherwise} \\
r & = & \texttt{true}, & \textbf{if } \rho = s \\
 & = & \texttt{false}, & \textbf{otherwise}
\end{array}
\end{aligned}
$$

5.5.3 Compiling Strictness Declarations

The $\mathcal{D}$ scheme declares the strictness signature of each function. The final parameter is an environment detailing which constant or strictness signature corresponds to each field. For a constant or strictness signature, c its corresponding field is $\rho(c)$.

$$
\begin{aligned}
& \mathcal{D}\ (f_1, \ldots, f_m)\ cls\ gs\ ls\ ds\ \rho \\
& \quad = \texttt{put(}p_1\texttt{, }c_1\texttt{, }n_1\texttt{, }s_1\texttt{);} \\
& \qquad \ldots \\
& \qquad \texttt{put(}p_m\texttt{, }c_m\texttt{, }n_m\texttt{, }s_m\texttt{);} \\
& \qquad \mathcal{I} cls\ gs\ ls\ ds\ \rho \\
& \qquad \textbf{where} \\
& \qquad\quad \begin{array}{rcl}
p_i & = & package(f_i) \\
c_i & = & class(f_i) \\
n_i & = & name(f_i) \\
s_i & = & \rho(strictness(f_i))
\end{array}
\end{aligned}
$$

5.5.4 Compiling Imports

The $\mathcal{I}$ scheme deals with creating the code needed to import the various classes and libraries.

$\mathcal{I}(c_1, \ldots, c_n)\ (g_1, \ldots, g_m)\ (l_1, \ldots, l_k)\ ds\ \rho$

$=$ `importClass("`c_1`");`

`...`

`importClass("`c_n`");`

`importGinger("`g_1`");`

`...`

`importGinger("`g_m`");`

`System.loadLibrary("`l_1`");`

`...`

`System.loadLibrary("`l_k`");`

`}`

$\mathcal{H}\ ds\ \rho$

Note the terminating brace for the definition of `__initialise_class__` started by the $\mathcal{S}$ scheme.

5.5.5 Compiling and Optimising Local Blocks

Our Aladin compiler contains a facility to eliminate common subprograms in a definition. Consider the following definition of the function `subseqs` which returns all the subsequences of a list:

```
subseqs :: s -> l
subseqs xs =
```

```
    if isEmpty xs then [[]]
    else subseqs (tl xs) ++ map ((:) (hd xs)) (subseqs (tl xs))
    endif;
```

Evaluation of this function applied to an argument has to evaluate the `subseqs (tl xs)` and `tl xs` twice. We would expect the user to recognise this and lift the repeated applications into a `let`. However there is no reason why we cannot let the compiler do it instead.

For each program definition, we examine its definition for any repeated non-trivial subprograms. For our purposes, a non-trivial subprogram is an application or a function, the latter because the code to construct a function is done using a hash table lookup and it is quicker if we only have to do this lookup once. So, for example, the above definition of `subseqs` is optimised to:

```
subseqs xs =
    let
        v5 = _op_list_empty;
        v3 = (subseqs (tl xs));
        v0 = _op_list_cons;
    in
        (((_op_if (isEmpty xs)) ((v0 v5) v5))
         ((_op_list_cat v3) ((map (v0 (hd xs))) v3)))
    endlet
```

Since both the functions `_op_list_empty` and `_op_list_cons` occur more than once in the definition of `subseqs` they are lifted into a local definition, similarly with the application `subseqs (tl xs)`. Note that by lifting the recursive call to `subseqs` into a local definition we now only have one occurrence of `tl xs` and so this does not have to be lifted out into a separate definition. This is ensured by a complimentary optimisation: eliminating redundant `lets`, that is local definitions whose variable occurs only once in the entire definition.

While these optimisations are not critically important in normal Aladin programs, since an experienced programmer should be able to spot when to introduce local definitions, they are important when we partially evaluate Aladin programs. This is because all knowledge of local definitions is lost after compilation and so the definition of any residual programs may contain repeated subprograms. See the next chapter for further details.

Before compilation all local variables are 'floated' to the top level using the *fl* function. This function splits a program into a list of declarations and a program containing no `lets`:

$$
\begin{array}{rcl}
\mathit{fl}\ ((\texttt{let}\ v\ \texttt{=}\ D\ \texttt{in}\ B\ \texttt{endlet})\ C) & = & ((v, D') : ds\ +\!\!+\ es, P) \\
& & \textbf{where} \\
& & \quad \begin{array}{rcl} (ds, P) & = & \mathit{fl}\ (B\ C) \\ (es, D') & = & \mathit{fl}\ D \end{array} \\
\mathit{fl}\ (C\ (\texttt{let}\ v\ \texttt{=}\ D\ \texttt{in}\ B\ \texttt{endlet})) & = & ((v, D') : ds\ +\!\!+\ es, P) \\
& & \textbf{where} \\
& & \quad \begin{array}{rcl} (ds, P) & = & \mathit{fl}\ (C\ B) \\ (es, D') & = & \mathit{fl}\ D \end{array} \\
\mathit{fl}\ (BC) & = & (ds\ +\!\!+\ es, B'\ C') \\
& & \textbf{where} \\
& & \quad \begin{array}{rcl} (ds, B') & = & \mathit{fl}\ B \\ (es, C') & = & \mathit{fl}\ C \end{array} \\
\mathit{fl}\ P & = & (\langle\rangle, P)
\end{array}
$$

In a conventional compiler we would have to be careful about how far outwards we floated local variable declarations as we might end up unnecessarily building the graph of programs that could otherwise be avoided. For instance, in a conditional expression a conventional compiler can exploit its knowledge of the conditional primitive to build only the graph related to the 'true' branch of the conditional and ignoring all the rest. Since Aladin knows very little about how its primitives work we can perform no such optimisation, but this does mean we do not have to be as careful about where we place our local variable declarations as in a conventional compiler.

The $\mathcal{H}$ scheme compiles all the definitions in a script plus creates headers for any primitives written in C into the script:

$$\begin{aligned}\mathcal{H}(f_1, \ldots, f_n)\rho \quad &= \quad \mathcal{F}\ f_1\ \rho \\ & \quad\ \ \ldots \\ & \quad\ \ \mathcal{F}\ f_n\ \rho\end{aligned}$$

The scheme $\mathcal{F}$ compiles an individual function. If the function is written in C or C++ we just need to declare a native method:

$$\mathcal{F}\ f^n\ \rho = \texttt{public final native static Prog}\ f\texttt{(Var x\_1, ..., Var x\_}n\texttt{);}$$

where $\texttt{x\_}i$ are dummy parameter names and n is the arity of the function. Each definition first has all its variables renamed so that they are all distinct and then compiled using the $\mathcal{F}$ scheme:

$\mathcal{F}\ (f\ x_1 \ldots\ x_n\ \texttt{=}\ E)\ \rho$

$=$ `public final static Prog` f`(Var` x_1`, ..., Var` x_n`)` $\{$

`Var` v_1 `=` $\mathcal{C}\ D_1\ \rho\ vs$`;`

$\ldots$

`Var` v_m `=` $\mathcal{C}\ D_m\ \rho\ vs$`;`

`return` $\mathcal{R}\ E'\ \rho\ vs$`;`

$\}$

where

$(\langle(v_1, D_1), \ldots (v_m, D_m)\rangle, E') = \mathit{fl}\ E$

$vs = \{x_1, \ldots, x_n, v_1, \ldots, v_m\}$

5.5.6 Compiling Simple Programs

The $\mathcal{R}$ and $\mathcal{C}$ schemes compile a simple program. They differ only in how they treat the outermost part of a program when that part is an application. If we have a constant then we simply have to load the relevant field:

$$\mathcal{C}\ c\ \rho\ vs = \mathcal{R}\ c\ \rho = \rho(c)$$

If we have an identifier then we either have a function or a variable. We can tell the difference by seeing if the identifier is in the set of variables passed to $\mathcal{C}$:

$$\begin{aligned} \mathcal{C}\ id\ \rho\ vs &= id, && \textbf{if}\ id \in vs \\ &= \texttt{get(}id\texttt{)}, && \textbf{otherwise} \end{aligned}$$

If we have a variable then the code to compile is thus that variable, otherwise we presume it is a function name and look it up in the heap using the `get` function inherited from the `AAM` superclass.

If we have an application, then the $\mathcal{C}$ scheme compiles the functor and the argument using the $\mathcal{C}$ scheme and then forms then into an `App` which is pointed to by a `Var` (achieved using the two-argument constructor of `Var`):

$$\mathcal{C}\ (f\ a)\ \rho\ vs = \texttt{new Var(}\mathcal{C}\ f\ \rho\ vs\texttt{, }\mathcal{C}\ a\ \rho\ vs\texttt{)}$$

If we are compiling the outermost application then we do not need to create the `Var`. Hence $\mathcal{R}$ is defined as:

$$\mathcal{R}\ (f\ a)\ \rho\ vs = \texttt{new App(}\mathcal{C}\ f\ \rho\ vs\texttt{, }\mathcal{C}\ a\ \rho\ vs\texttt{)}$$

5.5.7 Compiling the Target Code and Using the Resultant Classes

The code created by the compiler is placed in a `.java` file which is compiled into a class file using the `javac` compiler. If the script directly imports any C/C++ functions then `javah` is run over the generated class to create the header file defining the prototypes for the imported C/C++ functions. For example, suppose we have the following definitions in the script `foo.as`:

```
import fp.aladin.stdlib;
import fp.aladin.gstdlib;


importg bar
    lists   :: s -> l
    hdlists :: s -> l;


main :: -> l
main = hdlists 10;
```

where `lists` and `hdlists` are as defined in Section 5.4.3. Then this script is compiled into the Java file `foo.java` (the indenting has been added by hand):

```
import fp.aladin.*;


public class foo extends AAM {
  private static fp.aladin.Var CONST_2 = new fp.aladin.Var(10);
  private static fp.aladin.StrictnessSig CONST_0 =
    new fp.aladin.StrictnessSig(new boolean[] {true}, false);
  private static fp.aladin.StrictnessSig CONST_1 =
    new fp.aladin.StrictnessSig(new boolean[] {}, false);
```

```
  public static void __initialise__class__() throws Exception {
    importClass("fp.aladin.stdlib");
    importClass("fp.aladin.gstdlib");
    putGinger("", "bar", "lists", CONST_0);
    putGinger("", "bar", "hdlists", CONST_0);
    importGinger("bar");
    System.loadLibrary("Aladin");
    put("", "foo", "main", CONST_1);
    importClass("foo");
  }

  public static void main(String[] args)  throws Exception {
    __initialise__class__();
    parseAndEval(args);
  }

  public static Prog main() {
    return new fp.aladin.App(get("bar.hdlists"), CONST_2);
  }
}
```

This is then compiled into the class `foo.class`. The class `fp.aladin.stdlib` contains various standard functions, while `fp.aladin.gstdlib` contains import declarations and strictness signatures for Ginger primitives and functions in its standard prelude.

We can now evaluate programs. To evaluate `main` we simply pass `foo` as an argument to `java` with the `-pp2` argument to specify that we want to pretty-print lists using the square bracket notation:

```
gem:~/Aladin/progs> java foo -pp2
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can also apply `hdlists` to a different argument:

```
gem:~/Aladin/progs> java foo -pp2 hdlists 20
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Since we import `stdlib` into our script, we have access to standard functions such as arithmetic and comparison operators, and hence we can evaluate programs using these functions *via* the `foo` class:

```
gem:~/Aladin/progs> java foo '(3 < 2) | (2 >= 8 + 5)'
false
```

As well as command-line evaluation, we provide a graphical environment in which the user can interactively load, discard and evaluate programs. Figure 5.5 shows an example of the above three programs being evaluated in the graphical environment. Further examples of running Aladin programs can be found in the next chapter.

