The Provision of Non-Strictness, Higher Kinded Types and Higher Ranked Types on an Object Oriented Virtual Machine

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in the
University of Canterbury

by

Oliver Hunt

Examining Committee

Dr. Wolfgang Kreutzer	Supervisor
Dr. Jeremy Gibbons	Examiner
Prof. K John Gough	Examiner

University of Canterbury

2006

To Mum and Dad

Abstract

We discuss the development of a number of algorithms and techniques to allow object oriented virtual machines to support many of the features needed by functional and other higher level languages. These features include non-strict evaluation, partial function application, higher ranked and higher kinded types.

To test the mechanisms that we have developed we have also produced a compiler to allow the functional language Haskell to be compiled to a native executable for the Common Language Runtime. This has allowed us to demonstrate that the techniques we have developed are practically viable.

Table of Contents

List of Figures **v**

Chapter 1: Introduction **1**

1.1 Topic 1

1.2 Problems 2

1.2.1 Non-Strict Evaluation 2

1.2.2 Higher Kinded Types 3

1.2.3 Higher Ranked Types 3

1.3 Evaluation 4

1.4 Overview 4

Chapter 2: Background and Related Work **7**

2.1 Functional Programming Languages 7

2.2 Virtual Machines 9

2.3 Targeting an OOVM 12

2.4 Existing Functional Language to Typed Virtual Machine Compilers 13

2.5 Interfacing Haskell to an Object Oriented Virtual Machine . . 15

Chapter 3: Functions and Non-Strictness **17**

3.1 Lambda Expressions 17

3.2 Partial Application 20

3.2.1 Using The Push-Enter Model 21

3.2.2 Using The Eval-Apply Model 23

3.2.3 Transforming Partial Applications Into Complete Applications . . . 23

3.2.4 Discussion . . . 26

3.3 Non-Strictness . . . 26

Chapter 4: Types 29

4.1 Functional Language Types . . . 29

4.1.1 Algebraic types and Records . . . 29

4.1.2 Parametric Polymorphism . . . 31

4.1.3 Type Classes and Instances . . . 32

4.1.4 Higher Ordered, Ranked, and Kinded Types . . . 33

4.2 Imperative Object-Oriented Types . . . 34

4.2.1 Classes . . . 34

4.2.2 Generic Classes and Methods . . . 37

4.2.3 Primitive and Value Types . . . 38

Chapter 5: Converting Functional Types to an Imperative Object Model 39

5.1 Algebraic Types . . . 39

5.2 Type Classes and Instances . . . 42

5.3 Higher Order Types . . . 45

Chapter 6: Providing Non-Strictness 47

6.1 Support for Non-Strictness on Conventional Architectures . . . 48

6.2 Non-Strictness on OOVMs: Current Developments . . . 49

6.2.1 JIT Objects . . . 50

6.3 Algebraic types vs. JIT Objects . . . 51

6.4 Non-Strictness for Functional Languages . . . 53

6.4.1 Non Strictness for Boxed Primitives . . . 54

6.4.2 Non-Strict Function Values 55
6.5 Performing Non-Strict Evaluation 57

Chapter 7: Higher Kinded Types 61
7.1 Background 61
7.1.1 Higher-kinded type variables in functions 61
7.1.2 Higher-kinded type variables in type classes 62
7.2 Solution 64
7.2.1 Type Erasure 64
7.2.2 Explicit Instantiation Types 65
7.3 Problems 68

Chapter 8: Higher Ranked Types 71
8.1 What are higher ranked types? 71
8.2 Solution 72
8.2.1 Basic solution 72
8.2.2 Supporting Multiple Libraries 74
8.3 Problems 80

Chapter 9: Experimental Compiler 83
9.1 Design Decisions 83
9.2 Type Generation 84
9.3 Architecture 85
9.3.1 Stage 1: Initial Processing 85
9.3.2 Stage 2: Converting to a Typed Structure 86
9.3.3 Stage 3: Transformations 87
9.3.4 Stage 4: Final Compilation 91
9.4 Performance 93
9.5 Summary 94

Chapter 10: Future Work and Conclusion **97**

10.1 Future Work 97

10.1.1 Performance Improvements 97

10.1.2 Virtual Machine Level Support 98

10.1.3 Integration of External Functions into a Functional Language 98

10.2 Conclusions 99

References **101**

List of Figures

3.1 Algorithm to perform a function call using the push-enter model. 22

4.1 Definition of instance for *Eq (List a)* 33

5.1 Classes defined for the *List* type. 41

5.2 Classes defined for the *List* type with tags. 42

5.3 Reference summary of *Eq*, *Eq Bool* and *Eq (List a)* 43

5.4 Example of type class to algebraic type conversion 43

5.5 The *Eq* type class, and the instance for *Eg (List a)*. 44

5.6 Illustration of class for higher order type 45

5.7 Generic *Function* class for representing higher order types. . . 46

6.1 The JIT Object structure for the C# *List* type. 51

6.2 List structure with non-strict support – first attempt. 52

6.3 Final structures for non-strict algebraic types 55

6.4 *Boxed* type for non-strict evaluation of primitive types 56

6.5 Classes for non-strict evaluation of higher order types 57

6.6 *IThunkable* Interface for type parameters 59

7.1 Conversion algorithm for higher kinded types to an OOVM . . 66

7.2 Example application of higher kinded type conversion algorithm 69

8.1 Example of the problems of higher ranked types across multiple modules 75

8.2 Demonstration of how to share higher ranked types across multiple modules 76

Acknowledgments

I would like to thank Dr. Nigel Perry for his constant support and advice, even in the face of adversity. Without his support this thesis would never have been possible. I also thank my fellow occupants of MSCS344: Ryan, Taher, Michael, and Jason, they have made the last 16 months much more enjoyable and interesting that they might otherwise have been.

Finally many thanks Alex, Amy, Amanda, Dave, Graham, Jay, Lee, Mukundan, The Other Oliver, Richard, Wal and others far to numerous to mention for willingly discussing the many and varied topics of conversation (or rants) I have come up with over the past 16 months.

Chapter I

Introduction

1.1 *Topic*

Since the introduction of the Java Virtual Machine (JVM) the use of virtual machines in mainstream programming has been increasing. The execution of programs inside a virtual machine (VM) offers numerous advantages over native execution, such as improvements to code reliability, maintainability, interoperability, and security. However these VMs are often tailored to a specific language, due in part to a bias by the original developer. In the case of the JVM, that language is its namesake, 'Java'. More recently, however, Microsoft has produced the .NET Common Language Runtime, which has been standardised to the ECMA/ISO Common Language Infrastructure (CLI). Unlike other VMs, the CLI is designed as a general purpose VM, intended to efficiently execute code from a number of different programming languages.

One of the CLI's design goals was to support many different programming languages, and a number of compilers for languages have since been produced that target the CLI directly (including C#, COBOL, Delphi, Fortran, and numerous others). These languages however share many common features, including similar type systems and execution models, thus allowing the CLI to use a similar type system and execution model to those languages. The final result of this simplification is a virtual machine capable of being targeted relatively easily by any object oriented imperative programming language.

However there are programming languages that are not imperative or object oriented, and such languages may require a number of features not present

on an object oriented virtual machine. While a number of these features can be mapped easily to an object oriented virtual machine there are a some for which the necessary transformations are non-trivial. This means that a number of languages; including Haskell, LISP, and some dynamic languages; are not easily able to target OOVMs. In this thesis I will describe a number of algorithms and techniques that allow a number of these features, most notably non-strict evaluation and higher order, higher kinded, and higher ranked types, to be provided on an otherwise conventional OOVM. As all of these features are used in functional languages, it is functional languages that our thesis will focus on.

1.2 Problems

While there are a number of difficulties in providing support for functional languages in object oriented systems many of these, particularly representation of algebraic types, have been discussed previously, or are trivial extensions to prior work. In addition to the problems encountered when attempting to converted functional languages to OO environments, the typed memory models of OOVMs also place other restrictions how certain features may be supported.

While previous projects have aimed to provide support for full functional languages on OOVMs , these have either discarded most of the static type information from the original language, or provided only partial feature support. In this thesis we describe techniques and algorithms that provide full support non-strict evaluation, higher kinded types, and higher ranked types, while retaining almost all static type information.

1.2.1 Non-Strict Evaluation

Non-strict evaluation is the act of delaying evaluation of an expression until the computed value is actually needed. It has existed in many forms, from the *pass-by-name* semantics of Algol 60 through to non-strict functional languages and manual *proxy* objects in modern OO languages. Full

non-strictness in a language has many advantages as it relieves the programmer from the task of controlling when expressions are evaluated as they will be evaluated only when (or if) they are needed.

Due to the typed memory model of an OOVM the traditional methods used to provide non-strictness can no longer be used, however this thesis describes a technique that provides non-strictness over all algebraic types. Unlike previous techniques this approach minimises the visibility of any delayed evaluation, hopefully improving interoperability with other languages operating on the VM.

1.2.2 Higher Kinded Types

Higher kinded types are an extension to the basic concept of generic types that allow type parameters to be instantiated with functions over types (or open types) rather than just types (or closed types). The functional language Haskell supports a limited form of higher-kinded types, where type functions are limited to being first-order (ie. functions over closed types), and we have only addressed this level of support. We have developed an algorithm to map types with higher-kinded parameters onto OOVM ones with simple type parameters. The algorithm provides a high degree of static type-checking but some dynamic checks are still required to satisfy the OOVM – all such checks will succeed if the original higher-kinded types are type correct.

1.2.3 Higher Ranked Types

Higher-ranked types are those in which the scope of quantification of the type variables can be made local to just part of a type expression. This is in contrast to the usual approach to quantification employed in functional languages where the scope is always the whole type expression. In operational terms this means that a polymorphic function may be passed as an argument *without* being specialised to some particular type, and then used with different specialisations within the body.

In OOVM terms support for higher-ranked types requires the ability to pass open types as parameters, something which is not supported by current

OOVMs. We have developed an algorithm whereby wrapping a type within an interface allows it to be passed as an open type. There are a number of subtleties that must be handled for this to be successful and these are all dealt with.

1.3 Evaluation

To evaluate the algorithms we have chosen build an implementation of Haskell (with higher ranked types provided as an extension, as in the Glasgow Haskell Compiler), which provides us with many advantages over other similar languages. It is used in both academic and commercial environments, it has a largely standardised base [44], and most importantly for our purposes, it supports all of the above features, thus allowing us to use a consistent environment when discussing each feature. While many other similar languages exist, they are either no longer widely used (eg. Hope [3], Miranda [64]), primarily academic (eg. Mondrian [40]), or do not support all of the features we wish to develop (eg. Erlang [1], Nemerle [36], ML [16]). The use of Haskell for evaluating our techniques and algorithms does not limit them and they are applicable to any language that uses these features.

1.4 Overview

This thesis starts with a discussion of the history and development of both virtual machines and functional programming languages, as well as previous attempts to combine the two. Chapter 3 follows with an introduction to some of the more complex features of functions in functional languages, such as partial applications and lambda functions, and discusses how they are provided on OOVMs.

Chapter 4 introduces the type systems of the CLI and that of functional languages. This leads to a discussion of the mechanisms and transformations required to support the basic features of a functional type system on a standard OOVM (in Chapter 5).

Non-strict evaluation is introduced in Chapter 6, in which we describe a number of techniques to extend the techniques of Chapter 5 to support non-strict evaluation. We then discuss the algorithms required to support higher kinded types on an OOVM in Chapter 7, and higher ranked types in Chapter 8.

Chapter 9 describes the implementation of a compiler that uses the techniques we have developed to support the full *Haskell 98* language [44] on the CLI.

Finally, Chapter 10 concludes this thesis with a discussion of the possible extensions to our work and summarises what we have accomplished and the techniques that we have developed.

Chapter II

Background and Related Work

The problem of supporting different programming models on VMs designed for a different model has been addressed previously [10, 37, 38, 61]. One of the more recent approaches is the 'Iron Python' project in which an implementation of the Python programming language was created for the .NET CLI [23]. There have also been numerous projects to develop compilers that allow functional languages (either pre-existing, or developed for the purpose) such as Mondrian, F#, Lambada, and others to be compiled to an OOVM. In this chapter we will discuss the history of functional languages (Section 2.1), VMs (Section 2.2), and the attempts to bring them together (Section 2.4). Finally Section 2.5 will discuss the special requirements for targetting an object oriented VM.

2.1 Functional Programming Languages

Unlike imperative programming languages, in which a function is treated as a series of instructions, functional programming languages treat a function as a mathematical expression transforming that function's input. An immediate side effect of this interpretation of functions is the loss of variables and assignment operations, meaning that a purely functional program can not have any side effects. This is widely regarded as being one of the great strengths of functional languages as it greatly eases the process of proving program correctness [22].

The *Information Processing Language* is widely regarded as the one of the first true functional programming languages, however one of the first 'functional' languages to be widely used was the LISP programming language [19, 31]. While LISP was not a purely functional language (as functions could have side effects) it introduced many of the features found in more recent

functional languages, including features such as lambda expressions, and list processing. In the 1970's the language ML was developed, however it too allowed the use of imperative coding and therefore side effects. The KRC language [63] was the first purely functional programming language; this introduced the concept of pattern matching, allowing function definitions to be based on a case analysis of the argument values.

Another feature many functional languages support is non-strict evaluation (initially described by Friedman and Wise[12]). Under non-strict evaluation an expression is not evaluated when first encountered but rather later when, and if, its value is needed. For example when a function call is encountered the argument expressions are not immediately evaluated and evaluation of the functions definition proceeds; only when, and if, the definition requires the value of an argument is the argument expression evaluated. Among other things this allows for the construction of 'infinite' lists and other 'infinite' data structures[21]. Non-strict evaluation is not required for a functional language, and for this reason there are some languages that provide it, and some that do not. If a language provides non-strict evaluation it is considered to be *non-strict*, and conversely a language without support for non-strictness is referred to as *strict*. Many modern functional languages use the Damas-Milner type system as the basis for their type system as it allows complete inference of all type information [50].

The *Miranda* language [64] later introduced *currying*, a mechanism first described by Schonfinkel [42] and later extended by Curry [6]. In essence currying is the process of representing a function of n arguments as a function that takes 1 argument, and returns a function taking the remainder of the arguments. Here we provide an example of currying a function:

```
add a b = a + b
inc = add 1
```

In this example we have defined a function *add*, that takes two arguments, a and b, and returns the result of adding them. We then define *inc* to be a *partial application* of *add*, resulting in a function that only takes one argument, and returns the result of incrementing that argument by 1. So calling *add* 1 2 or *inc* 2 will both produce a result of 3.

ML later branched into a number of other languages, including SML, Caml, and OCaml. Late in the 1980's the Haskell language was developed. It brought features from many other functional languages including LISP and ML together into a single functional language. Haskell has since become the most widely used pure functional language, both in industry and academia.

Haskell is now one of the primary languages used for research on functional languages, as it presents a well defined language and a number of implementations that can be used as the basis for research. The most recent standard definition is Haskell 98[44], although different implementations provide extensions to the standard.

Functional languages are strongly typed; this means that any value must have a fixed type, be it a tuple, a list, an object, or some other primitive type (such as integers, and floating point values) etc. This does not necessarily mean that the types of function arguments must be declared, as in many cases it is possible for a compiler to infer what the types must be in order to be correct. Most functional languages are also strongly polymorphically typed, which allows the definition of generic functions, such as this:

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

The *reverse* function will be able to operate on a list of any type, meaning it only needs to be implemented once regardless of how many different types of list elements it is expected to work with.

2.2 Virtual Machines

The most general definition of a virtual machine is that it is a system that provides an abstraction from the native hardware of a computer. The most frequently encountered example of a VM with this definition is a computer's operating system (OS). The OS provides an abstraction from the underlying hardware of a computer to increase the portability of programs that it executes, as these programs are no longer reliant on specific hardware. This abstraction allows larger and more complicated programs to be written, and

increases the reliability of these programs, as it allows the developers themselves to work at a higher level of abstraction. This abstraction means that the underlying nature of the hardware from which a computer is built becomes less and less relevant. An early example of this is the IBM Virtual Machine OS, which provided one of the first multiuser environments on a computer. It did this by providing a complete abstraction of the computer, which was called a “Virtual Machine”. By allowing multiple VMs to run on a single system they were able support multiple simultaneous users, where individual programs did not need to be aware of the multi-user nature of the system.

The next level of virtual machines are the runtime systems of programming languages. For example the FORTRAN runtime system provides a uniform I/O system for programs regardless of the operating system upon which they are executing.

Building on this is the virtual machine which abstracts away further from the execution model of the hardware/operating system to provide a model better suited to a particular programming language or languages. For example, the LISP language which was first implemented using a VM in the form of an interpreter [31]. By using a VM it becomes easier to write a compiler, as the compiler can target the VM, which is likely to provide an abstraction from the native machine. For example both the JVM and the CLI provide low level instructions for loading and storage of method arguments, whereas a compiler for a native machine would need to track the arguments manually, including where they are on the stack, or in which register the arguments are.

Another branch of virtual machines are those that emulate one computer hardware model on another, enabling all the software from operating system upwards that is built for one set of hardware to be executed on a completely different set. A current example is Virtual PC which runs on IBM’s PowerPC system and emulates Intel’s Pentium hardware. A different approach to the same idea has lead to the development of systems such as the Parallel Virtual Machine (PVM)[58]. The PVM system provides an abstraction from the actual details of the systems running the virtual machine and allows a group

of computers to appear as a single parallel VM on which software can be executed.

The use of VMs brings many benefits to the language implementor as well as the programmer. The key benefits relate to what the VM is able to do automatically when executing a program. A common feature of modern VMs is the addition of automatic memory management or garbage collection. While a VM is not necessary to provide garbage collection [44], VMs can enforce restrictions on the memory model that make supporting garbage collection easier.

