THE UNIVERSITY OF NOTTINGHAM

DOCTORAL THESIS

The Modular Compilation of Effects

Author:
Laurence E. DAY

Supervisor:
Graham M. HUTTON

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

in the

Functional Programming Laboratory
School of Computer Science

October 2015

THE UNIVERSITY OF NOTTINGHAM

Abstract

The Modular Compilation of Effects

by Laurence E. Day

The introduction of new features to a programming language often requires that its compiler goes to the effort of ensuring they are introduced in a manner that does not interfere with the existing code base. Engineers frequently find themselves changing code that has already been designed, implemented and (ideally) proved correct, which is bad practice from a software engineering point of view.

This thesis addresses the issue of constructing a compiler for a source language that is modular in the computational features that it supports. Utilising a minimal language that allows us to demonstrate the underlying techniques, we go on to introduce a significant range of effectful features in a modular manner, showing that their syntax can be compiled independently, and that source languages containing multiple features can be compiled by making use of a fold.

In the event that new features necessitate changes in the underlying representation of either the source language or that of the compiler, we show that our framework is capable of incorporating these changes with minimal disruption. Finally, we show how the framework we have developed can be used to define both modular evaluators and modular virtual machines.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Graham Hutton, for both agreeing to supervise me in the first place and then putting up with me for the subsequent four years. I am not a model student, and his patience and shared knowledge has made me both a better scientist and a better person. Thank you, Graham.

Secondly, I would like to thank the members of the Functional Programming Laboratory – in particular those in Room A04 – for making it a particularly exciting and entertaining place in which to study. My particular thanks go to Thorsten Altenkirch, Henrik Nilsson, Venanzio Capretta, Neil Sculthorpe, Joey Capper, Jennifer Hackett, Ambrus Kaposi, Bas van Gijzel and Nicolai Kraus. I'm sorry that I will no longer be around to distract you, and wish you every success.

I had the pleasure of meeting Patrick Bahr whilst I was in the first year of my PhD, and the good fortune to have him join the laboratory in Nottingham on a research grant for six months in 2013. Patrick, you probably taught me more about how Haskell works than any other source, and as a co-author you're a genuine godsend. My doctorate is largely indebted to you, and I owe you drinks from now until the end of time.

Outside of the laboratory, there are a number of people – both academic colleagues and friends – who have been significant players in the time that has lead up to the conclusion of my doctorate. In particular, I would like to thank Michael Hoy, Joe Nash, Paula Besson, Alex Pinkney, and Michael Gale. You're all life-savers.

It is appropriate for me to pay homage to the person who first set me on my path as a scientist. To this end, my high school mathematics teacher, Karen Hulme, is in no small part responsible for my decision to both study mathematics at University and undertake graduate study. May you continue to inspire the South African youth, Karen.

Of course, my parents, Edward and Noeleen Day, are the ones who raised me and educated me such that I was in a position to undertake tertiary study in the first place. I appreciate everything you have done for me more than you can know, Mum and Dad, and I promise I'll get a real job now.

Finally, I would like to thank my fiancee, Dr Nichola Eales, for her unwavering support throughout the entirety of my time at Nottingham. More than anyone else, you are the one most responsible for my successes, both as an academic and otherwise. I love you.

Contents

For Nichola.

Chapter 1

Introduction

For as long as programs have been written, significant effort has gone into the process of making it easier to do. Whilst the first programs were written in machine code and executed directly by a processor, the evolution of programming languages has both significantly increased the productivity of programmers and the readability of their programs. One consequence of this is the existence of an ever-widening semantic gap between the language that a program is written in and its realisation in machine code. The tools used in order to close this gap represent an important branch of software engineering. We call such tools *compilers*, and the topics involved in their development represent a microcosm of modern computer science.

1.1 An Aside: What Are Compilers?

Put simply, compilers translate programming languages. The general case is that the source language – the language we compile *from* – is at a higher level of abstraction than the target language we compile *into* (although *decompilers* – which perform the reverse operation of 'abstracting' low-level code into a more human-readable form – are also common). With this said, compilers that translate *between* high-level source languages do exist, and are known as source-to-source compilers, or transpilers for short.

Whilst the history of compilation theory and implementation is a substantial topic on its own, credit is generally given to Grace Hopper for creating the first compiler in 1952 whilst working on the UNIVAC project at Remington Rand [Hop87]. However, the FORTRAN compiler of IBM – released five years later by Backus et al [BBB+57] – is considered the first compiler designed for a standalone source language. The motivation for these projects was more pragmatic than anything else - machine code was tedious to write by hand, and had to be rewritten for each new architecture.

Whilst the introduction of the first generation of compilers undoubtedly spared software writers from significant effort by way of code maintenance, the code that they produced was often slower than manually-written, hand-optimised machine code. To this end, the work of Frances Allen and John Cocke [All70, AC76, AC71] introduces some of the first optimising code transformations such as common subexpression elimination and operator strength reduction. Today, modern compilers can produce better quality code than expert human programmers.

1.2 An Aside: How Are Compilers Constructed?

A typical compiler consists of multiple distinct phases. Whilst individual compilers can vary greatly in their internal workings, the overarching themes of these phases – and their relationships – are given below:

$$\begin{tikzcd}
\texttt{Lexical Analysis} \arrow[d] \\
\texttt{Syntax Analysis} \arrow[d] \\
\texttt{Semantic Analysis} \arrow[d] \\
\texttt{Intermediate Code Generation} \arrow[d] \\
\texttt{Code Optimisation} \arrow[d] \\
\texttt{Target Code Generation}
\end{tikzcd}$$

It is not the purpose of this section to explain the complexities involved in the manipulations and transformations that comprise an industrial strength compiler. For this, a number of seminal, comprehensive works are available [ASU86, App97, PJ87]. However, we briefly sketch out the primary purpose of each phase below:

- **Lexical Analysis**: the *lexer* of a compiler steps through the textual representation of a source program and produces a sequence of tokens, or *lexemes*, which comprise the individual syntactic components of a program. For example, for C-like languages the lexer produces a token corresponding to a separator when it encounters a plaintext semicolon, and distinguishes between the reserved words of the language and other strings used as variable identifiers when it encounters alphabetic strings.
- **Syntax Analysis**: the *parser* of a compiler works in conjunction with the lexer to produce the *abstract syntax tree* of the source program.
- **Semantic Analysis**: the *typechecker* performs static checks on operands and variables within the program to avoid errors such as trying to add a string to an integer. This phase also performs any syntactic checks that do not fall within the remit of a parser, such as ensuring that any occurrences of `break` occur within conditional blocks.
- **Intermediate Code Generation**: generates an intermediate representation (IR) of the source program that is suitable for further

manipulation. The exact structure of an IR varies depending on the language in question, but in general IRs need to be rich enough to successfully capture the semantics of the source language whilst simultaneously being close enough to the target language so as to simplify the final code generation phase.

- **Code Optimisation**: the *optimiser* of a compiler applies a series of transformations to the IR in order to produce better target code. The optimisation phase can run as a fixed-point computation: optimisations are repeatedly applied until no change is detected in the resulting code. As a result, optimisation can be the most time-consuming task of a compiler.
- **Target Code Generation**: the final phase of a compiler produces target code from the optimised IR. This phase deals with low-level issues, such as register allocation and assignment of explicit jump addresses.

1.3 Re-Envisioning Intermediate Code Generation

The intermediate code generation phase can be viewed as the 'middle-end' of a compiler, with the collection of all phases occurring before it referred to as the *front-end*, and everything after as the *back-end*. If we assume that the IR being used is universal these collections are composable, with the existence of m front-ends and n back-ends giving rise to $(m \times n)$ distinct compilers. In practice, however, this is not the case, with the internal structure of various classes of IR differing greatly between compilers (representative instances include RTL [DF80], SSA [CFR+91] and CIL [NMRW02]).

We note that the analyses performed in the front-end generally exist to guarantee the sound and well-formed nature of the syntax of a source program. Further, they happen at a fairly high level, with the deepest analysis taking place typically being static type inference.

1.4 What Does This Thesis Present?

This thesis presents a unique re-factorisation of the *intermediate code generation* phase of a compiler with a particular style of IR representation at the middle-end, namely a stack-based sequence of low-level instructions. We explore the idea that the intermediate code generation phase can be constructed in a manner wherein said IR is modular in the features of the source language: in particular, we will see that the necessary IR can be *derived* for a given source language, and the amount of work required to implement the entire phase is reduced to defining the 'compilation semantics' for each feature in isolation.

We present our implementation of this concept within the pure, statically-typed functional language Haskell [HHPJW07]. More precisely, we present an embedded compiler (i.e. the compiler is written *in* Haskell as a function) that operates over domain-specific languages (DSLs) also defined within Haskell, with an online code repository available here [1]. The word compiler is overloaded here; from this point onwards, when we say 'compiler' we refer to the function performing the intermediate code generation, 'source language' refers to the domain-specific language representing the abstract syntax tree of a source program, and 'target language' refers to the IR we are compiling into.

1.4.1 Contributions & Thesis Structure

The main contributions of this thesis are as follows:

- Chapter 5.2 shows how the usage of generalised algebraic datatypes to model particular syntactic constructs permits the capture of existential type constraints in a clean and modular manner.

[1] https://www.fpcomplete.com/user/laurence.e.day/phd-thesis-code

- Chapter 5.3 extends our compilation framework with syntactic support for both mutable state and variable binding via the lambda-calculus, with support for two distinct evaluation schemes.

- Chapter 5.4.1 examines the issue of effects that do not commute (e.g. exceptions and mutable state), which may require a source program to be compiled in different ways depending on the order in which effects are manifested. We present three distinct techniques that can be used to solve this concern, operating over both the type and function level.

- Chapter 6.1 gives our source language further expressive power by introducing syntax supporting for-loops, while-loops, conditionals and sequencing. As a consequence of this, we identify a potential class of ill-formed program that can arise when programming imperatively without sufficient safeguards. We eliminate this concern by refactoring the source language as a typed variant of Johann and Ghani's fixpoint representation of generalised algebraic datatypes [JG08].

- Chapter 6.4.1 demonstrates the usage of Oliveira and Cook's structured graphs [OC12] as the underpinning of a refined representation of the target language, and show how the additional structure that they provide allows the compilation of non-cyclic control structures in a modular manner without code duplication. More importantly, this new representation permits the modular compilation of cyclic control structures, which we demonstrate by way of an extended example.

The ideas that we present first appeared in the following series of papers, with the author of this thesis serving as the lead author for each paper:

1. Laurence E. Day and Graham Hutton [DH11], "Towards Modular Compilers for Effects", in the *Proceedings of the 12th International Symposium on Trends in Functional Programming.*

2. Laurence E. Day and Graham Hutton [DH13], "Compilation à la Carte", in the *Proceedings of the 25th International Symposium on the Implementation and Application of Functional Languages.*

3. Laurence E. Day and Patrick Bahr [DB14], "Pick'n'Fix: Capturing Control Flow in Modular Compilers", in the *Proceedings of the 15th International Symposium on Trends in Functional Programming.*

We highlight at this point that the initial idea to use structured graphs when treating cyclic control structures is credited to Dr. Patrick Bahr, and thank him for this substantial contribution. The remainder of the ideas presented are those of the author himself.

The thesis is hereafter structured in the following way.

Chapter 2 frames the problem this thesis aims to solve by way of an extended example involving the extension of a minimal example with additional syntax implementing a new effect.

Chapter 3 provides a primer in the background theory required, together with a review of existing literature on topics related to language semantics and modelling computation.

In Chapter 4, we lay the groundwork for our modular compilation framework, introducing the notions of modular syntax for a small language, its modular semantics, and a first attempt at a modular compilation function.

We extend this framework in Chapter 5, adding further expressive power by way of new syntax supporting mutable state and the lambda calculus, and refine the representation of the source language in order to better enforce constraints upon individual syntax fragments. Furthermore, we present a number of novel solution to the issues that arise when the modular syntax of different effects interacts in unexpected ways.

In Chapter 6 we introduce the notions of control flow and conditionals, and further refine the source language with a surface-level type system in the presence of the various syntactic categories needed to eliminate a particular class of ill-formed program that result. Following this, we turn our attention to the representation of the target language, opting for a more flexible approach better suited to cyclic programs.

Chapter 7 provides a thorough treatment of a modular implementation of the semantics of the refined target language, by way of a virtual machine.

Finally Chapter 8 concludes the thesis by reflecting upon the overall work, and by discussing several potential research avenues aimed at extending the topics that have been presented throughout.

Chapter 2

Setting the Scene

In this chapter, we introduce the problem domain that we address in this thesis by way of an extended example. Specifically, we detail the issues that arise when we extend a domain-specific language (DSL) embedded in Haskell with syntax associated with a new computational effect. We note that a number of existing datatypes and functions associated with the DSL require extension in order to accommodate the extension, either by altering their type signatures or by introducing new constructors and definitions. We recognise that this scenario is a classic example of the Expression Problem [Wad98], and position ourselves to present the fundamentals of our solution framework in the next chapter.

2.1 Building From The Ground Up

Consider a simple language `Expr` comprising integer values and binary addition, for which we can give a denotational semantics by means of a function that evaluates an expression to an integer value (we make the assumption here that expressions are both finite and everywhere well-defined):

```
data Expr = Val Int | Add Expr Expr


eval           :: Expr -> Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
```

Alternatively, expressions can be compiled into a sequence of low-level instructions to be operated upon by a virtual machine, whose behaviour is defined as a (small-step) operational semantics. We can compile an expression to a list of operations as follows:

```
type Code = [Op]
data Op   = PUSH Int | ADD


comp   :: Expr -> Code
comp c = comp' c []
```

```
comp'               :: Expr -> Code -> Code
comp' (Val n)    c = PUSH n : c
comp' (Add x y) c = comp' x (comp' y (ADD : c))
```

Note that the above compiler is defined in terms of an auxiliary function `comp'` which accepts an additional `Code` argument playing the role of a continuation, thereby avoiding the use of the append operator (`++`) and enabling simpler proofs as in Chapter 13 of [Hut07]. We execute the resulting `Code` on a (partial) virtual machine operating over a `Stack`:

```
type Stack = [Item]
data Item  = INT Int

exec :: Code -> Stack
exec c = exec' c []

exec' :: Code -> Stack -> Stack
exec' []            s = s
exec' (PUSH n : c) s = exec' c (INT n : s)
exec' (ADD : c)     s = let (INT y : INT x : s') = s in
                             exec' c (INT (x + y) : s')
```

The correctness of the abovementioned compiler can now be captured by stating that the result of evaluating a *finite* expression is equivalent to first compiling and then executing it, and then selecting the topmost value on the stack. This can be expressed in diagrammatic form as follows:

$$\begin{tikzcd}
\texttt{Expr} \arrow[r, "\texttt{eval}"] \arrow[d, "\texttt{comp}"'] & \texttt{Int} \\
\texttt{Code} \arrow[ur, "\texttt{head} \circ \texttt{exec}"'] &
\end{tikzcd}$$

We make explicit at this point that we do not consider infinite source expressions to be well-typed for the purposes of this thesis.