5.6 Summary

The Aladin Abstract Machine is a useful tool for investigating aspects of functional programming, since its simplicity allows us to concentrate on the vital issues. Its control over strictness allows the user to specify when and if parts of a program should be evaluated, useful when importing primitives from a variety of languages, each with differing evaluation strategies. We have given the semantics of the Aladin Abstract Machine and an implementation of the machine, written in Java and creating a Java class. At present, our implementation is able to import primitives from

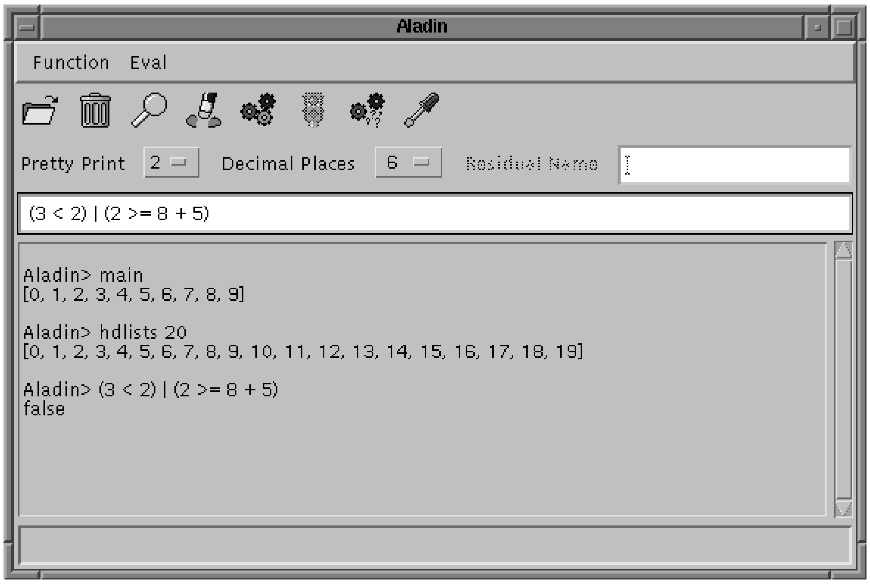


Figure 5.5: A GUI for Aladin

programs written in Java, C/C++, Ginger and the simple Aladin scripting language that we have defined in this chapter, using the fact that there are compilers available that target Java class files for all three of these languages.

Since the semantics of Aladin concern only the evaluation of programs, the construction of data structures and other such entities can only be done by means of function applications. However, since functions can be written in any language of the user's choice, there is no fixed template that the user must adhere to, and they have the freedom to implement the structures that their programs use in any way they see fit. The Aladin scripting language which we developed in Section 5.2, while outwardly resembling Ginger, is in fact a sugared way of constructing Aladin programs (as graphs of functions applied to arguments). Data structures such as lists and tuples are reduced to function applications and it is expected that the user provide suitable implementations of these functions.

Ginger, on the other hand, has a fixed set of data structures which are built into the language and which the language's compiler is aware of. This allows the

Ginger compiler to generate code which will directly construct data structures in some cases (such as lists — see Section 4.5.2) and also to directly call functions. In effect, the compiler is doing some evaluation of the program to be compiled in order to achieve these optimisations.

Another area where Aladin differs from other Ginger and other functional languages is in its use of strictness. In a conventional functional language compiler, whatever strictness information there is — obtained by strictness analysis, for example, or by defining a known set of primitives to be strict as in Ginger (see Section 4.4) — is used by the compiler to optimise code and is discarded at run-time. Strictness information is an integral part of Aladin, however, and is used to control every step of the evaluation process at run-time and, as we shall see in the next chapter, to control partial evaluation.

In essence, Aladin offers simplicity and flexibility at the cost of slower running times when compared to Ginger and other functional languages, a familiar trade-off in all branches of programming.

Chapter 6

Partial Evaluation in Aladin

In the previous chapter, we mentioned that Aladin could be used for partial evaluation and designed some of the primitives so that they could be more effectively used in partially-evaluated programs. In this chapter we shall make good on our earlier promises and show how partial evaluation can be done in Aladin.

Partial Evaluation [29, 51] is a means by which specialised programs can be obtained from more general ones by evaluating the general one with only some of its arguments instantiated. The specialised programs are typically considerably faster than the more general ones — an order of magnitude faster is not untypical.

The simplicity of Aladin, and its control over strictness, make this a suitable vehicle for partial evaluation, as first noted in [11]. Simplicity is advantageous when partially evaluating as it keeps to a minimum the cases when partial evaluation is different from non-partial evaluation, and makes these differences easier to determine. The advantage of strictness control when partially evaluating is that we know exactly what parts of a program it is safe to try to partially evaluate and which are not. We shall also see that because Aladin has no fixed set of primitives, the user can implement the primitives that their program uses so that they exploit partial evaluation to the full.

6.1 The Denotational Semantics

Partially evaluating a program involves evaluating it when some of its inputs are not known, the result being a residual program which will yield the same result as the original program when the rest on the inputs are provided. In the context of an Aladin program, this means that some of the variables occurring in the program have no value in the heap the program is being evaluated with respect to. That is, the program is being evaluated in a heap Γ and contains some variable x such that $\Gamma\ x = \perp$. We will refer to such a variable as an *unknown variable*. In the literature, if a parameter to a function contains unknown variables it is classed as *dynamic*, otherwise it is classed as *static*.

Our denotational semantics require a few modifications to handle partial evaluation. First, if we have an unknown variable applied to a number of arguments (possibly none) then we just return the original heap, as in the case when we have a data object applied to a number of arguments:

$$\mathbf{U}\ \Gamma[x_0 \mapsto p \in D \cup \{\perp\}, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]\ x_n = \Gamma \tag{6.1}$$

If we have a function applied to too few or too many arguments, then the rules are the same as in the non-partial evaluation case and given in Rules 5.9 and 5.10 respectively.

If we have a function applied to the exact number of arguments we first evaluate any strict arguments — using $\mathbf{A}$ which retains the same definition as that given in Rule 5.13 — and then see if it is possible to apply the function. This is the case each argument contains no unknowns (in other words, is static) or the function being applied is lazy in that argument (and hence the value of the argument may not be needed).

$$\mathbf{U}\,\Gamma \left[\begin{array}{l} x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho \\ x_1 \mapsto x_0\ y_1, \ldots, x_m \mapsto x_{m-1}\ y_m \end{array}\right] x_m = \Gamma''$$
$$\begin{array}{rcll} \multicolumn{4}{l}{\textbf{where}} \\ \Gamma'' & = & \mathbf{P}\,\Gamma'\,x_m, & \textbf{if } \bigwedge_{i=1}^{m}(\mathit{static}\ \Gamma'\ y_i) \vee (\sigma_i = l) \\ & = & \Gamma', & \textbf{otherwise} \\ \Gamma' & = & \mathbf{A}\,\Gamma\,x_m & \end{array} \tag{6.2}$$

The function *static* returns whether a program contains no unknowns:

$$\begin{array}{rcll} \mathit{static}\ \Gamma\ x & = & \mathit{static}\ \Gamma\ f \wedge \mathit{static}\ \Gamma\ a, & \textbf{if } \Gamma\ x = f\ a \\ & = & \Gamma\ x \neq \bot, & \textbf{otherwise} \end{array}$$

The $\mathbf{P}$ meta-function deals with the job of triggering the actual application of the program (using the @ meta-function):

$$\mathbf{P}\,\Gamma \left[\begin{array}{l} x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho, \\ x_1 \mapsto x_0\ y_1, \ldots, x_m \mapsto x_{m-1}\ y_m \end{array}\right] x_m = \Gamma'''$$
$$\begin{array}{rcll} \multicolumn{4}{l}{\textbf{where}} \\ \Gamma''' & = & \mathbf{U}\,\Gamma''\,x_m, & \textbf{if } \rho = l \\ & = & \Gamma'', & \textbf{otherwise} \\ \Gamma'' & = & \mathit{update}\ \Gamma'\ x_m\ r, & \textbf{if } r \neq \bot \\ & = & \Gamma', & \textbf{otherwise} \\ (r, \Gamma') & = & f@(\Gamma, y_1, \ldots, y_m) & \end{array} \tag{6.3}$$

The 'answer' part of the result (r in the above definition) of @ can be $\bot$ if either an error occurs or the value of an unknown variable is required.

6.2 The Operational Semantics

The changes in the operational semantics to handle partial evaluation mirror those made in the denotational semantics. In the first case, if we have an unknown as well as a data object at the top of the stack, then no further work can be done and we need to make no changes. If the dump is non-empty we restore it (which remains the same as described in Rule 5.23), else we terminate as no more rules apply.

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle\rangle \\ \langle x_0, x_1, \ldots, x_n \rangle & \Longrightarrow & \langle\rangle \\ \Gamma[x_0 \mapsto p \in D \cup \{\bot\}] & & \Gamma \\ \Delta & & \Delta \end{array} \tag{6.4}$$

The rules for unwinding an application at the head of the stack and, when we have a function applied to too few or too many arguments at the head of the stack, remain the same as described in Rules 5.16, 5.17 and 5.18 respectively.

If we have a function of arity m at the top of the stack and m other elements at the top of the stack then we first evaluate the necessary arguments and then see if it is possible to trigger the application (as in equation 6.2):

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle \texttt{EVALARGS}\ m \rangle \ +\!\!+\ C \\ \langle x_0, x_1, \ldots, x_m \rangle & & \langle x_0, y_1, \ldots, y_m \rangle \ +\!\!+\ S \\ \Gamma \left[\begin{array}{l} x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho, \\ x_i \mapsto x_{i-1}\ y_i \end{array} \right] & \Longrightarrow & \Gamma \\ \Delta & & \Delta \end{array} \tag{6.5}$$

$$\begin{array}{lcll} \textbf{where} & & & \\ (C, S) & = & (\langle \texttt{APPLY} \rangle, \langle x_m \rangle), & \textbf{if } \bigwedge_{i=1}^{m}((\mathit{static}\ \Gamma\ y_i) \vee (\sigma_i = l)) \\ & = & (\langle\rangle, \langle\rangle), & \textbf{otherwise} \end{array}$$

The rules for the `EVALARGS` and `EVALITH` stay the same as in the non-partial case and defined in Rules 5.20 and 5.21 respectively. The `APPLY` instruction triggers the primitive application as in equation 6.3:

$$
\begin{array}{lcl}
\langle \texttt{APPLY} \rangle & & C \\
\langle x_0, y_1, \ldots, y_m, root \rangle & \Longrightarrow & \langle root \rangle \\
\Gamma[x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho] & & \Gamma'' \\
\Delta & & \Delta
\end{array}
$$

$$
\begin{array}{lrcll}
\textbf{where} & & & & \\
& C & = & \langle\rangle, & \textbf{if } \rho = s \vee r = \bot \\
& & = & \langle \texttt{EVAL} \rangle, & \textbf{otherwise} \\
& \Gamma'' & = & \mathit{update}\ \Gamma'\ \mathit{root}\ r, & \textbf{if } r \neq \bot \\
& & = & \Gamma', & \textbf{otherwise} \\
& (r, \Gamma') & = & f@(\Gamma, y_1, \ldots, y_m) &
\end{array}
\tag{6.6}
$$

6.3 Implementation

To represent unknowns, we introduce the following class to our Aladin implementation described in the previous chapter:

```
public final class Unknown extends Prog {
  private String name;


  // ...
}
```

We store the identifier used to refer to the unknown inside the `Unknown` object since identifier names are otherwise lost at compile time. To signify that an unknown value has been encountered where one was not expected, we create the `UnknownValueException` class, instances of which will be thrown in such cases.

We now need to adapt the evaluation mechanism to deal with unknown values. First of all, we have the `eval` method of the `Var` class. This checks to see if the

`Var` object being evaluated is an unknown and if so throws an exception, if not it continues as before.

```
public Prog eval() throws UnknownValueException {
  if (p instanceof App ||
      p instanceof Function && ((Function) p).arity == 0)
    // only need to bother evaluating if this variable refers to an
    // App or a CAF.
    (new VarStack(this)).transform();
  else if (p instanceof Unknown)
    // Can't evaluate this var further; use the exception to notify
    // the caller that an Unknown was encountered.
    throw new UnknownValueException();

  return get();
}
```

The only required change to the `transform` method, first introduced in Section 5.3.3, is a check to make sure that after any strict arguments have been evaluated no strict argument contains unknowns. This is done by the `checkForStrictUnknowns` method:

```
private void checkForStrictUnknowns(Function f, Var[] args) {
  for (int i = 0; i < args.length; i++)
    if (f.strictIn(i + 1) && args[i].containsUnknowns())
      throw new UnknownValueException();
}
```

and `transform` now becomes:

```
public void transform() throws EvaluationException {
  for (;;) {
    Var v = head();

    Object head = v.get();

    if (head instanceof App)
      push(((App) head).functor);
    else if (head instanceof Function) {
      Function f = (Function) head;
      int no_args = count - 1;

      if (no_args == f.arity) {
        Var[] args = evalArgs(f);
        clearAllButRoot(); // root is elements[0]

        if (AAM.partial)
          checkForStrictUnknowns(f, args);

        Prog res = f.primApply(args);
        elements[0].update(res);

        // stop evaluation if f has a strict result
        if (f.hasStrictResult())
          return;
      }
      // as before
```

```
    }
    // as before
  }
}
```

The `evalArgs` method must also be modified to deal with unknowns:

```
private Var[] evalArgs(Function f) {
  // we need to preserve the ordering -- the stack is held reversed
  // for efficiency when pushing
  Var[] args = new Var[count - 1];
  UnknownValueException unknown = null;

  for (int i = 0; i < count - 1; i++) {
    // get the argument part
    args[i] = ((App) elements[count - (i + 2)].get()).arg;

    if (f.strictIn(i + 1)) {
      try {
        args[i].eval();
      }
      catch (UnknownValueException e) {
        // don't want to throw the exception until we have
        // attempted to evaluate all arguments, so just make
        // a note of its existence for now
        unknown = e;
      }
    }
```

```
  }

  if (unknown != null)
    throw unknown;

  return args;
}
```

Since we need to evaluate every strict argument of the function being applied, we catch any `UnknownValueException`s and only throw one at the end of the method when we have attempted to evaluate all strict arguments.