Many modern VMs are more strongly typed than hardware machines and restrict what operations can be performed using memory references. Such restrictions simplify the task of garbage collection as pointers are more readily verifiable [2, 7]. VMs can also improve the security of software by blocking any attempt to access invalid locations in memory; thus avoiding a number of major problems in software development.

Furthermore since the VM enforces type safety it can detect an incorrect argument, and prevent the function from being executed, once again making it easier to find or prevent errors from occurring [60]. This validity checking can be extended to runtime validity checks of the code itself, decreasing the chance of a corrupt executable causing data loss or corruption [52].

In the JVM and CLI the protection aspect of VMs has been further extended, allowing fine grained permissions in the VM to limit access to features that a particular program should not access [14, 57]. As these restrictions are implemented at the VM level, a program should not be able to circumvent the restrictions, even in the case of program errors.

Finally VMs can be used to ease the development of cross platform solutions. This is possible as instead of having to port an application to each target platform, only the VM needs to be ported. For a single application this is not significantly important, but when a large number of applications are involved the savings can be dramatic. For this task the JVM and the CLI are the most popular VMs, as Sun Microsystems has provided JVMs for many platforms [17], and the CLI is implemented on numerous platforms by Microsoft Corporation (through the cross-platform 'Rotor' implementation of

the CLI)[57] and by the Mono project [35]. In order to improve the execution speed of these VMs they tend to be optimised around the concepts used by the principal languages they will be executing. For the JVM that language is Java, the CLI was designed to be a platform that could be targeted by many different object oriented imperative languages [15].

2.3 Targeting an OOVM

Despite the many advantages of virtual machines, there is a potentially significant downside. By definition virtual machines are providing some abstraction of the actual underlying system and in doing so may constrain the range of operations that can be performed. OOVMs enforce a particular object-oriented model and in doing so constrain or restrict the generality of the system in order to provide the guarantees, such as type safety, required for that model.

The design of any high-level VM is a balance between maintaining generality while enforcing the required semantics and, at least in current designs, the former loses somewhat to the latter. This loss of generality means that programming languages that can operate directly on hardware, may not be able to effectively target a given high-level VM.

The challenge is to develop algorithms to increase the range of languages that can be supported on a given OOVM design; or to determine what, if any, changes might be made to its design to increase its applicability without undue negative impacts on the languages currently supported.

In this thesis we restrict ourselves to looking at a small number of features that current OOVMs where not designed to directly support: Damas-Milner style polymorphism, higher rank types, higher kinded types, and non-strict evaluation. Other features, such as multiple inheritance (as in C++, and Eiffel), are not addressed and many have been dealt with elsewhere (for example multiple inheritance by Eiffel for .NET[25]).

Attempts to provide support for features that the OOVM itself does not perform may be non-trivial, and may have significant computational overhead.

For this reason there are a number of decisions that need to made when developing the algorithms.

A basic technique that is often applicable to supporting different type semantics is *erasure* in which some or all static type information is removed from the compiled source and replaced with a the use of a base type such as *Object* and dynamic type checks. Such an approach may provide a simple way for new semantics to be supported however it immediately reduces a key advantage of an OOVM (static type safety), and the dynamic type checks which may have a significant performance impact. For this reason algorithms which do not use, or limit, type erasure are to be preferred. That said, systems can, and have been built almost entirely based on type erasure, for example the York Haskell Compiler [54].

2.4 Existing Functional Language to Typed Virtual Machine Compilers

A number of attempts have been made to compile functional languages to imperative VMs[33, 32, 37]. The problem in attempting to do this is that a VM designed for non-functional languages will tend to not support the features that are required for functional languages to be compiled and executed efficiently[61]. There are a number of different approaches for allowing functional languages to be compiled for efficient execution, and we will discuss these and their advantages and disadvantages.

The first approach that we will discuss is to avoid the features of functional languages that don't cleanly match the target VM. This approach has been used in a number of cases, such as the Pizza[37] programming language. As the CLI (and the JVM) are designed for object oriented languages they do not support features that an Object Oriented imperative language does not need. This includes functions implemented as first class objects (and hence function currying) and non-strict evaluation. To allow efficient execution, these features are removed or altered in order to better match the base paradigm of the VM. Thus the end result may be support for only a subset of the original language to be compiled. However this approach brings with

it the benefit of being completely compatible with the VM, and thus the produced code is usable by any other languages that also target that particular VM.

Another major approach used to let functional languages run on imperative VMs is to modify the VM itself by adding extensions that are needed by functional languages [61]. This type of approach allows for full implementation of the functional language, whilst retaining fast execution. However this has the side effect of requiring the user and developer to acquire non-standard versions of the VM, as any new features will not be present on the standard implementation. To make such a feature useful outside of purely research environments it would need to be incorporated into the major versions of the target VM. Achieving this can be a non-trivial task, especially if the virtual machine is backed by a formal standard and any modifications to the VM must be approved by the appropriate standards body. That said, it is occasionally possible for such features to be integrated into such a VM as a standard feature; such as parametric polymorphism, which is now incorporated into the CLI [27].

Finally there exists the possibility of avoiding writing the compiler at all, and instead relying on the ability of a VM – typically the JVM or CLR – to call functions in external libraries. In this case a functional program is executed natively by getting the VM to call the external code. This is the approach used by the *Lambada* project [33] to provide Haskell support on the JVM, and by the "Hugs for .NET" system [9]. The same mechanisms that allow the execution of external code also allow calls to be made into the VM, thus providing the functional languages with access to the libraries belonging to the VM. This approach allows the full feature set of the functional language to be interfaced with the target VM, and given the functional language is being executed in it's own virtual machine, there should be little if any performance reduction. The principal disadvantage of this approach is it's reliance on the external execution of the functional language, this requires that either the VM for the functional language or the executable itself can be tied to a specific platform, limiting its portability. For the Haskell language this is a less significant problem as there are already implementations for many

different platforms. However Haskell programs and libraries would still need to be recompiled for each target platform.

2.5 Interfacing Haskell to an Object Oriented Virtual Machine

Simply developing algorithms to enable a functional language to be compiled for a given VM is only part of the task. Most VMs come with large software libraries, which naturally are designed to be used from the languages the VM itself is designed for. Using the libraries is obviously beneficial; for example allowing code sharing and assisting language inter-working; but their computation model may present a problem. For functional languages this has been investigated by a number of researchers[10, 47, 38].

While there are many aspects to this problem. A simple mismatch which makes sharing libraries more difficult is that functional languages are built around modules and functions, while OOVMs are built around the class and method, One approach to this is to use a class to encapsulate any compiled module, and define all functions as static members of that class [9, 33, 59].

Related to this issue is the reliance on classes and sub-typing on OOVMs which is not present in type systems of most functional languages. This presents a problem mapping types between the two systems[38]. To address this problem the *Lambada* system used the Java Native Interface to provide a bridge[33], and used Haskell type classes to encode object oriented hierarchies. In other Haskell related work Finne introduced 'Phantom types' to handle the subtyping of classes[10].

Another hurdle is that of enforcing the non-side-effecting semantics of a purely functional language when interacting with an impure VM. Any function in a functional language is wholly dependant on its arguments, a constraint not enforced by imperative VMs where side effects are required.

Functional language compilers can perform code optimisations which would be invalid in the presence of side-effects. To address this some mechanism is required to isolate the functional and imperative code if inter-working is to be successful. The Glasgow Haskell compiler provides such a mechanism in its Foreign Function Interface (FFI). By using the FFI it is possible for a

developer to reference an externally defined function, and through the use of monads ensures correct semantic meaning in both Haskell and the external language [43, 49].

Chapter III

Functions and Non-Strictness

In this chapter we will introduce some of the more advanced features of non-strict functional languages and discuss a number of methods in which they can be implemented. Initially we will discuss lambda expressions followed by a discussion of the methods by which partial applications can be processed. The final feature we will discuss in this chapter is non-strict evaluation.

3.1 Lambda Expressions

Unlike imperative languages, where a very strong distinction is made between data and functions, functional languages do not distinguish between them in the same way, allowing both to be used as first-class values. Since they are first-class values they can be stored, retrieved, and manipulated in the same way. This has a number of effects on what features functional languages provide to a developer, including partial applications, which we will describe in Section 3.2, and lambda expressions.

A natural extension of functions being values is to have expressions which can create new function values. Such an expression is referred to as a lambda expression in functional languages and, more recently, as an anonymous delegate in C# [26]. While obviously not constrained to functional languages (lambda expressions were available in Algol 60), it is here that lambda expressions are used most frequently.

This allows a programmer a degree of flexibility that would not otherwise be available. While the transformations required to allow a lambda expression to be used on an OOVM are already well understood, it is necessary to explain them as a prerequisite to later discussions on how we handle more advance features.

If we take a simple function $f\ x\ y = x + y$ we can see that is is logically equivalent to the declaration $f = \lambda\ x\ y \rightarrow x + y$. Both result in the function f being of type (Number $\rightarrow$ Number $\rightarrow$ Number), and they are just different representations of the same function. Of course, such a use of a lambda expression has little advantage over a standard function declaration. Lambda expressions are much more useful when declared as part of a larger expression. For example wo could use a lambda expression as an argument to *map*, thus allowing us to increment each item in a list, without having to define a separate function:

$$f\ l = map\ (\lambda\ x \rightarrow x + 1)\ l$$

In this example the lambda expression is very simple, however it can easily be used to demonstrate the process known as lambda 'lifting' [42]. This is a process whereby we 'lift' a lambda expression out of the expression it is in, and convert it to a separate function. The original lambda expression is then replaced by a reference to the new function. Lambda lifting is used as part of the implementation strategy for lambda expressions.

By applying the above process to our example produces the following pair of functions:

$$g\ x = x + 1$$

$$f\ l = map\ g\ l$$

Here the function g names the lambda expression and a reference to g is used in place of the lambda expression. Such a simple function could also have been achieved using partial evaluation, as we will discuss in Section 3.2.

Lambda lifting is more involved than simply extracting sub-expressions. For example, consider the following generalised version of the above function:

$$f\ inc\ l = map\ (\lambda\ x \rightarrow x + inc)\ l$$

Here simply extracting the sub-expression would result in the created function referencing *inc*, but *inc* will not be in scope:

$$g\ x = x + inc$$

$$f\ inc\ l = map\ g\ l$$

Variables such as *inc* are termed *free variables*, in a similar way *free types* can also occur. To handle free variables and types an *environment* is introduced, indicated by $\{\ldots\}$:

$$g\{inc\}\ x = x + inc$$

$$f'\ inc\ l = map\ g\{inc\}\ l$$

By applying lambda lifting the function space is made 'flat', without any function definitions nested inside others. This greatly simplifies implementation.

There are two common implementation strategies for handling the environment. The first is to represent it explicitly, the free types and variables being provided to the function through some specialised mechanism. The second is to represent it implicitly by adding the free variables and types to the argument list of the lifted function:

$$g\ inc\ x = x + inc$$

$$f'\ inc\ l = map\ (g\ inc)\ l$$

In this case g has no free variables, but when it is called, only one argument is used instead of two. This is referred to as a *partial application*, and we will discuss these next.

3.2 Partial Application

A partial application is the result of calling a function without passing all of the arguments it needs to evaluate completely. Partial applications occur frequently in programs written in functional languages, and can be used for many purposes; for instance incrementing every element in a list of numbers could be achieved by passing the partial application (1+) as the function parameter to *map*. Given the importance of partial applications to functional languages, they are a feature that is critical to implement as efficiently as possible. In this section we will discuss the mechanisms by which partial applications have been implemented, and raise efficiency concerns for each method.

Before considering how partial applications might be implemented we first need to review the two main approaches to implementing ordinary (all arguments provided) function calls. Ignoring unimportant, in this context, details such as register usage the two approaches in summary are:

Push-Enter In this model argument expressions are pushed right-to-left onto a stack and then the callee is 'entered' (called). The callee removes the arguments it requires from the stack and returns its result. If the stack is not yet empty the *returned result* is in turn entered – in a type correct program it must itself be a function – and so on until the stack is empty. This is best shown by example, consider:

$$f\ x = \lambda\ z \rightarrow x + z$$
$$f\ 3\ 4$$

To evaluate this expression the values 4 and then 3 are pushed onto the stack and the function f entered; this removes its argument, 3, from the stack and returns the function value $\lambda\ z \rightarrow 3 + z$; the stack is not empty so this returned function value is entered, 4 is removed from the stack and 7 returned. The stack is now empty and the final result is 7.

Eval-Apply This is the traditional method used by most imperative languages. It is the responsibility of the *caller* to make the correct number

of arguments available, usually by pushing onto a stack, and the *callee* assumes they are all present. In a strongly typed system the correct number of arguments is always made available, though there are some weakly-typed systems which allow variable numbers of arguments they are not relevant to our discussion. The name *Eval-Apply* comes from function arguments usually being evaluated first and then the function applied. However when used by a non-strict language arguments may be passed unevaluated; the important aspect of the model is the number of arguments passed is known and fixed. The Hope+C[41] functional language uses this model.

We will now look at how partial applications may be supported using these two approaches, starting with *Push-Enter* as this is one of its primary purposes.

3.2.1 Using The Push-Enter Model

The *Push-Enter* model is used by a number of functional language implementations, including the *Three Instruction Machine*, or TIM[8, 46], and the *Spineless Tagless G-Machine*[42] used by the Glasgow Haskell Compiler[45] and Mondrian[40].

From the description above it can be seen that this model can easily support partial applications – when a callee is entered if insufficient arguments are available on the stack the available ones can be removed and some representation of a partial application constructed and returned. Indeed this is one of the primary purposes of the *push-enter* model.

The algorithm used on function entry is shown in Figure 3.1. Note that this is essentially an interpretive algorithm, at runtime tests are performed and the appropriate action determined. By using threading-style compilation techniques these tests can be exchanged for indirect-jumps, and though this may reduce the runtime cost it is not eliminated.

```
enter function
  if (normal function)
    if (enough arguments on stack)
      evaluate function normally
    else
      remove available arguments from stack
      build a partial application closure containing
        arguments and pending arg count
      return closure
  else // applying an existing partial application
    if (enough arguments on stack to satisfy pending
        arg count)
      evaluate function using args on stack and args
        in closure
    else
      remove available args from stack
      build new partial application closure with union
        of removed args and args from current partial
        application closure
      calculate its pending arg count
      return new closure
```

Figure 3.1: Algorithm to perform a function call using the push-enter model.

3.2.2 *Using The Eval-Apply Model*

The *eval-apply* model is based around placing the correct number of arguments on the stack *before* calling a function, so the mechanism does not trivially support partial applications at all. However it is the model commonly used for imperative languages, and more importantly in this context, is the model supported by the major OOVMs.

It is possible to implement the *push-enter* model on an OOVM which uses the *eval-apply* one, for example as done by the Mondrian[40] system; but as shown by that system this introduces inter-working issues and requires the use of dynamic rather than static type-checking, with consequential impacts on performance. Fortunately, as shown for example by Hope+C[41], it is possible to transform a program which uses partial applications into one which does not.

We therefore chose to support partial applications by transforming them away, inline with our goal of maximising compatibility with other code executing on the OOVM. In doing so the runtime overhead present in the *push-enter* is shifted to compile time.

3.2.3 *Transforming Partial Applications Into Complete Applications*

The simplest algorithm for transforming partial applications into complete ones is to wrap each partial application inside a lambda expression which takes the missing arguments and performs a complete application. For example the partial application (1+) can be replaced by the lambda expression $(\lambda\ z \rightarrow 1{+}z)$. Once this transformation is done lambda lifting can be applied as above.

However in some languages, including Haskell which we are using, determining whether an application is complete may not be simply a case of comparing the number of arguments supplied against the type of the function. Consider the two functions:

$$p\ x\ y = (\text{if}\ x > 0\ \text{then}\ \sin\ \text{else}\ \cos)\ y$$

$$q\ x = (\text{if } x > 0 \text{ then } \sin \text{ else } \cos)$$

Both p and q are of type $Integer \rightarrow Real \rightarrow Real$. Now if we consider the applications p 3 and q 3 it is not possible based on types to determine that the former is a partial application and the latter a complete one. Furthermore the definitions of the functions may not be available – e.g. they could have been separately compiled and being used from a library – so they cannot be examined to determine the difference.

To handle this we introduce the concept of arity, that is the number of parameters required for a function to actually compute a value. In our example p has arity 2 while q has arity 1.

Using eval-apply and handling partial applications in the compiler rather than using push-enter shifts the cost of supporting partial applications from execution time to compile time, but at the cost of some added complexity in the compiler. When using the eval-apply method a compiler needs to be aware of the arity of all function references. The compiler must then ensure that anytime a function reference is used, the function has the correct arity; if necessary by performing some type of transform of the underlying expression. This is required because functions are now called directly – it is no longer possible to pass some arbitrary number of arguments to a function and assume the virtual machine will ensure that any applications happen in the correct order. The problem the compiler now faces is that is is possible to have multiple functions or expressions that have the same base type, but may have different arities.

Consider again the functions p and q above. The former requires two arguments in order to produce a full application, whereas the latter requires two sequential applications of one parameter each in order to completely evaluate. This means that the compiler may be required to transform any function or lambda expression used as an argument to the correct arity for that parameter.

For the following examples we will use the notation $\{type, arity\}$ to communicate both the type and the arity of a function. For future references we will calls this tuple a 'signature'. In the eval-apply model, all function references

(not necessarily the function declarations or lambda expressions) will have an arity of $n-1$, where n is the number of parameters in the type signature of the function reference. Consider:

$$f\ a\ b\ c = a\ b\ c$$

where the function f has the type $(t \to t1 \to t2) \to t \to t1 \to t2$, where t, $t1$, and $t2$ are type variables. The full signature of the function f is $\{\{(t \to t1 \to t2), 2\} \to t \to t1 \to t2, 3\}$ (eg. the first parameter is a function $t \to t1 \to t2$ with an arity of 2, while f itself has an arity of 3). In this case there is no difference between the signature of an equivalently typed anonymous function reference and the signature of the function declaration.