2.2 Adding A New Effect

Suppose now that we wish to extend our language with a new effect, in the form of exceptions. We consider what changes will need to be made to the language syntax, semantics, compiler and virtual machine as a result of this extension. First of all, we extend `Expr` with two new constructors:

```
data Expr = Value Int | Add Expr Expr
          | Throw   | Catch Expr Expr
```

The newly introduced `Throw` constructor corresponds to an uncaught exception, while `Catch` is a handler construct that returns the value of its first argument unless it is an uncaught exception, in which case it returns the value of its second argument instead.

From a semantic point of view, adding exceptions to the language requires changing the result type of the evaluation function from `Value` to `Maybe Value` in order to accommodate potential failure when evaluating expressions, noting that whilst the original `eval` function has result type `Int`, it can implicitly be seen as being wrapped in an `Identity` monad: an idea that we shalll explore in later chapters. In turn, we must rewrite the semantics of values and addition accordingly, and define an appropriate semantics for throwing and catching (once again abusing Haskell syntax for the purposes of defining our denotational semantics):

```
eval :: Expr -> Maybe Value
eval (Val n)     = return n
eval (Add x y)   = eval x >>= \n ->
                   eval y >>= \m ->
                   return (n + m)
eval (Throw)     = mzero
eval (Catch x h) = eval x `mplus` eval h
```

In the above code, we exploit the fact that `Maybe` is monadic, as we will see in Chapter 3.3.2. In particular, because `Maybe` is also a *monoid* — a structure with an associative binary operation and an identity element —

we make use of the monoidal methods `mzero`, corresponding to failure, and `mplus`, for sequential choice. Finally, in order to compile exceptions we must introduce new stack instructions to the virtual machine, and extend the compiler accordingly:

```
data Op = PUSH Int  | ADD | THROW
        | MARK Code | UNMARK


comp' :: Expr -> Code -> Code
comp' (Val n)     c = PUSH n : c
comp' (Add x y)   c = comp' x (comp' y (ADD : c))
comp' (Throw)     c = THROW : c
comp' (Catch x h) c = MARK (comp' h c)
                      : comp' x (UNMARK : c)
```

Intuitively, `THROW` is an operation that throws an exception, `MARK` makes a record on the stack of the handler code to be executed should the first argument of a `Catch` expression fail, and `UNMARK` indicates that no uncaught exceptions were encountered in the most recent `Catch`-block and that the topmost handler code on the stack can be removed. Note that the accumulator `c` plays a key role in the compilation of `Catch`, being used in two places to represent the code to be executed after the current compilation. Also note, however, that this leads to explicit code duplication!

Because we now need to keep track of handler code on the stack as well as integer values, we must also extend the `Item` datatype and the virtual machine to cope with both the new operations and the potential for failure:

```
data Item = INT Int | HAN Code

exec :: Code -> Maybe Stack

exec c = exec' c []



exec' :: Code -> Stack -> Maybe Stack

exec' []            s = return s

exec' (PUSH n : c) s = exec' c (INT n : s)

exec' (ADD : c)    s = let (INT y : INT x : s') = s in

                          exec' c (INT (x + y) : s')

exec' (THROW : _)  s = unwind s

exec' (MARK h : c) s = exec' c (HAN h : s)

exec' (UNMARK : c) s = let (v : HAN _ : s') = s in

                          exec' c (v : s')
```

The auxiliary `unwind` function used in the above implements the process of invoking handler code in the case of a caught exception, by executing the topmost `Code` record on the stack, and failing if no such record exists:

```
unwind :: Stack -> Maybe Stack

unwind [] = mzero

unwind (INT _ : s) = unwind s

unwind (HAN h : s) = exec' h s
```

2.3 The Problem At Hand

As the reader can appreciate, extending such a simple source language with constructors which enable additional expressive power (which we will refer to as an 'effect' from this point onwards) can result in multiple changes to existing code being required. For our example in particular, we needed to extend three datatypes (`Expr`, `Op` and `Item`), change the return type and existing definition of three functions (`eval`, `exec` and `exec'`), and extend the definition of all functions involved.

The need to modify and extend existing code for each new effect we wish to introduce to our language is clearly at odds with the desire to structure a compiler in a modular manner, and raises a number of problems. Importantly, unless there is familiarity with the workings of *all* aspects of the language rather than just the feature being added, changing existing code that has already been designed, implemented and (ideally) proved correct is bad practice from a software engineering point of view.

At this point, however, we should clarify our intentions towards the usage of the word ‘modular’ throughout the rest of this thesis. Firstly, when we refer to a modular language, we refer to the syntax for the various effects it supports being defined in a manner such that we are capable of selecting the language that contains precisely the expressive power we desire. Whilst languages can also be modularised along the axes of static vs dynamic semantics, we will not explore this idea further here. Secondly, when we refer to a modular compiler, we allude to a very high-level ‘black box’ that accepts a source language expression and produces corresponding, lower-level code according to a dynamic/execution semantics, with the box itself constructed from several smaller boxes, with each one entirely self-contained and responsible for translating the constructors associated with a particular effect.

One might wonder why we have chosen to explore this problem domain by defining compilers for embedded DSLs (i.e. languages defined within Haskell itself) in a modular (as we have circumscribed the word) manner. We have made this choice because it is an ideally rich source topic from which to draw interesting and complex problems, concerning both extensibility and the interaction of common effects. With this in mind, we present an initial – rudimentary – solution in Chapter 4, which we will then refine and extend throughout the remainder of the thesis.

One final point to be made here is that whilst Haskell serves as a reasonable vehicle for modelling what *happens* on a virtual machine, a real virtual machine wouldn't be implemented in Haskell! A more realistic choice would be C, however the translation between the two is fairly straightforward, modulo a more formal treatment of the instruction set and addressing modes.

Chapter 3

Background Knowledge

/begincomment As mentioned previously, we have chosen Haskell as the implementation language for the work we will present throughout this thesis. However, Haskell – as a pure language – is well-known in the wider programming community for its approach to implementing side-effects [Jon01]. Given that the meaning of a source program is often derived from the effects that it invokes, it is particularly important that we review the techniques that Haskell uses to manifest impurity.

This chapter consists of two halves: a primer and a literature review. The primer presents reviews of the major approaches to defining the semantics of programs, the fundamentals of category theory as a formalism, and the usage of functors and monads (both category-theoretic constructs) in Haskell to implement side-effects. The intended audience for the primer (especially as far as monads are concerned) is the functional programming community with prior experience in using monads to structure their programs, however the references provided throughout Chapter 3.4.1 serve as a rigorous introduction for the non-FP reader. The literature review that follows places this thesis in its wider context, discussing relevant work published on the various characterisations of computation, compilation and proving correctness, and defining programming constructs in a modular, extensible manner.

3.1 Semantics

When we talk about what a program 'does' or 'means', we are often referring to its *semantics*. There are multiple ways in which we can specify the underlying meaning of a program, and each has their appropriate use-cases. In this section we present the fundamentals of the two main approaches to defining the semantics of a program: *denotational* and *operational*.

3.1.1 Denotational Semantics

The *denotational semantics* of a language, as originally investigated by Scott and Strachey in the 1960s [SS71], is defined as a mapping from a program to a mathematical object which captures the essential meaning of the program. We read $[\![\mathbf{x}]\!]$ as the *denotation* of program **x**. Candidates for the set of denotations (the *semantic domain*) vary widely, but as a rule require the ability to differentiate between the 'amount' of information carried by a program [SHLG94]. In general, a denotational semantics should be *compositional*: that is, the meaning of a program phrase should only be constructed as a function of the meaning of its subphrases (however, as we shall see in Chapter 6, this need not always be the case!). To illustrate with an example – that uses Haskell *syntax* rather than Haskell itself, as we don't want anything infinite – , consider the following language:

```
data Basic = Value Int | Add Basic Basic
```

We can define a denotational semantics for `Basic` as follows, mapping into the set of integer values $\mathbb{Z}$:

$$
\begin{array}{lll}
[\![\_]\!] & :: & \texttt{Basic} \rightarrow \mathbb{Z} \\
[\![\texttt{Value n}]\!] & = & \texttt{n} \\
[\![\texttt{Add x y}]\!] & = & [\![\texttt{x}]\!] + [\![\texttt{y}]\!]
\end{array}
$$

In the above, the semantics of a `Value` constructor is simply the associated integer, and in turn the semantics of an `Add` is the sum of the denotations of the two argument expressions.

3.1.2 Operational Semantics

In contrast to the denotational approach of constructing a single interpretation function, a language can also be given an *operational semantics* by describing the syntax-directed behaviour of individual constructs of a language via *inference rules* which detail the desired transitions. An operational semantics can be given in one of two forms: either *structural* (or small-step, as developed by Plotkin [Plo81]), or *natural* (big-step, as developed by Kahn [Kah87]), and the key difference between these is whether a transition details a single computational step or a complete evaluation.

Recalling the `Basic` datatype introduced in the previous subsection, we define the rules for the structural operational and natural semantics for illustration. Note that we use the metavariables n_1, n_2, . . . to range over integer values, `(+)` for integer addition, and a_1, a_2 over terms in `Basic`:

Structural Operational Semantics Rules

$$\frac{}{(\texttt{Value}\ n) \Rightarrow \texttt{Value}\ n}\ (S1)$$

$$\frac{}{\texttt{Add}\ (\texttt{Value}\ n)\ (\texttt{Value}\ m) \Rightarrow \texttt{Value}\ (n + m)}\ (S2)$$

$$\frac{a_1 \Rightarrow a_1'}{\texttt{Add}\ a_1\ a_2 \Rightarrow \texttt{Add}\ a_1'\ a_2}\ (S3)$$

$$\frac{a_2 \Rightarrow a_2'}{\texttt{Add}\ (\texttt{Value}\ n)\ a_2 \Rightarrow \texttt{Add}\ (\texttt{Value}\ n)\ a_2'}\ (S4)$$

Natural Semantics Rules

$$\frac{}{(\texttt{Value}\ n) \Rightarrow \texttt{Value}\ n}\ (N1)$$

$$\frac{a_1 \Rightarrow \texttt{Value}\ n_1 \quad a_2 \Rightarrow \texttt{Value}\ n_2}{\texttt{Add}\ a_1\ a_2 \Rightarrow \texttt{Value}\ (n_1 + n_2)}\ (N2)$$

Inference rules with no premisses (S1, S2 and N1) are referred to as *axioms*. We refer to the application of operational semantic rules to an expression as the *execution* of the expression [Hut10] (assuming that the application of existing operations such as integer addition is automatic), in contrast with its *evaluation* with respect to a denotational semantics. The implementation of an operational semantics is frequently referred to as a *virtual machine*, and we will see this term heavily used throughout this thesis.

3.2 Category Theory

Category theory is a mathematical tool used to capture the essence of interactions between mathematical objects with common structure [Pie91]. Within the scope of this thesis, being able to precisely describe generic structure of mathematical objects is important, and so we give a brief introduction to the key concepts within this section. Whilst there are several ways to characterise a category, here we define a *category* $\mathbb{C}$ to be:

1. A collection $\mathbb{C}_{ob}$ of *objects*, denoted by metavariables $\mathbf{X}$, $\mathbf{Y}$, ...
2. A collection $\mathbb{C}_{ar}$ of *morphisms* between the objects of $\mathbb{C}_{ob}$ (also sometimes called *arrows*). For example, $f : \mathbf{X} \to \mathbf{Y}$ is a morphism between objects $\mathbf{X}$ and $\mathbf{Y}$, provided both objects are defined in the collection $\mathbb{C}_{ob}$. We denote morphisms with metavariables *f*, *g*, ...
3. A pair of functions *dom* and *cod* defined over morphisms from $\mathbb{C}_{ar}$, detailing the domain and codomain of a given morphism (taking the view of morphisms as functions); e.g. given $f : \mathbf{X} \to \mathbf{Y}$, then we have $dom(f) = \mathbf{X}$, $cod(f) = \mathbf{Y}$.

Furthermore, a category must adhere to the following:

1. For all objects $\mathbb{C}$, there is an associated morphism $id_{\mathbb{C}} : \mathbb{C} \to \mathbb{C}$.
2. For all morphisms $f : \mathbb{B} \to \mathbb{C}$, $id_{\mathbb{C}} \circ f = f = f \circ id_{\mathbb{B}}$.

3. For all morphisms f and g where the domain of g matches the codomain of f, there is an associated morphism $(g \circ f) : dom(f) \rightarrow cod(g)$.

4. For all morphisms f, g and h of the appropriate type, it must be the case that $(f \circ g) \circ h = f \circ (g \circ h)$.

Categories can represent a vast array of mathematical structures, depending on the choice of $\mathbb{C}_{ob}$ and $\mathbb{C}_{ar}$. For instance, the category **Grp** has groups as objects and group homomorphisms as morphisms, and the category **Set** has sets as objects and total functions as morphisms.

Further, in some categorical instances, particular objects can be denoted as being special in one way or another. We call an object $\mathbf{X} \in \mathbb{C}_{ob}$ *initial* if for all other objects $\mathbf{Y} \in \mathbb{C}_{ob}$, there is a unique morphism $i : \mathbf{X} \rightarrow \mathbf{Y}$. Similarly, we call $\mathbf{X}$ terminal if there is a unique morphism $t : \mathbf{Y} \rightarrow \mathbf{X}$. For instance, within the category **Set**, the empty set $\emptyset$ is initial and any one element set $\{x\}$ is terminal; i.e. $i_{\mathbf{Set}} : \mathbb{C}_{ob} \rightarrow \emptyset$ and $t_{\mathbf{Set}} : \mathbb{C}_{ob} \rightarrow \{x\}$ $\forall \; x \in \mathbb{C}_{ob}$.

Adding further abstraction to this notion of common mathematical structure, the category **Cat** can be constructed with (small) categories as objects and structure-preserving constructs known as *functors* as elements of $\mathbb{C}_{ar}$. Given two categories $\mathbb{C}$ and $\mathbb{D}$, a functor F between them must obey the following conditions:

1. For all objects $\mathbf{X}$ of the source category $\mathbb{C}$, there is a corresponding object $F(\mathbf{X})$ in the target category $\mathbb{D}$.
2. For all morphisms $f : \mathbf{X} \to \mathbf{Y}$ of the source category $\mathbb{C}$, there is a morphism $F(f) : F(\mathbf{X}) \to F(\mathbf{Y})$ in the target category $\mathbb{D}$ such that:
   (a) For all objects $\mathbf{A}$ of the source category $\mathbb{C}$, $F(id_{\mathbf{A}}) = id_{F(\mathbf{A})}$.
   (b) For all morphisms $f : \mathbf{X} \to \mathbf{Y}$ and $g : \mathbf{Y} \to \mathbf{Z}$ in the source category $\mathbb{C}$, there is a morphism $F(g \circ f) = F(g) \circ F(f)$ in the target category $\mathbb{D}$.