6.4 Partially Evaluating Programs

Recall that our Aladin implementation takes programs written in the Aladin scripting language and compiles them to Java class files. Suppose we have the following function defined in the file `power.as`:

```
power :: l * s -> l
power x n = if n == 0 then 1 else x * power x (n - 1) endif;
```

Compiling this function yields a Java class file, `power.class`. We can then use this class file along with the `java` interpreter to evaluate Aladin programs. For example:

```
> java power 'power 12 3'
1728
```

We can also supply an unknown value as the first argument of `power` and obtain a specialisation. For instance, we can specialise `power` to a cube function *viz*:

```
> java power -p -pp -n cube 'power x 3'
```

```
cube :: l -> l

cube x = x * (x * (x * 1));
```

The `-p` option denotes that we wish to do partial evaluation (any undefined identifiers are treated as unknowns rather than errors); `-pp` indicates that we wish to 'pretty' print the result (in this case, write the multiplication as an infix function using the `*` symbol, rather than as the alpha-numeric prefix version as would be the normal case); and the `-n` option supplies the name of the residual function. The result of the partial evaluation is an Aladin function, with all unknown values becoming arguments of that function, which can be redirected to a file and compiled.

It is the fact that `power` is lazy in its first argument which allows us to specialise the function like this; if it were strict then no evaluation could be done. This suggests a general rule of thumb when it comes to strictness and partial evaluation: if a parameter at a control point, for example the antecedent part of an `if` expression, then that parameter is strict and cannot have an unknown value; otherwise it is lazy and can have an unknown value. This correspondence was noted by Launchbury [62], who showed that doing strictness analysis and *binding-time analysis* — a pre-evaluation process which determines which expression in a program can be partially evaluated given the limited amount of data that will be present — at the same time could prove useful.

So far, we have only described how to use Aladin to partially evaluate programs at the command line. The residual function must then be put manually into a program file. This has several advantages, for instance, it makes testing easier and printing a program forces all of its components to be (partially) evaluated (for example, every element of a list will be evaluated and printed, rather than just leaving the list in its initial cons form). However, this restricts us to partially evaluating one function at a time and requires the intervention of a human user.

We therefore allow the user to specify that some partial evaluation be done at compile time, by allowing the user to denote that the body of a function is the result of partially evaluating another. For instance, we can define the `cube` function from above as:

```
cube :: s -> l
cube x => power x 3;
```

Here we use the `=>` in place of `=` to indicate that we want to partially evaluate the RHS of the definition, regarding the parameters of the LHS as unknowns, before assigning it as the definition of `cube`. We could legitimately partially evaluate the RHS of *all* function definitions — which would give us such compiler enhancements as constant folding (see Section 14.7.1 of [86] for example), and function unfolding [17, 16]. However, partially evaluating programs may not terminate or doing so may take so much time that the compiler becomes unfeasibly slow.

Compile-time partial evaluation has other advantages. For example, in the above we have specified that `cube` is strict in its argument, whereas producing the function at the command line makes `cube` lazy in its argument by default.

Since our implementation can only evaluate compiled functions (compiled by both Aladin and the Java compiler) any functions which appear on the RHS of function definitions which we wish to partially evaluate must have been pre-compiled, in other words, they cannot appear in the same file as the function being partially evaluated. This is only a minor inconvenience and does not affect our analysis.

Partially evaluating a program may lead to a residual program that is less space efficient than the original one. This is because the original function(s) may have been implemented to use space-saving devices such as accumulating parameters and tail recursion which might be expanded in the definition of the residual function(s). It is hard to predict in advance when this might occur to a detrimental effect since predicting the space behaviour of functional programs is notoriously difficult [15, 86, 97], and it is up to the user whether the the speed-up due to partial evaluation is outweighed by any additional space used. We are only concerned with the time behaviour of partially-evaluated programs in this chapter, and will not consider space behaviour further. However, we have not yet encountered any problems caused by space inefficiency during our investigations into partial evaluation.

6.5 Primitives and Partial Evaluation

Some primitives can be rewritten to more comprehensively handle partial evaluation. For instance, consider the comparison operations. For an unknown `x`, we know the following:

$$\begin{aligned}
\texttt{x == x} &\Longrightarrow \texttt{TRUE}\\
\texttt{x \~{}= x} &\Longrightarrow \texttt{FALSE}\\
\texttt{x <= x} &\Longrightarrow \texttt{TRUE}\\
\texttt{x < x} &\Longrightarrow \texttt{FALSE}\\
\texttt{x >= x} &\Longrightarrow \texttt{TRUE}\\
\texttt{x > x} &\Longrightarrow \texttt{FALSE}
\end{aligned}$$

The above equations can be incorporated into the code for the primitives, at the cost of a little more programming effort and making the operators lazy in both their arguments. For example, we can rewrite the method `_op_eq` (the prefix form of the equality operator `==`) defined in Section 5.4.4 as:

```
public final static Prog _op_eq(Var v, Var w) {
  Prog a = v.eval_nu();
  Prog b = w.eval_nu();
```

```
  if (a.containsUnknowns() || b.containsUnknowns()) {
    if (a instanceof Unknown && b instanceof Unknown && a.equals(b))
      return Bool.TRUE;
    else
      // one or both of v or w contains an unknown and they are not
      // the same unknown, hence we can't complete the equality
      throw new UnknownValueException();
  }
  else if (a instanceof Comparable)
    return ((Comparable) a).eq(b);
  else
    return (a.equals(b)) ? Bool.TRUE : Bool.FALSE;
}
```

The `eval_nu` method is similar to the `eval` method except that it catches and discards any `UnknownValueExceptions`. The `containsUnknowns` method returns whether the program it was invoked upon contains any unknowns.

The method first of all evaluates its arguments (and does not have to worry about any `UnknownValueExceptions`) and then checks to see if either contains any unknown values. If so, it checks to see if they are in fact the same single unknown. If so we know they have to be equal, despite not knowing their actual values, otherwise we cannot complete the inequality and throw the exception to indicate this.

As an example, consider the function `min` which returns the minimum of its two arguments (and stored in the script `power.as`):

```
min :: l * l -> l
min x y = if x <= y then x else y endif;
```

Partially evaluating this with two unknown, but equal, arguments gives:

```
> java power -p -pp -n min_xx 'min x x'
min_xx :: l -> l
min_xx x = x;
```

Using such an implementation of the comparison operators can have an effect on the termination properties of residual functions, in that the residual function can terminate in more cases than the original function when restricted to the same arguments as the residual. For example, consider the simple function:

```
foo :: l * l -> l
foo a b = if a == b then 1 else 0 endif;
```

Partially evaluating this with two unknown, but equal, arguments gives:

```
> java foo -p -pp -n foo1 'foo x x'
foo1 :: l -> l
foo1 x = 1;
```

Now `foo1` $\perp$ = `1`, but the equivalent call to `foo`, `foo` $\perp$ $\perp$, returns $\perp$, since $\perp$ `==` $\perp = \perp$. This is a beneficial side-effect of partial evaluation, analogous to the fact that lazy functional programs can terminate in more cases than their strict equivalents [15]. Note that the reverse situation cannot be true, since the set of individial evaluations done during partial evaluating a program then evaluating the residual function is a subset of the set of individual evaluations done during full evaluation of the same program. If the result of a program is $\perp$ then the result of at least one of these individual evaluations must be $\perp$ and *vice versa* — hence if the residual function returns $\perp$ when applied to some particular arguments, then the original function must also return $\perp$ when applied to the equivalent arguments.

The comparison operators are not the only functions which can be re-coded to exploit partial evaluation. We can also re-implement the binary logical primitives to take advantage of the following rules:

```
 x & true  =  true & x  =  x
x & false  = false & x  = false
 x | true  =  true | x  = true
x | false  = false | x  =  x
```

Note this holds true in the fuzzy as well as the boolean case. The standard implementation of these primitives have the strictness $s \times l \rightarrow l$. Logical conjunction is performed by the `_op_and` method defined as:

```
public static Prog _op_and(Var x, Var y) {
  Prog a = x.get();

  if (a instanceof Logical)
    return ((Logical) a).and(y);
  else
    throw new EvaluationException("Can't take the logical " +
                                  "conjunction of " + a +
                                  " and " + y);
  }
}
```

with logical disjunction (`_op_or`) being defined similarly. Rewriting the definitions of these primitives to take advantage of partial evaluation first requires us to to make them lazy in both arguments as with the comparison operators. Also (exploiting commutativity) if the first argument contains unknowns we switch the order of the arguments and see if we can obtain a result by treating the second argument as the first and *vice versa*:

```
public static Prog _op_and(Var x, Var y) {
```

```
  Prog a = x.eval_nu();

  if (a instanceof Logical)
    return ((Logical) a).and(y);
  else {
    Prog b = y.eval_nu();

    if (b instanceof Logical)
      return ((Logical) b).and(x);
    else if (a.containsUnknowns() || b.containsUnknowns())
      throw new UnknownValueException();
    else
      throw new EvaluationException("Can't take the logical " +
                                    "conjunction of " + a +
                                    " and " + y);
  }
}
```

The definition of the `and` method can stay the same as in the non-partial case in whatever class implements it. For instance, in the `Bool` class we have:

```
public Prog and(Var v) {
  return (value) ? (Prog) v : (Prog) Bool.FALSE;
}
```

where `value` is the boolean value of the `Bool` object. As an example, we have:

```
> java power -pp -p -n prog 'x <= 3 & (y < 2 | 13 > (5 + 6))'
prog :: l * l -> l
prog x y = x <= 3;
```

This strategy is not only useful for partially evaluating binary logical primitives: it can be used on any commutative primitive which can be made lazy in one of its arguments. It is especially effective if the operator has an annihilator since it can be used to avoid evaluating one of the arguments altogether. For instance, from above we see that `true` is an annihilator of logical disjunction and `false` one of conjunction, but also 0 is an annihilator of multiplication and division. This is used to great effect in Section 6.6

6.5.1 Partial Data Structures

Aladin treats data objects as indivisible entities, in particular, as far as it is concerned the list `[1, y, 3, x, 5]` does *not* contain any unknowns, since by default data objects do not. This means that such a list can be used as a strict argument of a function and still allow some useful partial evaluation to be done. For instance, given the following definition of `reverse`:

```
reverse :: s -> l
reverse xs = rev xs [];


rev :: s * s -> l
rev xs ys =
    if isEmpty xs then ys
    else rev (tl xs) (hd xs : ys)
    endif;
```

partially evaluating `reverse [1, y, 3, x, 5]` yields:

```
> java power -p -pp2 -n rev1 'reverse [1, y, 3, x, 5]'
rev1 :: l * l -> l
rev1 y x = [5, x, 3, y, 1];
```

To be able to do more useful partial evaluation on lists and other data structures it is useful if we can do (in)equality tests on them. For instance, we could partially evaluate `[x, y, z] == [1, 2, 3]` to `x == 1 & y == 2 & z == 3`. This requires us to alter the implementation of the `Comparable` interface (see Section 5.4.4) in the `SumProd` class so that it can handle unknown values. For the `eq` method we have:

```
public Prog eq(Prog p) {
  if (this.getClass().equals(p.getClass())) {
    SumProd sp = (SumProd) p;

    Prog result = Bool.TRUE;
    Var and = AAM.get("_op_and");
    Var eq = AAM.get("_op_eq");

    for (int i = 0; i < fields.length; i++) {
      Prog e = (new Var(eq, fields[i], sp.fields[i])).eval_nu();

      if (e instanceof Bool) {
        // equality went to completion
        if (e.equals(Bool.FALSE))
          // found fields that don't correspond, so can return now
          return Bool.FALSE;
      }
      else
        // must have unknowns so add to list of conjunctions
        result = (result == Bool.TRUE) ? e : new App(and, e, result);
    }
```

```
    return result;
  }

  // If we've got here then can't be equal
  return Bool.FALSE;
}
```