However, if we refer to our previous examples of identical type but different arities:

$$p\ x\ y = (\text{if } x > 0 \text{ then } \sin \text{ else } \cos) y$$

$$q\ x = (\text{if } x > 0 \text{ then } \sin \text{ else } \cos)$$

The signature of p is $\{Int \to Real \to Real, 2\}$ whereas q has the signature $\{Int \to Real \to Real, 1\}$. Consider now using these as parameters to the function f:

$$h1 = f\ p\ 1\ 2$$

$$h2 = f\ q\ 1\ 2$$

For correct operation not only must the types match but also the arities. Examining $h1$ shows that all the parameters have the correct arity, however when examining $h2$ reveals that the function q does not match the arity required (2). Therefore the body of $h2$ must be transformed to supply an argument of the correct arity. This is simply achieved by wrapping the expression in a lambda of the correct arity:

$$h2' = f\ (\lambda\ a\ b \rightarrow q\ a\ b)\ 1\ 2$$

All arguments now have the correct arity, but there is one minor transformation required – g is being applied to two arguments, but it only takes one and returns another function which consumes the second. It is a trivial transformation to examine the expression and break it into its constituent applications producing:

$$h2'' = f\ (\lambda\ a\ b \rightarrow (g\ a)\ b)\ 1\ 2$$

3.2.4 Discussion

We can see now that the two mechanisms used to provide partial applications differ drastically in complexity and in the amount of work required by the compiler. Currently there is much debate over which mechanism is the 'best'. Many functional languages use the push-enter model as their primary execution model, although they often optimise full applications of global functions by reducing them to direct calls [42, 45]. Despite this, some recent studies have shown that the eval-apply method result in faster execution at runtime [30]. Given the reports of improved performance of the eval-apply model, and the fact that it is the underlying model of the CLI we have chosen to use it rather than push-enter for our experimental compiler, as discussed in Section 9.

3.3 Non-Strictness

Non-strict evaluation is a mechanism by which no expression is evaluated until the result of the expression is required. Non-strict evaluation can take many forms, either through the implicit laziness of Haskell and similar functional languages, or explicitly, as in the LISP programming language. A limited form of non-strictness is provided in many imperative languages, where it takes the form of *short-circuit* evaluation of boolean expressions (and the conditional operator if the language supports it).

In addition to language level support, non-strict evaluation can also be manually implemented by developers in other languages; for a very wide range of tasks. The archetypal example of non-strict evaluation in such cases is the *Proxy* design pattern [13], which describes a number of uses for proxies, eg. cases that benefit from non-strict evaluation. These include such applications as web browsing, graphics, text editing and numerous other tasks from many different fields. An advantage often cited in fully non-strict languages is the seamless handling of potentially infinite computations, such as generating lists containing values from an infinite series.

Non-strict evaluation is not a required feature of functional programming (many examples of strict functional languages exist, such as ML and its descendants), however it is present in many, including Haskell.

Fully automatic non-strict evaluation is not practical within object oriented languages, as they (along with most imperative languages) rely heavily on a predefined order of evaluation[29] at execution time. Since non-strict evaluation is not present in object oriented languages, OOVMs do not provide native support for non-strict evaluation. For this reason it is necessary to develop a mechanism by which non-strictness can be provided.

The first approaches to implementing non-strict evaluation on top of object oriented virtual machines required explicit evaluation (through calls to 'eval' functions)[34] code. As such, while suitable for functional language only environments, they do not meet our goal of inter-working well with other code executing on the OOVM.

A more recent development is JIT Objects[39]. By leveraging the OOVMs sub-typing mechanisms JIT Objects provide a transparent way to suspend the evaluation of expressions in both functional and imperative languages. The original work demonstrates JIT Objects in use in C#.

However JIT Objects as originally specified have a mismatch with those languages, such as Haskell, where rather than the exact type of a suspended expression being known it is only known to be one of a family of related subtypes. This occurs in Haskell where the suspended expression has algebraic type. We return to this problem in a later chapter.

Chapter IV

Types

In this chapter we introduce the key aspects of the type systems used by functional and imperative object-oriented languages that are relevant in our context of supporting functional languages on OOVMs. To make the discussion concrete we will use Haskell in the functional language description and C# when describing object-oriented languages. In the next chapter we will look at how the type system used by functional languages may be mapped onto an OOVM.

4.1 Functional Language Types

Algebraic types and records will be the first feature we will discuss, as they are central to functional languages. This introduces parametric types and this leads to parametric polymorphism (Section 4.1.2). Section 4.1.3 will describe the concept of type classes provided by Haskell.

4.1.1 Algebraic types and Records

The *algebraic type* is the basis of typing in a functional language. In this section we will introduce the features of algebraic types, and discuss how they are used. We will also examine Haskell *records* and show that they are equivalent to algebraic types, and hence do not need to be treated differently.

Algebraic types are the basic type construct of functional languages, allowing a user to define structured types and unions. A data type consists of a name, a set of type parameters, and a set of constructors, eg.

```
data IntList = Cons Integer IntList | Nil
```

In this example *IntList* is the name of the type, and *Cons* and *Nil* are termed *constructors*. The *Nil* constructor is what is a referred to as a *nullary* constructor as it does not contain any fields. The *Cons* constructor does have fields however, one of type *Integer*, the other of type *IntList*. In other words *IntList* is a tagged (*Cons* and *Nil*) disjoint union of two types (the pair *Integer IntList* and the void type).

Algebraic types may also be parameterised so that they describe a family of types:

```
data List a = Cons a (List a) | Nil
```

In this example *List* is termed a *type constructor*, *a* is a type parameter. Such a type is called a *parametric type*.

To handle algebraic types a mechanism called *pattern matching* is provided which combines constructor testing and field extraction [20]. Simple matching, in which no conditions are made when matching against a field, is also known as *decomposition*. The following example (using the above *IntList* type) demonstrates both constructor matching and decomposition:

```
length :: IntList -> Integer
length Nil = 0
length (Cons head tail) = 1 + (length tail)
```

This function computes the length of a *IntList*, the first line matching against the *Nil* constructor, the second matching the more complex *Cons* constructor. This basic match has separated handling of the *Nil* and *Cons* constructors, in a fashion similar to switch on type technique used for unions in imperative languages. When we examine the *Cons* branch of *length* we can see it is matching the two fields of the constructor to two variables, *head* and *tail*.

Since the fields in a constructor are not labelled, complex constructors may become difficult to manage and use. For this reason Haskell [44] provides support for *records*. *Records* are a special kind of data type, where there is only one constructor, however that constructor is able to name each field. It

is important to realise that *records* are not in fact any different from standard algebraic types, but instead are a purely syntactic addition, for this reason we will not discuss them further.

Functional languages also usually provide special syntax for *tuple* types. The type $(t_0, ..., t_n)$ (where $n \neq 1$) is equivalent to the parametric algebraic type:

data Tuple(t_0, ..., t_n) = Tuple t_0 ... t_n

4.1.2 Parametric Polymorphism

Most functional languages support *parametric polymorphism* where functions may be defined to operate over many types. Consider the following version of *length* function in Haskell:

```
length :: List Integer -> Integer
length Nil = 0
length (Cons head tail) = 1 + (length tail)
```

This computes the length of a value of type $ListInteger$, a parametric type. Clearly an almost identical function could be written to compute the length of a $ListChar$ – the value of the parameter to $List$ does not effect the algorithm. To address this functional languages support parametric polymorphism where type *variables* may be used in place of types:

```
length :: List a -> Integer
length Nil = 0
length (Cons head tail) = 1 + (length tail)
```

Here a is a type variable and $length$ may now be applied to a list of *anything*. Type variables are not restricted to being used in parametric type signatures. For example:

```
revApp :: a -> (a -> b) -> b
revApp x f = f x
```

Defines a 'reverse application' function, such that $revApp\, y\, g$ is equivalent to $g\, y$, which can be applied to any functions and values which match the signatures $a \rightarrow b$ and a respectively.

4.1.3 Type Classes and Instances

Type Classes were developed as a mechanism to provide a higher-level and more capable alternative to ad-hoc polymorphism (overloading) to functional languages in general, and Haskell in particular [65, 18]. A type class enables the types that may be substituted for a type variable to be constrained to those which provided a particular set of functions, namely those specified in the type class. For example consider the standard Haskell class *Eq*:

```
class Eq a where
  (==) :: a -> a -> Bool
```

This defines the class *Eq* to contain a single function (==) of type $a \rightarrow a \rightarrow Bool$. In a type signature the term $(Eq\, a)$ represents a *context* or constraint on a requiring any type substituted for a to implement the *Eq* class, which in turn means that a (==) function is available over values of type a even though the actual type substituted for a is unknown. For example:

```
not_equal :: (Eq a) => a -> a -> Bool
not_equal a b = not (a == b)
```

In order to use the *not_equal* function we need to define an *instance* of *Eq* for some type. For example consider an instance with the *Bool* type:

```
instance Eq Bool where
  (==) = boolEquals


boolEquals True  True  = True
boolEquals False False = True
boolEquals _     _     = False
```

The *not_equal* function can now be used with *Bool*, and any other type for which an instance of *Eq* is defined.

We have shown that when defining a function a context may be provided to allow that function to use a class, this can also be applied to the definition of instances. For example by providing a *context* an instance for *List a* for all types *a* for which *Eq* is provided can be defined (as shown in Figure 4.1).

```haskell
instance (Eq a) => Eq (List a) where
  (Cons lh lt) == (Cons rh rt) = if (lh == rh) then
                                   lt == rt
                                 else
                                   False
  Nil == Nil = True
   _  ==  _  = False
```

Figure 4.1: An *instance* of the *Eq* type class covering *Lists*. We can see the use of the context *Eq a*, thus allowing *Eq List a* to exist for all types that provide an *instance* of *Eq*.

4.1.4 Higher Ordered, Ranked, and Kinded Types

In this section we discuss a series of extensions to the basic types that we have already discussed, starting with simple higher order types, followed by an introduction to higher ranked types. Finally we discuss higher kinded types.

Higher ordered types are one of the most fundamental types of any functional language, they represent the ability of a function itself to be treated as a value (a task imperative languages tend to use function pointers or *delegates* for). This allows functions to be passed to and returned from other functions, thus providing a very simple mechanism to control what actions are performed inside that function. Many standard design patterns [13] (*Iterator* and *Functor* being prime examples) can be replaced trivially with higher ordered types.

However higher ordered types cannot reference polymorphic functions, instead the polymorphic functions must have had any required type arguments bound in advance. There are rare cases where it may be desired to have a function reference that is itself polymorphic, and in these cases it may be difficult to remove this requirement [48], for this reason a number of functional languages now provide support for them (including Haskell through an extension provided by the Glasgow Haskell Compiler). These polymorphic function references are referred to as *higher ranked* types, and we discuss them in further detail in Section 8.

Finally Haskell supports *higher-kinded* types. Higher-kinded types are the

type equivalent of higher-order functions. In the parametric types described so far a type variable may only be substituted by a *type*. In a parametric type declaration, such as *List a*, the type name (usually termed a *type constructor*), *List*, is a *function over types*; that is it takes a type '*a*' and returns another type '*List* of *a*'. A higher-kinded type variable may only be instantiated with a type constructor [55]. Using higher kinded types allow a function to operate over any algebraic type, rather than a specific type such as *List*. Through this ability a function may be defined to allow it to operate over a list or a tree structure, without requiring different definitions. Higher kinded types are described in more detail in Chapter 7.

4.2 Imperative Object-Oriented Types

There are many imperative languages in existence today and a large number of these have support for object oriented concepts. There are dynamically typed languages like Python [51], Ruby [53], and Smalltalk [24], as well as large numbers of statically typed languages ranging from Java [17] to C++ [56] to Eiffel [25], each with varying degrees of 'purity'. For these reasons this section will be devoted primarily to the most basic concepts of imperative OO languages.

As they are the building blocks of OO languages the first constructs we discuss are classes and structures (Section 4.2.1). This discussion is followed by a brief coverage of primitive and value types provided by the CLI itself in Section 4.2.3.

4.2.1 Classes

Classes are the central type structure of object-oriented languages (there are OO languages which are not centrally class-based, we shall not refer to these further). A basic class is a collection of named fields, a product in type terms – much like the tuples in functional languages. For example consider:

```
class Point
{
```

```
    float xCoord;
    float yCoord;
}
```

This defines a C# class *Point* with two *float* fields named *xCoord* and *yCoord* (a similarity to tuples and records in Haskell should be apparent at this point). If the *Point* class is subclassed (see Section 4.2.1) the fields *xCoord* and *yCoord* will be inherited by the subclass.

Methods

Functions defined as part of a class are referred to as *methods*, and we will use this terminology to distinguish between functions defined globally, and those defined as part of a class. As with fields, methods are inherited when subclassing, however how they are inherited, and what subsequent behaviour they specify can be controlled.

For example consider:

```
class Rectangle
{
    Point corner;
    float width;
    float height;

    public float Area() { return width * height; }
}
```

This represents a rectangle with a method to compute the area. The syntax of how these methods are called is not important for our purposes. However it is worth noting the distinctly different approaches of functional and OO languages here: in a functional language functions and data are usually defined separately while in on OO language the methods are defined as part of the classes. This will effect how functional languages are mapped onto OOVMs.

Subclassing

Classes can depend, or be *derived* from, other classes using *subclassing*, or *inheritance*, which is mechanism which combines method overriding and *subtyping*. If a class A subclasses a class B then class A can be used in any place that class B would be required. While it is technically possible to provide support for *multiple inheritance* in an object oriented system [25, 56] modern OOVMs provide only limited support for it, we discuss these restrictions later in this section.

For example consider:

```
class ColouredPoint : Point
{
    Colour c;
}
```

This defines a class $ColouredPoint$ which has three fields in total; c, $xCoord$ and $yCoord$. A value of type $ColouredPoint$ may be used wherever one of type $Point$ is required, but *not vice-versa*.

The methods defined over a class may either be *virtual*, *non-virtual*, or *abstract* (exact terminology varies between languages). When a method is implemented it may be defined as either virtual or non-virtual. If the method is virtual any subclass of the containing class may redefine the method, thus providing an ability to *override* the original implementation of that method, in this case a class is effectively combining the goals of a type class and an instance. Any attempts to call the original method on an instance of the subclass will be diverted through to the new implementation. If a method is non-virtual it is not possible for the method to be overridden.

If a method is declared to be abstract it means there is no initial implementation, eg. it behaves in the same way a function defined in a type class behaves. If a class contains any abstract methods it is called an *abstract class*. Abstract classes cannot be directly instantiated, instead only subclasses of the abstract class may be, provided the subclass has provided implementations of the abstract method. By definition an abstract method must be virtual

and thus may be overridden by a subclass of the class that initially provides in an implementation.

Interfaces

As mentioned earlier imperative object oriented languages may provide support for multiple inheritance, but this is not supported in the major virtual machines. Instead OOVMs such as the CLI and JVM provide a restricted form of multiple inheritance through the use of *interfaces*. An interface is a special construct that can only define abstract methods, and cannot contain data. In this way they match the behaviour of type classes almost exactly, though the interface is implemented through subtyping, rather than declaring an instance of the interface that operates over a particular data type.

In order to better describe the OO structures we will be referring to in this thesis we will be using the Unified Modelling Language (UML) [11]. A discussion of UML is beyond the scope of this document, so it assumed that the reader has at least a passing knowledge of the meaning of UML class diagrams.

4.2.2 Generic Classes and Methods

Some OO languages and OOVMs also support *generics* which equate to the parametric types and polymorphic functions found in functional languages.

For example the following defines a generic C# class similar to the 2-element tuple of Haskell:

```
class Pair<A, B>
{
    A fst;
    B snd;
    ...
}
```

And a generic method to swap elements of a *Pair* could follow the template:

```
Pair<B, A> swap<A, B>(Pair<A, B> arg) { ... }
```

Generics are a recent addition to OOVMs and not only enhance the OO environment but greatly assist the mapping of functional languages with polymorphic type systems, as covered in the next chapter.

4.2.3 Primitive and Value Types

While early OOVMs followed the 'semi-pure OO' model, where all values are objects except the built-in primitive types (such as integers etc.), more recent OOVMs provide direct support for general user-defined value types in addition to object-based ones.

In particular the CLI provides full support for value types; which can have their own methods and implement interfaces, as with classes. However sub-classing is not provided. Finally for every value type defined a matching object type, termed the *boxed* type, is also declared and operations to covert back and forth from the value to boxed types provided. In C# these operations are largely automatic.

In addition to the primitive types and custom value type the CLI also provides support for *enumerated types*. Enumerated types provide a mechanism to define a set of constant values which may be used with a degree of static type safety (there are a number of mechanisms that can violate the safety of an enumerated type on the CLI, though these are beyond the scope of this thesis).

Chapter V

Converting Functional Types to an Imperative Object Model

In this chapter we will introduce the basic mechanisms used to convert the core of the Haskell type system to the structures of an OOVM. Much of this material has previously been introduced previously by the Mondrian Project [40], the F# [59] language, or plain common sense. While some new approaches have been used, the primary purpose of this chapter is to provide a context for the following chapters on non-strictness, higher ranked types, and higher kinded types.

Section 5.1 will discuss the mechanisms and constructs used to support standard *Haskell's* algebraic types. This is followed by the systems required to support type classes and instances in Section 5.2, and finally we discuss support for higher order types in Section 5.3.

5.1 Algebraic Types

As discussed in Section 4.1.1 algebraic types are the fundamental form of structured type used by Haskell, and most other functional languages. Therefore it is critical that a fast and effective mechanism is available to support both the types themselves, and access to their constructors. In this section we will describe the mechanisms developed by Mondrian and F# to support algebraic types.