Given a functor $F : \mathbb{C} \to \mathbb{C}$ over a category $\mathbb{C}$ (which we refer to as an *endofunctor*), we call a pair $(\mathbf{A}, \alpha)$ with an object $\mathbf{A} \in \mathbb{C}_{\mathbf{ob}}$ and an arrow $\alpha : F\mathbf{A} \to \mathbf{A} \in \mathbb{C}_{ar}$ an *F-algebra* on F. In the category **F-Alg** with F-algebras as objects, we define the morphism h between $(\mathbf{A}, f)$ and $(\mathbf{B}, g)$ as the morphism making the below diagram commute:

$$\begin{tikzcd}
F\mathbf{A} \arrow[r, "F(h)"] \arrow[d, "f"'] & F\mathbf{B} \arrow[d, "g"] \\
\mathbf{A} \arrow[r, "h"'] & \mathbf{B}
\end{tikzcd}$$

The most important definition we introduce in this section is that of a catamorphism. We define a *catamorphism on f* to be the unique morphism h between an *initial algebra* $(\mathbf{A}, in)$ and any other F-algebra $(\mathbf{B}, f)$. This is also referred to as a *functorial fold*, and we will see it heavily used throughout the remainder of the thesis.

3.3 Computational Effects Using Monads

We have mentioned several times that it is tricky to manifest effectful computation in Haskell as a direct result of its purity. In this subsection, we show how the concept and usage of *monads* allows us to write programs that indirectly *simulate* effects (in that we avoid the need to painstakingly thread the appropriate data structures through a 'monad-free' program). More specifically, we give the categorical definition of monads as a specific type of augmented functor, show how this concept can be implemented in Haskell (and why the Haskell definition of monads uses a 'non-canonical' choice of methods), and how monads thus implemented can be used to define effectful language semantics.

3.3.1 Monads in Category Theory

Given a category $\mathbb{C}$, we define a monad on $\mathbb{C}$ to be an endofunctor T that comes equipped with two 'morphisms between morphisms' (known as *natural transformations*), namely $\eta : 1_{\mathbb{C}} \to T$ (where $1_{\mathbb{C}}$ is the *identity functor* on $\mathbb{C}$) and $\gamma : T^2 \to T$. A monad thus equipped must meet the conditions laid out by the two diagrams below, with the underlying intuition given in Chapter refintroducing-monads:

$$\begin{tikzcd}
T^3 \arrow[r, "T\gamma"] \arrow[d, "\gamma T"'] & T^2 \arrow[d, "\gamma"] & & T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T^2 \arrow[d, "\gamma"] \\
T^2 \arrow[r, "\gamma"'] & T & & T^2 \arrow[r, "\gamma"'] & T
\end{tikzcd}$$

3.3.2 Implementing Monads in Haskell

To reiterate, at this point we consider monads to be endofunctors augmented with two natural transformations which obey ‘appropriate’ laws [Mog89]. In order to understand the *implementation* of monads in Haskell [Wad92], we must first consider the implementation of functors.

Functors in Haskell

We have introduced functors as structure-preserving morphisms *between* the objects and morphisms of categories. In Haskell, the `Functor` typeclass captures this notion as follows:

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

That is to say, an instance of a Haskell `Functor` is a type constructor `f` which can have its associated datatype mapped over via `fmap`. We require `fmap` to adhere to two laws:

1. $\mathit{fmap}\ \mathit{id}_{\mathbf{X}} = \mathit{id}_{f(\mathbf{X})}$

2. $\mathit{fmap}\ (g \circ f) = \mathit{fmap}\ g \circ \mathit{fmap}\ f$

By way of example, consider the following instantiation of a polymorphic binary tree as a `Functor`:

```
data Tree a = Leaf a | Node Tree Tree
instance Functor Tree where
  fmap f (Leaf n)   = Leaf (f n)
  fmap f (Node l r) = Node (fmap f l) (fmap f r)
```

An intuitive view of a functor is that of a container, the contents of which can have functions applied to them. The shape of this container can vary widely, but the core functionality remains.

Introducing Monads

The *categorical* notion of monad can be represented in Haskell via the following datatype, with η renamed to `return` and γ to `join`:

```
class Functor m => Monad m where
  return :: a -> m a
  join   :: m (m a) -> m a
```

The `join` method characterises what it means to 'be' a monad:

1. `join` $\circ$ `fmap join` $=$ `join` $\circ$ `join`
2. `join` $\circ$ `fmap return` $=$ `join` $\circ$ `return` $= id$
3. `join` $\circ$ `fmap` (`fmap` f) $=$ `fmap` $f \circ$ `join`

However, prior to GHC 7.10 [1], Haskell did not enforce the requirement that all monads are functors – disqualifying the usage of `fmap` –, using the following formulation instead:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The above two definitions of monads are equivalent, with `join` and `(>>=)` being related by the following equations:

1. `fmap` f $\mathbf{m} = \mathbf{m}$ `>>=` (`return` $\circ f$)
2. `join` $\mathbf{n} = \mathbf{n}$ `>>=` id
3. $\mathbf{m}$ `>>=` $f \equiv$ `join` (`fmap` f $\mathbf{m}$)

[1]The work within this thesis was developed using GHC 7.8.3.

The formulation in terms of `return` and `(>>=)` is the one that shall be used throughout this thesis. Moreover, we require that `return` and `(>>=)` adhere to the 'monad laws' of identity and associativity:

1. $\texttt{return}\ \mathbf{x}\ \texttt{>>=}\ f \equiv f\ \mathbf{x}$
2. $\mathbf{m}\ \texttt{>>=}\ \texttt{return} \equiv \mathbf{m}$
3. $(\mathbf{m}\ \texttt{>>=}\ f)\ \texttt{>>=}\ g \equiv \mathbf{m}\ \texttt{>>=}\ (\lambda\mathbf{x} \rightarrow f\ \mathbf{x}\ \texttt{>>=}\ g)$

For example, the `Maybe` datatype can be instantiated as a monad which is used to model exceptions:

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
  return x       = Just x
  Nothing >>= f = Nothing
  Just x   >>= f = f x
```

Monadic Semantics

As we shall see shortly, Moggi's original 1989 paper [Mog89] utilised monads in order to structure denotational semantics, and Wadler expanded on this idea to produce modular interpreters in 1992 [Wad92]. Liang and Hudak continued this line of work, developing the notion of modular monadic semantics in 1998 as a *structured* denotational semantics, revolving around monads and monad transformers as the key descriptive mechanism [Lia98].

As we have seen, a monad in Haskell is a type constructor `m` with methods:

```
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
```

Using these methods, it can be shown that a language can be given a denotational semantics that is parameterised by a monad encapsulating the result type. The modular monadic semantics for a language `L` supporting addition can be given in the following way:

```
eval :: Monad m => L -> m Int
eval (Add x y) = eval x >>= \n1 ->
                 eval y >>= \n2 ->
                 return (n1 + n2)
```

Depending on the choice of monad `m`, the concrete semantics that result may differ, hence the usage of the term 'modular' in this context. For example, observe the `return` and `(>>=)` definitions for the monads representing state and environment respectively:

```
type m a = \s -> (a, s) -- Mutable State

  return x = \s -> (x, s)