This method first checks that the two objects to be compared are of the same class and then steps through each of the fields, seeing if they are equal. If it finds two that do not match, then it immediately returns false; else it builds the result into a conjunction (eliminating any redundant trues). We can define `ne` similarly. So, for example, we have:

```
> java power -p -pp -n eq3 '[x,y,z] == [1,2,3]'
eq3 :: l * l * l -> l
eq3 x y z = (z == 3 & y == 2) & x == 1;
```

and:

```
> java power -p -pp -n neq2 "(x, y, 'a') ~= (false, 2, 'a')"
neq2 :: l * l -> l
neq2 x y = y ~= 2 | x ~= false;
```

As a further example, consider the function `palindrome` in `power.as`:

```
palindrome :: s -> l
palindrome xs =
    let
        n = #xs / 2;
    in
```

```
        take n xs == take n (reverse xs)
    endlet;
```

We can specialise this to a function which checks if a list of length 5 is a palindrome:

```
> java power -p -pp -n pal5 'palindrome [a,b,c,d,e]'
pal5 :: l * l * l * l * l -> l
pal5 a b c d e = b == d & a == e;
```

This partial evaluation leads to arity raising which is normally considered a good thing since we avoid having to construct and deconstruct data structures [35, 95]. However it may be preferable to have the residual function take a single list as its only argument. For this to work, we have to either have an alternative form of `palindrome` which takes the length of the list as a parameter, or some other way of deconstructing the list into its (here 5) constituent parts. We choose the latter, using the function `decons` to deconstruct a list into a specified number of elements:

```
decons :: s * l -> l
decons n xs = if n == 0 then []
              else hd xs : decons (n - 1) (tl xs)
              endif;
```

Note that by making the second (list) argument lazy, we can use an unknown in its place. So, for instance:

```
> java power -p -pp2 -n list5 decons 5 xs
list5 :: l -> l
list5 xs = [hd xs, hd (tl xs), hd (tl (tl xs)),
            hd (tl (tl (tl xs))),
            hd (tl (tl (tl (tl xs))))];
```

Note that the common subprograms here will be lifted out by the compiler as described in Section 5.5.5. So, the above functions is optimised to:

```
list5 xs =
    let
        v2 = hd;
        v1 = tl;
        v4 = (v1 xs);
        v5 = (v1 v4);
        v3 = (v1 v5);
        v0 = (:);
    in
        (v0 (v2 xs)
            (v0 (v2 v4)
                (v0 (v2 v5)
                    (v0 (v2 v3)
                        (v0 (v2 (v1 v3))
                            _op_list_empty)))))
    endlet;
```

However, for purposes of readability, all future examples of residual functions will be given in non-optimised form.

We can now produce a specialisation of `palindrome` which takes a single argument:

```
> java power -p -pp2 -n pal5 'palindrome (decons 5 xs)'
pal5 :: l -> l
pal5 xs = hd (tl xs) == hd (tl (tl (tl xs))) &
          hd xs == hd (tl (tl (tl (tl xs))))
```

The above data structures are lazy in that the evaluation of their components is not done until required, which might not be until the printing stage. This can prevent some useful partial evaluation being done at compile time if the components of a data structure can be evaluated but evaluation has stopped because Aladin does not evaluate the components of data structures. For instance, `[(1 + 2) .. 10]` evaluates to `1 + 2 : [((1 + 2) + 1) .. 10]`. If we evaluated this program at the command line then we would not notice this, since printing the list forces its evaluation, but if the list was evaluated at compile time then it would only be evaluated to the first cons. To alleviate this problem, we provide the primitive `force`, as in Miranda and Ginger, which forces evaluation of all parts of its argument in the exact same way as printing it would. Care must be taken not to use `force` on infinite data structures.

For instance, the specialisation of matrix multiplication to 2×2 matrices used in Section 6.6.1 is coded as:

```
mult_2x2 :: l -> l
mult_2x2 p =>
    let
        m1 = dedecons 2 2 (fst p);
        m2 = dedecons 2 2 (snd p);
    in
        force (mult m1 m2)
    endlet;
```

Here `dedecons` n m deconstructs a list of lists into a list of n elements each of which is a list of m elements.

6.5.2 Partial Evaluation and C Primitives

To be able to partially evaluate C/C++ primitives we need to be able to deal with `UnknownValueException`s. Also, being able to detect unknown values is also useful if we wish to do something other than abort evaluation when unknown values occur (see below). First of all, the following macro will abort a function if *any* exception occurs:

```
#define abortIfException(env) \
if ((*(env))->ExceptionOccurred(env)) return 0
```

The result of the function in which the exception occurred will be null and the exception is thrown back up to the JVM. For example, consider the following power function, which will be imported into an Aladin script `update.as`. This function is declared to have strictness $l \times s \to s$ and its implementation is:

```
JNIEXPORT jobject JNICALL Java_update_power
  (JNIEnv *env, jclass cl, jobject x, jobject y)
{
  jint power = getInt(env, y);

  if (power == 0)
    return makeInt(env, 1);
  else {
    jint base = evalInt(env, x), result = base;

    // abort execution if any exceptions have occurred
    abortIfException(env);

    for (; power > 1; power--)
```

```
      result *= base;

    return makeInt(env, result);
  }
}
```

This gets the value of `y` and if it is zero immediately returns 1. Otherwise it evaluates `x` and extracts its integer result. At this point, if `x` contains unknowns an `UnknownValueException` will have been thrown. This will be detected by the `abortIfException` macro and evaluation will be immediately aborted. If things went well, we just compute the power using an iterative method and return.

An alternative way is to evaluate `x` and explicitly test to see if it contains unknowns. In this case, we can unfold the exponentiation into multiple application of the multiplication function, `_op_times`. We provide a `containsUnknowns` function analogous to its Java namesake and functions and macros which will create and throw the appropriate exceptions. So, we can rewrite `power` as:

```
JNIEXPORT jobject JNICALL Java_update_power
  (JNIEnv *env, jclass cl, jobject x, jobject y)
{
  jint power = getInt(env, y);

  if (power == 0)
    return makeInt(env, 1);
  else {
    eval(env, x);

    if (containsUnknowns(env, x)) {
```

```
      /* return multiple applications of '*' */
      jobject timesx = makeApp(env,
                               getFunction(env, "_op_times"), x);
      jobject mult = x;

      for (; power > 1; power--)
        mult = makeApp(env, timesx, mult);

      return mult;
    }
    else {
      /* can work out the full answer */
      jint base = getInt(env, x);
      jint result = base;

      for (; power > 1; power--)
        result *= base;

      return makeInt(env, result);
    }
  }
}
```

6.5.3 Partial Evaluation and Ginger Primitives

As our Ginger compiler produces Java byte-code (Chapter 4), we do not have to worry about explicitly detecting and throwing exceptions since this is handled by the JVM. We do, however, have to worry about `Unknown` objects since there is no equivalent in Ginger and we need to convert all Aladin programs into Ginger ones before applying the Ginger primitive. To deal with this we introduce the `GingerProg` class which represents a 'suspended' conversion:

```
public final class GingerProg extends fp.gingerc.App {
  protected Var aladin;

  public GingerProg(Var v) {
    aladin = v;
    args = empty;
    total_app = true;
    in_whnf = false;
  }

  public Object eval() {
    aladin.eval();

    return aladin.toGinger();
  }
}
```

All *lazy* arguments of a Ginger primitive will be converted to the above class; strict arguments cannot contain unknowns (since otherwise we would not be applying the primitive) and so can be safely converted. When (and only when) Ginger attempts to evaluate an object, the Aladin program is evaluated and then converted to its Ginger representation. This means that if a lazy argument of a Ginger primitive contains unknowns, we can still execute the Ginger primitive and only if the argument contains unknowns after it has been evaluated (which may not be required) will an exception be thrown, since trying to convert an `Unknown` to a Ginger representation will raise an `UnknownValueException`. This has the added benefit that if a lazy argument is part of the result of a Ginger primitive and is unevaluated throughout then no conversion from Aladin to Ginger and *vice versa* has to be done: we can simply extract the value of the `aladin` field.

6.6 Further Examples and Results

One question needs to be asked: does partially evaluating Aladin programs have any benefit? In this section we shall present five further, more substantial examples of programs where some partial evaluation can be done and then compare performances. Note that we have given the results of partial evaluation at the command line since this is the most readable form, but all partial evaluation described below is done at the compile time for the purposes of obtaining results. Note that for display purposes we have split the answer Aladin returns into separate lines and indented appropriately, but no other manipulation of the answers has been done.

6.6.1 Matrix Multiplication

Consider the following program to multiply two matrices together, where a matrix is represented by a list of lists:

```
row :: s * s -> l
row n xss = xss ! n;


col :: s * s -> l
col n xss = map (flip (!) n) xss;


no_rows :: s -> l
```

```
no_rows xss = # xss;


no_cols :: s -> l

no_cols xss = # (hd xss);


mprod :: s * s * s * s -> l

mprod xss yss r c =

    if r == no_rows xss then []

    elsif c == no_cols yss then [] : mprod xss yss (r + 1) 0

    else

        let

            m = mprod xss yss r (c + 1);

            v = sum (zipWith (*) (row r xss) (col c yss));

        in

            (v : (hd m)) : (tl m)

        endlet

    endif;


mult :: s * s -> l

mult xss yss = mprod xss yss 0 0;
```

Matrix multplication has many uses. One such use is the rotation of points about some origin in an n-dimensional space. For example, the result of rotating a two-dimensional point (x, y) clockwise about the origin by θ radians is given by the result of the matrix multiplication:

$$\left(\begin{array}{cc} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{array} \right) \cdot \left(\begin{array}{c} x \\ y \end{array} \right)$$

This can be coded as the following Aladin function:

```
// coords as a pair, theta in radians
rotate :: l * l -> l
rotate theta coords =
    let
        x = fst coords;
        y = snd coords;
        rcoords = mult [[cos theta, sin theta],
                         [neg (sin theta), cos theta]]
                        [[x], [y]];
    in
        (hd (rcoords ! 0), hd (rcoords ! 1))
    endlet;
```

The application of `mult` used in the definition of `rotate` is fixed to matrices of a specific size, namely a 2×2 matrix multipled by a 2×1 matrix, and hence we have scope for partial evaluation:

```
> java matrix -p -pp2 'rotate theta (x, y)'
rotate :: l * l * l -> l
rotate theta x y = (cos theta * x + (sin theta * y + 0),
                    neg (sin theta) * x + (cos theta * y + 0));
```

All list operations, conditionals and recursive calls have disappeared and all we are left with are simple mathematical operations.

More partial evaluation can be done by fixing `theta` to be a specifc angle, for example $\pi/2$:

```
> java matrix -p -pp2 -n quarter 'rotate (pi / 2) (x, y)'
quarter :: l * l -> l
quarter x y = (0 * x + (1 * y + 0), -1 * x + (0 * y + 0));
```

These residual programs contain some redundant operations, namely the addition of zero and multiplication by one and zero. We can eliminate this by providing alternative definitions of `_op_plus` and `_op_times`. Rather than re-implementing the functions from scratch, we can provide wrapper functions to the original functions to do the job. First of all we define the functions `isZero` and `isOne` of strictness $l \rightarrow s$ which tests if a value is definitely zero or definitely one. We can implement `isZero` *viz*:

```
public static Prog isZero(Var x) {
  try {
    Prog p = x.eval();

    if (p instanceof Int)
      return ((Int) p).value == 0 ? Bool.TRUE : Bool.FALSE;
    else if (p instanceof Real) {
      double d = ((Real) p).value;

      return (-1.0E-7 < d && d < 1.0E-7) ? Bool.TRUE : Bool.FALSE;
    }
    else
      return Bool.FALSE;
  }
  catch (UnknownValueException e) {
    return Bool.FALSE;
  }
}
```

If x contains unknowns then it cannot definitely be zero, hence we return false. Note that if x refers to a real we return true if the value is in the range $(-1 \times 10^{-7}, 1 \times 10^{-7})$ so as to cope with rounding errors. The function isOne is defined similarly. We can now define our wrapper functions, for example, _op_times *viz*:

```
_op_times :: l * l -> l
_op_times x y =
    if isZero x | isOne y then x
    elsif isZero y | isOne x then y
    else Operators._op_times x y
    endif;
```

where the method Operators._op_times is the original definition of multiplication defined in Aladin. Note that if either x or y contain unknowns then we default to the standard definition and so do not drag the graph of the if statement around while evaluating the rest of the program. Our specialisation of rotation for an angle of $\pi/2$ is now:

```
> java matrix -p -pp2 -n quarter 'rotate (pi / 2) (x, y)'
quarter :: l * l -> l
quarter x y = (y, -1 * x);
```

In the tests which we obtain results for, we shall specialise rotate to angles of $\pi/2$, π and $3\pi/2$ and rotate 256 points by these angles.