For the following discussion we will use the simple List type, and the corresponding *length* function:

```
data List a = Cons a (List a) | Nil
```

```
length :: List a -> Int
length Nil = 0
length (Cons _ l) = 1 + (length l)
```

The first definition describes a new *List* type, with a type parameter *a*, and two constructors; *Cons* and *Nil*. In this case *Cons* has two fields, one of type *a*, and the other of type *List a*. The second definition describes a *length* function that determines the length of the given *List*. Together these two definitions use the important features of the Haskell algebraic types, and therefore provide an ideal base from which to described the conversion process.

An algebraic type is a tagged (the constructors) disjoint union, one possible mapping of this onto an OOVM is to use one class, termed the root class, to represent the algebraic type itself, and sub-classes of the root class to represent each constructor [37, 40, 59]. In the event that the type has type parameters we have the choice of either performing type erasure, in which we replace all type parameters with a generic *Object* reference, or relying on VM support for parametric types and adding these type parameters to the root class of the algebraic type. As the type erasure approach effectively removes any ability to retain static type information of algebraic types with type parameters we will ignore this mechanism, and instead assume that the target VM supports parametric typing. Figure 5.1 shows the result of applying these techniques to the *List* type above.

In order to prevent modification (from other languages accessing the data structure) we shall actually map constructor elements to private fields and provide accessor methods (or properties) for them.

Functional languages perform 'switch on constructor' operations to determine which code branch to take. Using the class/subclass mapping this can be done using dynamic type checks and an if/then/else if pattern. However this would be relatively slow. Functional language implementations on standard machines typically support this feature by storing a small tag value in each constructor representation. We adopt the same technique and add a tag property to the root class.

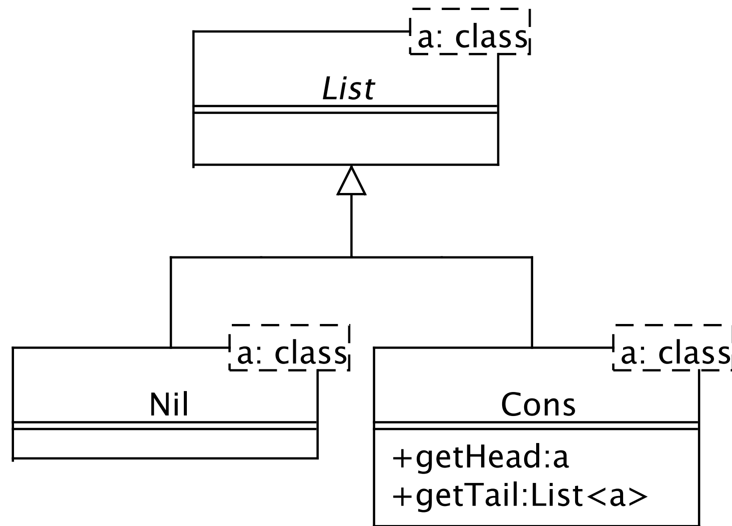

Figure 5.1: Classes defined for the *List* type.

While the tags could be any quickly comparable type, we have used an *enumeration* as these also provide much greater feedback to developers accessing Haskell algebraic types from other languages. The other concern is how the tags are accessed; if the tag is stored by value in the root class of a type it can be accessed through a direct field operation (once again the tag field would be encapsulated by a method, but modern OOVMs attempt to inline trivial *accessor* methods [57]) however doing so would result in every instance of the type requiring more memory than might otherwise be required. The alternative is to use virtual methods that return the tag explicitly, this saves on the space requirements of storing the tag value itself, but adds the relatively expensive cost of a virtual call. Whichever approach is taken, the net effect is the same as it will be possible to query the tag of any algebraic type instance, and hence perform the switch on constructor type.

This final structure gives us full support for strict algebraic types, for details of how support for non-strictness is added to this structure see Chapter 6. The final system produced for the *List* type we have been using is shown in Figure 5.2.

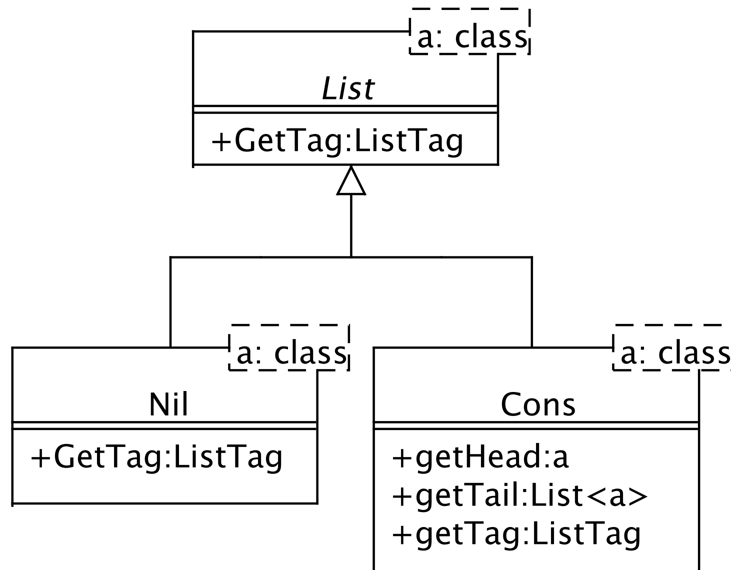

Figure 5.2: Classes defined for the *List* type with tags.

5.2 Type Classes and Instances

Type classes and instances were described in Section 4.1.3 as a mechanism to provide support for function overloading. While described as a distinct concept *Haskell* compilers typically compile classes into method dictionaries, that is records containing one method reference per class member, and calls to class methods first index into this dictionary to obtain the applicable method reference. By transforming the *Eq* class and *Eq Bool* instance (repeated in Figure 5.3) we can produce the set of functions and types shown in Figure 5.4.

Since we can transform type classes and instances into standard types there is no specific need to treat them distinctly. While such an approach would work, the way in which type classes are used is very well structured, and is amenable to a more OOVM friendly implementation. Section 4.1.3 discussed the similarity between type classes and interfaces on an OOVM, and we will now make use of this fact.

As most uses of fields of the type class constructor will be direct application of the functions contained, we can convert the explicit dictionary extraction

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  (==) = boolEquals

instance (Eq a) => Eq (List a) where
  a == b = ...
```

Figure 5.3: The *Eq* class and the *Eq Bool* and *Eq List instances* described in Section 4.1.3.

```
data Eq a = Eq (a->a->Bool)

dictEqBool :: Eq Bool
dictEqBool = Eq boolEquals

(==) :: (Eq a) -> a -> a -> Bool
(==) (Eq eqFun) = eqFun

not_equals :: (Eq a) -> a -> a -> Bool
not_equals context a b = not ((==) context a b)
```

Figure 5.4: Conversion of *Eq* class, *Eq Bool* instance, *dictEqBool*, and the *not_equals* function to use standard types.

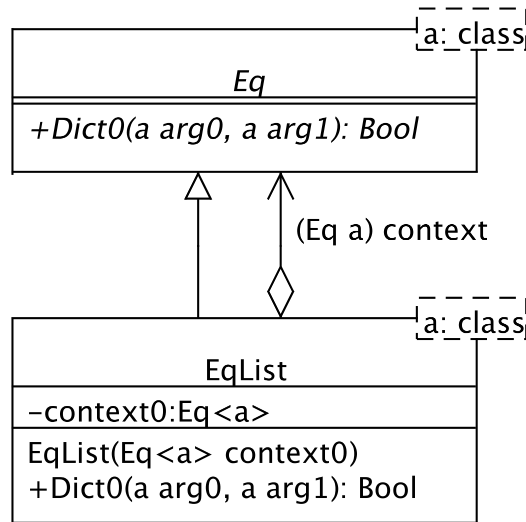

Figure 5.5: The *Eq* type class, and the instance for *Eg (List a)*.

above into an implicit dictionary using virtual methods. This is done by converting the dictionary type into an abstract class, as shown in Figure 5.5.

Instances can than be implemented through subclassing the dictionary class, although care must be taken when an instance depends on the existence of another instance, as in the case of *Eq (List a)*. In that case references to required instances are stored as fields of each instance of the dictionary class. The superclass structure defined for type classes is, in effect, a standard application of a *context*. For this reason any super classes are accessed through fields in the same manner as function members of the type class. Figure 5.5 shows the structure for the instance *Eq (List a)*.

It may at first be thought that OOVM inheritance could be used implement the dependency relation between type classes. However a type class can depend on multiple other type classes, and most modern OOVMs do not support multiple inheritance in any form other than the limited features of *interfaces*. These limitations, combined with the potential for ambiguous functions (when inheriting from numerous type classes with the same function types) mean that using multiple interface dependencies would be difficult for type class dependencies. Therefore we have followed the standard approach to compiling type classes, and provide access to super classes through explicit

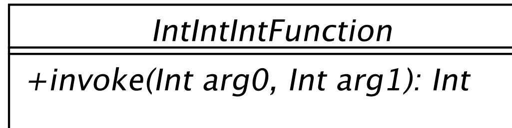

Figure 5.6: A class representing a function of type $Int \rightarrow Int \rightarrow Int$ with arity two.

fields and properties in the type class, as we do with a standard *context*.

5.3 Higher Order Types

Higher order types, or function values, are relatively trivial to provide on an OOVM simply by using the *Command* design pattern [13]. In this we first declare a base type for a given function type which will act as the base OOVM type for that function type. As with the command pattern an invocation method is present in this class (in all of our examples this will be the *invoke* method) with a signature that matches the type encoded. Figure 5.6 gives the class produced for the type $Int \rightarrow Int \rightarrow Int$ with an arity of two (see Section 3.2 for a description of arity).

Creating a separate interface or class for every function type would result in large numbers of distinct types and be difficult to coordinate. Furthermore most OOVM's use name, and not structural, equality; if one compilation produced a class for a given function type it would not be directly compatible with a class produced in a different compilation for the exact same function type; severely impacting the use of separate compilation and libraries. For this reason we use a number of parametric classes to represent different function types. Each type parameter represents the type of the corresponding type in the function type signature, and different instantiations of these parametric types with the same type parameters are compatible, as shown in Figure 5.7.

This approach raises the question of how many of these parametric function types should be provided; they cannot simply be created on-the-fly as different compilations must use the same types or they will not be compatible. In

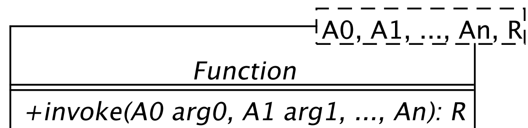

Figure 5.7: Generic *Function* class for representing higher order types.

practise functions do not have large numbers of parameters. Therefore only a small number of these types may be provided (the experimental compiler described in Chapter 9 provides support for a maximum of 7 parameters), without imposing a significant limitation on programs.

Whenever it is necessary to create a higher order type (eg. a partial application, lambda expression, or passing a function reference) the expression to be passed is placed inside the invocation method of a subtype of the appropriate function type. As discussed in Section 3.1 it is possible that free variables or types may be present in the expression we are attempting to pass. This environment can be trivially stored either as fields in the case of free variables, or as type parameters to the subtype, as in the case of free types.

Chapter VI

Providing Non-Strictness to a Strict VM

Historically it has been difficult to allow a developer to use code which was written to benefit from a certain language feature in one language from within another language. Doing so often relies on awkward and complex mechanisms, such as GHC's Foreign Function Interface [4]. Such mechanisms are cumbersome but nonetheless allow a degree of interaction between languages.

The development of the CLI (see Section 2.2) however has provided a virtual machine that allows many languages to coexist and communicate uniformly, thus allowing a developer to easily use code written in multiple languages; taking advantages of the features offered in each. Ideally this platform would permit developers using strict languages to make use of the lazy evaluation features of languages like Haskell in order to lighten development efforts.

Our goal is to support non-strict evaluation on an OOVM so that systems for non-strict languages can target the OOVM. Furthermore we aim to do this in a way which allows non-strict values to be consumed by strict code with the minimal impact on those languages, if not transparently, so that those languages need never know the values they are consuming are being generated lazily by functional language code.

In the Section 6.1 we describe how non-strictness has been traditionally supported on conventional architectures. This is followed by an overview in Section 6.2 of current developments for non-strict support on OOVMs. We then discuss why the existing methods are not suitable for our purpose and introduce a development of them in Section 6.3. In Section 6.4 we then present a new algorithm developed from JIT Objects for supporting non-strictness for functional languages. Finally Section 6.5 will discuss the steps

necessary to support non-strict evaluation in a functional language system based on the techniques presented in Section 6.4.

6.1 Support for Non-Strictness on Conventional Architectures

As stated in Section 3.3, a number of languages currently support non-strictness at the language level, so automatic mechanisms to compile non-strictness have been developed. The standard approach used to provide non-strictness on real hardware is to create a *thunk*. Thunks were originally used to provide the *call-by-name* semantics of *Algol 60* (a precursor to modern non-strict evaluation), but in modern OO systems would be considered to be an implementation of the *Proxy Pattern*. The processes of delaying evaluation of an expression by placing it into a thunk is referred to as *suspension*, and the expression itself is said to be *suspended*.

A thunk consists of a reference to code to evaluate the suspended expression, and an environment containing data needed for that evaluation. Once the result is requested the code referenced by the function is executed and the resultant value returned. So called 'fully lazy' implementations optimise this and store the returned value in some way, removing the need to re-evaluate it should it be required again – this of course can only be done if the value computed by the suspension is immutable. Functional languages are usually implemented using the full laziness technique for efficiency.

The fully lazy method may introduce 'indirection' nodes into the system – once evaluated the thunk is marked and the result cached within it, subsequent references first find the thunk and then the by now computed result within it. Various techniques are used to remove this indirection once evaluation has taken place.

Provided some mechanism exists that can be used to reference code (such as a function pointer or virtual functions) the implementation of a thunk is a trivial matter. A number of techniques have been developed to allow the use of thunks to provide non-strictness [39], however most take the approach of allowing data references to refer to either a thunk or an actual value and then using flags to distinguish between thunk and value references. In type terms

a location of type T needs to be given the type such as $T \cup (void \rightarrow T)$. This is required as it cannot in general be known in advance whether a particular expression needs to be suspended, and so whether a given value will be a suspension or not.

Implementing this technique on real hardware is relatively simple, as the underlying memory model is untyped storing either a value or a thunk reference poses no difficulties. However once we move to an OOVM, such as the CLI, this picture changes dramatically. One of the key features of an OOVM is a typed memory model.

By using a typed memory model an OOVM can prevent unsafe operations from corrupting data, but in the process it must restrict what a program can do. In particular the type systems of current OOVMs do not support typing locations/references with a type such as $T \cup (void \rightarrow T)$, exactly the kind of type used in the implementation of non-strictness. In the next section we will overview existing work to overcome this restriction and why it does not fit algebraic types well. We then introduce our new algorithm which does handle algebraic types, and the full type system, of non-strict functional languages.

6.2 Non-Strictness on OOVMs: Current Developments

The simple model described in Section 5.1 provides full support for all data types in the Damas-Milner type system on an OOVM, however it does not provide any clear way to provide non-strict evaluation. Many methods have previously been described to extend (or replace) this model in order to provide support for non-strictness on an OOVM, however these frequently rely on either complete type erasure (replacement of all explicit types with generic *object* references) [54] or explicitly checking whether it is necessary to evaluate a value prior to use [40]. We have already said that our goal is to statically type all generated code, so type erasure is not an ideal solution. Explicit checks also directly contradict our goal of making non-strictness transparent to other languages.

Recently however the JIT Objects technique was developed that allows non-strict evaluation to occur transparently.

6.2.1 JIT Objects

The JIT Object [39] model was developed to enable any language to create and consume suspended instances of object types on OOVMs. The basic algorithm is to use a *proxy* object when a 'suspended' instance of an object is required. This proxy embodies the code and environment needed to create the real object when demanded. To fit into the OOVM type system as transparently as possible these *proxy* classes are generated as subtypes of the type for which non-strictness is desired. For example, consider a C# style *List* class:

```
class List<A>
{
    private A head;
    private List<A. tail;

    public A getHead() { return head; }
    public List<A> getTail() { return tail; }
}
```

This is a linked-list in OO style. A JIT Object for *List* is an automatically created subclass which embodies the computation needed generate a particular instance of *List*. As shown in Figure 6.1, the proxy class acts as a caching wrapper for a thunk (as described in Section 6.1).

A crucial benefit of the JIT Object approach is its transparency. As the automatically generated proxy object is a subclass of the type being suspended standard OOVM type testing and casting operations work as expected. For example, using C# the expression:

obj is List<int>

will evaluate to true for instances of both $List < int >$ and the proxy type $NSList < int >$. It is only *creator* of the object that knows it is suspended, the *consumers* of the object can be completely unaware.

The JIT Object model was demonstrated for the C# language, and such diversions as 'infinite' lists of primes easily produced.

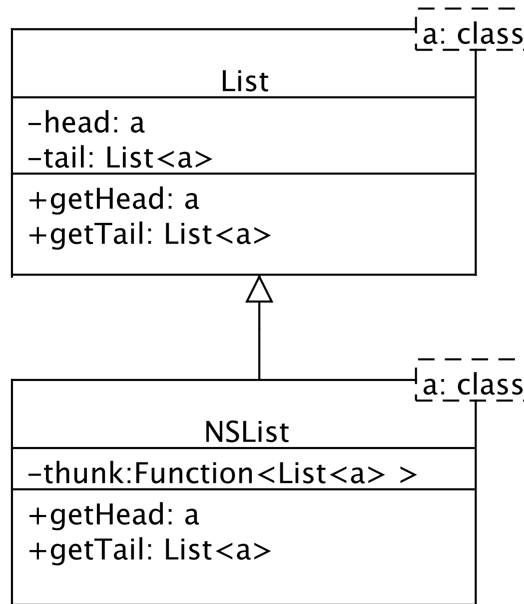

Figure 6.1: The JIT Object structure for the C# *List* type.

6.3 Algebraic types vs. JIT Objects

JIT Objects and algebraic types are somewhat of a mismatch. To demonstrate this we compare the C# *List* type in the previous section with that produced for the standard functional language algebraic *List* by the transformations described earlier:

```
// data List a = Cons a (List a) | Nil
abstract class List<A> {}
class Cons<A> : List<A> { ... }
class Nil<A> : List<A> { ... }
```

A JIT Object for C# can proxy the single *List* type, but in the above there are three types: *List*, *Cons* and *Nil*. A naive application of JIT Objects would allow proxies for each of these types, however in source language terms only values of type *List* are suspended, *Cons* and *Nil* are not types. The first observation is therefore that only JIT Objects for *List* should be generated. The proposed type structure is shown in Figure 6.2.

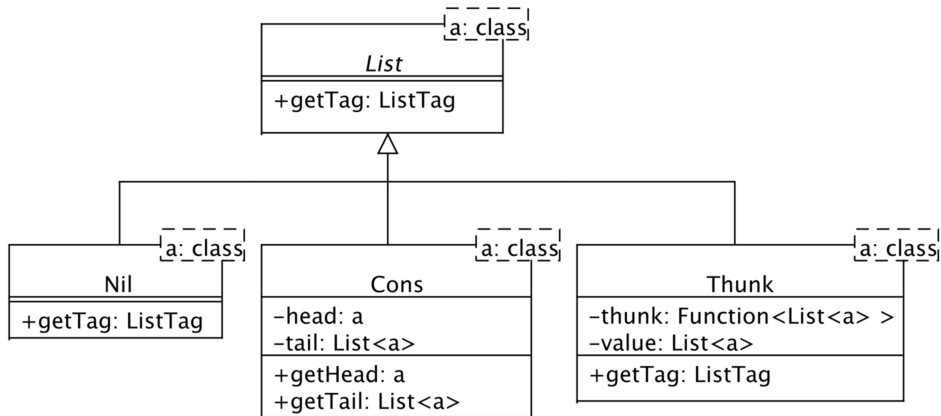

Figure 6.2: List structure with non-strict support – first attempt.

Consider the following simple Haskell function which filters a list by removing all elements for which the supplied predicate is true:

```
filter :: (a -> bool) -> [a] -> [a]
filter _ [] = []
filter p (h:t) | p x = h : (filter p t)
               | otherwise = filter p t
```

When translated onto an OOVM an application of *filter* may return an instance of *Cons* – for non-empty results – or an instance of *Nil* – for empty results. Which will occur is data dependent.

Consider suspending a call to *filter* and using JIT Objects as the implementation method. The type of a call to *filter* is *List*, so a JIT Object of *List* is created. A JIT Object provides all the methods of the object it is proxying, in this case we only have *getTag*.

Having created a suspension we demand the value. This will occur when compiled code examines the result of the *filter* call by calling *getTag* to determine if it is a *Cons* or *Nil*. The call to *getTag* will trigger evaluation of the JIT Object, an instance of either *Cons* or *Nil* will be created, and the resulting instances' *getTag* invoked to obtain the final result – all transparently behind the scenes by the JIT Object machinery.

Up until this point nothing has gone awry, everything behaves as expected. However, having now determined which of the constructors is present, the compiled code may need to access the constructor, and following the algebraic transformation model that will be done using an OOVM type cast. Unfortunately the JIT Object is a proxy of *List* and cannot be cast to either a *Cons* or *Nil*. Casting to *List* is not an option, though it is type correct to do so, as *List* does not provide the members of either *Cons* or *Nil* and the purpose of the cast is to access such members.

The difficulty has arisen due to the nature of JIT Objects – they work by proxying an object and in doing so must provide the same members as the object they proxy. What a JIT Object does not do is return the object it is proxying, and in the case of the algebraic transformation model that is what is required. Which suggests a solution to the difficulty, as explained in the next section.

6.4 Non-Strictness for Functional Languages

We adapt the algebraic transformation/JIT Object combination model to include methods on the root class which 'cast' an instance to one of the child types which represent the alternatives of the algebraic type. In conjunction with the existing *getTag* method we now have both type testing and casting operations as required. The new type structure is shown in Figure 6.3.

Unfortunately this change in approach reduces the transparency of JIT Objects when applied to algebraic types In the original strict transformation for algebraic types each constructor is a subclass of the root class; a reference with the static type of the root class will have a dynamic type of one of the constructor classes and standard OOVM type test and cast operations may be used to determine and access the dynamic type.

However with the introduction of non-strictness a dynamic instance representing a particular constructor may either be an instance of the constructors class or an instance of the thunk class. A standard OOVM type test will not identify the thunk class as anything other than the thunk class, and as the thunk class is a sibling of the constructor class an instance of it cannot be

cast to the constructor class. Therefore, unlike with the original JIT Objects, the machinery can be seen.

There is a positive benefit that balances this loss of transparency. Under the original JIT Object model the proxy may remain in place long after evaluation has taken place, depending on the particular implementation of JIT Objects and how deeply they are embedded into the OOVM. The casting operations of the algebraic JIT Objects return a direct reference to the evaluated object, thus removing the indirection through the proxy.

The added casting operations are provided by virtual methods defined on the root class of the algebraic type:

A default implementation in the root class throws an exception (to improve consistency with the rest of the CLI our experimental compiler throws *ClassCastExceptions* as would occur from an illegal cast). Each particular constructor subtype overrides the appropriate cast method and simply return *this*. The JIT Object proxy subtype, following the model, just proxies the cast operations.

This approach provides full support for non-strictness, however this is at the cost of some transparency as we have lost support for standard casts. This method is an improvement over earlier techniques where a developer would be required to manually trigger the computation of a thunk, whereas our mechanism allows a consistent approach to all non-strict types that does not rely on external knowledge of the underlying data or thunk.

6.4.1 Non Strictness for Boxed Primitives

While the solution given in Section 6.4 provides full support for non-strict evaluation, in cases where only one subtype is present, in particular for the primitive types, a more optimised approach is possible.

For the primitive types we use a general parameterised class to wrap them all, as shown in Figure 6.4. This effectively merges the thunk class and the constructor class into a single class. While this marginally increases memory usage when evaluation has not been suspended the burden is countered by

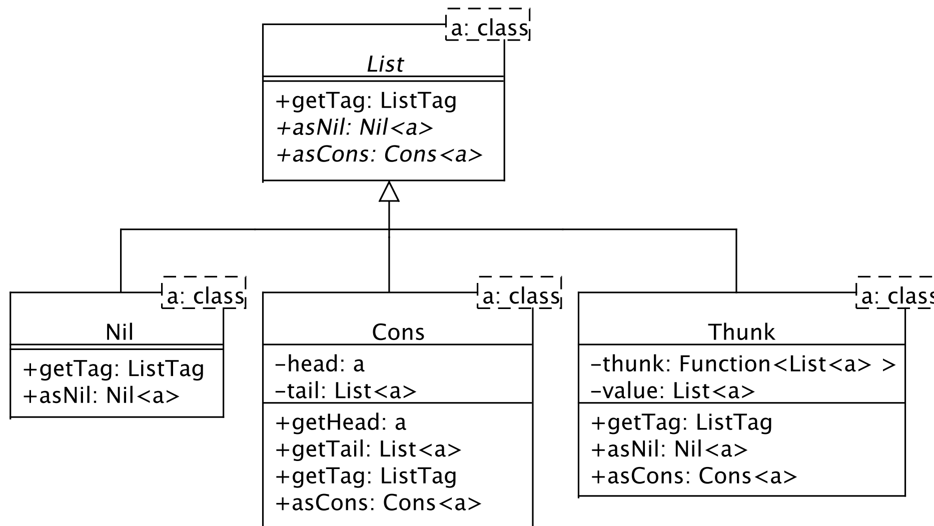

Figure 6.3: The final structure used to store the List type. Note the *asNil* and *asCons* methods.

the many advantages of this mechanism. One of the most significant is the improved performance attainable by removing virtual calls from accessing fields. There are a number of other possible performance and resource improvements that can be made but they are dependant on a large number of relatively minor design decisions.

A secondary advantage of this mechanism is that it aids interoperability of primitive types, as these are the most likely types to be explicitly manipulated by external languages (as described in Chapter 5 the data types of functional languages are much more readily manipulated by a functional language). This is done by including user-defined conversions (casts) between the non-strict boxed type and the OOVM native primitive types. While not a part of the type system directly (the compiler must replace casts with a function call itself) they are transparent to any developer using them.

6.4.2 Non-Strict Function Values

In Section 5.3 we discussed support for higher order types (or function values) on an OOVM. The methods discussed there support all the operations

```
class Boxed<ValueType>
{
    private ValueType value;
    private Function<ValueType> thunk = null;
    public Boxed(ValueType value)
    {
        this.value = value;
    }

    public Boxed(Function<ValueType> thunk)
    {
        this.value = default(ValueType);
        this.thunk = thunk;
    }

    public ValueType Value
    {
        get
        {
            if (thunk != null)
            {
                this.value = thunk.invoke();
                thunk = null;
            }
            return this.value;
        }
    }
}
```

Figure 6.4: C# code illustrating the basic *Boxed* type for allowing non-strictness for primitive types.

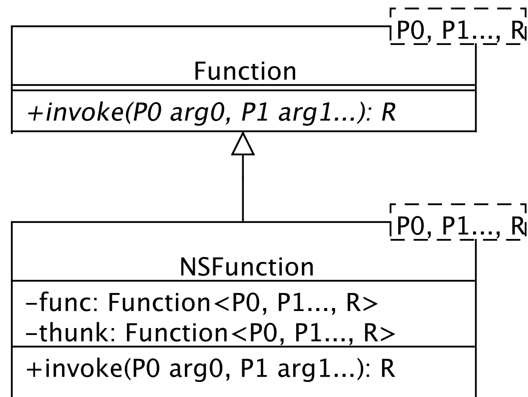

Figure 6.5: The class structure to allow non-strict evaluation of higher order types.

required for them to be used correctly, but do not provide any mechanism to allow function values themselves to be generated non-strictly.

While a function value is just represented by a class type, and so the above techniques for non-strict algebraic types can be applied, given the obvious need for all functions to be callable as efficiently as possible, we specialise the handling of non-strict function types. This handling is based on the observation that to evaluate a thunk some code must be called, and to apply a function some code must be called. Therefore a suspended function value can be represented in the same way as a non-suspended one, just the code reference needs to refer to code to evaluate the function value and then apply it rather than to just apply it.

The specialisation is performed through standard subclassing of the function types defined in Section 5.3 as shown in Figure 6.5. In this way we are able to streamline the invocation of all functions, whether the function reference represents a thunk or an actual value.

6.5 Performing Non-Strict Evaluation

Now that we have mechanisms in place to support non-strictness we need to generate code that will be non-strict. Existing compilers for non-strict

languages obviously already handle this process and the algorithms are well-known. The only change required is the relatively mechanical one of producing code for the OOVM platform with non-strictness rather than for a conventional architecture. We outline the process here.

There are three steps in this process. The first step is to determine which expressions should not be evaluated immediately, this is achieved through the use of a standard strictness analysis algorithm [42]. Following strictness analysis any expressions that should be delayed need to be converted into code that builds thunks rather than code that will evaluate. This step is performed through the simple mechanism of treating all expressions that should be suspended as nullary lambda expressions, and then lifting them in the manner described in Section 3.1. For example the function

```
foo f g x y = f (g x) (g y)
```

Will be transformed into the following set of functions after strictness analysis and lifting (in which '∧' indicates a suspension)

```
generated1 {g, x} = g x
generated2 {g, y} = g y
foo f g x y = f ∧generated1{g, x} ∧generated2{g, y}
```

Once the expressions to be suspended have been lifted, they will have been replaced by references to the functions generated during the lifting process and references to the free types and variables they require to evaluate correctly. To create the correct thunk all that is now required is to create an instance of the lifted function, supplying the appropriate environment parameters (as for any lambda expression). The lambda instance is then passed as the function for the target type's thunk constructor. So the final generated imperative code for the above example will be akin to (for the sake of simplicity we assume all x and y are $Ints$):

```
class generated1 : Function<Int>
{
  Function<Int, Int> g;
  Int x;
  generated1(Function<Int, Int> g, Int x) { ... }
```

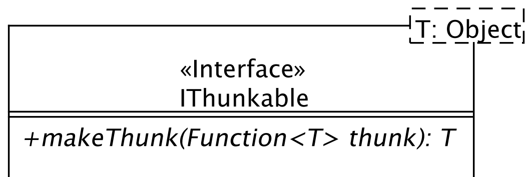

Figure 6.6: *IThunkable* interface used to allow non-strict construction of type parameters.

```
  Int invoke() { return g.invoke(x); }
}

class generated2 : Function<Int> { ... }

Int foo(Function<Int, Int, Int> f,
        Function<Int, Int> g, Int x, Int y)
{
  return f.invoke(new IntThunk(new generated1(g, x)),
                  new IntThunk(new generated2(g, y)));
}
```

The only element of non-strict evaluation remaining is the suspension of expressions returning type parameters. In this case it is not known the exact name of the thunk class that must be instantiated, as the type that will be substituted for the type parameter is unknown. To allow non-strict evaluation of parametric types we there use the OO standard approach of requiring the type parameter to implement a well known interface. The interface contains a single method which parallels the types constructor method (see Figure 6.6). This mechanism has a potentially significant performance impact as calling an interface method requires an instance of the type to be made, and so every thunk created may require both a virtual call and an additional object construction. There are a number of mechanisms that may be able to reduce this cost however they are heavily tied to specific implementation details, and therefore we will not discuss them.

Chapter VII

Higher Kinded Types

This chapter introduces the concept of higher kinded types. Section 7.1 describes what higher kinded types are and why they are useful. This will be followed by a description of the mechanisms we have developed to allow code using higher kinded types to be compiled to a virtual machine without native support in (Section 7.2). Finally, Section 7.3 will discuss the shortcomings of our approach.

7.1 Background

We first introduced higher kinded types in Section 4.1.4 as an extension to simple type parameters that allows type parameters to be *functions over types* [55].

In our work we have not attempted to support general higher-kinded type variables, but instead limit ourselves to provide support sufficient to support Haskell. In Haskell higher-kinded type variables are limited to being first order functions; this is exactly the same as type constructors in Haskell which cannot themselves take higher-kinded type arguments. We are not aware of any language in common use which supports more general higher-kinds so restricting ourselves to support Haskell is not a real limitation.

As we are limiting ourselves to supporting Haskell we will use examples from that language. Higher kinded type variables are introduced in two situations in Haskell: as parameters to functions; and as parameters to type classes. The primary use is in type classes, in particular in the standard `Monad` class.

7.1.1 Higher-kinded type variables in functions

The following is a use of a higher-kinded type variable in a Haskell function:

```
aId :: f a -> f a
aId x = x
```

The above defines the identity function, but only over parametric algebraic types of one argument. This is clearly rather trivial! The following is a function over unknown type constructor, f, that takes an input structure of type *f of Int* and returns an *f of Real*:

```
convert :: (f Int -> Int) -> (Real -> f Real) -> f Int -> f Real
convert g1 g2 x  = g2 (fromIntegral (g1 x))
```

In this case *convert* can operate over *List*, *Tree*, or any other type with a single type parameter; provided the required injection/extraction functions are available of course.

What should be clear is that higher-kinded type variables share the same restriction as standard type variables – a function has no knowledge of the type structure and so can perform only limited operations over the type. This means that using just functions it is difficult to implement complex functions using higher kinded types. Which brings us to higher-kinded type classes.

7.1.2 Higher-kinded type variables in type classes

Consider the function *map*, which can apply a function to each element in a list to produce a new list of transformed values

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = (f x):(map f xs)
```

Clearly a similar function could be defined which applies a function to each element of, say, a tree or indeed any 'container' type. Each particular function will know the structure of type it operates over, you cannot write such a function without that knowledge!

Type classes (described in Section 4.1.3) allow a type parameter to be constrained to types which provide a given set of functions. However you cannot

define a type class which specifies a function over constructed types, such as map above. Higher-kinded type classes address this.

For example, Haskell defines a `Functor` class that provides an *fmap* function that is intended to provide *map*like behaviour over many different types:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The key point about using a higher-kinded type class is that any instance will know the structure of what it is operating over. For example when defining an *instance* of *Functor* f will be bound to an appropriate type constructor, so the instance will be able to manipulate the data itself. For example:

```
instance Functor List where
  fmap = map
```

As with ordinary type classes, while each individual implementation of *fmap* is not able to work over every type, it can be used in a generic manner, as in these functions to scale a set of values:

```
scale :: [a] -> a -> [a]
scale elems factor = map (factor *) elems

fscale :: Functor f => f a -> a -> f a
fscale elems factor = fmap (factor *) elems
```

The first example shows an implementation of a function that scales each element in a list. The second function, *fscale*, uses a higher kinded type (the type parameter f) and *fmap* to define a function that can scale lists, trees, or any other type that has an associated instance of *Functor*. So we can see that by using higher kinded types (especially in conjunction with type classes) a function can be given a much greater level of abstraction, thus removing the developer's burden of repeatedly implementing almost identical code.

OO languages would typically use inclusion polymorphism to achieve the goal of higher kinded types, and for this reason do not have higher kinded types. For this reason OOVMs frequently do not provide support for higher-kinded

type parameters and thus we need to develop mechanisms that allow the semantics of higher kinded types to be supported.

7.2 Solution

In this section we present the system that we have developed to allow higher kinded types to be supported on an OOVM. Section 7.2.1 looks at type erasure and why it does not work in this case. Section 7.2.2 describes a mechanism that generates correct type information. Finally, Section 7.2.2 discusses what steps are necessary to allow higher kinds to be used safely within the context of classes and instances.