6.6.2 Gaussian Elimination

Gaussian Elimination (see [59], for example) is a systematic way of solving systems of linear equations. If we have a system:

$$a_{11}x_1 + \ldots a_{1n}x_n = b_1$$

$$\vdots$$

$$a_{m1}x_1 + \ldots a_{mn}x_n \quad = \quad b_m$$

Then we can form the *augmented* matrix:

$$\begin{pmatrix} a_{11} & \ldots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{m1} & \ldots & a_{mn} & b_m \end{pmatrix}$$

and by a combination of *forward elimination* and *back substitution* we can obtain a solution, if one exists. This is a problem ripe for partial evaluation, especially if the a_{ij} are known and our matrix is sparse (that it, has a lot of zero entries). This was first recognised in [31] though not described as partial evaluation, as such. The Aladin program implementin Gaussian elimination is:

```
// augment an m x n1 matrix with an m x n2 one
augment :: -> s
augment = zipWith snoc;


// add an element onto the end of a list
snoc :: s * l -> l
snoc xs x = xs ++ [x];


// partition xs into a pair of lists, the first element of which
// is all those elements of xs satisfies the predicate p, the
// second one all those that don't
splitWith :: s * s -> l
splitWith p xs =
    if isEmpty xs then ([], [])
    else
```

```
        let
            ps = splitWith p (tl xs);
            x  = hd xs;
        in
            if p x then (x : fst ps, snd ps)
            else (fst ps, x : snd ps)
            endif
        endlet
    endif;

// reorder xs, s.t. all those elements of xs which satisfy p
// come before those that don't
reorder :: s * s -> l
reorder p xs =
    let
        ps = splitWith p xs;
    in
        fst ps ++ snd ps
    endlet;

// returns whether the kth element of xs is not zero
nonZeroKth :: s * s -> l
nonZeroKth k xs = (xs ! k) ~= 0.0;

// pivot/reorder the rows of the matrix xss s.t. all the rows of
// xss with a non-zero kth element come before those that don't
pivot :: s * s -> l
```

```
pivot xss k =
    let
        p   = nonZeroKth k;
        yss = reorder p xss;
    in
        if p (hd yss) then yss else error "Singular Matrix" endif
    endlet;

// subtract a constant multiple of the first row of the given matrix
// from all proceeding rows so that in the result matrix all bar the
// first row have a zero element in the kth position (forward
// elimation) the resultant matrix will be in triangular form
forwardElim :: s * s -> l
forwardElim k xss =
    if isEmpty xss then []
    else
        let
            pss    = pivot xss k;
            first  = hd pss;
            rest   = tl pss;
            factor = first ! k; // Non-zero
            rows   = map (subtractRows first k) rest;
        in
            first : forwardElim (k + 1) rows
        endlet
    endif;
```

```
// auxiliary function for forwardElim: performs the actual
// multiplication/subtraction
subtractRows :: s * s * s -> l
subtractRows xs k ys =
    let
        m = (ys ! k) / (xs ! k);
    in
        zipWith (-) ys (map ((*) m) xs)
    endlet;

// back substitutes the values obtained for each column of the
// matrix (i.e., variable when the matrix is considered as a system
// of equatiuons) starting from the last row, which should have
// only one non-zero column in the unaugmented part.
backSub :: s -> l
backSub rs =
    if isEmpty rs then []
    else
        let
            xs = backSub (map tl (tl rs));
            r  = hd rs;
            a  = hd r;
            as = tl r;
            b  = last r;
        in
            ((b - sum (zipWith (*) as xs)) / a) : xs
        endlet
```

```
    endif;

// reduce a given augmented matrix to triangular form and use back
// subsitition to solve the underlying simultaneous equations
solve :: -> s
solve = backSub . forwardElim 0;
```

Now, suppose we have the following system of equations:

$$\begin{aligned} 3x_1 - 2x_2 &= 1 \\ 2x_1 + 5x_2 &= 26 \end{aligned}$$

Then we can include the following matrix in our program:

```
two_x_two :: ->l
two_x_two = [[3.0, -2.0], [2.0, 5.0]];
```

The above system of equations can be solved by augmenting this matrix with the values of the RHSs of the above equations and using this augmented matrix as the argument to the `solve` function:

```
> java gauss_npe -pp2 -p 'solve (augment two_x_two [1, 26])'
[3.0, 4.0]
```

That is, $x_1 = 3$ and $x_2 = 4$. Alternatively we can choose to leave the RHSs of the system of equations unknown:

$$\begin{aligned} 3x_1 - 2x_2 &= b_1 \\ 2x_1 + 5x_2 &= b_2 \end{aligned}$$

and augment the matrix with these unknown values. Partial evaluation can then give us values for x_1 and x_2 in terms of these unknowns:

```
> java gauss_pe -pp2 -p -dp 3 'solve (augment two_x_two [b1, b2])'
solve :: l * l -> l
solve b1 b2 = [(b1 - -2 * ((b2 - 0.667 * b1) / 6.333)) / 3,
               (b2 - 0.667 * b1) / 6.333];
```

where the value of x_1 is the first element of the list on the RHS of the function definition, and x_2 the second element.

In the tests we shall give times for, we have the following system of equations (from [31]):

$$\begin{array}{cccccccl}
7x_1 & -3x_3 & & & -x_5 & & = & b_1 \\
2x_1 & +8x_2 & & & & & = & b_2 \\
 & & +x_3 & & & & = & b_3 \\
-3x_1 & & & +5x_4 & & & = & b_4 \\
 & -x_2 & & & +4x_5 & & = & b_5 \\
 & & & -2x_4 & & +6x_6 & = & b_6
\end{array}$$

which we solve for $\{b_1, \ldots, b_6\} \in$ *permutations* $\{1, \ldots, 6\}$.

6.6.3 Exponentiation

We can use the definition of e^x (where $e = 2.718\ldots$):

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

to obtain an exponential function in Aladin:

```
// return the exponential of z, using n iterations
exp :: s * l -> l
exp n z =
    if n == 0 then 1.0
    else exp1 n z 1.0 1.0 1.0 1
```

```
    endif;


// return the exponential of z, using n interations; e_z is the
// current approximation; num the current numerator and denom the
// current denominator of the term of the power series; and i the
// current power of the power series
exp1 :: s * l * l * l * s * s -> l
exp1 n z e_z num denom i =
    if i == n then e_z
    else
        let
            new_num   = z * num;
            new_denom = i * denom;
            new_e_z   = e_z + new_num / new_denom;
        in
            exp1 n z new_e_z new_num new_denom (i + 1)
        endlet
    endif;
```

This can be specialised for a specific n, for example:

```
> java exp -p -pp2 -n exp6 'exp 6 z'
exp6 :: l -> l
exp6 z =
    ((((1.0 + z) + (z * z) / 2) + (z * (z * z)) / 6) +
     (z * (z * (z * z))) / 24) + (z * (z * (z * (z * z)))) / 120;
```

In the tests we shall do, we shall specialise n to 20 and use it to work out e^x for $x \in \{0, \pi/720, \ldots, \pi\}$.

6.6.4 Polynomials

Polynomials can be represented as lists of their coefficients. For example, $a_0 + a_1 x + \ldots + a_n x^n$ can be represented as the list `[`a_0`,`a_1`, ...`a_n`]`. Evaluation of a polynomial for a specific value of x can be coded in Aladin as:

```
eval :: s * l -> l
eval p x =
    if isEmpty p then 0
    else hd p + x * eval (tl p) x
    endif;
```

It is fairly simple to add two polynomials together:

```
add :: s * s -> l
add p q =
    if isEmpty p then q
    elsif isEmpty q then p
    else hd p + hd q : add (tl p) (tl q)
    endif;
```

Once we have defined addition of polynomails, we can define multiplication:

```
mult :: l * l -> l
mult p q = foldr add [] (summands 0 p q);


// return a list of the summands in the sum; each summand is the
// product of the polynomial q with each element of q; ord is the
// current power
summands :: s * s * l -> l
summands ord p q =
```

```
    if isEmpty p then []
    else
        let
            s = rep ord 0 ++ map ((*) (hd p)) q;
        in
            s : summands (ord + 1) (tl p) q
        endlet
    endif;
```

These functions can be specialised for polynomials of a certain degree, or even for polynomials for fixed coefficients. For example, we can obtain a specialised function which multiplies two quadratics together:

```
> java poly -pp2 -p -n mult2 'mult [a0,a1,a2] [b0,b1,b2]'
mult2 :: l * l * l * l * l * l -> l
mult2 a0 a1 a2 b0 b1 b2 = [a0 * b0,
                           a0 * b1 + a1 * b0,
                           a0 * b2 + (a1 * b1 + a2 * b0),
                           a1 * b2 + a2 * b1,
                           a2 * b2];
```

In the tests we shall do, we perform 720 quadratic multiplications.

6.6.5 Integration by Simpson's Rule

Simpson's rule for integration of a function f over the range $[a, b]$ over $2n$ sub intervals is given as:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots 2f_{2n-2} + 4f_{2n-1} + f_{2n})$$

where $h = (b-a)/2n$ and $f_i = f(a + hi)$. This can be encoded as the following Aladin program:

```
// integrate f over the range [a, b] which is split into 2n intervals
integrate :: s * l * l * l -> l
integrate n a b f =
    let
        h = (b - a) / (2.0 * n); // force real division
    in
        (h / 3) * (sumInt (2 * n - 1) f h (a + h) (f a + f b))
    endlet;

// returns the sum of integrating the f over m intervals, where
// each interval starts at x and is h wide; sub_total is the
// cumulative total
sumInt :: s * l * l * l * l -> l
sumInt m f h x sub_total =
    if m == 0 then sub_total
    else
        let
            c = if m % 2 == 0 then 2 else 4 endif;
        in
            sumInt (m - 1) f h (x + h) (sub_total + c * f x)
        endlet
    endif;
```

There is much scope for partial evaluation here. We can specialise `integrate` on a fixed number of sub-intervals, 4 for example:

```
> java integrate -p -pp2 -n integrate2 'integrate 2 a b f'
integrate2 :: l * l * l -> l
```

```
integrate2 a b f =
    (((b - a) / 4.0) / 3) *
    ((((f a + f b) +
       4 * f (a + (b - a) / 4.0)) +
       2 * f ((a + (b - a) / 4.0) + (b - a) / 4.0)) +
       4 * f (((a + (b - a) / 4.0) + (b - a) / 4.0) +
              (b - a) / 4.0);
```

There is also the possibility of specialising for a fixed interval too, $[0, 1]$, say:

```
> java integrate -p -pp2 -dp 4 -n int2_01 'integrate 2 0.0 1.0 f'
int2_01 :: l -> l
int2_01 f =
    0.0833 * ((((f 0 + f 1) +
                4 * f (0 + 0.25)) +
                2 * f ((0 + 0.25) + 0.25)) +
                4 * f (((0 + 0.25) + 0.25) + 0.25));
```

Note that the argument to the occurrences of `f` in the sum have not been evaluated, even though they could have been. This is because `sumInt` is lazy in its third and fourth arguments (the width of the interval, `h`, and the current value `x`) to allow partial evaluation to be done when the range is not known. If the range is known, we can change the strictness of `sumInt` to $s \times l \times s \times s \times l \to l$ and do more partial evaluation:

```
> java integrate -p -pp2 -dp 4 -n int2_01 'integrate 2 0.0 1.0 f'
int2_01 :: l -> l
int2_01 f =
    0.0833 * ((((f 0 + f 1) + 4 * f 0.25) + 2 * f 0.5) + 4 * f 0.75);
```

In the tests we shall do, we shall use the above to calculate values for the standardised normal distribution, that is evaluate $\Phi(z)$ for $z \in \{-4, -3.99, \ldots, 4\}$ where:

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{\infty}^{z} e^{-x^2/2}\, dx$$

This is encoded in Aladin as:

```
e_minus_half_x2 :: l -> l
e_minus_half_x2 x = MathPrimitives.exp ((x * x) / -2.0);


phi :: s -> l
phi z =
    let
        f = 1 / sqrt (2 * pi);
    in
        f * integrate 8 -6.0 z e_minus_half_x2
    endlet;
```

This uses a lower bound of -6 (the integral over $(-\infty, -6)$ is sufficiently small to be ignored) and 16 sub-intervals. We can partially evaluate this function simply by replacing `phi z =` with `phi z =>` and letting the compiler do the work.

6.6.6 Fuzzy Systems

Recall from Section 5.4.5 the definition in Aladin of the shower controller:

```
change_valves :: s * s -> l
change_valves temp flow =
    let
        defuz = centroid changedom;
        changes = rulebase (applyBin add) [
```

```
        applyUn (when (cold temp & weak flow))   (pm,  z),
        applyUn (when (cold temp & right flow))  (pm,  z),
        applyUn (when (cold temp & strong flow)) ( z, nb),
        applyUn (when (ok temp & weak flow))     (ps, ps),
        applyUn (when (ok temp & strong flow))   (ns, ns),
        applyUn (when (hot temp & weak flow))    ( z, pb),
        applyUn (when (hot temp & right flow))   (nm,  z),
        applyUn (when (hot temp & strong flow))  (nb,  z)];
    in
       (defuz (fst changes), defuz (snd changes))
    endlet;
```

where `changedom` is the list `[-0.2, -0.175 .. 0.2]`. The function `centroid` is defined as:

```
centroid :: s * l -> l
centroid dom f =
    let
        fdom = map f dom;
    in
        sum (zipWith (*) dom fdom) / sum fdom
    endlet;
```

Naturally, we would expect do be able to partially evaluate `centroid` by fixing `dom` to be `changedom` and indeed we can do so to beneficial effect. We would not expect to be able to fix either of the arguments of `change_valves` since it is expected that the function is to be used in a dynamic situation where the parameters are constantly changing. However, there is a somewhat surprising scope for partial evaluation arising from the fact that our fuzzy subsets may overlap, but not to any great extent.

Consider the calculation needed to compute the change in the hot valve (the first element of the pair in the rule base). We weight and sum the fuzzy subsets and pass this as the `f` parameter to `centroid`. This has then to work out the degree to which each element of `changedom` is in this sum, amongst other things. Now, the way that fuzzy sets are normally arranged in a fuzzy system means that a value usually occurs in at most two different fuzzy subsets to a non-zero degree. For instance, -0.2 is only in the fuzzy subset `nb` to a non-zero degree. This means that for each element of `changedom` only a limited amount of rules can possibly contribute to the weighted sum, in the case of -0.2 only the last rule contributes. For each element of `changedom` and we can partially evaluate the weighted sum of eight fuzzy subsets down to the weighted sum of one or two. For the changes to the hot valve (the changes to the cold valve partial are similar) over `changedom` the weighted sum reduces to:

```
[(-0.2,   up 36 75 temp & up 12 25 flow),
 (-0.175, (up 36 75 temp & up 12 25 flow) * 0.833),
 (-0.15,  (up 36 75 temp & up 12 25 flow) * 0.667),
 (-0.125, (up 36 75 temp & up 12 25 flow) * 0.5),
 (-0.1,   (up 36 75 temp & up 12 25 flow) * 0.333),
 (-0.075, (up 36 75 temp & atri 9 ((9 + 15) / 2) 15 flow) * 0.667 +
          (up 36 75 temp & up 12 25 flow) * 0.167),
 (-0.05,  (up 36 75 temp & atri 9 ((9 + 15) / 2) 15 flow) * 0.667),
 (-0.025, atri 32 ((32 + 40) / 2) 40 temp & up 12 25 flow),
 (0,      (down 15 36 temp & up 12 25 flow) +
          (up 36 75 temp & down 0 12 flow)),
 (0.025,  atri 32 ((32 + 40) / 2) 40 temp & down 0 12 flow),
```

```
(0.050,  (down 15 36 temp & down 0 12 flow) * 0.667 +
         (down 15 36 temp & atri 9 ((9 + 15) / 2) 15 flow) * 0.667),
(0.075,  (down 15 36 temp & down 0 12 flow) * 0.667 +
         (down 15 36 temp & atri 9 ((9 + 15) / 2) 15 flow) * 0.667),
(0.100,  0),
(0.125,  0),
(0.150,  0),
(0.175,  0)];
```

We use the definitions of multiplication and addition in Section 6.6.1 to eliminate any redundant arithmetic operations. Note that the definitions of `hot`, `strong`, etc. have been evaluated down to their representations as standard fuzzy subsets and also that `atri` is lazy in its second argument (the value at which the membership function hits the value 1) and hence `(9 + 15) / 2` and `(32 + 40) / 2` remain unevaluated.

We can now partially evaluate `change_valves`, though we have to make it lazy in both its arguments and use a `force` to force evaluation of the two fields of the resultant pair. In the tests we give results for, we calculate how long it takes the shower to come to a satisfactory temperature and flow for hot and cold valves set to the values $0.0, 0.2, \ldots, 1.0$ (that is, 36 runs of the shower program).

6.6.7 Results

Table 6.1 gives running times for evaluating Aladin programs with and without partial evaluation, using the same machine as was used to obtain the results for Ginger in Section 4.5.6. As with our Ginger compiler, we also have the overhead of initialising the JVM (roughly 0.5–1.0 seconds). Partially evaluating programs can result in a significant decrease in running times, with a decrease of an order of

	Running times (s)		Speed up Factor
	No Part. Eval.	Part. Eval.	due to Part. Eval.
Matrix Multiplication	13.0	2.3	5.7
Gaussian Elimination	318.6	12.1	26.1
Exponentiation	12.0	5.1	2.4
Polynomials	24.0	8.8	2.7
Integration	17.3	8.6	2.0
Shower Controller	92.5	6.4	14.5

Table 6.1: Comparisons of running times of Aladin programs with and without partial evaluation

magnitude in the case of the Gaussian elimination and shower controller programs. Our results suggest two classes of speed-ups that can be obtained from partially evaluating Aladin programs. The first comes from simply unfolding the definition of a function so that control structures are replaced by combining lots of simple function applications in one single program. This was observed in the exponential, polynomial and integration programs and results in a speed-up factor of 2–3. The second class arises when, after or during unfolding, many of the simple function applications can be eliminated by exploiting the algebraic rules of the functions involved (implementing the functions involved accordingly). This was seen in the matrix multiplication, Gaussian elimination and shower controller programs and the speed-up factor was typically an order of magnitude. The extremes of this can be seen with the Gaussian Elimination test — because the matrix is sparse, many operations (and hence heap allocations) can be avoided entirely by exploiting the arithmetic rule $\forall x.x \times 0 = 0 \times x = 0$ — and with the fuzzy shower controller which fixes the domain over which our weighted sum is defuzzified and hence many fuzzy membership tests are eliminated.

The results in Table 6.1 do not include the time taken to do any partial evaluation, just the time taken to evaluate the original or residual program. In an

	Compile + Run times (s)	
	No Part. Eval.	Part. Eval.
Matrix Multiplication	5.8 + 13.0 = **18.8**	6.1 + 2.3 = **8.4**
Gaussian Elimination	5.4 + 318.6 = **324.0**	8.8 +12.1 = **20.9**
Exponentiation	5.7 + 12.0 = **17.7**	6.1 + 5.1 = **11.2**
Polynomials	4.6 + 24.0 = **28.6**	5.6 + 8.8 = **14.4**
Integration	5.3 + 17.3 = **22.6**	8.8 + 8.6 = **17.4**
Shower Controller	7.0 + 92.5 = **99.5**	12.2 + 6.4 = **18.6**

Table 6.2: Compile + Run times of Aladin programs with and without partial evaluation

environment where the resiudal program is used many times, the cost of partially evaluating the original program can be amortized over each evaluation of the residual program and become negligible. However, in a situation where the residual program is only run one or two times, the cost of the partial evaluation becomes more significant and, if too high, negates the argument for using partial evaluation in the first place. Since all partial evaluation is done at compile time, the cost of partial evaluation can be calculated by examining the compile times of the programs with and without partial evaluation.

Table 6.2 gives the times for compiling and running a single time each of our example programs. Compiling involves compiling the Aladin code into Java and then compiling the Java code into Java byte-code, as well as partially evaluating when necessary.

It turns out that compiling and running a program which does some partial evaluation during compilation is faster than compiling and running a program without partial evaluation, at least in these cases. This was noted by Jones *et al* [51] among others, and is analogous to the situation that compiling and running a program is often faster than interpreting it.

Since Aladin uses the same syntax as Ginger, it is not too difficult to use Ginger to evaluate the results of partially evaluating an Aladin program, giving us a way of partially evaluating Ginger programs. (A summary of the differences between Aladin and Ginger can be found in Section 5.6.) This involves partially evaluating the required Aladin functions at the command line and manually cutting-and-pasting the results into Ginger programs (the automation of this process is discussed in Section 7.1). For instance, in our test of the matrix multiplication program we use the following functions:

```
quarter, half, three_quarters :: -> s
quarter        = rotate (pi / 2);
half           = rotate pi;
three_quarters = rotate (3 * pi / 2);


rotations :: l * l -> l
rotations x y = (quarter (x, y), half (x, y), three_quarters (x, y));
```

The function `rotations` is partially evaluated at the command line using Aladin:

```
> java matrix -pp2 -dp 0 -p -og 'rotations x y'
rotations :: l * l -> l
rotations x y =
    let
        v2 = (*) -1;
        v1 = (v2 y);
        v0 = (v2 x);
    in
        mkTuple_3 (mkTuple_2 y v0) (mkTuple_2 v0 v1) (mkTuple_2 v1 x)
    endlet;
```

Here the `-og` option tells Aladin to optimise the residual function for pasting into a Ginger program. This eliminates common subprograms and redundant local defini-

	Running times (s)		Speed up Factor
	No Part. Eval.	Part. Eval.	due to Part. Eval.
Matrix Multiplication	4.4	1.3	3.4
Gaussian Elimination	57.7	4.4	13.1
Exponentiation	3.6	1.6	2.3
Polynomials	5.7	2.8	2.0
Integration	4.4	3.6	1.2
Shower Controller	15.3	2.1	7.3

Table 6.3: Comparisons of Running times of Ginger programs with and without partial evaluation

tions on the residual program but, unlike when optimising Aladin programs, functions are not treated as individual subprograms since Ginger functions are compiled to field accesses and in some cases may be directly applied (see Sections 4.5.2 and 4.6.1).

This new definition of `rotations` can then be manually cut-and-pasted into the Ginger program and used instead of the old one. Definitions of `mkTuple_2` and `mkTuple_3` also have to be supplied. Table 6.3 shows the running times of evaluating these partially-evaluated programs. The time required by partial evaluation remains the same as that in Table 6.2. Again, we achieve considerable speed-ups, but notice they aren't quite as big as those achieved by Aladin. This is because Aladin, or at least out particular implementation of it, can benefit more from partial evaluation than other languages since nearly everything has to be done *via* heap allocation and, as we saw in Section 4.7, Java is not particularly fast at heap allocation when compared to a dedicated functional engine like the Haskell interpreter Hugs [50]. Hence, the more heap allocation we can avoid doing, the proportionally better performance becomes.

6.7 Summary

We have shown that Aladin is a suitable machine for partial evaluation, by giving the denotational and operational semantics for a version of the AAM with unknown values, and then producing an implementation based on these semantics. By using the fact that Aladin's primitives may be implemented in any language and none are built in to the language, we have produced a set of primitives and data structures which exploit partial evaluation to the full. We have shown that using strictness declarations not only allows us to make lazy arguments strict for efficiency purposes, but strict arguments (or at least arguments that can be made strict without affecting the termination properties of the program) lazy for partial evaluation purposes.

Another advantage of using Aladin to partially evaluate programs is the fact that it has no built-in primitives, letting the user supply their own. This means that the primitives can be adapted to exploit partial evaluation to the full. A good example of this is with the multiplication function: by using the arithmetic rule $\forall x.x \times 0 = 0 \times x = 0$ and partially evaluating programs in which this rule could be exploited, we could circumvent large amounts of evaluation and reduce run times by an order of magnitude.

In general, partially evaluating Aladin programs results in a significant decrease in running times, and in some cases these running times are shorter than those of the equivalent programs in Ginger. The results of partially evaluating programs can also be evaluated using Ginger, albeit with a little manual intervention on the user's part.

Chapter 7

Conclusion and Further Work

Our stated aim of this thesis, back in Chapter 1, was to tackle the downsides of functional programming, by expanding the range of applications in which functional programming could be used, increasing the portability of functional programs, enabling functional languages to be interfaced with other languages, and using novel evaluation strategies to increase the performance of functional language implementations. Have we achieved these aims?