7.2.1 Type Erasure

We have previously discussed type erasure as the worst-case, but workable, solution to supporting type systems which cannot be mapped onto the OOVMs type system. Erasing all types is undesirable, however it is possible to erase the types of just higher-kinded type variables, an approach we shall refer to as 'partial type erasure'. However in the case of higher-kinds erasure has further limitations which make it unacceptable.

For example, a function $g :: \forall f \, . \, f \, Integer \rightarrow [f \, Integer]$, would become $g :: Object \rightarrow [Object]$, following type erasure. This function will not be type safe, regardless of what type is bound, for example, $g\,[1]$ should return a $[[Integer]]$, yet the erased result will be $[Object]$. For this to work on an OOVM its type system must allow $[Object]$ to be convertible to $[[Integer]]$, but this type of covariance is not supported by any of the current OOVMs as it is not type safe (notably both the CLI and the JVM allow this form of covariance for array types by using dynamic type checks to enforce type safety). Without this conversion the list of *Object* would need to be copied an element at a time, converting each one – a conversion which is valid and would succeed. Such copying would be unacceptable.

Attempts to erase even more type information result in similar problems.

So though type erasure might be made to function, the cost would be prohibitively high.

7.2.2 Explicit Instantiation Types

In this section we will introduce the mechanisms and algorithms that support higher kinded types with a significant level of static type safety. We will divide our discussion into separate sections which discuss the basic steps required for the transformation and the specialisations required for classes and instances.

Basic Algorithm

Section 7.2.1 demonstrated that it is not possible to erase to object, as the resultant type requires an invalid type conversion. Our algorithm solves this problem through the addition of "explicit instantiation types". The basic approach is: while the OOVM type system does not allow an open type; the OOVM equivalent of a type constructor; to be used as a type parameter it does allow a closed type; the equivalent of an applied type constructor. We therefore transform declarations using higher-kinded type variables so that instead of passing a higher-kinded type variable which is subsequently applied to a type (either constant or another type variable) we 'lift' out each such application and pass it as an additional type parameter. The type applications are then performed at the point of instantiation and only closed types are passed – so the instantiation is 'explicit', giving us our name.

The pseudo-code for the basic algorithm to perform these transformations is given in Figure 7.1.

Applying our algorithm to the earlier *fmap* example:

$$\forall\, f\, a\, b\, .\, (Functor\ f) => (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

results in the replacement type:

$$\forall\, f_a\, f_b\, a\, b\, .\, (Functor) => (a \rightarrow\ b) \rightarrow\ f_a \rightarrow\ f_b$$

```
makeExplicit definition
  expr = expression of definition
  sig = signature of definition
  typeparams = type parameters of defn

  foreach hk in
      (higher-kinded type variable applications in sig)
    tv = new type variable name
    sig = sig replacing hk with tv
    expr = expr replacing hk with tv
    typeparams = typeparams ∪ tv

  foreach hk in
      (higher-kinded type variable applications in expr)
    tv = new type variable name
    expr = expr replacing hk with tv
    typeparams = typeparams ∪ tv

  foreach hk in
      (higher-kinded type variable appearing alone in
       sig or expr)
    sig or expr = sig or expr with hk elided when alone

  return (typeparams, sig, expr)
```

Figure 7.1: Basic algorithm to convert from higher kinded types to *explicit instantiations*.

Note the eliding of f from $Functor$. This is required as applications $(f\,a)$ and $(f\,b)$ cannot be lifted to $Functor$ as a and b are only local to $fmap$. This causes the loss of static type checking and casts are inserted into $Functor$ instances to assert that a correct instance has been passed. This is covered below.

Type Classes and Instances

As discussed in Section 5.2 type classes and instances are processed differently from other types, the result of which is that higher kinds also need to be processed slightly differently.

In the standard model for converting a class (described in Section 5.2) the type parameter of the type class becomes a type parameter of the generated OOVM class. When the type parameter is a higher kinded type this isn't possible, and it must therefore be removed. The function signatures for each method in the type class are then restructured as described above (shown in Figure 7.2).

Higher-kinded *instances* however require slightly different handling, which mirrors the handling of other instances. Recall (4.1.3) that when defining a polymorphic function the details of the actual type which will applied in an application are not available and the definition of the function is type-agnostic. Type classes enable the types that may be supplied to be constrained to provide certain functions, and then the function definition may use those functions on instances of the type parameters. However, when an *instance* is created the function supplied is specific to the type the instance is being defined for and those specific types. In non-higher-kinded type classes the required type information is preserved as type parameters to the OOVM classes, and so the instances can operate on the *actual* types (5.2).

Due to our need to remove higher-kinded type variables the situation is slightly more complicated for higher-kinded type classes. For example, consider:

```
class Countable s where
  Count :: s a -> Integer
```

```
sum :: [a] -> Integer

instance Countable List where
  Count = sum
```

Applying the above algorithm for higher-kinded type removal to $Total$ produces the signature:

$$\forall\, s_a\, a.(Countable) => s_a \rightarrow Integer$$

In our example in the instance definition a will be substituted by $Integer$, just as for non-higher-kinded classes (5.2). However a cast is required to recover the type of s_a as $[a]$ before it can be passed to sum. The OOVM code produced for the instance must therefore contain the fragment:

```
... sum( (List<a> arg) ...
```

Depending on the instance being compiled this fragment is either included inline or a type casting wrapper function is produced which performs the any required casts on the arguments and return values, and this wrapper function is used as the function in the instance.

The algorithm for inserting the casts is a simple development of the one in Figure 7.1; when types are replaced casts are inserted. Figure 7.2 shows the result of applying this algorithm to the above *Functor* type class and the instance for *Functor List*.

7.3 Problems

The algorithm lacks the ability to produce code that can be statically verified by current OOVMs. Assuming the original source is itself type-safe, as it is in this situation, the generated code will be correct. However runtime type casts are required to meet the requirements of current OOVM type systems, and even though they will all succeed they do carry a cost. Despite this

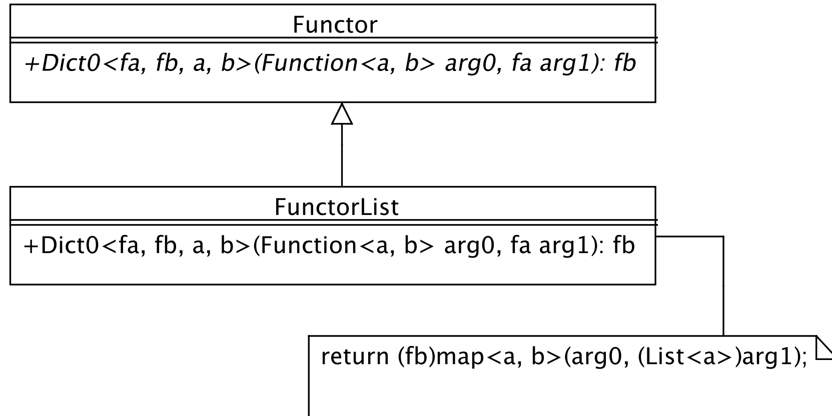

Figure 7.2: The generated structure the *Functor* class and *Functor List* instance.

shortcoming our approach is a significant improvement over complete type erasure, as it retains complete static type safety when higher kinds are not used. Unlike partial type erasure it produces completely correct code without resorting to copying.

The other disadvantage of the algorithm we have developed is that it has the potential to cause a combinatorial explosion in the number of type arguments required for otherwise simple methods. While this is not strictly a disadvantage from the point of view of the source language, it represents a significant difficulty when attempting to access a method with higher kinded type parameters from other languages. The issue is compounded further by having constraints that cannot be *statically* enforced, eg. where $\forall\, f\, a\,.\, a \rightarrow f\, a$ becomes $\forall\, f_a\, a\,.\, a \rightarrow f_a$ the connection between f, a, and f_a cannot be enforced.

Chapter VIII

Higher Ranked Types

In this chapter we will be discussing higher ranked types. An introduction to higher ranked types and their uses is given in Section 8.1. This introduction is followed by a discussion of the methods we have developed to support higher ranked types on an OOVM (Section 8.2). Finally Section 8.3 discusses the problems we have found in this approach.

8.1 What are higher ranked types?

As discussed in Section 4.1.4 *higher ranked* types are an addition to the higher order types provided by many functional languages that allow unbound polymorphic methods to be treated as values [55]. While this feature is not frequently required, if it is needed it is often not possible to do without [48]. To illustrate the purpose of higher ranked types we will use this trivial function:

```
f g = (g [0,1,2], g [True, False])
```

The function f attempts to apply g to a *List of Integers* and also to a *List of Bool*, however without higher ranked types g can only operate over a single type. If we wished to apply the function $reverse$ to each list g must have the type $\forall a\,.\,List\,a \rightarrow List\,a$, however traditional higher ordered types would not allow this. Instead a type would need to be bound to $reverse$ prior to it being passed to the function f, immediately removing any possibility of polymorphism. This is known as the *monomorphism restriction.* By allowing higher ranked types this problem is removed as it is not necessary to bind any types to the $reverse$ function, thus g can be bound locally.

As with higher kinded types this is not a feature that is prevalent in OO languages, and therefore it is not actively supported by OOVMs. For this

reason it has been necessary for us to develop a mechanism that supports the semantics of higher ranked types. It is this mechanism that we will be discussing in this section.

8.2 Solution

While the mechanism we have developed to support higher ranked types is conceptually simple, there are a a number of aspects that require special attention. For this reason Section 8.2.1 will first introduce the basic technique, followed by a discussion of the steps required to be able to safely use higher ranked types across multiple libraries (in Section 8.2.2).

8.2.1 Basic solution

In our earlier discussion of higher order types in Section 5.3 we introduced a family of parametric types to represent function values, $Function < A_1, ..., A_n, R >$ for the function types $A_1 \rightarrow ... \rightarrow A_n \rightarrow R$. Using these the Haskell function:

```
f :: a -> (a -> b) -> b
```

is transformed to the OOVM method:

```
B f<A, B>(A a1, Function<A, B> a2) { ... }
```

where $Function$ follows the template:

```
class Function<A1, ..., An, R>
{
  R invoke(A1 a1, ..., An an) { ... }
}
```

and 'class' may in practice be an interface or abstract class.

Our transformation is based on the observation that though an instance of an OOVM parametric type must be *closed* when passed as parameter, that type may contain parametric methods and these are *open*. Using this we may relocate the type parameters from $Function$ to $invoke$ to support higher-ranked types. For example, consider the Haskell function:

```
f :: a -> b -> (forall c. c -> c) -> (a, b)

f x y g = (g x, g y)
```

This can be transformed to the OOVM method (assuming the tuple type (a, b) is transformed to $Pair < A, B >$):

```
Pair<A, B> f<A, B>(A a1, B, a2, FunctionCC a3)
{
   ... a3.invoke<A>(a1) ...
}
```

where *FunctionCC* follows the template:

```
class FunctionCC { C invoke<C>(C a₁) ... }
```

Despite the relative simplicity of the concept, there are a number of complexities that must be handled. The first of these is that a parameter may contain both local and globally quantified type variables. For example consider the Haskell function:

```
f :: a -> b -> (forall c. c -> d) -> (d, d)

f x y g = (g x, g y)
```

Such cases combine higher-order and higher-rank and the obvious transformation is to combine the transforms for these two; globally quantified variables being placed on *Function* and locally quantified ones on *invoke*. Applying this transform to the above example produces:

```
Pair<D, D> f<A,B,D>(A a1, B, a2, FunctionC<D> a3) { ... }
```

and:

```
class FunctionC<D>
{
  D invoke<C>(C a1){ ... }
}
```

The next complexity should now be apparent, how should the *Function* types be named? For higher ordered types we predefined a number of generic *Function* classes (see Section 5.3). Such a simple solution is not possible in the case of higher ranked types, since an arbitrary combination of local and globally quantified type parameters could be present.

Our solution to this is to generate an interface or abstract class for each unique higher ranked type encountered while compiling a program. This allows a program to use any combination of higher ranked types, and does not rely on any infeasibly large collection of parameterised function types. While this approach does not require any specific naming convention for these thunk types, we have found that, for reasons discussed in Section 8.2.2, it is useful to produce names derived from the type itself.

8.2.2 Supporting Multiple Libraries

The ability to create independent libraries of functions that can be compiled and distributed separately from those systems that use them is a clear boon to the programmer. In this section we shall discuss the features required to allow the use of higher ranked types across multiple independently compiled libraries. First we discuss how to ensure type safety and consistency when passing higher ranked types from one library to another, then we discuss how to ensure that any higher ranked types returned from a separate library can be handled correctly and efficiently.

Higher Ranked Types as Parameters

As stated in Section 8.2.1, it is not possible to pre-generate generic higher ranked type classes in advance. We must therefore generate them at compile time. This means that each library that uses higher ranked types will have its own types to represent each higher ranked type that it uses. This creates a significant problem when higher ranked types are passed from one library to another. Since the CLI (and many other OOVMs) does not allow structural equivalence of types, the classes defined for higher ranked types in separately compiled libraries cannot be treated interchangeably. An instance of a higher

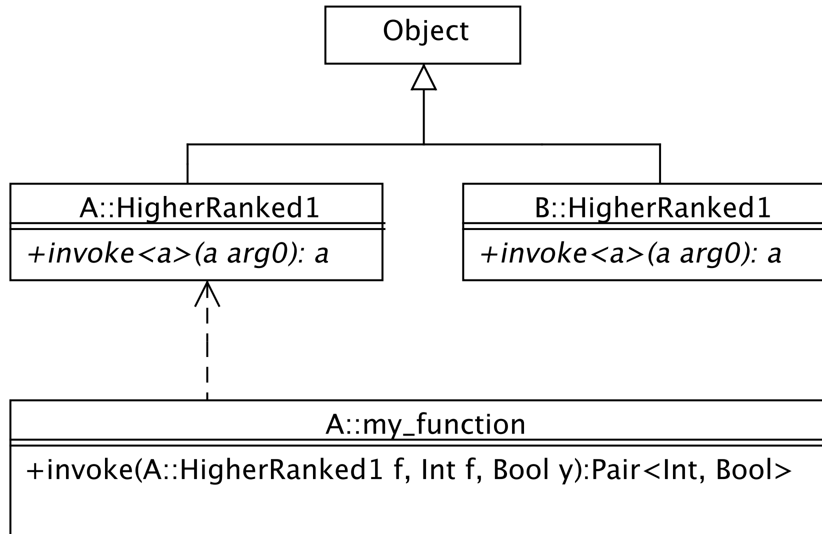

Figure 8.1: We can see in this example that the *HigherRanked1* class of module B is not substitutable for that of module A, so higher ranked types generated in module B will not be usable in module A.

ranked type can therefore not be passed trivially from one library as an argument to a function in another. An example of such a situation is given in Figure 8.1.

To address this for any type T introduced to represent a higher-ranked type that must be passed to an external library as type S we make T implement S; as shown in Figure 8.2. To make the process of creating and discovering the names of introduced types easier, we construct the names based on the module they are in and their usage.

As a higher-ranked parameter may be passed to functions from multiple libraries, the introduced type itself may have to subclass multiple types. This is why we cannot simply name T as S. Further given that most mainstream OOVM's, including the CLI, restrict multiple inheritance to interfaces, the introduced type must be an interface and not an abstract type. This requirement does place certain restrictions on where higher ranked types can be used, since an interface cannot force an implementing type to have a default constructor; a constraint required to allow non-strict evaluation in certain cases, as discussed in Section 6.5.

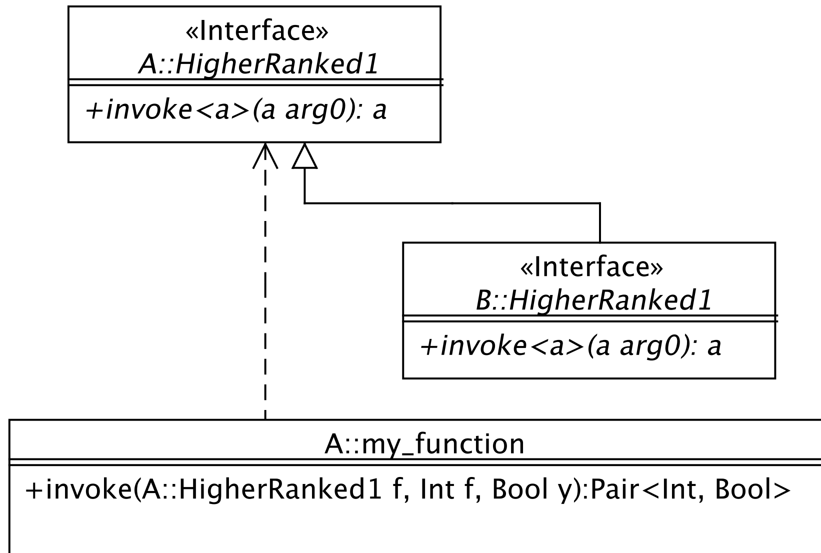

Figure 8.2: By making the higher ranked class for module B a subclass of the equivalent class in module A, higher ranked references from both modules can be passed to functions in module A.

Returning Higher Ranked Types

So far we have described techniques which allow us to pass higher ranked types from one library to another with complete, statically verifiable, type safety. However we must also handle the return of higher ranked types from functions in other libraries. This problem is more difficult to solve than that of passing higher ranked types as arguments. When generating a type, T, to represent a given higher-ranked parameter it is easy to determine all the uses of that parameter and hence all the types S_i that T must implement, as described above. However when the higher-ranked type occurs as the return type of a function it is not possible to determined all the places that function may be called, and hence the set of types that are used in all those calling locations to represent the higher ranked type. Therefore when transforming a call to a function which returns a higher rank type with introduced name T in a context where the higher rank type has been called S a conversion from T to S needs to be performed. This issues stems from OOVMs using name rather than structural equivalence for types.