Our implementation of fuzzy logic in a number of functional languages is elegant and concise. Once the synonymity of functions and fuzzy subsets is recognised, applications involving fuzzy logic become programs involving higher-order functions, and such programs are most easily expressed in a functional language since this is one of the *raisons d'etre* of functional programming. While we only gave a couple of small examples of the applications of fuzzy logic in a functional language, Jan Skibinski at Numeric Quest Inc. [100] has developed larger Haskell programs which utilise our original work on fuzzy logic.

We then tackled the problem of portability of functional programs by producing a compiler for the functional language Ginger which compiles programs that run on the Java Virtual Machine (JVM). While this gave us the desired portability, since many machines have an implementation of the JVM, and also a way of using Java methods in functional programs, since we represented functions as static Java methods, performance was disappointing with programs running an order of magnitude slower than the equivalent ones using the Haskell interpreter Hugs. This was due to the expense of heap allocation in both the Java implementations tried when compared to the allocator used by Hugs (written in C). This level of performance was noted by David Wakeling who produced a Haskell compiler which targetted the JVM.

Our experience with Ginger led us to develop an implementation for Aladin which was designed to integrate many languages, functional and imperative, into a pure, lazy functional machine. This first required us to develop a reasonably efficient denotational and operational semantics for Aladin, which was used as the basis for the implementation. The implementation allowed us to define primitives in Aladin itself, Ginger, Java and C/C++, and combine them into Aladin programs.

The purity and simplicity of Aladin has other applications, too, and one we examined was the use of Aladin to partially evaluate programs. This was greatly aided by the requirement that the user specify the strictness of all Aladin functions. The implementation used these strictnesses to find out which parts of a program it could attempt to safely evaluate. The ability of the user to define *all* primitives used by Aladin programs was also beneficial where partial evaluation was concerned, since their definitions could be fine-tuned by the user to exploit partial evaluation to the full. This led to significant performance benefits, in particular with the sample fuzzy system we encoded in a functional style in Chapter 3 (a shower controller) we were rewarded by a speed up of an order of magnitude when we converted the code into Aladin and partially evaluated it.

In summary, we have found a new application area for functional programming, developed a system for making functional programs more portable, developed a machine that integrates many languages into a functional context, and used this machine to increase the efficiency of functional programs by means of partial evaluation.

We have thus accomplished what we set out to do — what possibilities are there for future development of our work?

7.1 Integrating Aladin and Ginger

We have already seen in Section 6.6.7 how we can use the results of partially evaluating an Aladin program in a Ginger program, albeit in a manner which requires a manual cut-and-paste by the user, and obtain significant speed ups in running time by doing so. It is thus reasonable to ask if and how we can automate this process.

Integrating Aladin into Ginger would allow us to integrate partial evaluation, strictness signatures and the inclusion of primitives written in a number of languages into Ginger. The AAM would be used as a secondary evaluation mechanism, used by Ginger to partially evaluate programs and to apply non-Ginger primitives, while Ginger's evaluation mechanism would be used in all other cases. Aladin's strictness signatures could also be used by Ginger to compile more efficient code, for instance, by being able to use the $\mathcal{E}$ scheme in more places.

7.2 Ginger and Type Systems

One of the downsides of Ginger being weakly-typed is that all overloading must be resolved at runtime. This means that even simple operations such as arithmetic, comparison and logical operations must be done *via* method calls even though the JVM provides instructions to perform these operations directly on the stack. This approach would be faster and allow us to avoid having to construct some objects in the heap. It would be fairly simple to encode fuzzy logic operations using a combination of stack operations. Making Ginger a strongly typed language also has the correctness benefits described in Section 2.4.

This leaves the question of how we resolve the overloading. Haskell's type classes are certainly powerful and flexible, but perhaps too large and complex for a small and simple language like Ginger, and Miranda's type system ignores overloading all together. A similar approach to that of ML (see Section 2.4) would thus seem to be best for Ginger, since all overloading is resolved at compile time. As well as allowing us to do simple operations on the stack in some cases, this method would allow us to remove run-time resolution of overloading in the function definitions.

As an example, consider the operator - (subtraction) which is used to subtract integer and real arguments. In Ginger this is achieved by means of a single method, _minus (see Section 4.4). If we had the application of - to two integer arguments then we could adapt the $\mathcal{E}$ compilation scheme to produce more efficient code than before:

$\mathcal{E}$ (- i_1 i_2) ρ fs v =

new $Long$

dup

$\mathcal{E}$ i_1 ρ fs v

invokevirtual $Long$.getLong()J

$\mathcal{E}$ i_2 ρ fs v

invokevirtual $Long$.getLong()J

lsub

invokespecial $Long$/<init>(J)V

This scheme compiles the two arguments and extracts their long values. The subtraction is then performed and the result put into a Long object. This method not only avoids run-time type checking and type casting, but also avoids a method call. A similar strategy can be used to subtract two real (`double`) arguments. This is just an outline of how such a scheme would precede — Peyton Jones [86] describes the $\mathcal{B}$ scheme which compiles such expressions much more efficiently.

We may not be able to directly apply the overloaded primitive in code, for instance in the expression `map ((-) x) xs` the subtraction is not applied until the elements of the list are required and thus must be done by a generic function application. However, if we adopt ML's system of overloading resolution we must specify whether the `x` and `xs` are integers or reals and the compiler can use this information to compile the occurrence of `-` here to a call to the correct function: either one that subtract integers or one that subtracts reals. So, instead of the single method definition as in Section 4.4 we need two:

```
public class StrictPrimitives extends Node {
  public final static Class TYPE = StrictPrimitives.class;
  public static Object _minus_int(Object lhs, Object rhs) {
   return new Long(((Long) eval(lhs)).longValue() -
                   ((Long) eval(rhs)).longValue());
  }
  public final static Object _minus_int =
    Function.make(TYPE, "_minus", 2);

  public static Object _minus_real(Object lhs, Object rhs) {
    return new Double(((Number) lhs).doubleValue() -
                      ((Number) rhs).doubleValue());
  }
  public final static Object _minus_real =
```

```
    Function.make(TYPE, "_minus", 2);
  // ...
}
```

The compiler then has to decide whether to compile - to code which loads the value of the field `_minus_int` or `_minus_real`.

7.3 Parallel Aladin

The strictness of function arguments in Aladin is used to determine which arguments can be safely evaluated, and those that we wish to leave unevaluated. This was used to great effect when we partially evaluated Aladin programs; this information can also be used to evaluate the various parts of an Aladin program in parallel.

Parallelism has long interested researchers into functional programming [33, 91]. The purity of a functional language means that evaluation order doesn't matter and hence it is safe to evaluate the various parts of a functional program in parallel. This can be done in a variety of ways. The user can create individual processes which will evaluate different parts of a program concurrently, with communication being done by passing messages. This approach is used by Erlang [6]. Alternatively, an attempt can be made to evaluate the various parts of the graph representing the program in parallel. This approach is taken by GAML [69] and the original version of Ginger [52], and parallel abstract machines such as GRIP [20] and the $\langle \nu, G \rangle$ machine [9].

The latter approach raises the question of how to discover which parts of a program can be safely evaluated in parallel. The program can rely on user annotations, as in GAML or Ginger; use strictness analysis and other such methods to determine which parts of a program can be safely evaluated in parallel [55]; or even speculatively evaluate parts of a program in the hope that the results can be used later on in the evaluation sequence [71].

As Aladin already has strictness declarations, we can use these to determine which parts of the program to evaluate in parallel. With regards to our particular implementation, this would involve modifying the `evalArgs` method from Section 5.3.3 so that each of the strict arguments of a function are evaluated concurrently in separate threads in parallel — in Java this would involve creating a new thread for each strict argument and evaluating each strict argument in that thread. We would also have to adapt the implementation so that it performed the various housekeeping tasks required by a parallel evaluator, for example, making sure that two threads don't try to evaluate or update the same variable, locking the heap so that two threads don't try to write to it at the same time, and managing the various threads of execution.

However, strictness information by itself may not be enough. Kaser *et. al.* [55] showed that even if we know the strict arguments of a function we may not be able to extract and exploit enough information for maximal parallelism, that is, when all available processors have work to do. This is because most lazy functional languages only evaluate arguments down to Weak Head Normal Form (see Section 2.2) whereas more parallelism can be achieved if we know which arguments can be evaluated all the way down to Normal Form as well as those that can be evaluated to WHNF. For instance, in the case of a list, it is in WHNF once we have evaluated down to the first list constructor (the one at the head of the list), but is not in NF until all elements of the list have been evaluated.

Kaser called this three-level strictness *ee-strictness* and used a form of strictness analysis to determine the ee-strictness of functions. Aladin is not restricted to evaluating expressions to WHNF or NF, it depends on the strictness and definitions of the functions used, but a function cannot evaluate a program down to WHNF in one context and down to NF in another without the definition or strictness of the function being changed. However, we could alter Aladin's concept of strictness to enable the user to specify that an argument should be evaluated all the way down to normal form and provide a suitable implementation using [55] as a basis.

Another method of introducing parallelism into Aladin programs could be to supply parallel primitives. For instance, we could define `par` of strictness $l \times l \to l$ which would evaluate its two arguments in parallel and return the result of applying the first to the second. For instance, `par ((+) x) y` would return the result of `x + y` but evaluate `x` and `y` in parallel (presuming `+` is strict in its first argument so that `x` is evaluated). We could also define other parallel primitives, such as parallel versions of `map`, `fold` — which would be useful in fuzzy systems, for instance, to evaluate the rules of a fuzzy system in parallel — and `filter`.

7.4 Other Aladin Enhancements

Our implementation of Aladin was developed as much to show the correctness and applicability of the Aladin semantics as to have a machine to develop and work with. This means we have mirrored the semantics in the implementation as much as possible. Any future implementation may wish to use a more obscure interpretation of the semantics in order to implement compiler enhancements, such as those for Ginger described in Chapter 4.

We mentioned above the possible use of Aladin to partially evaluate Ginger programs. Such a use for Aladin is not restricted to Ginger — Aladin could be used to partially evaluate programs written in *any* pure functional language, provided suitable implementations of the primitives and functions of that language were provided.

Another possible area for investigation is the language used to implement Aladin. Java has the advantages of portability, ease of use, being able to be run in applets, and an in-built garbage collector, but it has one large disadvantage: current implementations are slow. It may be advantageous to write any future implementation in C or C++ and take advantage of the existence of fast compilers for these languages. This approach would also make the task of importing C/C++ primitives into Aladin (the present method is a little involved) though would complicate the importing of Ginger and Java primitives. A C/C++ implementation would be more involved, for instance, we would have to encode our own memory management scheme.

The development of Aladin programs is sometimes complicated by the fact that it is very weakly typed, which can make debugging programs hard. To alleviate this problem we could investigate adding a type system, probably based on the Hindley-Milner system described in Section 2.4. This leaves us with the problem what to do with the type of data objects. Aladin allows the user to introduce new types at will, and in no particularly systematic way. Some way would have to be found to allow the user to declare new types, including container types, and some way of allowing the overloading of functions using these types, possibly based on Haskell's type classes or *subtyping* [19, 101]. Alternatively, we could use a single type to represent all Aladin data objects, though this would reduce the strength of the typing system.

Bibliography

[1] Aptronix Ltd. Focusing system. http://www.aptronix.com/fuzzynet/applnote/focusing.htm, 1992.

[2] Aptronix Ltd. Washing machine. http://www.aptronix.com/fuzzynet/applnote/wash.htm, 1992.

[3] Aptronix Ltd. Fuzzy java. http://www.aptronix.com/fuzzynet/applnote/java.htm, 1996.

[4] J.L. Armstrong, S.R. Däcker, B.O.and Virding, and M.C. Williams. Implementing a functional language for highly parallel real-time applications. In *Software Engineering for Telecommunication Switching Systems, Florence*, March/April 1992.

[5] J.L. Armstrong and S.R. Virding. Erlang — an experimental telephony programming language. In *XIII International Switching Symposium, Stockholm*, May/Jun 1990.

[6] J.L. Armstrong, S.R. Virding, and M.C. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[7] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 2nd edition, 1998.

[8] L. Augustsson. A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, pages 218–27, August 1984.

[9] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the $\langle \nu, G \rangle$-machine. In *Functional Programming & Computer Architecture*, pages 202–214. ACM, 89.

[10] Tom Axford and Mike Joy. Aladin: An Abstract Machine for Integrating Functional and Procedural Programming. *Journal of Programming Languages*, 4:63–76, 1996.

[11] Tom Axford and Mike Joy. Lazy Partial Evaluation in Aladin. Unpublished paper, September 1997.

[12] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[13] H.P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland Publishing Company, 1981.

[14] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[15] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.