The most simple solution to this problem is to introduce wrapper classes which perform the required type conversion to encapsulate any higher ranked type that is returned from another library.

For example, using C# on the CLI as our target platform, the following example shows such a wrapper class. First the definition for the locally introduced type to represent $\forall a, b.a \rightarrow b$ which has been given the name $GeneratedNameOne$:

```
interface GeneratedNameOne { B invoke<A, B>(A a1) { ... } }
```

Now assume we need to call the library method $Widget$ which returns a value of type $\forall a, b.a \rightarrow b$ which in this case has been given the name $AcmeGeneratedName$, in the library this will look something like:

```
interface AcmeGeneratedName { B invoke<A, B>(A a1) { ... } }

public AcmeGeneratedName Widget(...)
```

Back in our calling location we introduce a wrapper class to 'convert' a $AcmeGeneratedName$ to a $GeneratedNameOne$:

```
class AcmeWrapper : GeneratedNameOne
{
    private AcmeGeneratedName widgetResult;

    public AcmeWrapper(AcmeGeneratedName result)
    {
        widgetResult = result;
    }

    public B invoke<A, B>(A a1)
    {
        return widgetResult.invoke<A,B>(a1);
    }
}
```

Finally we wrap the call to $Widget$:

```
GeneratedNameOne ret = new AcmeWrapper( Widget(...) );
```

While simple, there are a number of issues with this approach, not least of which is its lack of elegance. A more fundamental concern is the potential for significant overhead, especially if a single higher ranked instance is passed repeatedly through multiple modules. In such a case there would be a significant overhead from creating each wrapper. Every wrapper place around a higher ranked typed requires another level of indirection, so that even if no further wrappers are applied, every subsequent call to the function will be more expensive than a standard call.

While we have been unable to develop a mechanism that will allow us to completely remove all uses of wrapper classes, we have developed an algorithm that allows us to safely use higher ranked types returned from other functions without necessarily requiring a wrapper. Our approach requires simple analysis of any uses of higher ranked types that are returned from other libraries. There are three cases to consider; namely where the returned higher ranked type is

returned from the current function: In this case we would ideally make the function return the appropriate type for the library to which the call was made. Unfortunately this would complicate any attempts to use the function from other modules, since it would effectively violate the encapsulation of the module. Such a violation could result in future changes to the implementation of the function changing its return type. Since this would require recompilation of all dependant libraries and programs the only sensible option is to build a wrapper around the returned value, thus ensuring that the return type remains consistent with the current library or program.

passed as a parameter to another function: The case is more flexible. By definition the type passed to a function must be either the type expected or a subtype of the type expected. In the event that the return value comes from the same library as the function it is passed to, or the called function is part of the code currently being compiled,

the delegate type will be correct and can be passed safely. In any other case more careful consideration is required to ensure that all generated code will be valid.

While it would technically be possible to examine the type of the returned value statically and determine whether it implements the appropriate interface (and would thus not require a wrapper), this would lead to the same violation of encapsulation mentioned above and is therefore not useable. This leaves two possibilities, either applying a wrapper class on every call, or using a dynamic type check to determine whether a wrapper is necessary at runtime. The first option is much simpler and results in completely statically verifiable code, but may wrap instances that would not need to be wrapped. The second option does not have that shortcoming, but is not statically verifiable and the performance impact of the dynamic check may outweigh any benefits of reducing the number of wrapped functions.

assigned to a local variable: This case is the most complex, since there are many ways in which the variable can be used; and it can be used multiple times. The first stage in the processing of this case is to find every use of the variable that has been assigned to and determine the expected type at each of these places. When determining the expected type of a use of a variable we must apply the rules described above. If the variable is ever returned as the result of the current function, it is expected to be the appropriate introduced type for the current library. When passed as a parameter to another library it is expected to be a subclass of the introduced type for that library. If every use of the variable expects the same type as the value originally assigned, the type given to the variable should match the introduced type of the source library. If the expected types include types other than that of the source, itself more care must be taken. The most trivial case is to declare the variable to be of the correct type for the current library and create a wrapper around the returned value. Needless to say, such an approach could be unnecessarily inefficient, especially if some of the uses were expecting the exact type of the returned value. Therefore

if any of the uses of the variable expect its actual type, a reference should be created to its original unwrapped value and the appropriate uses should be replace with references to the temporary store.

In this final case there are a significant number of tradeoffs to be made. Should the value be wrapped immediately or should it be wrapped at the site of each use? Should we use the dynamic checks mentioned above to see if a wrapper is even necessary? These decisions are difficult, since the most efficient choice may depend on how the function being compiled will be used, or how it is implemented. If the return value is wrapped too early, it is possible it may not be used, so time was wasted creating the wrapper. However if the wrapper is applied late (eg. wrapping at each use), then it may be unnecessarily wrapped multiple times. Moreover, these factors may be influenced externally by factors such as the way objects are created or dynamic type checks are made, either of which could influence the overall efficiency of the generated code.

8.3 Problems

While the techniques we have discussed in this section have resolved some of the difficulties of providing support for higher ranked types, of these a number of problems remain to be solved. The first of these is the potential for these techniques to generate a vast number of interfaces; one for each unique higher ranked type. While this is an unavoidable problem, every distinct type that exists has an overhead on many OOVMs, including the CLI, and excessive interface generation may therefore prove a burden on the target virtual machine.

The major shortcoming of this solution, however, is the high overhead from the wrapping of higher ranked values as they are passed from one library to another. This overhead manifests itself in three ways. The first is the obvious cost associated with instantiating the wrapper object and the second is the increased memory usage required for each wrapper instance. Thirdly the last of the wrapper related problems is the increased levels of indirection in

subsequent calls to a wrapped delegate.

Finally, there is the constraints problem referred to in Section 8.2.2. In order to allow higher ranked types to be used across multiple libraries the introduced types must be declared as interfaces. However, as interfaces they cannot provide default constructors. As stated in Section 6.5, the CLI offers no mechanisms to define virtual static functions, so a generated interface cannot provide a static function to create a *thunk*. Therefore creating a non-strict thunk for a parameterised type requires creation of an instance of that type, a feat that can only be accomplished through the use of the default constructor constraint. As higher ranked types are interfaces and cannot have a default constructor, they cannot be provided as type parameters to any function that would attempt to create a non-strict thunk. This is an unavoidable problem of our current solution.

All of these problems are caused by the lack of any form of structural equivalence in the CLI and similar OOVMs. If support for structural equivalence were provided by the VM, all of these problems could be resolved trivially, since it would no longer be necessary to use any form of wrapping to pass higher ranked types between different libraries. The removal of the need for wrapper instances would allow classes to be used in place of interfaces, thus allowing a default constructor to be present and allowing higher ranked types be be safely used as type parameters.

Chapter IX

Experimental Compiler

In this chapter we document the construction of a compiler for the Haskell functional programming language that targets the CLI. We have used this compiler as a platform to demonstrate that the techniques we have presented in this thesis work, and are efficient enough to be considered practical.

The first section in this chapter will discuss the design decisions made during during the development of this compiler. Section 9.3 follows with a description of the compiler architecture and descriptions of what each part of the compiler does. Section 9.4 provides a brief discussion of the performance of our compiler. Finally, Section 9.5 summarises this chapter.

9.1 Design Decisions

When developing the compiler a number of designs decisions were made for both practical and aesthetic reasons. In this section we explains what these decisions were and why they were made.

The first implementation-centric decision we made was to use another Haskell compiler as the basis for our own. The reasoning for this was twofold. First by using another compiler as the base we immediately gain all of the optimisations it implements for free. The second reason is that doing so removes the burden of implementing a complete type inference engine. As both of these features have been thoroughly covered in other papers [42, 45, 28, 65, 66] there would be little gain in implementing either feature ourselves.

This decision lead to the question *'what compiler do we use as our base?'*. While a number of Haskell compilers exist, the Glasgow Haskell Compiler

(GHC) stood out as having the best library support, a very powerful optimiser, and, importantly, it provides an external representation of it intermediate language; Core [62]. The external Core output from GHC is produced after GHC's type inference and optimisation stages, and therefore allows us to avoid implementing either of these phases.

Having decided on what the compiler would use as its initial code source, we needed to choose what the target language would be. As the compilers final target is to be the CLI, there are two options; the CLI intermediate language, CIL, or some other higher level language with its own compilers that target the CLI. By targeting CIL the compiler has much greater flexibility in what it can do, and it can have guaranteed access to all of the features the CLI provides. However, this comes at the cost of much greater complexity when generating code. Alternatively if the compiler targets a higher level language, code generation is much simpler and we gain any optimisations the higher level language compiler provides. Unfortunately this means that if the compiler for the target language does not support some feature of the CLI our compiler also does not have access to it.

Due to time constraints we chose to target a higher level language. In this case the C# language was the obvious choice as it is a defined standard and it supports every feature of the CLI that we require. There are a number of trade offs in this choice, most notably the CLI C# compiler does not currently generate the tail call [5] instructions supported by the CLI. This leads to potential overflowing of the program stack during deep recursion and we will discuss (in Section 9.3.3) one attempt to mitigate this problem.

9.2 Type Generation

Generation of all algebraic types, type classes, and instances is performed almost exactly as described in earlier chapters. We have used specific features of the CLI in an attempt to maximise possibility of VM optimisations occurring, and to aid interoperability with other languages on the CLI.

The most basic of these was restructuring the code generated for algebraic types. Rather than having the constructor and *thunk* subtypes contained

globally, they are defined as inner classes of the class defining the algebraic type. We can then *seal* the class preventing any other (unsafe) subtypes from being created. We also use the CLI's *reference constraint* on type parameters wherever possible as this has the potential to allow the VM to better handle many instances of a polymorphic method.

If a global function is ever used as a parameter (rather than being called) a wrapper class is created for it, however rather than creating a new instance of the wrapper for each use, a single static instance is defined as a member of the wrapper. This approach matches the *Singleton* pattern [13] and thus allows a single wrapper to be used for multiple references to the same function.

In addition to these minor optimisations a number of idiosyncrasies of the *Core* output of GHC result in partial application of class type constructors. This requires the addition of an explicit implementation of the type class constructor that in effect matches the structure of a standard algebraic type constructor.

9.3 Architecture

We have used a simple pipeline model for our experimental compiler, in the module being compiled is passed through a pipeline of operations. Each operation, or stage, of the pipeline performs a specific task, the result of which is then passed to the next stage. The following sections describe in detail what each task does, and, in more complex cases, how that goal is achieved.

9.3.1 Stage 1: Initial Processing

The first stage in the compilation pipeline is the parsing of the input Core files into useable syntax trees. This stage is completely standard and so we will not discuss it in any significant detail. Following the initial parse we encounter actual transform stages, the first of which are the *Core Name* and *Function Type* filters.

The purpose of the *Core Name* filter is to convert type variable names into names that are guaranteed to be safe, to partially decode the z-coded identifiers in native Core, and to rename certain standard identifiers to be more human friendly. Modifications to the type arguments are made to ensure that there are no name clashes between value and type variables, since the CLI cannot distinguish between identically named type and value variables. The remaining name changes merely ease human comprehension of output code.

The *Function Type* filter is a filter that converts implicit function types in Core to explicit ones. This is needed since certain versions of GHC producing function types as applications of types to a function type, *GHC.Prim.*→, rather than using Core's build in function type construct; eg. *GHC.Prim.*→ *Int (GHC.Prim.*→ *Int Int)* instead of *Int* → *Int* → *Int*. This filter simplifies later stages that would otherwise have to check for multiple type definitions with the same meaning.

After these filters have modified the Core syntax tree, it can be safely converted to a more useful structure that isn't as tightly coupled to the underlying Core syntax.

9.3.2 Stage 2: Converting to a Typed Structure

At this point the compiler has a version of the Core syntax tree of the current module that is safe to process. The next step is to load the root symbol information from this module – that is, the list of all function, type and constructor names listed in the module. This information is then inserted into a symbol table, along with links to the data required to fully evaluate the function as well as data types where a complete definition is needed (this is a case where language level non-strict evaluation would have been useful).

Once all the symbols in the module have been loaded, the next stage converts the Core body of a function into a useable typed structure. This produces a new tree, which is structurally equivalent to the original Core tree, but has been saturated with useable Type information (and the appropriate symbol table references). At this point the *Reference Function* filter converts alias

functions to actual function applications, eg. the function $foo = (+)$ becomes $foo\,arg0\,arg1 = (+)\,arg0\,arg1$. This new tree is still not all that different from the original Core syntax tree, except its structure is no longer confined to the Core syntax and it has recorded all function applications as being unchecked. We will discuss why this is important in the next section.

9.3.3 Stage 3: Transformations

The compiler has not really accomplished much at this point. It has a symbol table containing a whole lot of type information, and function bodies are now stored in a typed tree structure, but all of these are still equivalent to the original Core tree. Next the compiler starts actually modifying the functions bodies to prepare them for compilation. It does this by running a series of transformations over the body of every function in the module. These transformations are grouped into a number of phases, the first of these consists of a number of basic transformations that merely simplify certain structures in the tree. The second phase covers a number of tasks, focusing on the different aspects of function evaluation. Phase three follows with a series of filters to perform any required expression lifting. Finally, phase four performs target specific transformations.

A number of the transformations in this phase generate new functions. To ensure correct compilation these newly created functions are added to the queue of functions that this stage has to process, and thus are themselves eventually correctly transformed.

Phase 1: Initial Simplification

This phase consists of a number of simplification transforms. The underlying goal of these transforms is to remove superfluous information from the tree. This information may be a trivial local definition of the form $a = b$, or the calling of a constructor for a primitive type and a number of other artifacts of the Core semantics.

- The *Simplify Literals* transformation removes constructor calls from

any literal values. By doing this we avoid the need to generate wrappers for strict literal values, since literals can now be trivially emitted.

- The *Simplify Select* transformation simplifies the scrutinee (the value being examined) of the select statement, by lifting any scrutinee that is not a variable reference. By lifting any non-trivial scutinee out of the select statement we remove the burden of validating the scutinee from the code generator.

- The *Dictionary Select Removal* transform is the first transform that has a significant impact on the underlying expression structure. This transform identifies *select* expressions that are used to access the functions in dictionary types (see Section 4.1.3). Once one is found it replaces all subsequent references to the extracted fields with explicit *dictionary select* nodes. By doing this we have now explicitly marked calls to the functions from a type class.

- The *Simple Let Removal* transformation is a very basic process that removes trivial local definitions of the form $a = b$, replacing subsequent references to a with b. Its other task is to find any local definition that produces a variable that is used only once, and remove it by inlining the expression.

The final step of the this phase is the inlining of nullary functions. Unlike other transforms that operate on individual expressions the transform for inlining nullary transforms operates over every function in a module. This transforms finds all trivial nullary functions – functions that take no arguments – and inlines them wherever they are used. This reduces the impact of elements such as literal values being referenced through function calls.

Phase 2: Code Validation

Once the initial code tidying has been completed, the compiler moves on to code validation. In this phase a number of transforms process the tree

to ensure that correct Haskell semantics are enforced, and to ensure correct typing of expressions.

- *Partial Application Identification* is the first step of code validation. As we are using the *eval-apply* model of function evaluation it is necessary to ensure all function applications are correct prior to code generation. For this reason we use this transform to ensure that all function evaluations are correct. As noted in Section 9.3.2, all of the function applications currently in the expression tree have been recorded as being unchecked. By doing this we have allowed the transform to process the tree and validate or repair any function evaluations.

  Any given application may have either the correct number of parameters, too many, or too few. If the correct number of parameters are given, we merely mark the expression as being a ordinary function application. The case of too many parameters is processed by simply creating a new application with the correct number of parameters and performing an unchecked application of the remaining arguments to the result. The transform is then applied recursively to the resultant expression, thereby validating the new application. The single remaining case is a partial application. This is processed by using a lambda expression, as discussed in Section 3.2.

- Following the corrections of the function applications we use the *Arity Matching* transform to ensure that all function references are of the correct arity. This is necessitated by our use of the *eval-apply* model. If the arity of a function reference is incorrect, it is fixed through the use of a lambda expression, as described in Section 3.2.2.

- The *Suspension Filter* enforces the full non-strictness semantics of Haskell by processing the tree and wrapping all expressions that should be delayed inside special suspension nodes. By doing this we have produced all the information needed to correctly produce non-strict code.

Phase 3: Expression Lifting

This phase performs all the required lifting of lambda expressions and suspensions. At the end we are left with a tree in which all lambda expressions and artifacts of non-strict evaluation will be explicit.

- The *Lambda Lifter* exists merely to lift lambda expressions out of function bodies. To do this it uses the methods described in Section 3.1. When lifting lambda expressions with free variables we attach an explicit environment to the resultant function, as described in Section 5.3.

- The *Suspension Lifter* continues the enforcement of Haskell's non-strictness semantics by lifting suspended expressions into separate functions. This is achieved by wrapping the suspended expression inside a lambda expression, and then calling the Lambda Lifter to lift the generated lambda. Once that is complete, the resulting expression is marked as being non-strict. This means that the code generator is able to produce the correct thunk.

Phase 4: Target Specific Transformations

At this point we have performed a number of transformations to the original tree, we have removed some superfluous information, and have made what was once implicit information explicit. However even at this point the tree is still largely as it was before and could almost trivially target any non-strict functional language.

The next phase however ends any such ability, as it is here that we convert the tree from its current expression oriented structure to the statement oriented imperative structure we are targeting. Doing this requires only a few small transformations we will discuss below.

- The *Tagged Select* transformation operates on select statements performing a switch on type (Section 5.1). When such a structure is found, the transform searches for uses of any of the deconstructed fields from

a constructor of that alternative. For each used field a local definition with the appropriate name the new definition is assigned a special block which indicates that a field selection needs to be inserted. By taking this approach we have removed the need to copy every field (even those that are unnecessary) out of a data type whenever a switch on type is performed.

- The *Statement Transformer* is the final step in the transformation to imperative code. Since we are targeting an imperative model we need to modify the expression tree to use a statement based model. With only a few exceptions, such as conditional expressions, value returning constructs are limited to literals, variables, and function calls. This means that many constructs currently in the expression tree, such as local definitions and select statements, are not valid in value contexts; eg. as function parameters. To correct this, the Statement Transformer adds special blocks representing sequential actions and lifts any illegally placed expressions out of invalid contexts, and replaces them with the appropriate expression. Due to the insertion of sequential expression blocks, the expression tree is no longer expression oriented and has finally developed the structure suitable for an imperative virtual machine.

- The *Tail Call transformation* is an optional optimisation step, introduced to combat the problem of deep recursion. While the CLI does support tail calls natively, the C# compilers do not currently emit the required instructions. For this reason deeply recursive code that might work in Haskell natively may fail to execute correctly on the CLI. To reduce (but not remove) this problem, we have added the *Tail Call transform*. This transform identifies simple, direct self-recursion and replaces it with an iterative loop.

9.3.4 Stage 4: Final Compilation

We have now converted the original expression-oriented AST into an imperative statement oriented tree with all lambda expressions and partial evalu-

ations made completely explicit. At this point we have to actually generate the final code. To do this we generate a C# source file that can be passed to a standard C# compiler.

The first stage in the code generation is production of the data types, for which there are a number of different facets. For each new data type and type class declared in the module we are compiling, we generate new classes using exactly the form described in Sections 5.1 and 5.2. There are a few minor implementation specific details (such as the use of the CLI's *sealed* classes) to ensure that the structures for these cannot be incorrectly subclassed or otherwise cause harm to the integrity of the stored data. The only variation is the removal of non-strict thunk types for any strict algebraic types.

Data type creation is followed by the generation of executable code. This comes in two forms: functions and instances. Generation of function bodies is a trivial conversion of the statement-oriented expression tree to C#. An exception to this are functions with attached environments. These are wrapped inside Function objects, as described in Section 5.3. In order to improve efficiency of the output code we inline certain functions; such as the functions for performing arithmetic on primitive types.

When generating the code for functions there is the possibility a function will have no parameters, and will therefore always return the same value. To produce a completely lazy executable it would be necessary to cache the result of these functions upon their first use, thereby preventing a potentially complex computation from being performed multiple times. Retaining these values without creating space leaks has been an area of prior research by others. Building on this we have utilised the CLI's weak pointers for our implementation.

Once code generation is complete the Haskell symbol table is emitted as an attribute, allowing the compiled code to be referenced from other modules easily. At this point the generated code is finally compiled by the standard CLI C# compiler. If the module being compiled contains a Main module and *main* function, the output is a executable for the CLI, otherwise a library is produced (though the compiler can be forced to generate an library regardless of whether it encounters a *main* method).

9.4 Performance

As the primary purpose of this compiler has been to test the algorithms and techniques developed earlier we have performed only a rudimentary study of the performance, our main metric during development has been 'wall clock' time – does the compiler and compiled code perform acceptably, which it does – and not absolute performance.

Using a VM normally has some performance impact, but the extent of that impact varies by VM and actual computation so merely comparing the performance of our compiler to a native one would not necessarily produce meaningful results.

In addition our compiler is built using one architected to target conventional machines, and as commented elsewhere there are cases where its design would not be the one chosen if the compiler has been built from scratch – something outside of the scope of this project and also to a large extent would involve applying standard compiler engineering.

Building performance test suites which isolated just the parts of the language on which we worked proved difficult. It became obvious to us that any detailed analysis would require a substantial amount of work and was unfortunately something we did not have the time to do.

However we have performed a few simple tests of performance which show our compiler is approximately half the speed of the native implementation for simple programs, such as the following factorial function:

```
factorial :: Int -> Int
factorial n = if n <= 0 then 1 else n * (factorial (n-1))

main = putStr (show (factorial 20) )
```

and a similar slowdown with the following fibonacci function which involves the lazy construction and circular referencing:

```
fibonacci :: [Int]
fibonacci = 1:1:zipWith (+) fibonacci (tail fibonacci)
```

```
main = putStr (show (take 1000 fibonacci) )
```

As we increased the size of the list evaluated by the fibonacci function the relative performance stayed relatively constant, this seems to imply that the initial cost associated with VM startup has no significant impact on overall performance. However we have no relative metrics for the native GHC memory allocator/garbage collector (GC) and the one provided by the VM, so it is possible that GC is contributing to the slowdown, although this is unlikely to make up for a 50% slowdowm, which is worse than we had hoped for.

9.5 Summary

In this chapter we have described the development of a compiler that supports all the features of *Haskell 98*, as well as supporting the *higher ranked type* extension offered by GHC. This has acted as a validation of the functionality of many of the algorithms and techniques that we have developed. Our compiler is able to compile very large and complex libraries (namely the GHC base libraries) to native CLI libraries, and is able to subsequently use those libraries in separately compiled programs.

Unfortunately due to time constraints we have been unable to completely implement support for higher kinded types, and therefore are unable to demonstrate complete functionality of the associated algorithms within our compiler. The approach we have taken is effectively that partial type erasure (as discussed in Section 7.2.1). While this approach does not work in all circumstances, it provides enough functionality to give basic use of the Haskell *Monad* and *Functor* classes.

While we expected the performance degradation when moving *Haskell* onto the CLI to be more significant than that experienced by imperative languages, the difference was greater than we anticipated. There are a number of possible causes for this

- While GHC performs a significant number of optimisations, some of them produce structures and expressions that are not very amenable

to the techniques we have developed. The most damaging of these is the partial application of a type class constructor; time constraints prevented us from developing a suitable technique to handle this, and so we have fallen back on the less efficient method of providing a constructor class for type classes. This allows instances to be created dynamically by wrapping a set of higher order types.

- The *Core* output from GHC does not directly support type classes or instances and instead emits them following the transform described in Section 5.2. In order to emit instances in the way described in Section 5.2 our compiler must effectively reverse this transform. In most cases this is trivial, however complex recursive instance definitions can produce complex code that our compiler can have difficulty reconstructing from. In some cases the compiler may not be able to reconstruct the instance as efficiently as possible, leading to poor performance of the generated code.

- Finally, our comparison is against a commercial grade compiler, and we do not have the same time and resources to expend on our compiler. As a result it is quite possible that significant further optimisations could be made to our compiler that would reduce the performance gap.

A number of steps could be taken to improve this situation, obviously the final point could be remedied with further time and resources, however the first two points are less trivial. Both are related to our use of GHC as a front end. By using GHC we removed the burden of implementing the type inference and optimisation stages of a Haskell compiler, however a number of the transforms and optimisations are obviously not suitable for an OOVM. If instead a compiler were built with the primary intent of targeting an OOVM it may be able to make better decisions when transforming and optimising code.

Chapter X

Future Work and Conclusion

10.1 *Future Work*

While our thesis has introduced a number of algorithms and techniques that allow non-strict evaluation, higher kinded and higher ranked types on an OOVM we have found a number of areas for further research.

10.1.1 Performance Improvements

As this thesis has focused primarily on what is needed in order to allow non-strict functional languages to operate on statically typed virtual machines, we have primarily examined data structures. Another field of considerable importance is the performance of the resultant code. Section 9.4 showed that in our experimental compiler there was a significant performance penalty when converting from native application execution to execution on a virtual machine. While some performance degradation is expected, the results in our tests demonstrate that the degradation is much greater than that for imperative languages. Pinpointing the reason for this discrepancy has proved difficult, since it is likely due to multiple causes.

These performance problems provide an ample supply of further research problems, ranging from optimising the structures used for non-strict evaluation (although such optimisations would likely be platform specific), to improved handling of function calls and values. One of the more immediate problems, tail calls, has already been commented on in Section 9.3.4. Tail calls are frequently used by functional languages to improve performance, and also for limiting stack overflows. However, initial tests demonstrate that

on the CLI tail calls reduce performance and currently the high level compiler we use does not emit them. Although it may be possible to emit tail calls by changing the final code generation phase a careful study of the tradeoffs of such a step would be prudent.

10.1.2 Virtual Machine Level Support

If it is found that there is no mechanism to achieve high levels of performance on current OOVMs it may prove necessary to modify the VM itself. While supporting strict algebraic types is trivial on an OOVM, non-strict types have proved to be a different story, requiring a number of steps to allow only semi-transparent non-strictness. The addition of VM support for non-strictness could resolve this problem through incorporating the mechanisms we described in Chapter 6 into the VM's type system. Such a modification would render non-strictness completely transparent to other languages.

Higher kinded types and higher ranked types would both benefit from the addition of VM support as well. VM support for higher kinded types would improve our current solution by allowing code with higher kinds to be statically verified. Higher ranked types would potentially receive a significant performance improvement through no longer requiring extensive wrapping, they would also no longer require the generation of large numbers of interfaces (as described in Section 8.2.1).

10.1.3 Integration of External Functions into a Functional Language

The techniques we have discussed allow code written in a functional language to be executed on an object oriented virtual machine, and also allow it to be called from other languages. However, we have not worked on a mechanism, such as GHC's foreign function interface, to provide functional language level calling of external functions. Such an interface between languages with dissimilar semantics is a research project in its own right.

However, given that our design follows the calling conventions of the host OOVM, calling code written in other languages can be done; requiring only that the signatures be described in the appropriate manner to the Haskell

compiler. In this situation though no guarantees are provided that the correct semantics of the functional language code will be maintained, this must be checked by manual means.

10.2 Conclusions

We have developed a number of techniques that provide support for non-strict evaluation, higher kinded types, and higher ranked types on the CLI. These techniques provide support for partial evaluation of functions, non-strict evaluation and higher order, higher ranked and higher kinded types.

The techniques to convert and integrate the algebraic types of a functional language (and the *type class* feature of the *Haskell* language) with that of the CLI were discussed in Chapter 5. These techniques allow transparent interaction with algebraic types from other languages the targeting the CLI. This is achieved through the use of specially arranged inheritance to represent the disjoint union structure of algebraic types.

Chapter 6 describes the mechanisms we have developed to extend this basic model to support the non-strict evaluation semantics of non-strict languages like *Haskell*. Unfortunately the techniques we have developed are not completely transparent, however they are a substantial improvement over previous techniques.

We describe the algorithm used to process higher kinded types in Chapter 7. While the algorithm we have developed provides complete support for higher kinded types, we have not been able to provide support for full static typing. This results in a potential performance impact through the requirement for runtime type checking. However, unlike other approaches static type information is retained for all non-higher kinded types in all generated code.

Chapter 8 introduced a series of algorithms to support higher ranked types on an OOVM. While the basic idea described was an obvious extension to existing programming practises it has proved necessary to develop numerous techniques to allow such higher ranked types to be used in multiple libraries.

The algorithms and techniques have been used to develop an experimental compiler for the full *Haskell 98* language specification, including the GHC

extension of higher ranked types (see Chapter 9). While time constraints prevented us from being able to completely validate our support for higher kinded types, the compiler demonstrates that our techniques can be used to support an existing language on an OOVM with no modification to the source language.

In conclusion the techniques we have developed allow languages using non-strict evaluation, higher kinded types, or higher ranked types to be compiled to OOVMs such as the CLI without modification to the original language or the VM. Unlike other attempts to support these features our techniques do not rely on type erasure, and use dynamic typing only in specific cases, thus improving the ability of the generated code to safely interact with other languages on the VM.

References

[1] J. Armstrong. The development of erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM Press.

[2] H. J. Boehm. Space efficient conservative garbage collection. *SIGPLAN Not.*, 39(4):490–501, 2004.

[3] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143, New York, NY, USA, 1980. ACM Press.

[4] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Markow, E. Meijer, S. Panne, S. P. Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report. Technical report, 2002.

[5] W. D. Clinger. Proper tail recursion and space efficiency. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 174–185, New York, NY, USA, 1998. ACM Press.

[6] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[7] H. Evans and P. Dickman. Garbage collection and memory management. pages 138–143, 1997.

[8] J. Fairbairn and S. Wray. TIM: a simple, lazy abstract machine to execute supercombinators. In *Proc. of a conference on Functional pro-*

gramming languages and computer architecture, pages 34–45, London, UK, 1987. Springer-Verlag.

[9] S. Finne. Hugs98 for .NET. `http://galois.com/~sof/hugs98.net/`, 2002.

[10] S. Finne, D. Leijen, E. Meijer, and S. L. P. Jones. Calling hell from heaven and heaven from hell. In *International Conference on Functional Programming*, pages 114–125, 1999.

[11] M. Fowler. *UML Distilled, 3rd Edition*. Addison-Wesley, 2004.

[12] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In Michaelson and Milner, editors, *Automata, Languages and Programming*. Edinburgh University Press, 1976.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. GAM e 95:1 1.Ex.

[14] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, 1997.

[15] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. *ACM SIGPLAN Notices*, 36(3):248–260, 2001.

[16] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. Edinburgh LCF – A mechanised logic of computation. *LNCS*, 78, 1979.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[18] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[19] A. C. Hearn. Standard lisp (reprint). *SIGSAM Bull.*, (13):28–49, 1969.

[20] M. C. Henson. *Elements of Functional Languages*. Blackwell Scientific Publications, 1987.

[21] R. Hertzman. Lazy imperative languages - report on a project to examine the use of lazy evaluation in imperative languages. `http://citeseer.ist.psu.edu/hertzman95lazy.html`, 1995.

[22] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[23] Jim Hugunin. Ironpython: A fast python implementation for .net and mono. *PyCON*, 24 March 2004.

[24] D. H. H. Ingalls. The smalltalk-76 programming system design and implementation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 9–16, New York, NY, USA, 1978. ACM Press.

[25] ECMA International. *ECMA-367: Eiffel Analysis, Design and Programming Language*. 2005.

[26] ECMA International. *ECMA-334: C# Language Specification*. 2006.

[27] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2001. ACM Press.

[28] A. Kuhnemann. Benefits of tree transducers for optimizing functional programs, 1998.

[29] J. Launchbury. Lazy imperative programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92*, pages 46–56, 1993.

[30] S. Marlow and S. L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 4–15, New York, NY, USA, 2004. ACM Press.

[31] J. McCarthy. History of lisp. In *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*, pages 217–223, New York, NY, USA, 1978. ACM Press.

[32] E. Meijer and K. Claessen. The Design and Implementation of Mondrian. In *Haskell Workshop*. ACM, June 1997.

[33] E. Meijer and S. Finne. Lambada: Haskell as a better java. *Electronic Notes in Theoretical Computer Science*, 41, 2001.

[34] E. Meijer, N. Perry, and A. van IJzendoorn. Scripting .NET using Mondrian. *Lecture Notes in Computer Science*, 2072:150–??, 2001.

[35] The Mono Project. `http://www.go-mono.org`.

[36] M. Moskal, P. Olszta, and K. Skalski. Nemerle: Introduction to a Functional .NET Language. `http://nemerle.org/intro.pdf`.

[37] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[38] A. T. H. Pang and M. M. T. Chakravarty. Interfacing haskell with object-oriented languages. In P. Trinder and G. Michaelson, editors, *Implementation of Functional Languages: 15th International Workshop*, LNCS, pages 20–36. Springer-Verlag, October 2003.

[39] N. Perry. Implementing non-strict evaluation on OOVMs. *IEE Proceedings on Software*, 152(6):309–315, 2005.

[40] N. Perry and E. Meijer. Implementing functional languages on object-oriented virtual machines. *IEE Proceedings on Software*, 151(1):1–9, 2004.

[41] N. Perry and K. M. Sephton. The Hope$^+$C Compilation System. In *Proceedings of the 1989 Bejing International Symposium for Young Computer Professionals*, Bejing, 1989.

[42] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[43] S. L. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *Engineering theories of software construction, Marktoberdorf Summer School 2000*, pages 47–96, 2000.

[44] S. L. Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*, volume 13. Cambridge University Press, 2003.

[45] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. Partain, and P. Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993.

[46] S. L. Peyton Jones and D. R. Lester. *Implementing Functional Languages: a tutorial.* 22 January 1992.

[47] S. L. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM components in haskell. In *Fifth International Conference on Software Reuse*, Victoria, British Columbia, 1998.

[48] S. L. Peyton Jones and M. Shields. Practical type inference for arbitrary-rank types, March 2004. Under consideration for publication in J. Functional Programming.

[49] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993.

[50] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[51] The Python Programming Language. `http://www.python.org`.

[52] E. Rose. Towards secure bytecode verification on a Java Card. Master's thesis, 1998.

[53] The Ruby Programming Language. `http://www.ruby-lang.org`.

[54] T. Shackell. Yhc: The York Haskell Compiler. `http://www-users.cs.york.ac.uk/ndm/yhc/`, 9 February 2006.

[55] C. Shan. Sexy types in action. *SIGPLAN Not.*, 39(5):15–22, 2004.

[56] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[57] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly & Associates, Inc, 2003.

[58] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[59] D. Syme. The F# Programming Language. `http://research.microsoft.com/projects/ilx/fsharp.aspx`.

[60] D. Syme. Proving java type soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, 1999.

[61] D. Syme. ILX: Extending the .NET common IL for functional language interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[62] A. Tolmach. An external representation for the GHC core language. http://www.haskell.org/ghc/docs/papers/core.ps.gz.

[63] D. A. Turner. Recursion equations as a programming language. In John Darlington, Peter Henderson, and David Turner, editors, *Functional Programming and its Applications*, pages 1–28. Cambridge University Press, January 1981.

[64] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[65] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.

[66] S. C. Wray and J. Fairbairn. Non-strict languages — programming and implementation. *The Computer Journal*, 32(2):142–151, 1989.