[16] James M. Boyle. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, 10(5):574–588, September 1984.

[17] R.M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):33–67, January 1977.

[18] R.M. Burstall, D.B. MacQueen, and D.T. Sanella. Hope: an experimental applicative language. Technical Report CSR-62-80, Department of Computer Science, Edinburgh University, May 1980.

[19] Guiseppe Castagna, Giorgio Ghelli, and Guiseppe Longo. A calculus for overloaded functions with subtyping. In *ACM Conference on LISP and Functional Programming, San Francisco*, 1992.

[20] Chris Clack, Simon L. Peyton Jones, and Jon Salkild. Efficient parallel graph reduction on GRIP. Technical Report RN/88/29, Department of Computer Science, University College, London, July 1988.

[21] Earl Cox. *The Fuzzy Systems Handbook*. AP Professional, 1994.

[22] Antony J.T. Davie and David J. McNally. CASE — a lazy version of an SECD machine with a flat environment. Technical Report CS/90/19, Department of Computer Science, University of St. Andrews, 1990.

[23] Patrik Eklund and Frank Kwalonn. Neural fuzzy logic programming. *IEEE Transactions on Neural Networks*, 3(5):815–818, September 1992.

[24] Erricsson. CSLab — Erlang. http://www.ericsson.se:800/cslab/erlang/, 1999.

[25] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Functional Programming Langauges and Computer Architecture*, number 274 in LNCS, pages 34–45. Springer-Verlag, 1987.

[26] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[27] János Fodor and Marc Roubens. *Fuzzy Preference Modelling and Multicriteria Decision Support*. Kluwer Academic Press, 1994.

[28] Franz Inc. Home page. http://www.franz.com, 1999.

[29] Carsten K. Gomard and Neil D. Jones. A Partial Evaluator for the Untyped Lambda Calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[30] Rob Gordon. *Essential JNI: Java Native Interface.* Prentice Hall, 1998.

[31] F. G. Gustavson, W. Liniger, and R. Willoughby. Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations. *Journal of the ACM*, 17(1):87–109, January 1970.

[32] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[33] Kevin Hammond. Parallel functional programming: An introduction (invited paper). In *PASCO'94: First International Symposium on Parallel Symbolic Computation*. World Scientific Publishing Company, Sept 94.

[34] Chris Hankin. *Lambda Calculi — A Guide for Computer Scientists.* Oxford University Press, 1994.

[35] John Hannan and Patrick Hicks. Higher-Order Arity Raising. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 27–38, January 1999.

[36] Harlequin. Lisp products. http://www.harlequin.com/products/ads/lisp/, 1999.

[37] Harlequin. ML works. http://www.harlequin.com/products/ads/ml/, 1999.

[38] Pieter Hendrik Hartel and Henk Muller. *Functional C.* International Computer Science Series. Addison-Wesley, 1997.

[39] Report on the programming language Haskell 98. http://haskell.systemsz.cs.yale.edu/definition/, February 1999.

[40] Peter Henderson. *Functional Programming — Application and Implementation*. Prentice Hall, 1990.

[41] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[42] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

[43] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[44] J.M. Horvath. A fuzzy set model of learning disability. In Tamás Zétényi, editor, *Fuzzy Sets in Pyschology*, number 56 in Advances in Pyschology, pages 345–382. North-Holland, 1988.

[45] Ian Hoyler. *Functional Programming with Miranda*. Pitman, 1991.

[46] John Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison-Wesley, 1990.

[47] Persimmon IT. The Persimmon MLJ Compiler. http://research.persimmon.co.uk/mlj/.

[48] T Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the 11th ACM Symposium of Principles of Programming Languages*, pages 58–69, 1984.

[49] T Johnsson. Lambda-lifting — transforming programs to recursive equations. In *Conference on Functional Programming and Computer Architecture, Nancy*, number 201 in LNCS, pages 190–203. Springer-Verlag, 1985.

[50] Mark Jones. Hugs archive. http://www.cse.ogi.edu/~mpj/hugsarc/.

[51] Neil D. Jones, Carsten L. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[52] M. Joy and T. H. Axford. A parallel graph reduction machine. In H. Kuchen and R. Loogen, editors, *Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages.* Fachgruppe Informatik, 1992.

[53] Mike Joy. Ginger — A Simple Functional Language. Technical Report CS-RR-235, Department of Computer Science, University of Warwick, Coventry, UK, 1992.

[54] Mike Joy and Steve Matthews. Some experiences of teaching functional programming. *International Journal of Mathematical Education in Science and Technology*, 25(2):165–172, 1994.

[55] Owen Kaser, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. EQUALS —- a fast parallel implementation of a lazy language. *Journal of Functional Programming*, 7(2):183–217, March 1997.

[56] Arnold Kaufmann. *Introduction to the Theory of Fuzzy Subsets*, volume 1. Academic Press, 1975.

[57] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[58] Bart Kosko. *Fuzzy Thinking.* Flamingo, 1994.

[59] Erwin Kreyszig. *Advanced Engineering Mathematics*. Wiley, 6th edition, 1988.

[60] P.J. Landin. The Mechanical Evaluation of Expressions. *BCS Computing Journal*, 6(4):308–320, January 1964.

[61] J. Launchbury. A natural semantics for lazy evaluation. In *Principles of Programming Languages*, Charleston, 1993.

[62] John Launchbury. *Projection Factorisations in Partial Evaluation*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.

[63] Daan Leijen and Erik Meijer. HaskellScript. http://haskell.systemsz.cs.yale.edu/haskellscript, 1999.

[64] J.W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE '94*, 1994.

[65] Logic Programming Associates Ltd. FLINT toolkit. http://www.lpa.co.uk/fln.html, 1997.

[66] Jan Łukasiewicz. On the notion of possibility/On three-valued logic. In Storrs McCall, editor, *Polish Logic 1920–1939*, pages 15–18. Oxford University Press, 1967. Appeared originally under the titles 'O pojęciu możliwości' and 'O logice trojwartosciowej' in *Ruch Filozoficzny* 5 (1920), pp 169–171.

[67] Jan Łukasiewicz. Philosophical remarks on many-valued systems of propositional logic. In Storrs McCall, editor, *Polish Logic 1920–1939*, pages 40–65. Oxford University Press, 1967. Appeared originally under the title 'Philosophische Bemerkungen zu mehrwertigen Systemen des Aussagenkalküls' in *Comptes rendus des séances de la Societé des Sciences et des Lettres de Varsovie*, Cl. iii, 23 (1930), pp 51–77.

[68] E.H. Mamdani and S. Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Man-Machine Studies*, pages 1–13, 1975.

[69] Luc Maranget. GAML: a parallel implementation of lazy ML. In J. Hughes, editor, *Functional Programming & Computer Architecture*, pages 102–123. Springer-Verlag, August 1991.

[70] T.P. Martin, J.F. Balwdin, and B.W. Pilsworth. The implementation of FProlog — a fuzzy Prolog interpreter. *Fuzzy Sets and Systems*, 23:119–129, 1987.

[71] Jim S. Mattson Jr. Performance of parallel schedulers for distributed graph reduction. In *Nijmegen Workshop on the Implementation of Functional Languages*, 1993.

[72] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 8(3):184–195, March 1960.

[73] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, 2nd edition, 1965.

[74] Gary Meehan. Compiling functional programs to Java byte-code. Research Report CS-RR-334, Department of Computer Science, University of Warwick, Coventry, UK, September 1997.

[75] Gary Meehan. Fuzzy functional programming. Research Report CS-RR-322, Department of Computer Science, University of Warwick, Coventry, UK, April 1997.

[76] Gary Meehan. The Aladin Abstract Machine. Research Report CS-RR-355,

Department of Computer Science, University of Warwick, Coventry, UK, December 1998.

[77] Gary Meehan and Mike Joy. Animated Fuzzy Logic. *Journal of Functional Programming*, 8(5):503–525, September 1998.

[78] Gary Meehan and Mike Joy. Compiling Lazy Functional Programs to Java Byte-code. *Software — Practice and Experience*, 1999.

[79] Jon Meyer and Troy Downing. *Java Virtual Machine.* O'Reilly, 1997.

[80] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Science Systems*, 17(3):348–375, 1979.

[81] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[82] C.V. Negoita. *Expert Systems and Fuzzy Systems.* The Benjamin/Cummings Publishing Company, 1985.

[83] NRC-CNC Institute for Information Technology. Fuzzy CLIPS. http://ai.iit.nrc.ca/fuzzy/fuzzy.html, 1996.

[84] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings 24th ACM symposium on Principles of Programming Languages, Paris, France*, 1997.

[85] The Haskell 1.4 report. http://haskell.org/report/, April 1997.

[86] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[87] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[88] Simon L. Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: exploring the design space. In *Proceedings of the Haskell Workshop, Amsterdam*, June 1997.

[89] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages — A Tutorial*. Prentice Hall, 1992.

[90] Simon L. PeytonJones, Thomas Nordin, and Reid Alastair. Green card: a foreign language interface for Haskell. In *Haskell Workshop, Amsterdam*, June 1997.

[91] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[92] Rinus Plasmeijer and Marko van Eekelen. Concurrent Clean v1.1 language report. Technical report, University of Nijmegen, March 1996.

[93] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[94] Claus Reinke. Towards a Haskell/Java connection. In *Implementation of Functional Languages, the 10th International Workshop, IFL'98, London, UK*, number 1595 in Lecture Notes in Computer Science, pages 200–215. Springer-Verlag, September 1998.

[95] Sergei A. Romanenko. Arity Raiser and its Use in Program Specialization. In N. Jones, editor, *Proceedings of the European Symposiom on Programming, Copenhagen, Denmark*, number 432 in Lecture Notes in Computer Science, pages 351–360. Springer-Verlag, May 1990.

[96] Timothy J. Ross. *Fuzzy Logic With Engineering Applications.* McGraw-Hill, 1995.

[97] Colin Runciman and David Wakeling. *Applications of Functional Programming.* UCL Press, 1995.

[98] Stuart Russel and Peter Norvig. *Artificial Intelligence — A Modern Approach.* Prentice Hall, 1995.

[99] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[100] Jan Skibinski. Fuzzy oscillator. http://www.numeric-quest.com/haskell/oscillator/Fuzzy_oscillator.html, November 1998.

[101] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[102] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[103] Sun Microsystems, Inc. Javap — The Java Class File Disassembler. http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/javap.html.

[104] Kazuo Tanaka. *An Introduction to Fuzzy Logic for Practical Applications.* Springer-Verlag, 1997. First published in Japanese, 1991.

[105] Transvirtual Technologies. What is Kaffe Open VM? http://www.transvirtual.com/kaffe.html.

[106] Simon Thompson. *Miranda: The Craft of Functional Programming.* Addison-Wesley, 1995.

[107] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1996.

[108] Thomas W. Torgerson. *Visual Basic Professional 3.0 Programming.* Wiley, 1994.

[109] D.A. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9:31–49, 1979.

[110] D.A. Turner. Miranda — a non-strict functional language with polymorphic types. In Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy*, number 201 in LNCS, pages 1–16. Springer-Verlag, 1985.

[111] D.A. Turner. An overview of Miranda. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 1–16. Addison-Wesley, 1990.

[112] Jeffrey D. Ulman. *Elements of ML Programming.* Prentice Hall, 1994.

[113] Philip Wadler. Introduction to Orwell. Technical report, Programming Research Group, University of Oxford, 1985.

[114] Philip Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78. ACM, ACM Conferences, June 1990.

[115] Philip Wadler. The essence of functional programming. In *ACM Symposium of Programming Languages*, January 1992.

[116] Philip Wadler. An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, February 1998.

[117] Philip Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, 33(8):23–27, August 1998.

[118] David Wakeling. Mobile Haskell: Compiling lazy functional languages for the Java Virtual Machine. Unpublished paper.

[119] David Wakeling. VSD: A Haskell to Java Virtual Machine code compiler. In *Proceedings of the 9th International Workshop on the Implementation of Functional Languages*, 1997.

[120] Li-Win Wang. *Adaptive Fuzzy Systems and Control — Design and Stability Analysis.* Prentice Hall, 1994.

[121] Robert Wilensky. *Common LISPcraft.* W. W. Norton & Company, 1986.

[122] Barry Wilkinson. *Computer Architecture: Design and Performance.* Prentice Hall, 1991.

[123] Jun Yan, Michael Ryan, and James Power. *Using Fuzzy Logic.* Prentice Hall, 1994.

[124] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[125] L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Men and Cybernetics*, 3:28–44, 1973.

[126] H.-J Zimmermann. *Fuzzy Set Theory — and its Applications.* Kluwer Academic Publishers, 1991.