  x >>= f  = \s -> let (u, t) = x s in (f u) t


type m a = Env -> a -- Environment

  return x = \e -> x

  x >>= f  = \e -> f (x e) e
```

With this style of language specification, those features (i.e. data constructors) of a language which do not require the abstraction required by a monad are simply 'lifted' into one by way of `return` and `(>>=)`, whilst those that *do* require the monad are free to make full usage of any operations that its presence enables, a concept which we will explore fully in Chapter 4.

3.4 Associated Literature

In this section, we present a survey of the existing work that is most relevant to our work on the modular compilation of effects. These papers have been loosely grouped into subcategories to maintain coherence, and is current as of December 2014. For each paper, the major contributions are introduced and any particularly interesting or relevant concepts are explained.

3.4.1 Monads As Computation

In this section, we examine the work credited with both identifying and popularising the idea of modelling computational effects using monads within both the category theory and functional programming communities. Whilst these papers are not directly linked to the research presented within this thesis, they are fundamental to the notion of computation as enabled by Haskell as a host language.

Computational Lambda-Calculus and Monads, *Eugenio Moggi (1989)* [Mog89]: Moggi's 1989 paper was the first to investigate the idea that effectful computation can be modelled using category theory. The primary notion is that a program can be seen as a morphism between an object $\mathbf{A}$ of *values* of type `A` and an object $\mathtt{T}\ \mathbf{B}$ of *computations* of type `B`, where `T` is a monad which captures the side-effects that the computation may contain. This type of morphism belongs to the *Kleisli category* $\mathbb{C}_{\mathtt{T}}$ constructed from a base category $\mathbb{C}$, with identity and composition morphisms given by the natural transformations associated with `T` as defined in Chapter 3.2. For example, in the category **Set**, the monad $\mathtt{T}_{\mathbf{ND}}$ for nondeterminism is defined with $\mathtt{T}\ \mathbf{A} = \mathcal{P}(\mathbf{A})$, $\eta_{\mathbf{A}}(\mathrm{a}) = \{\mathrm{a}\}$ and $\gamma_{\mathbf{A}}(\mathbf{X}) = \cup\mathbf{X}$.

The paper goes on to discuss the interpretation of simple computations within Kleisli categories, and the conditions which must be met to extend the set of terms which can be interpreted. For example, to interpret lambda-terms the underlying monad must be extended to a strong monad via an additional natural transformation. The main contribution of the paper is the concept of $\lambda_c - models$ over a category $\mathbb{C}$ (where such a model is a strong monad with `T`-exponentials) and the formal system λ_c – the *computational lambda calculus* – which is sound and complete over λ_c-models, capable of establishing the categorical equivalence of terms written within it.

The Essence of Functional Programming, *Philip Wadler (1992)* [Wad92]: Following Moggi's discovery that effectful computation can be modelled using monads, Wadler began to investigate the application of monads to structure functional programs. The paper begins by discussing the changes that an interpreter written in a pure functional language would require to support a number of effects, and contrasts these changes to the fact that an impure interpreter would *need* no such restructuring. Following on from this, an interpreter for a language `Term` based upon the lambda-calculus is introduced in Haskell, and the functional characterisation of monads is established. It is then shown how the 'standard' interpreter is extended in order to support individual additional features, ranging from exception handling to nondeterminism. For each extension, the underlying monad `M` and associated methods `unitM` and `bindM` are redefined to support the feature in question, and the parts of the interpreter that are associated with the feature identified and changed appropriately. Further, the changes which need to be made to the interpreter such that it uses the call-by-name evaluation scheme – instead of call-by-value – are discussed in depth.

At this point, the monad laws are introduced, and it is discussed how the laws can alternatively be formulated using `unitM` and the monadic operations `mapM` and `joinM`, and that these two sets of laws necessarily follow from each other. The laws are then used to prove that binary addition is associative in *any* monadic interpreter.

The paper goes on to contrast the monadic style of programming used in the interpreter and its extensions to continuation-passing style (CPS). To this end, the continuation monad is introduced and used to structure the interpreter for the *original* `Term` language: it is observed that the resulting interpreter is similar to that which utilised the identity monad. Indeed, it is recognised that a suitable monad allows a monadic interpreter to be translated into a CPS interpreter – and vice-versa – by choosing a suitable answer space. However, there *is* a difference between monads and CPS concerning the degree of 'control' allowed in a datatype: for example, CPS cannot provide an 'error escape' for a language with exceptions.

Finally, it is noted that some syntactic sugar may go some way to aiding the comprehension of programs written in a monadic style – a 'letM' construct for a monad `M` is proposed –, an observation eventually realised in the form of **do**-notation.

Monads For Functional Programming, *Philip Wadler (1992)* [Wad95]: Building upon the work described in the previous paper, this paper identifies further application areas for monads within functional programs. The first half of the paper re-introduces the 'monadification' of an interpreter extended with various features. This version differs, however, in that the full *non-monadic* definition of the interpreter for each feature is given first, pointing out the common structure of each variation before generalising this pattern and revisiting each feature, defining its monadic counterpart. This approach makes explicit the benefit that the monadic style provides when function redefinition is required. For each feature, a number of sample expressions are evaluated to clarify the interpreter semantics, a notable omission from the original paper.

It is then observed that whilst the use of monads so far has been limited to describing existing features more effectively, they can also be used to aid in the definition of *new* features. To demonstrate, the following two sections treat (respectively) the implementation of an efficient in-place array update and the use of monads to construct recursive descent parsers. In the former, monads of state transformers are introduced as a way to transform and read arrays, with the fact that monads are represented as abstract datatypes ensuring the *single-threadedness* of the array (a crucial condition for updating an array safely). The latter identifies that parsers themselves form a monad, and concepts such as sequencing, alternation, filtering and iteration of such monadic parsers are defined and discussed.

3.4.2 Alternative Characterisations Of Computation

Whilst this thesis is heavily reliant on Haskell's use of monads and monad transformers to model computational effects, these are not the only techniques available. In this section we survey papers which seek to provide alternate characterisations of effectful computation.

Computational Effects & Operations: An Overview, *Gordon Plotkin and John Power (2002)* [PP04]: The commonly held view of operations associated with computational effects (i.e. `inEnv`, `callcc` etc) is that they are derived from an underlying monad `T`, and this is the view taken within this thesis. However, it is worthwhile knowing that there is a diametric view wherein the operations themselves are taken as primitive, and a monad is derived using the operations as constraints. This latter approach is referred to as the *algebraic theory of effects*, and makes use of *Lawvere theories*: collections of all equations that hold for a particular algebraic structure. More specifically, *countable* Lawvere theories are used, meaning that the operations and equations comprising the theory form a countable set.

The main notion of this paper is that a description of an effect can be given as a countable Lawvere theory freely generated by its associated algebraic operations [PP01], with a correspondence existing between the theory's morphisms and said operations. A number of examples are given in the paper, including nondeterminism and partiality, amongst others. We note that whilst commutative *product* Lawvere theories can be constructed from subtheories subject to adherence to commutativity laws (which would allow for arbitrary orderings of effects: more on this in Chapter 5.4.1), Haskell cannot enforce these laws at present (under GHC 7.8.4).

Handlers of Algebraic Effects, *Gordon Plotkin and Matija Pretnar (2009)* [PP09]: In a sense, the algebraic operations which characterise an effect can be seen as dual to the concept of computational effect handlers: the former can be viewed as constructors for an effect, and the latter as deconstructors that *manifest* an effect. Moreover, common constructs such as exception handlers are *not* algebraic operations [PP03]. This paper introduces the notion of a handler construct for arbitrary algebraic operations, and shows how a generalisation of the exception-handling construct of Benton and Kennedy [BK01] permits an algebraic treatment.

The underlying concept is that the handling of a computation corresponds to a homomorphism (a structure-preserving map), the domain of which is generated by the algebraic theory of the effects involved. The paper presents a logic for the handling of algebraic effects and a call-by-push-value calculus [PP08, Lev06] to formalise the ideas presented throughout.

Programming and Reasoning with Algebraic Effects and Dependent Types, *Edwin Brady (2013)* [Bra]: The previous paper presents formalisms which underpin the work on constructing handlers for algebraic effects, however these formalisms are defined over sets and functions, with the understanding that the concepts generalise in a straightforward manner to richer domains. This paper explicitly demonstrates this claim, implementing the handlers of algebraic effects within the general-purpose, dependently-typed functional programming language Idris [Bra13].

The notion of algebraic effects is implemented as a *domain specific language* (DSL) called `Effects`. This DSL makes use of the dependent types available in Idris in two essential ways: firstly by implementing a type-level check that the effect which we are invoking a handler for is indeed present in a source program, and secondly keeping relevant resources (data associated with effects) up-to-date throughout the lifetime of a program. An interesting research direction from the point of view of this thesis is the implementation of statically-enforced type-correct EDSLs – expanding upon the ideas presented in Chapter 6.2.1 – and their compilers within Idris.

3.4.3 Compilation & Correctness

In this section we survey a selection of papers associated with both the techniques utilised in the compilation of languages and methods of proving such compilation techniques to be semantically correct. We acknowledge at this point that the term 'modular compiler' can mean a great many things, with the axes upon which measures of modularity are obtained differing from one paper to the next. We point out such differences where they arise throughout.

Monad Transformers And Modular Interpreters, *Sheng Liang, Paul Hudak and Mark Jones (1995)* [LHJ95a]: This paper is considered the genesis of the usage of the monad transformer technique within Haskell, focussing upon the notion of using composable *building blocks* corresponding to individual computational features to form programming language interpreters. The data constructors and supported operations expected of a number of effects are introduced, alongside the monad transformers needed to implement them.

The source languages which result from this modularised approach are represented as a domain sum (implemented using an `OR` operator). The main interpreter function is defined as a method of a constructor class, which each sublanguage must be an instance of. What follows is a thorough discussion of the complications which arise when lifting certain monad transformers – more precisely, their operations – through each other. Two monad transformer laws detailing the conditions which the transformer method `lift` must satisfy, and the concept of a *natural lifting* of operations along a monad transformer is introduced. A natural lifting enforces the implicit constraint that a program not utilising a certain language feature behaves in the same manner if that feature is removed.

It is observed that if – categorically – a monad cannot compose with *all* other monads, a monad transformer variant cannot be defined for the feature it models. For example, the list monad can only compose with *commutative* monads, as discovered by Jones and Duponcheel [JD93]. As such, there is no monad transformer modelling nondeterminism. Due to this, we must often define a base monad to which transformers are applied. The paper goes on to discuss how certain monad transformers can be implemented by others: the examples given include environments modelled using the state monad transformer, and exceptions modelled using the continuation transformer. To conclude, a number of 'difficult' liftings – in the sense that the resulting semantics may be unclear – are given.

Modular Compilers Based On Monad Transformers, *William L. Harrison and Samuel Kamin (1998)* [HK98a]: Extending the work described above on using monad transformers to develop modular interpreters, this paper seeks to apply the same techniques to compiler construction. Such a compiler is constructed via a combination of *compiler blocks*, where a single compiler block is defined by the equations defining the 'compilation semantics' of an effect, and the monad transformers – and associated methods – needed to implement the effect. Given the monadic, CPS semantics of a feature, its compilation semantics are obtained via *pass separation*, the introduction of intermediate data structures with monad transformers.

The main result of the paper is the construction of a modular compiler for an Algol-like imperative language. The target language is a machine-language represented using appropriate combinators (`popblock`, `push` etc), an approach which takes advantage of the monadic structure of compilation semantics. Separate to this final result, the compilation of two languages supporting simple expressions and control flow is demonstrated, followed by their combination. The paper concludes by discussing a number of issues which may arise, such as metavariable scoping and recursion.

Compilation as Metacomputation: Binding Time Separation in Modular Compilers, *William L. Harrison and Samuel Kamin (1998)* [HK98b]: This paper builds upon the authors' previous work by observing that *metacomputations* (computations which produce computations) arise naturally in the compilation process. Compiler metacomputations can be obtained by compiling the *static* aspects of a program – such as code generation – into a computation which contains the *dynamic* aspects, i.e. stack manipulation. This staging of a program can be implemented using two monads – a static and dynamic monad – constructed via monad transformer. It is claimed that the composition of these two monads gives the correct domain for a modular compiler utilising staging.

Several of the compiler blocks from the previous paper are re-introduced, and the structure of the metacomputations produced by each is given. In each instance, the staging monads `Comp` and `Exec` are described via the operations which must be supported, and therefore which monad transformers must be used to construct them. The construction of these staging monads using transformers simplifies the combination of building blocks. Given the staging monads for any two compiler blocks, the staging monads for their *combination* are constructed using each of the monad transformers associated with the originals. Presuming that the specification of two languages are described by sets of equations, the specification of their combination is simply their set union.

Modular Compilers and their Correctness Proofs, *William L. Harrison (2001)* [Har01]: Harrison's doctoral thesis focusses the problem of modular compilation in the sense discussed by the previous two papers, via the construction and verification of *reusable compiler building blocks* (RCBBs) for various features of a source language. Demonstrations – and implementations – of the compilation of programs supporting various combinations are given, including static/dynamic scoping of variables, control flow and imperative features. Two distinct approaches to developing RCBBs are proposed, namely as metacomputations – defined previously – and as monadic code generators, targeting the same language as considered in his prior work [HK98a].

The examples and discussion relating to the metacomputational approach correspond strongly with that in the previous paper, mainly concerning theorems relating the standard and compilation semantics of languages. The alternative approach – compilation using monadic code generators (MCGs) – defines an MCG as a function `compile :: Source → m Target` for each feature, parameterised over a monad `m` – this approach being more closely related to that of our own research. The same features used as examples for the metacomputational approach are reused here, with MCGs for each introduced and explained in detail. The thesis then presents a case study of a source language `Exp` supporting exceptions, imperative features, blocks, booleans and control flow structure. The compiler for this language is verified correct by formulating and proving relations between the standard and compilation semantics of metacomputations and MCGs.

Certain conditions must be met in order to combine blocks to form a compiler for a non-trivial language, called *linking conditions*. The two linking conditions illustrated in the thesis relate imperative features to expressions, and control flow to booleans. To conclude, the notion of *observational program specification* is developed, a parametric monadic specification making minimal assumptions about the monad associated to a MCG, which proves useful in the verification process described.

Compiling Exceptions Correctly, *Graham Hutton and Joel Wright (2004)* [HW04]: Surprisingly, despite the amount of research into compilers within functional programming, correctness proofs for compilers dealing with non-standard features have been slow to emerge. To this end, this paper seeks to address this issue for exceptions, traditionally viewed as an advanced topic in compilation theory. The paper opens with the Haskell definition of a small language `Expr` consisting of integer values and addition, alongside an evaluator, compiler (targeting a stack-based list of instructions as the IR, as will be seen throughout this thesis) and virtual machine operating over it. The conditions for correctness are stated and proved for a general stack via induction, alongside a lemma detailing how code can be split up and executed in steps without affecting the result.

`Expr` is then extended with constructors enabling simple notions of exception throwing and handling, alongside extensions to the datatypes representing code. As a result, the evaluator, compiler and virtual machine are extended with new function definitions and edited where necessary to reflect the new semantics. A second proof of generalised compiler correctness relating to this extended language is given. Finally, in order to bring results more in line with 'realistic' languages, `Expr` is altered to include label jumps, and – instead of marking a stack with compiler handler code – makes reference to address locations.

Compiling Concurrency Correctly: Cutting Out The Middle Man, *Liyang Hu and Graham Hutton (2010)* [HH09]: Traditionally, compiler correctness for concurrent languages is proved by translating from both the source and target languages into an intermediate π-calculus (a formal system giving semantics to concurrent computations) and then proving equivalence via bisimulation. However, this method is overly complicated, requiring several extra layers of formalism, and this paper describes a simpler technique. Taking the same approach as the previous paper discussed, a language `Zap` of integer values and addition is defined, and – using Agda – given an operational semantics involving *actions* and *labels* which permits a simple notion of nondeterminism.

As nondeterminism introduces the possibility of *sets* of result values, the semantics of the compiler and virtual machine are given as a relation – rather than a set-valued function – coupled with a notion of weak bisimilarity (ignoring silent actions). The compiler correctness statement for `Zap` is formulated in terms of weak bisimilarity between two *combined semantics* expressions – the pairing of a `Zap` expression and a virtual machine – and proved in Agda. It has since been observed that this theorem captures precisely the same notion of compiler correctness as that of the compiler for exceptions discussed in the previous paper.

The `Zap` language is then extended to support explicit concurrency by forking expressions into new threads, resulting in the language `Fork`. The compiler is extended, new actions and labels are introduced, and the notion of a 'thread soup' as a list of concurrent threads is formalised. Finally, `Fork` is proved correct in a general setting using Agda.

A Formally Verified Compiler Back-End, *Xavier Leroy (2009)* [Ler09]: A large portion of the literature regarding compiler correctness considers a variation on the lambda calculus as the choice of source language. However, compiler correctness *is* possible for languages capable of producing critical software, as demonstrated by Leroy in this journal paper. Taking a significant subset of the C language (`Cminor`) as the source language, and targeting PowerPC assembly code, a Coq-verified compiler *Compcert* is presented. It is observed that the semantic preservation property of a compiler can be established by proving the *forward simulation for safe programs* property, which states that both the source and target programs produce the same 'set' of observable behaviours, with no behaviour classified as being 'incorrect'. We also require that the target language is deterministic, but since Compcert targets assembly code – deterministic by design –, only the forward simulation property requires attention.

The paper then discusses each stage of the compilation process using Compcert in detail. Each stage performs a different task, beginning with converting a source program from `Cminor` – for which a full semantics is given

– to `CminorSel`, a language making use of a processor-specific set of operators. Further stages address issues such as light optimisations (e.g. constant propagations) and register allocation. At each stage, semantic preservation is proved, and detailed semantics of intermediate languages (of which there are several) are given. The possibility of retargeting Compcert is also discussed, and shown to be possible by retargeting the ARM instruction set.

First-Class Syntax, Semantics, and their Composition, *Marcos Viera (2012)* [Vie12]: When describing extensible programming languages, a frequently used technique is that of *attribute grammars*, a formalism proscribing attributes to grammar productions, producing an abstract syntax tree with attribute information in the nodes [Joh87]. These attributes can be either *inherited* (passed down a tree, such as environments), or *synthesised* (passed up a tree, such as previously computed results). Viera's PhD thesis shows how a programming language can be implemented by composing attribute grammar fragments corresponding to individual aspects of the desired language, with the type system of the implementation language – Haskell – guaranteeing that conflicting compositions are not permitted. Parsers for these languages can be constructed on-the-fly in a similar manner. Throughout, it is shown how a compiler can be incrementally constructed for the imperative language Oberon0 [Mö93].

CoSy Compiler Phase Embedding with the CoSy Compiler Model, *Martin Alt, Uwe Aßmann and Hans van Someren* [AAvS94]: The desire to avoid handcrafting compilers for increasingly complex (and parallel) industrial-strength compilers led Associated Computer Experts bv. to embark in 1990 on the CoSy project, aiming to produce a model enabling the automatic generation of easily-targetable and flexible compilers from a high-level specification. The CoSy model does this by way of modular *engines* (compiler phases) that can be embedded into a compiler without recompiling existing source code. Engines are specified using three formalisms written in ISO C; namely the *Engine Description Language* (EDL), that describes the interactions between engines and the side-effects that engines have on the intermediate representation, the *Structure Definition Language* (SDL), which defines the common IR structures, and the *Compiler Configuration Language*, which assigns CoSy processes — such as engines — to the resources of the compiler-host. These specifications ultimately produce a *Data Manipulation and Control Package* (DMCP) that is then compiled and linked with the *Engine Control Package* (ECP), a package that implements CoSy facilities on the compiler host. This paper (one of nine currently available which are attributable to ACE) describes the overall process at a high-level, and demonstrates how newly introduced engines can be embedded into various compiler contexts such as parallel contexts and generate-and-test. However, at present the CoSy compiler is proprietary, and as it is written in C, may encounter significant difficulties if attempts to prove engines correct are desired, in contrast with the work and goals that are presented within this thesis.

Language Extension via Dynamically Extensible Compilers, *Sean Seefried* [See06]: Seefried's doctoral thesis approaches the issue of language extensibility by proposing *plug-in extensions*: libraries that either extend the syntax or semantics of a language, in much the same way as plug-in utilities are widely accepted and used in applications such as web browsers and text editors. More specifically, plug-ins are classified in two ways; *front-end* – improving upon the safety or expressivity of a dynamically extensible language – and *back-end* – improving code generation or optimisation – or alternatively *lengthening* – introducing new phases to the compiler – or *widening*, which adds to the functionality of existing compiler phases.

The thesis goes on to contrast the plug-in approach to that of meta-programming, concluding that whilst they are complementary, the barrier to extending a meta-programming language is greater than that of interfacing with the API of a plug-in compiler: nonetheless, Chapter 3 of the thesis shows how Template Haskell can be used to optimise an EDSL, using the Pan image synthesisation library as an example. The core notion of plug-in optimisations rests on the requirements of a small, well-understood IR, a scripting framework for composition of a pipeline of optimisations, and a communication framework for plug-ins to communicate: the latter is in contrast to the approach we present here, as we seek to completely avoid interaction between individual effects for the purpose of aiding correctness.

The implementation of plug-ins makes use of a solution to the expression problem based on open datatypes, which we encounter shortly [LH06], and the remainder of the thesis presents PHRaC: a real-world implementation of a dynamically extensible Haskell compiler, and demonstrates how a front-end PHRaC plugin can add list comprehension support to a simple functional language. This work primarily differs from that presented here in that our focus is on introducing wholly new computational features to an existing language, whereas plug-in compilers are more concerned with both simplifying the usage of existing languages and adding greater stability to the overall compiler. Nonetheless, the overlap is significant, and may prove to be an interesting future research direction.

3.4.4 Modular Semantics

In this section, we survey papers which develop techniques which allow for a greater degree of control in language definition and effectful computation.

Modular Denotational Semantics for Compiler Construction, *Sheng Liang and Paul Hudak (1996)* [LH96]: Extending the author's previous work introducing monad transformers [LHJ95b], this paper aims to derive compilers from modular interpreters which are based on modular monadic semantics, which links to the work presented here in that the compilers we produce for an individual effect are driven by the semantics of said effect within a source language. Firstly, examples are given defining the standard semantics of a number of language features in terms of effectful operations, referred to here as *primitive monadic combinators*.

It is observed that since a modular monadic semantics is no more than a structured denotational semantics, the monad laws described earlier can be used to β-reduce – and optimise – source programs. However, it is noted that program fragments utilising primitive monadic combinators such as `inEnv` *cannot* be β-reduced as they do not feature in the monad laws. To overcome this issue, the environment monad is axiomatised via four laws (e.g. inner environments supercede outer environments) and these are used to further simplify programs.

Lifting of Operations in Modular Operational Semantics, *Mauro Jaskelioff (2009)* [Jas09]: Jaskelioff's doctoral thesis addresses the lifting of operations through monad transformers in a uniform manner. The thesis begins by explaining why this is necessary, identifying a number of issues with the current monad transformer framework of Haskell [Gil14], such as the shadowing of operations wherein two applications of the same monad transformer result in one transformer being inaccessible.

Operations are defined to be mappings on the *category of monoidal categories*, and classified as either *H-algebraic*, *first-order* or *algebraic* depending on their structure. It is then demonstrated how several examples of such operations can be lifted in a canonical manner, and where several such liftings are possible, that they coincide. Key to this is the classification of a monad transformer as being *monoidal*, *functorial* or *covariant* (or potentially none of these).

The theory developed is then implemented in Haskell in the form of the *Monatron* library [Jas11]. Monatron differs from the *mtl* library in that it distinguishes between multiple classes of monad transformer (whilst this is a 'cleaner' treatment of monad transformers than that used within this thesis, we have chosen not to make use of it here as it would significantly raise the barrier to entry in terms of constructing new effect instances). Detailed examples are given, most notably for the exceptions monad. Here, the associated methods `throw` and `handle` are separated and lifted using the appropriate techniques for their algebraic class. Monatron is used extensively throughout the final two chapters, firstly in developing a modular interpreter for a language supporting processes, conditional arithmetic and exceptions using the à la carte technique, and secondly in implementing a modular operational semantics based upon work by Turi [TP97].

Semantic Lego, *David Espinosa (1995)* [Esp95]: Espinosa's doctoral thesis presents *Semantic Lego* (SL), a "language for describing languages" – implemented in Scheme – designed to build interpreters from component specifications. The theory of stratification (splitting a monad into 'levels') is introduced: for example, the monad $\mathtt{T}_1(\mathtt{A}) = \mathtt{S} \rightarrow \mathtt{A} \times \mathtt{S}$ is considered to be on a higher level than $\mathtt{T}_2(\mathtt{A}) = \mathtt{A} \times \mathtt{S}$ due to the extra argument of type `S`. Stratified levels can then be combined using *stratified monad transformers*, classified as either *top*, *bottom* or *around* transformers according to their structure; for example, the bottom exception transformer $\mathtt{F(T)(A)} = \mathtt{T(A} + \mathtt{X)}$, and the top environment transformer $\mathtt{F(T)(A)} = \mathtt{Env} \rightarrow \mathtt{T(A)}$.

SL divides a language semantics into two parts, a *computation* ADT and a *language* ADT. The former represents the actual semantics, the latter grammatical syntax, and it is the former which is implemented using stratification. An SL interpreter for a Scheme-like language is developed, with examples of the resulting semantics when considering the various orderings of nondeterminism and continuations. As with the previous paper, whilst the approach given here is technically elegant, the design decision was made to keep with the more accessible *mtl* approach to monad transformers for the purposes of this thesis.

A Modular Monadic Action Semantics, *Keith Wansborough and John Hamer (1997)* [WH97]: Action semantics [Mos96] and modular monadic semantics (MMS) [Lia98] are two existing approaches to defining semantics in a modular manner. These two semantic approaches differ in their underlying formalisms, and both styles suffer from issues regarding extensibility and accessibility: an action semantics cannot easily be extended to describe new features (inherently limiting it's usefulness within this thesis), and MMS uses syntax which can be confusing to the layman. This paper proposes merging the best of both styles, resulting in *modular monadic action semantics* (MMAS): an action semantics wherein combinators are described via MMS as opposed to SOS.

This change enables the extensibility that action semantics lacks: if a feature cannot be described with action notation (e.g. continuations) then we implement that feature via MMS instead. As MMS is itself no more than a structured denotational semantics, the usual methods for reasoning about programs – e.g. β-reduction – therefore apply. However, unaddressed issues remain: the underlying type-class system is that of Haskell, rather than the unified algebra foundations underpinning action semantics [Mos89], and some operations have their semantics altered as a result - for example, MMAS provides a 'lesser' implementation of non-deterministic choice than its action semantic counterpart.

3.4.5 Extensible Constructs

In this final section of our literature review, we present related work on the topic of defining programming language constructs in a manner that – by construction – renders them easy to extend.

Monads, Zippers and Views: Virtualising the Monad Stack, *Tom Schrijvers and Bruno C. D. S. Oliveira (2011)* [SO11]: As we have observed, a number of problems exist when lifting operations through a concrete monad stack. One problem we have not yet mentioned is that modifying the stack may result in the need to alter existing code to ensure that invocations to the `lift` operation refer to the correct transformers. This paper describes two techniques for 'virtualising' a monad which has been constructed using transformers: namely, the notions of *monad zippers* and *monad views*, which generalise to create *structural* and *nominal masks*. A monad zipper is a transformer which ignores a prefix of a monad stack using operations $\uparrow$ – a '`lift`' of sorts – and $\downarrow$, an inverse which has no typical equivalent. These operations allow methods associated with a monad transformer to be called without using `lift` to bypass other instances of the same transformer. A monad view is a monad *morphism* which allows for different versions of the monad stack to be presented to distinct parts of a computation. This in particular may allow for the issue of non-commutative effects presented in Chapter 5.4.2 of this thesis to be addressed without the need to further refine type-class instances in 'non-standard' cases.

Together, these notions permit the construction of a masking language which allows parts of a monad stack to either be hidden or have their access restricted at a point of usage. Structural masks are defined using primitives $\square$ and $\blacksquare$. To demonstrate, the mask $(\square ::: \blacksquare ::: \square)$ hides the second layer of the monad stack. Using different combinations of structural masks, several permutations of a single concrete stack can be presented when needed, eliminating the need for lifting. Nominal masks, in contrast, 'tag' each layer of a monad stack with a singleton type, distinguishing between multiple instances of transformers without the usage of `lift`.

Datatypes à la Carte, *Wouter Swierstra (2008)* [Swi08]: this paper describes a solution to the 'expression problem' [Wad98] – described in Chapter 2 – and forms a crucial part of the foundations for this thesis, and will be explored in depth in Chapter 4 – using techniques for constructing datatypes and functions in a modular manner by combining several previously known results. The leading example is an evaluation function over a simple language consisting of integers and addition. The core idea is that a datatype can be represented as the least fixpoint of a coproduct of functors – which represent constructor signatures –, and functions over such datatypes can be represented as categorical algebras.

Following a discussion of how certain well known monads (e.g. identity and `Maybe`) are free monads – essentially trees parameterised over a functor –, the paper demonstrates how both the state monad and certain aspects of the IO monad can be emulated in the à la carte style by implementing a simple calculator with memory for storing and recalling values.

Open Data Types and Open Functions, *Andres Loh and Ralf Hinze (2006)* [LH06]: This paper presents an alternative approach to solving the expression problem in Haskell, with the key mechanism being that of a novel syntactic extension to Haskell, in contrast to the à la carte technique which makes use of existing language features and compiler pragmas. In this approach, datatypes and functions that we may yet extend with new constructors or patterns are flagged with the **open** keyword. Extensions to either functions or datatypes can then be included *anywhere* in a source program, not just at the point of definition, with the final definitions grouped together at compile-time by a Haskell preprocessor.

One consequence of the fact that open functions can have their patterns dispersed throughout a source program is that the usual first-fit approach that Haskell takes to pattern matching is no longer appropriate. For example, if the default case for an open function appears at the top of a source file, the function is effectively closed, with no other definitions reachable. In response to this, open functions make use of *best-fit left-to-right* pattern matching instead (in which the most specific match is the one chosen, with wildcards less specific than explicit metavariables etc), rendering the order in which open function definitions appear irrelevant. The degree to which the work presented within this thesis overlaps with this paper will be an interesting research direction when the latter is integrated into GHC.

Parametric Compositional Datatypes, *Patrick Bahr and Tom Hvitved (2011)*: The aforementioned datatypes à la carte technique presents a particularly elegant solution to the expression problem in Haskell. However, the insights it reveals are of limited applicability outside the setting of the problem itself. Previous work by the authors of this paper [BH11] has sought to extend the *à la carte* approach in the form of *compositional datatypes* (CDTs). The CDT framework supports wider functionality for recursion and mutually recursive datatypes. However, CDTs cannot implement variable bindings due to issues concerning α-equivalence.

This issue is resolved by merging the à la carte technique with a parameterised variant of *higher order abstract syntax* (HOAS). HOAS represents variable bindings in a source language by using the binding mechanism of the metalanguage. However, this combination of methods cannot support recursion over abstract syntax trees, as data constructors which are defined in this way are not functors. The solution posed by this paper involves changing the signature representation to that of *difunctors* [MH95], and altering the structure of terms to that of a free monad. This combination of compositional datatypes and HOAS forms a technique referred to as *parametric compositional datatypes* (PCDTs).

The paper shoes how monadic computations can be implemented using both CDTs and PCDTs, before discussing *term algebras* – algebras defined with carrier `Fix f` for some signature `f` – and *term homomorphisms*, functions of type ∀ `a. f a` → `Context g a` for some free monad `Context`. It is then demonstrated how these algebras and homomorphisms can compose, and concludes by introducing *monadic* term homomorphisms and shows how mutually recursive datatypes and generalised algebraic datatypes can be constructed using PCDTs.

Extensible Effects: An Alternative to Monad Transformers, *Oleg Kiselyov (2013)* [KSS13]: The monad transformer framework is not without its issues, as we have alluded to when discussing the work of Jaskelioff above and will explore in detail in Chapter 5. An alternative approach developed around the time that monad transformers were introduced is that of *extensible denotational language specifications* (EDLS) [CF94], wherein an effect is viewed as a request to an external, inflexible 'authority' which interprets the request and either permits it and returns an appropriate continuation, or declines and aborts.

This paper by Kiselyov improves upon both the monad transformer and EDLS approaches by introducing an extensible 'bureaucracy' as part of the user program which consists of partial authorities (which correspond to algebraic handlers) that can either execute a request for pass the request to the appropriate authority. In conjunction with this, a type-and-effect system is introduced as an open union of effects: a handler will remove its associated effect from the union during execution in order to both accurately track the current state of a program, and guarantee that no 'dangling effects' exist in a program.

3.5 Justifying Our Position

In light of the related work that we have presented above, a summary of the positions and justifications taken within this thesis is desirable. To this end, we present the following:

- Monads, along with several techniques for simplifying their combination, are discussed heavily throughout. Despite the fact that the mathematical fundamentals of these techniques are often more precise than the existing status quo (the *mtl* library), we make use of the latter for the simple reason that it is both more the most widely understood way to simulate effects, and keeps the barrier for understanding at a reasonable level.
- We will represent the target intermediate representation as a simple stack for the purposes of clarity. Again, this is done for the purposes of understanding, however there is no explicit barrier preventing this work from targeting a more complex model such as RTL.

- An implicit goal of this thesis is to continue the work on splitting compilers into small, isolated pieces in the style of Hutton, with the future goal of individual correctness proofs, and we envisage Idris as being a particularly amenable host language for this purpose.

Chapter 4

Modular Compilers: Initial Steps

Now that we are familiar with the problem domain, we can begin to put together an initial framework that allows for the specification of compilation schemes between tree-based source languages and stack-based target languages, with each such scheme independent to the others by design. In this chapter, we will discuss how this can be done in Haskell by exploiting the typeclass system, and also how functors – as a core example of the typeclass system in action – allow for language syntax to be defined in a modular manner. We begin this chapter, however, with a discussion of how Haskell makes use of *multiple* side-effects within a single monad.

4.1 Modular Effects

In the previous chapter, we saw an example of the concept of computational effects being modelled using monads. Whilst each monad normally corresponds to a single effect, since most languages support more than one effect the issue of how to combine monads quickly arises. In this section, we briefly review the approach based upon *monad transformers* [LHJ95b].

4.1.1 Monad Transformers in Haskell

In Haskell, monad transformers have the following definition:

```
class (Monad m, Monad (t m)) => MonadT t where
  lift :: m a -> t m a
```

Intuitively, a monad transformer is a type constructor `t` which, when applied to a monad `m`, produces a *new* monad (`t m`). Monad transformers are required to satisfy two laws:

1. $\mathtt{lift} \circ \mathtt{return}_{\mathbf{M}} = \mathtt{return}_{(\mathbf{T\ M})}$
2. $\mathtt{lift}\ (\mathbf{m}\ \mathtt{>>=}\ \mathbf{k}) = \mathtt{lift}\ \mathbf{m}\ \mathtt{>>=}\ (\mathtt{lift} \circ \mathbf{k})$

The `lift` operation associated with every monad transformer is used to convert values in the base monad `m` to the new monad (`t m`). By way of example, the following table summarises five common computational effects, their monad transformer types, and corresponding implementations:

Effect	**Transformer Type**	**Implementation**
Exceptions	`ErrorT m a`	`m (Maybe a)`
State	`StateT s m a`	`s → m (a, s)`
Environment	`ReaderT r m a`	`r → m a`
Logging	`WriterT w m a`	`m (a, w)`
Continuations	`ContT r m a`	`(a → m r) → m r`

The general strategy is to 'stratify' the required effects, by starting with a base monad – often the `Identity` monad – and applying the appropriate transformers. There are some constraints regarding the ordering; for example, certain effects can only occur at the innermost level and certain effects do not commute [LHJ95b] (topics which we discuss in detail in Chapter 5.4.1), but otherwise effects can be ordered in different ways to reflect different intended interactions between the features of the language.

To demonstrate the concept of monad transformers, we will examine the transformer for exceptions in more detail. Firstly, its type constructor is declared in the following manner:

```
newtype ErrorT m a = E { runE :: m (Maybe a) }
```

Note that (ErrorT Identity) is equivalent to the Maybe monad. We can now declare ErrorT as a member of both the Monad and MonadT classes:

```
instance Monad m => Monad (ErrorT m) where
  return a    = E $ return (Just a)
  (E m) >>= f = E $ do v <- m
                       case v of
                        Nothing -> return Nothing
                        Just a  -> runE (f a)


instance MonadT ErrorT where
  lift m = E $ m >>= \v -> return (Just v)
```

In addition to the general monadic operations, we would also like access to other primitive operations related to the particular effect that we are implementing. In this case, we would like to be able to throw and catch exceptions, and we can enable this by having the relevant operations defined as methods of an *error monad class*:

```
class Monad m => ErrorMonad m where
  throw :: m a
  catch :: m a -> m a -> m a
```

We instantiate ErrorT as a member of this class as follows:

```
instance Monad m => ErrorMonad (ErrorT m) where
  throw = E $ return Nothing
  x ‘catch‘ h = E $ do v <- runE x
                       case v of
                        Nothing -> runE h
                        Just a  -> return v
```

More importantly, we can also declare monad transformers to be members of effect classes *other* than their own. Indeed, this is the primary purpose of the lift operation. For example, consider the state monad transformer:

```
newtype StateT s m a = S { runS :: s -> m (a, s) }

instance Monad m => Monad (StateT s m) where
  return x    = S $ \s -> return (x, s)
  (S g) >>= f = S $ \s -> do (x, t) <- g s
                             runS (f x) t

instance MonadT (StateT s) where
  lift m = S $ \s -> m >>= \x -> return (x, s)
```

We can extend `StateT` such that it supports exception handling by instantiating it as a member of the error monad class as follows:

```
instance ErrorMonad m => ErrorMonad (StateT s m) where
  throw = lift . throw
  x `catch` h = S $ \s -> runS x s `catch` runS h s
```

In this manner, a monad that is constructed from a 'base' monad using a number of transformers comes equipped with the associated operations for all of the constituent effects with necessary liftings handled automatically, with the caveat that they have to be manually implemented for any combinations one wishes to use, which can lead to boilerplate.

4.2 Modular Syntax & Semantics

The previous chapter made the observation that adding extra constructors to a datatype often requires the modification of existing code. In this section, we will first review the modular approach to datatypes and functions put forward by Swierstra [Swi08], known as the *datatypes à la carte* technique, and show how it can be used to obtain modular syntax and semantics for our simple expression language.

4.2.1 Datatypes À La Carte

The underlying structure of an algebraic datatype in Haskell – such as `Expr` from Chapter 2.1 – can be captured by a constructor signature. We can define distinct *signature functors* for the arithmetic and exceptional components of the `Expr` datatype as follows:

```
data Arith  e = Val Int | Add e e

data Except e = Throw   | Catch e e
```

These definitions capture the non-recursive aspects of expressions, in the sense that `Val` and `Throw` have no subexpressions, whereas `Add` and `Catch` have two apiece. We declare both `Arith` and `Except` as Haskell functors:

```
instance Functor Arith where

  fmap _ (Val n)     = Val n

  fmap f (Add x y)   = Add (f x) (f y)


instance Functor Except where

  fmap _ Throw       = Throw

  fmap f (Catch x h) = Catch (f x) (f h)
```

For any functor `f`, its induced recursive datatype, `Fix f`, is defined as the *least fixpoint* of `f`. In Haskell, this is implemented as follows [MH95]:

```
newtype Fix f = In (f (Fix f))
```

For example, `Fix Arith` is the language of integers and addition, while `Fix Except` is the language comprising throwing and catching exceptions. We shall see shortly how these languages can be combined.

Of particular importance is the name of the data constructor associated with this least fixpoint construct, namely `In`, which we notice is the same name given to the morphism associated with a categorical initial algebra (recall Chapter 3.2) . This is no coincidence, as it is appears when defining the `fold` operator [MFP91] used to write functions over `Fix f` [Swi08]:

```
fold :: Functor f => (f a -> a) -> Fix f -> a
fold f (In t) = f (fmap (fold f) t)
```

The parameter of type (`f a` → `a`) is the `f`-algebra, which can be intuitively viewed as a directive for processing each constructor of a functor. Given such an algebra and a value of type `Fix f`, the `fold` operator exploits both functional and recursive characteristics of `Fix` to process recursive values.

The aim now is to take advantage of the above machinery to define a dynamic semantics for our expression language in a modular fashion. Such a semantics will have type `Fix f` → `m Value` for some functor `f` that captures the syntactic structure of the source language, monad `m` that captures the computational effects that are required by the source language, and semantic domain `Value` that captures the notion of fully evaluated results. To define functions of the requisite type using `fold`, we require an *evaluation algebra*, which notion we capture by the following typeclass definition:

```
class (Monad m, Functor f) => Eval f m where
  evalAlg :: f (m Value) -> m Value
```

Using this typeclass, we can now define algebras corresponding to the semantics for both the arithmetic and exception components:

```
instance Monad m => Eval Arith m where
  evalAlg (Val n)   = return n
  evalAlg (Add x y) = x >>= \n ->
                      y >>= \m -> return (n + m)

instance ErrorMonad m => Eval Except m where
  evalAlg (Throw)     = throw
  evalAlg (Catch x h) = x ‘catch‘ h
```

There are three important points to note about the above instantiations. First of all, the semantics for arithmetic have now been completely separated from the semantics for exceptions, in particular by way of two separate instance declarations. Secondly, the semantics are parametric in the underlying monad, and can hence be used in multiple differing contexts. And finally, the operations that the underlying monad must support are explicitly qualified by class constraints, e.g. in the case of `Except` the monad must be an `ErrorMonad`. These latter two points generalise the work of Jaskelioff [Jas09] from a fixed monad to an arbitrary one supporting the required operations, resulting in a clean separation of the semantics of individual language components.

With this machinery in place, we can now define a general evaluation function of the desired type by folding an evaluation algebra:

```
eval :: Eval f m => Fix f -> m Value
eval = fold evalAlg
```

Note that this function is both modular in the syntax of the source language and parametric in the underlying monad. However, at this point we are only able to take the fixpoints of `Arith` or `Except`, not both. We need a way to combine signature functors, which is naturally done by taking their *coproduct* (or disjoint sum) [LHJ95b], noting that at this point we only consider the potential for a single ‘recursive knot’, namely the variable `e`. In Haskell, the coproduct of two functors can be defined as follows:

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

```
instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap f (Inl x) = Inl (fmap f x)
  fmap f (Inr y) = Inr (fmap f y)
```

We can now define the coproduct of evaluation algebras:

```
instance (Eval f m, Eval g m) => Eval (f :+: g) m where
  evalAlg (Inl x) = evalAlg x
  evalAlg (Inr y) = evalAlg y
```

The general evaluation function can now be used to give a semantics to languages with multiple features by simply taking the coproduct of their signature functors. Unfortunately, there are three problems with this approach. First of all, the need to include fixpoint and coproduct tags (`In`, `Inl` and `Inr`) in values is cumbersome unless we have a parser available. For example, if we wished to manually enter the concrete expression `1 + 2` with type `Fix (Arith :+: Except)` (for smoke testing purposes, perhaps), it would be represented as follows:

```
In (Inl (Add (In (Inl (Val 1)) (In (Inl (Val 2))))))
```

Secondly, the extension of an existing syntax with additional operations (i.e. extending a language from type `Fix (Arith :+: Except)` to `Fix (State :+: Arith :+: Except)`) requires the modification of existing tags — `Inl` would need to be replaced by `(Inl . In . Inr)` for example — which breaks modularity. And finally, `Fix (f :+: g)` and `Fix (g :+: f)` are isomorphic as languages, but require different values to be tagged in different ways. In the next two subsections, we will describe how the *datatypes à la carte* technique resolves these concerns.

4.2.2 Smart Constructors

Given the concerns just described, it would be helpful to have a way of automating the injection of values into expressions such that the appropriate sequences of fixpoint and coproduct tags are prepended. This can be achieved using the concept of a *subtyping relation* on functors, which can be formalised in Haskell by the following class declaration, wherein `inj` injects a value from a subtype into a supertype.

```
class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a
```

We can now define instance declarations to ensure that `f` is a subtype of any coproduct containing `f`, via the following declarations:

```
instance f :<: f where
  inj = id
```

```
instance f :<: (f :+: g) where
  inj = Inl

instance (f :<: h) => f :<: (g :+: h) where
  inj = Inr . inj
```

We note that an overlap occurs between the second and third instances, however this is not a problem in practice so long as right associativity is guaranteed via the use of an `infixr` declaration on `(:+:)`.

Using this notion of subtyping, we can define an *injection function*:

```
inject :: (g :<: f) => g (Fix f) -> Fix f
inject = In . inj
```

We use `inject` to define *smart constructors* which bypass the need to tag values when embedding them in expressions (using the `SC` suffix to differentiate between smart constructors and methods of `ErrorMonad`):

```
val       :: (Arith :<: f) => Int -> Fix f
val n     =  inject (Val n)
```

```
add        :: (Arith :<: f) => Fix f -> Fix f -> Fix f
add x y    =  inject (Add x y)


throwSC      :: (Except :<: f) => Fix f
throwSC      =  inject (Throw)


catchSC      :: (Except :<: f) => Fix f -> Fix f -> Fix f
catchSC x h =  inject (Catch x h)
```

Note the constraints stating that `f` must have the appropriate subtype functor; for example, in the case of `val`, `f` must support arithmetic.

4.2.3 Putting It All Together

We can now define language syntax in a modular manner. Using smart constructors, we can define values within languages given as fixpoints of coproducts of signature functors. For example:

```
ex1 :: Fix Arith
ex1 =  val 18 `add` val 24


ex2 :: Fix Except
ex2 =  throwSC `catchSC` throwSC
```

```
ex3 :: Fix (Arith :+: Except)
ex3 =  throwSC ‘catchSC‘ (val 1337 ‘catchSC‘ throwSC)
```

The types of these expressions can be generalised using the subtyping relation, but for simplicity we have given fixed types above (the most general type we can obtain with no available information is `Eval f m => Fix f`). In turn, the meaning of such expressions is given by our modular semantics:

```
> eval ex1 :: Identity Value
> 42

> eval ex2 :: Maybe Value
> Nothing

> eval ex3 :: Maybe Value
> Just 1337
```

Note the usage of explicit typing judgements to determine the resulting monad. Whilst we have used `Identity` and `Maybe` above, *any* monad satisfying the required constraints can be used, as illustrated below:

```
> eval ex1 :: Maybe Value

> Just 42


> eval ex2 :: [Value]

> []
```

One final point to make in this chapter is that the modular abstract syntax we have introduced is currently *single-sorted*: that is to say, we cannot differentiate between the *purposes* of expressions. This is quite limiting, and we shall see how this can be generalised in Chapter 6.2.1.

4.3 Modular Compilation Algebras

With the techniques described in the previous two sections within this chapter, we can now construct a modular compiler for our expression language. The instructions corresponding to the arithmetic and exceptional parts of the virtual machine (and a modular 'empty list' construct) are:

```
data ARITH  e = PUSH Int e | ADD e
data EXCEPT e = THROW e    | MARK e e | UNMARK e
data NULL   e = NULL
```

In the above, by defining the `Op` datatype (see Chapter 2.1) as a fixpoint, we have combined `Op` and `Code` into a single type defined using `Fix`, allowing code to be processed using the datatypes à la carte technique.

The desired type for our compiler is `Fix f` → (`MCode` → `MCode`), where `MCode` is a (closed!) datatype characterising the syntax of the source language. To define such a compiler using the `fold` operator, we require an appropriate *compilation algebra*, which notion we define as follows:

```
type MCode = Fix (ARITH :+: EXCEPT :+: NULL)


class Functor f => Comp f where
  compAlg :: f (MCode -> MCode) -> (MCode -> MCode)
```

In contrast with evaluation algebras, no underlying monad is utilised in the above definition, because the compilation process itself does not involve the manifestation of effects within the program itself being compiled (compiler *implementations* are themselves often stateful, but we are concerned only with the effects that a source program invokes). We can now define algebra instances for both the arithmetic and exceptional aspects of our modular compiler in the following manner:

```
instance Comp Arith where
  compAlg (Val n) = pushc n
  compAlg (Add x y) = x . y . addc

instance Comp Except where
  compAlg (Throw) = throwc
  compAlg (Catch x h) = \c -> h c `markc` x (unmarkc c)
```

Similarly to the evaluation algebras defined in Chapter 4.2.1, note that these definitions are modular in the sense that the two language features are being treated completely separately from each other. We also observe that because the carrier of the algebra is a function, the notion of appending code in the case of the `Add` constructor corresponds to function composition.

The smart constructors `pushc`, `addc` and so on can be defined as follows:

```
pushc     :: Int -> Code -> Code
pushc n c =  inject (PUSH n c)

addc      :: Code -> Code
addc c    =  inject (ADD c)
```

The other smart constructors are defined similarly. Finally, we can now define a general compilation function of the desired type by folding a compilation algebra, with an initial accumulator `empty`:

```
comp    :: Comp f => Fix f -> MCode
comp e  =  comp' e empty
comp'   :: Comp f => Fix f -> (MCode -> MCode)
comp' e =  fold compAlg e

empty   :: MCode
empty   =  inject NULL
```

For example, applying `comp` to the expression `ex3 = throw ‘catch‘ (val 1337 ‘catch‘ throw)` produces the following `MCode`, in which the fixpoint and coproduct tags `In`, `Inl` and `Inr` have been removed for readability:

```
MARK (MARK (THROW NULL) (PUSH 1337 (UNMARK NULL)))
     (THROW (UNMARK NULL))
```

4.4 Towards Modular Machines

What remains to complete our framework at this stage is the construction of a modular virtual machine which can execute code produced by the modular compiler defined in the previous section. We note that whilst a non-modular (or *universal*) virtual machine is also a viable candidate to target, we wish to explore the degree to which our treatment of datatypes and functions can accommodate *all* phases of our framework. We can now redefine the underlying `Stack` datatype of Chapter 2.1 in a modular manner, however the boilerplate code required to implement the appropriate execution algebras quickly becomes prohibitive. For this reason, we delay the implementation of modular virtual machines until Chapter 7.

4.5 Chapter Summary

At the end of each chapter from this one forwards, we will conclude with a section describing the general state of affairs of the framework we are developing to tackle the problem domain (as presented in Chapter 3), as well as explicitly identifying those tasks which we deferred treatment of until 'later on in the thesis':

In this chapter we have:

- Identified how the syntax of an embedded language can be written in a modular manner via functors, coproducts and least fixpoints.
- Identified how functions can be written over such modular syntax by applying an appropriate algebra to a fixpoint using a catamorphism.

- Shown how the notion of a subtyping relation on signature functors allows for the definition of constructors which derive the correct placement of a value within a modular source expression.

- Described how the semantics of a modular language defined using these techniques can be defined polymorphically over any monad supporting the requisite constraints and typeclass memberships.

- Defined a compilation scheme for our modular source language, mapping into a predefined modular target language, an approach which breaks modularity, as extending the chosen source would require editing the definition of the chosen target. A more elegant treatment would be to allow the algebra to target any language which meets the minimum requirements in terms of supported instructions. We treat this topic in the next chapter.

Chapter 5

Modular Compilers: Further Refinements

At this point, we have established the fundamentals of our solution to the problem introduced in Chapter 3: we can define the syntax of a source language as a combination of signature functors describing the data constructors associated with particular computational features, we can define interpreters over languages which are parameterised over the requisite monad class needed to define their semantics, and we can define compilation schemes between the syntax of a source language and the instructions for a stack-based target language in an independent manner. In each of these cases, the underlying functorial representation can be exploited to combine multiple definitions into a single compound instance. The upshot of using this approach is that a new feature can be defined and given a semantics and compilation scheme before being folded into the existing definition with no need for recompilation of code.

At this stage however, the expressiveness of our source language is limited to arithmetic and exception handling, and we determine the target language which we compile into in advance. In this chapter we introduce modular variants of the untyped λ-calculus and mutable state, describe how we can generalise the language we compile into to a parametrically polymorphic variant, how non-standard instructions can be accommodated by making use of generalised algebraic datatypes (GADTs), and also how the combination of effects which can interact in noncanonical ways can be compiled into the appropriate instruction sets by way of three distinct parameterisation techniques. We begin with the most pressing issue at present, namely the generalisation of the target language for the compiler.

5.1 Abstracting Over Algebras

Recall the type of the compilation algebra presented in Chapter 4.3:

```
class Functor f => Comp f where

compAlg :: f (MCode -> MCode) -> (MCode -> MCode)
```

We observe that the type of the algebra for `compAlg` is an endofunction over the target language `MCode`, which is defined as the least fixpoint of a number of signature functors characterising instructions for the virtual machine. More generally, we know that this algebra is an endofunction *regardless* of the signature functors contained within a target language.

These observations suggests that our modular compiler need not target a particular language defined in advance (with alterations to this language requiring recompilation of the class), but rather *any* language which contains (at least) the appropriate instructions. The modular counterpart of the compilation function `comp` should have type `Fix f` → `Fix g` → `Fix g`, for signature functors `f` and `g` that characterise the syntax of the source and target languages respectively. Furthermore, to supply an initial value for the accumulator (the second argument), we require that `NULL` — corresponding to the empty code fragment or the 'empty list', to serve as an initial accumulator — is a component of `g`. Putting this all together, we redefine the compilation typeclass as follows:

```
class (Functor f, NULL :<: g) => Comp f g where
  compAlg :: f (Fix g -> Fix g) -> Fix g -> Fix g
```

We can now instantiate the compilation algebras for `Arith` and `Except` using the subtype relation to constrain the target signature functor `g` to any language which supports the required signatures (where `push`, `add`, `throw` etc. are smart constructors):

```
instance (ARITH :<: g) => Comp Arith g where
  compAlg (Val n)   = push n
  compAlg (Add x y) = x . y . add


instance (EXCEPT :<: g) => Comp Except g where
  compAlg (Throw) = throw
  compAlg (Catch x h) = \c -> mark (h c) (x (unmark c))
```

The resulting modular compilation function is obtained by folding the compilation algebra over an empty accumulator `emptyC` as follows:

```
emptyC :: (NULL :<: g) => Fix g
emptyC = inject NULL


comp :: Comp f g => Fix f -> Fix g
comp x = fold compAlg x emptyC
```

5.2 Generalised Algebraic Datatypes

During the course of implementing a new effect, it may arise that a instruction associated with its virtual machine signature functor does not fit the pattern seen thus far. For example, consider the following. Over the course of implementing the `StrongAI` effect, we find that we need to make use of an instruction `THINK` parameterised over a polymorphic argument `mind` that is a member of the `Consciousness` class:

```
data STRONGAI e = THINK mind e | ...
```

The immediate issue that arises is that `mind` is not in scope. The natural solution is to extend the `STRONGAI` signature as follows:

```
data STRONGAI mind e = THINK mind e | ...
```

However, turning `mind` into a parameter in this manner essentially means that every language that makes use of `STRONGAI` now needs to explicitly refer to `mind`, which breaks modularity. Moreover, the fact that `mind` is an instance of a typeclass requires the presence of a class constraint, which could be done as follows:

```
data Consciousness mind => STRONGAI e = THINK mind e | ...
```

Unfortunately, as of GHC version 7.2.1 this is no longer possible using the algebraic datatypes of Haskell[1]. The solution to the deprecation of this feature is to define those signatures which contain constructors requiring

[1]The pragma that allowed this – `-XDatatypeContexts` – was deprecated at this point, being widely considered a misfeature. The listed constraints were not actually enforced!

constraints (such as `THINK`) as *generalised algebraic datatypes* (GADTs), permitting individual constructors to be typed explicitly with their own class constraints. For example, consider the GADT representation of the *non-modular* variant of `Expr` as originally presented in Chapter 3:

```
data Expr e where
  Val   :: Int -> Expr Int
  Add   :: Expr Int -> Expr Int -> Expr Int
  Throw :: Expr e
  Catch :: Expr e -> Expr e -> Expr e
```

Note that this representation enables a level of type-safety that was previously unavailable. For example, consider the `Add` constructor, which dictates that only subexpressions which represent an `Int` can be added together. Whilst this type-safety is a desirable feature to have, we are primarily utilising GADTs to leverage existential types into our framework. By describing constructors as methods associated with a type, we can now impose constraints on individual constructors without affecting the datatype as a whole (and therefore achieve what the `-XDatatypeContexts` pragma was designed to do). Using this idea, the signature functor for `STRONGAI` can be redefined as follows:

```
data STRONGAI e where
  THINK :: Consciousness mind => mind -> e -> STRONGAI e
  ...
```

By using GADTs to define source signatures, we grant ourselves the flexibility to constrain arguments to individual data constructors without including this constraint in the top-level definition (and thereby constraining the other constructors similarly).

In the next section, we show how to extend the modular source language with support for variable binding, and in particular the role that GADTs play in defining its modular semantics.

5.3 The Untyped λ-Calculus

The ability to abstract over variables in the body of a function is a near-universal feature in programming languages, and in this section we will introduce variable binding into our modular framework using the untyped lambda calculus of Church [Chu36]. We will begin by formally introducing the notation and reduction-rules of the lambda calculus, before moving on to our treatment of a modular representation within our framework.

5.3.1 A Formal Treatment Of The λ-Calculus

Initially constructed by Alonzo Church in 1932 as a model of effective computability [Chu32], the first variant of the λ-calculus concerning the foundations of mathematics was proven logically inconsistent with Curry's combinatory logic [Cur30] via the Richard paradox by the Kleene-Rosser paradox [KR35] in 1935.

As a result, the portion of the λ-calculus solely devoted to computation was isolated and published seperately in 1936 [Chu36], and is today referred to as the untyped λ-calculus. Whilst a typed variant was produced in 1940 [Chu40], this thesis considers only the former.

Definition Of The Untyped λ-Calculus

Untyped lambda expressions are constructed from the following:

- Variables v_1, v_2, v_3, $\ldots$
- The terminal symbols λ and $(.)$.
- Left and right parentheses `(` and `)`.

The set of all lambda expressions Λ is defined inductively:

1. If v is a variable, $v \in \Lambda$.

2. If v is a variable and $\mathbf{M} \in \Lambda$, then $(\lambda v\ .\ \mathbf{M}) \in \Lambda$.

3. If $\mathbf{M}$ and $\mathbf{N} \in \Lambda$, then $\mathbf{(M\ .\ N)} \in \Lambda$.

A variable associated with a λ symbol is said to be *bound* if it appears within the body of an abstraction (an *anonymous function*), and the `(.)` symbol is notation for function application. For example, in the lambda expression $(\lambda$`x. x y x)`, `x` is bound. In contrast, those variables that are not bound are said to be *free.* The set of free variables of an expression `M`, denoted $\mathbf{FV}(\mathtt{M})$ is defined recursively over the structure of the expression:

- $\mathbf{FV}(v) = \{\ v\ \}$, where v is a variable.

- $\mathbf{FV}(\lambda v\ .\ \mathtt{M}) = \mathbf{FV}(\mathtt{M}) \setminus \{\ v\ \}$

- $\mathbf{FV}(\mathtt{M}\ .\ \mathtt{N}) = \mathbf{FV}(\mathtt{M}) \cup \mathbf{FV}(\mathtt{N})$

An expression `M` containing no free variables (i.e. $\mathbf{FV}(\mathtt{M}) = \emptyset$ is *closed.* For the purposes of this thesis, closed expressions are particularly important as they represent programs which can be fully evaluated.

At this point, we must recognise that lambda expressions are given meaning by the *way* in which they are evaluated. There are multiple notions of reduction and conversion which can be applied when manipulating said expressions, but for our purposes we focus on three in particular, namely α-conversion, β-reduction, and term substitution.

α-Conversion

Also known as α-renaming, the process of renaming the bound variables of a lambda expression in a manner producing an expression describing the same function is referred to as α-conversion. For example, the expression `(λx.λy. x z y)` can be α-converted to `(λa.λb. a z b)`. Note that care must be taken to avoid *name capture*, e.g. renaming `(λx. x y)` to the *different* expression `(λy. y y)`.

β-Reduction

The application of an argument to a function is commonly known as a β-*reduction*. More generally, we often think of such a reduction as a single computational step. For example, given the expression `(λx.λy. x + y)` (assuming both `x` and `y` represent integers, with addition defined in the usual manner), its application to the argument `40` can be β-reduced to `(λy. 40 + y)`. Importantly, we observe that β-reduction is defined in terms of *capture-free substitution*, as shown in the next section.

Term Substitution

Substitution, an operation which we denote as `E[V := R]`, is the replacement of all *free* instances of the variable `V` in the expression `E` with the expression `R`. Substitution over a λ-expression is defined recursively (below, x and y are variables, and $\mathbf{M}$ and $\mathbf{N}$ are metavariables over λ-expressions):

- $x\,\texttt{[}x\ \texttt{:=}\ \mathbf{N}\texttt{]} \equiv \mathbf{N}$
- $y\,\texttt{[}x\ \texttt{:=}\ \mathbf{N}\texttt{]} \equiv y$ if $y \neq x$
- $(\mathbf{M}_1\ .\ \mathbf{M}_2)\ \texttt{[}x\ \texttt{:=}\ \mathbf{N}\texttt{]} \equiv (\mathbf{M}_1\texttt{[}x\ \texttt{:=}\ \mathbf{N}\texttt{]}\ .\ \mathbf{M}_2\texttt{[}x\ \texttt{:=}\ \mathbf{N}\texttt{]})$
- $(\lambda x\ .\ t)\ [x\ \texttt{:=}\ r] \equiv (\lambda x\ .\ t)$
- $(\lambda y\ .\ t)\ [x\ \texttt{:=}\ r] \equiv \lambda y\ .\ (t\ [x\ \texttt{:=}\ r])$ if $x \neq y$, $y \notin \mathbf{FV}(r)$

As the latter two cases above indicate, we need to be certain that we are not going to substitute in a term containing a variable that is already bound by the term being substituted into (referred to as *capturing* a variable). This is undesirable, as doing so can change the semantics of a lambda expression.

Capture-Free Substitution

One potential solution [2] for performing substitution in a capture-free manner lies in renaming the conflicting bound variables from the term being

[2]We will encounter other solutions shortly.

substituted to unique identifiers, thereby assuring that the substitution does not capture any existing variable. By α-converting the relevant variables in such a way that fresh names are used where needed, we can reformulate the case for substituting into an abstraction as follows:

- $(\lambda y\ .\ t)\ [x\ := r] \equiv (\lambda y\ .\ t)$

  if $x \equiv y$

- $(\lambda y\ .\ t)\ [x\ := r] \equiv \lambda y\ .\ (t\ [x\ := r])$

  if $x \not\equiv y \wedge y \notin \mathbf{FV}(r)$

- $(\lambda y\ .\ t)\ [x\ := r] \equiv (\lambda z\ .\ (t\ [y\ := z]))\ [x\ := r]$

  if $x \not\equiv y \wedge z \notin \mathbf{FV}(r) \wedge z \notin \mathbf{FV}(t)$

The notion of β-reduction can now be defined simply in terms of capture-free substitution via the equation $(\lambda x\ .\ t)\ u \equiv t[x\ := u]$.

One may wonder why we are going through such pains to describe substitution in a capture-free manner. Whilst it is indeed important that the substitution operation is semantically sound, we do so here to illustrate the fact that there are implementation issues that require additional boilerplate code to resolve, particularly when managing variable names. An implementation that avoids these concerns is preferable, and it is this point which justifies the model that we use within our modular framework.

5.3.2 A Modular λ-Calculus

Although the variables used for abstraction purposes in lambda terms are often given names in the same way that we would name other variables, there are many alternative ways to model bindings, including such approaches as *higher order abstract syntax* (HOAS) [PE] and de Bruijn indices [dB72], amongst others. The HOAS approach approach uses the binders of Haskell to describe the binding structure of the language being implemented, via a datatype such as:

```
data LambdaH e = App e e | Lam (e -> e)
```

The argument to the `Lam` constructor in the above explicitly prevents `LambdaH` from being an instance of the `Functor` typeclass, as we cannot apply the `fmap` method to this argument in a sensible manner. More formally we observe that the `Lam` variable `e` appears in both a covariant and contravariant position; this point is discussed in finer detail in [MH95]. We can still define catamorphisms over `LambdaH` as a *difunctor* by using a more refined fold operator; however this adds significant amounts of both complexity and boilerplate to the underlying technique. As such, for simplicity's sake – alongside the desire to avoid implementing capture-free substitution – in this thesis we use a *de Bruijn indexed* encoding of the lambda calculus. In this encoding, the syntax of lambda expressions is defined in the following manner:

```
data Lambda e = Index Value | Abs e | Apply e e
```

The `Value` type associated with an `Index` constructor represents *some* datatype that gives rise to an integer constructor that can represent a variable, where the integer refers to the number of lambda operators in scope [3] before its binding site (note that we have previously encountered `Value` in Chapter 4.2: we state at this point that this type should be modular to account for the potential for other features being included, but we do not do so here for ease of explanation). The `Abs` constructor indicates the presence of a binder, and `Apply` represents the substitution of lambda terms, and is passed both a function body and its argument as subexpressions.

However, by choosing not to use the HOAS approach, an issue arises when considering the `Apply` constructor. Specifically, when defining a modular semantics for the `Lambda` signature, the carrier of the evaluation algebra determines that both subexpressions would be typed `(m Value)`, a problem avoided by HOAS by only deeming valid those expressions wherein the first subexpression is a `Lam` constructor. The following attempt at defining the evaluation algebra illustrates the problem:

```
instance Monad m => Eval Lambda m where
  -- evAlg :: Lambda (m Value) -> m Value
  evAlg (Apply f x) = f >>= \f' -> ...
```

[3]An interesting potential research direction is the usage of the type system to ensure scope correctness for terms constructed in this way.

The definition of `Apply` cannot be completed in a sensible way, because the semantic domain is not *expressive* enough. In particular, the result of binding the function body `f` has the primitive type `Value` which accepts no arguments (whereas the `Lam` constructor of `LambdaH` is typed (`e` $\rightarrow$ `e`)). Moreover, binding the result of `f` breaks the abstraction that a function body represents.

Our solution to this issue is to extend the semantic domain `Value` with support for *closures*: pairs consisting of functions and environments defining their non-local variables. To do this, we define `Value` as a GADT [4] here rather than as a modular datatype simply to avoid boilerplate:

```
data Value m where
  Num  :: Int -> Value m
  Clos :: Monad m => [Value m] -> m (Value m) -> Value m
```

Above, the `Num` constructor represents an integer value, and the `Clos` constructor takes as arguments a list of values (i.e. an environment) and a computation which represents an unevaluated function body. There are two points to note about this definition. Firstly, we would not be able to represent closures in this way without the `(Monad m)` constraint (as we need to suspend the evaluation of the function body within a monad until

[4]A second usage of GADTs within our framework, and the one likely to see more usage in practice.

required) and secondly, this constraint is only imposed within the `Clos` constructor, meaning that those subexpressions that do *not* make use of lambda expressions can safely use `()` as the parameter to `Value` rather than a monad which is not used.

To make use of closures when giving a semantics to the lambda calculus, we define a class `CBVMonad` of operations associated with the *call-by-value* evaluation scheme, which reduces arguments to values prior to their use:

```
class Monad m => CBVMonad m where
  env  :: m [Value m]
  with :: [Value m] -> m (Value m) -> m (Value m)
```

Intuitively, the `env` operation provides the list of values that are currently in scope, and the `with` operation takes both an associated environment and a computation, returning the result of performing substitution. We can now give a semantics to the λ-calculus signature, using the `CBVMonad` class constraint to allow the use of the `env` and `with` methods as follows:

```
instance CBVMonad m => Eval Lambda m where
  evAlg (Index i)   = do e <- env
                         return (e !! i)
  evAlg (Abs t)     = do e <- env
                         return (Clos e t)
```

```
evAlg (Apply f x) = do (Clos ctx t) <- f
                       c            <- x
                       with (c:ctx) t
```

In the above, a de Bruijn index is evaluated by looking up the index in the current environment, a lambda abstraction is packaged up with the current environment to form a closure, and substitution of lambda expressions is performed by evaluating argument `x`, adding this value to the environment of the closure associated with the function body, and finally evaluating the function body with respect to this updated environment. Implicit in the above is that all lambda expressions we define must be *closed*, as permitting open terms would require that we also keep track of the largest de Bruijn index at a given point in an expression.

We can now write expressions in our modular source language that make use of variable binding. For example, consider the following example (where `apply`, `abs`, etc. are the appropriate smart constructors, and `Identity` is used in order to permit the do-notation in the evaluation algebra for the `Lambda` subsignature):

```
e :: Fix (Lambda :+: Arith)
e = apply (abs (ind 0)) (add (val 1) (val 2))
```

```
> eval e :: [Value Identity]

[Num 3]
```

The source language used in this example is capable of using both variable binding and arithmetic. The expression `e` represents the lambda expression `(λx . x)(1 + 2)`, and evaluating `e` with respect to (for example) the list monad, which can readily be made into an instance of the `CBVMonad` class, returns the singleton value `Num 3`.

We can also define multiple evaluation schemes for terms within the lambda calculus. A common alternative to call-by-value is *call-by-name*, which does not evaluate arguments before applying them to a function body. The difference between this and call-by-value is that environments contain *computations*, not values. Another class, `CBNMonad`, is needed:

```
class Monad m => CBNMonad m where

  env  :: m [m (Value m)]

  with :: [m (Value m)] -> m (Value m) -> m (Value m)
```

Constraining by this class allows a call-by-name semantics to be defined for the lambda calculus signature as follows:

```
instance CBNMonad m => Eval Lambda m where
  evAlg (Index i)   = do e <- env
                         (e !! i)
  evAlg (Abs t)     = do e  <- env
                         e' <- sequence e
                         (Clos e' t)
  evAlg (Apply f x) = do (Clos ctx t) <- f
                         with (x : map return ctx) t
```

This definition is similar to that for call-by-value evaluation, the main difference being that the substitution of terms does not bind the argument `x` to a value prior to using it.

We have presented two separate evaluation algebras, both defined over a signature `Lambda`. However, despite the differing contexts, Haskell does not permit the two algebras to coexist in the same source file, stating that they are overlapping instances: GHC does not consider differing class constraints a sufficient distinction between two instances as to suggest uniqueness. One possible solution is to define two source signatures `LambdaCBV` and `LambdaCBN` which contain appropriately tagged constructors to avoid naming conflicts. An alternative involves parameterising the evaluation algebra class with a tag that can be pattern-matched upon, and we will see this idea in action in the final section of this chapter when discussing a solution to the issue of noncommutative effects.

5.3.3 Compiling λ-Calculi

In order to execute programs which make use of the lambda calculus as defined in the previous section, instruction sets for the two variants (call-by-value and call-by-name) must be defined separately (`newtypes` could be used to distinguish the evaluators, but this may prove quite confusing in practice). To this end, we define two target signatures, corresponding to the stack instructions for variants of a Categorical Abstract Machine (CAM) [CCM85] and Krivine machine [Ler90, Cur91], respectively:

```
data LAMBDAV e = IND Int e | CLS e e | RET  e | APP e
data LAMBDAN e = ACS Int e | PSH e e | GRAB e
```

The above constructors are sufficient for us to look up values in environments by de Bruijn index, build closures, evaluate function bodies and arguments and execute code with a given environment: the *semantics* of these constructors are based upon the following compilation schemes $\mathbb{C}$ and $\mathbb{K}$ targeting the CAM and Krivine machine respectively:

$$\mathbb{C}[n] = [\texttt{IND}\ n]$$

$$\mathbb{C}[\lambda\ t] = [\text{CLS}\ (\mathbb{C}[t] \mathbin{+\!+} [\texttt{RET}])]$$

$$\mathbb{C}[f\ x] = \mathbb{C}[x] \mathbin{+\!+} \mathbb{C}[f] \mathbin{+\!+} [\texttt{APP}]$$

$$\mathbb{K}[n] = [\texttt{ACS}\ n]$$

$$\mathbb{K}[\lambda\ t] = [\texttt{GRAB}] \mathbin{+\!+} \mathbb{K}[t]$$

$$\mathbb{K}[f\ x] = [\texttt{PSH}(\mathbb{K}[x])] \mathbin{+\!+} \mathbb{K}[f]$$

We have chosen to show the compilation schemes for the above rather than the code representing their compilation algebras, as we believe that the above is better suited to demonstrating the translation between the source and target languages; however the definitions for said algebras follow directly. For example, an abstraction over a term is translated under the Krivine machine model into a `GRAB` instruction appended to the result of recursively translating the underlying term itself. As alluded to at the end of Chapter 4, we defer the implementation of the *operational semantics* of these two machines until Chapter 7.

Having successfully implemented variable binding modelled using the lambda calculus in a modular manner and presented the compilation schemes for two distinct models of execution, we now consider how to introduce persistent, updatable state into our modular compilation framework.

5.4 Introducing Modular Mutable State

In programming languages, a commonly used feature is that of *mutable* state variables that can change their value over time. In this section, we extend the expressive power of a modular source language by introducing the notion of modular mutable state. To begin with, we consider a single-celled state consisting of an integer as the state domain, although as we have seen with the lambda-calculus, lifting these definitions into a modular `Value` datatype is straightforward. This single-celled state is presented simply as a proof of concept, however, as we shall see in Chapter 7 how this can be generalised to a countably infinite key-value table of states. The syntax associated with such an updatable integer value is given by the following signature functor:

```
data State e = Get | Set Int e
```

In the above, the `Get` operation returns the current state, and the `Set` operation takes an integer and an expression which treats this new value as the current state. As with each new feature, we define a class `StateMonad` of associated operations:

```
class Monad m => StateMonad m where
  update :: (Int -> Int) -> m Int
```

The `update` operation takes a function `Int` → `Int` and uses it to alter the state variable. By passing different functions to `update`, it can be used to define an evaluation algebra for the `State` signature:

```
instance StateMonad m => Eval State m where
  evAlg (Get)     = do n <- update id
                       return (Num n)
  evAlg (Set v c) = update (\_ -> v) >> c
```

When evaluating a `Get` constructor, the `update` operation is passed the identity function `id`, leaving the state value unchanged. This value is then bound to `n` and embedded into the `Value` domain. In turn, when evaluating a `Set` constructor, `update` is passed an anonymous function overwriting the state to `v` before evaluating subexpression `c`.

We can now write terms in our modular source language that utilise an integer state variable. To illustrate, consider the following two terms, built from languages supporting both arithmetic and state, and state and exception handling, respectively:

```
x :: Fix (Arith :+: State)
x = set 1 (add get (val 2))
```

```
y :: Fix (State :+: Except)
y = set 1 (catch throw get)
```

Informally, the expression `x` first sets the state to have value `1`, then adds the current state to the number `2`. In turn, expression `y` first sets the state to number `1`, then immediately throws an exception that is handled by returning the current value of the state.

Recall that in our modular compilation framework, we evaluate a modular expression with respect to a monad that has been constructed by applying the appropriate monad transformers to a base monad, for which purposes we often use the identity `Identity`. The underlying machinery associated with the monad transformer class allows access to the operations associated with each constituent feature (such as `throw`, `update`, `env` etc.) at the top level, with all of the necessary lifting handled automatically.

Recall that each monad transformer comes equipped with an *accessor function* – such as `runS` and `runE` – with allow access to the underlying representation. By first evaluating an expression and then applying the desired series of accessor functions, we obtain a final value, as illustrated below (using `()` as the parameter to `Value` as we do not require the closures from the lambda calculus):

```
newtype StateT s m a =
  S { runS :: s -> m (a, s) }
newtype ErrorT m a =
  E { runE :: m (Maybe a) }
```

```
> let a = eval x :: StateT Int Identity (Value ())
> runS 0 (runId a)
Num 3
```

```
> let b = eval y :: ErrorT (StateT Int Identity) (Value ())
> runE (runS 0 (runId b))
Just (Num 1)
```

In both of the above evaluations, we see that modular expressions involving state are given a semantics by applying the `StateT` state monad transformer at some point when building the monad, and similarly that the `ErrorT` exception monad transformer is applied when handling exceptions modularly.

However, an issue arises when considering the *order* in which certain monad transformers are applied, namely that of *noncommutative effects*. To illustrate, consider the following:

```
instance MonadT (StateT s) where
  lift m = S $ \s -> do x <- m
                        return (x, s)


instance Monad m => Monad (StateT s m) where
  return x    = S $ \s -> return (x, s)
  (S g) >>= f = S $ \s -> do (x, t) <- g s
                             runS (f x) t



instance Monad m => StateMonad (StateT Int m) where
  update f = S $ \s -> (s, f s)
```

The above instantiations and instance declarations of the `StateT` monad transformer appear at first glance to be no different to that of any other transformer associate with a particular feature. However, in the next section we shall see that defining `StateT` in this manner leads to noncommutativity concerns.

5.4.1 The Noncommutativity Of Effects

We have just seen how monad transformers are used to access the operations needed to define evaluation algebras. However, in some cases separate features can interact in multiple ways, and this is reflected when applying the associated monad transformers in different orders. Consider the following expression `demo`, constructed from a modular source language which supports arithmetic, mutable state and exception handling:

```
demo :: Fix (Arith :+: Except :+: State)
demo =  set 0 (catch (add (set 1 get) throw) get)
```

The `demo` example must be evaluated within a monad that supports both exception and state, and therefore must contain both of the relevant monad transformers. It is less obvious, however, that switching the order in which these two transformers are applied has an observable effect on the resulting semantic domain. Assuming that no other features are present, and using `Identity` as the base monad, the types resulting from the two possible orderings are:

```
type LocalM a =
  StateT Int (ErrorT Identity) a
= Int -> ErrorT Identity (a, Int)
= Int -> Identity (Maybe (a, Int))
= Int -> Maybe (a, Int)
```

```
type GlobalM a =
  ErrorT (StateT Int Identity) a
= StateT Int Identity (Maybe a)
= Int -> Identity (Maybe a, Int)
= Int -> (Maybe a, Int)
```

In particular, when applied to a parameter `a`, the underlying representation of the `LocalM` monad takes an `Int` and either successfully returns a pair `(a, Int)`, or an exception in the form of `Nothing`. In turn, the `GlobalM` monad also takes an `Int` but *always* returns a pair, where the first element can return `Nothing`.

More specifically, when handling an exception the 'local state' monad restores the state to its most recent value prior to entering the catch-block that threw the exception, while the 'global state' monad treats any updates to the state value as irreversible. Specifically, `demo` produces the value `Num 1` when evaluated with respect to `GlobalM`, and the value `Num 0` with respect to `LocalM`.

These are both sensible results, and depend on how we wish to order the underlying effects: the local version has a transactional nature to it, which may better capture the particular requirements of a given situation. The natural progression at this point is to address the issue of compiling expressions with multiple interpretations, such as `demo`, in a modular manner.

Our modular compiler will currently compile demo to the following code sequence (written using Haskell list notation):

```
> comp demo []
[SET 0, MARK [GET] [SET 1, GET, THROW, ADD, UNMARK]]
```

The above code is associated with the *global* approach to state, as the SET operation within the catch-block cannot be reversed when the THROW instruction is encountered. To model the behaviour associated with the local approach to state, two additional operations are required:

```
> comp demo []
[SET 0, MARK [RESTORE, GET]
             [SAVE, SET 1, GET, THROW, ADD, UNMARK]]
```

The SAVE operation records the current value of the state on the stack, and in turn the RESTORE operation restores the state to its previous value before the handler code is executed.

Both of the above results are valid, corresponding to compiling demo with respect to a particular ordering of effects. However, a modular compiler is only capable of generating *one* such program in any particular session, as the compilation algebra class is only parameterised by the source and target signatures, with no information available concerning intended semantics.

Clearly, there is a need for a more flexible compilation algebra that is aware of the *context* of an argument expression. To do this, we must allow the compilation algebra to examine the monad in which an expression is evaluated, as the semantics are defined by the order in which certain monad transformers are applied.

5.4.2 Monadic Parameterisation

In this section, we propose three distinct techniques for directing the modular compilation of an expression by inspecting its underlying semantic monad. As we have seen, in our framework we make use of monads that have been constructed by applying a sequence of transformers to a base monad. Taking advantage of the fact that monad transformers are defined as `newtype`s, we can inspect their constructors at the type level, giving rise to our first technique:

Type-Level Monadic Parameterisation

```
class (Functor f, Functor g, Monad m) => Comp f g m where
  compAlg :: f (m () -> Fix g -> Fix g)
             -> m () -> Fix g -> Fix g
```

In the above, the compilation algebra class is parameterised by a monad. The algebra carrier then includes a monadic computation as an argument, however this computation is parameterised by the void type `()` to indicate that the monad is not explicitly used in the compilation process, but rather used as a context reference.

In this manner, multiple instances of a compilation algebra can be defined for a single source signature by pattern-matching upon constructors associated with monad transformers. This allows for expressions such as `demo` (defined in the previous section) to be compiled using different schemes for different orderings of effects. For example, the compilation schemes for the two different orderings of exceptions and state can now be defined:

```
-- global compilation scheme
instance (EXCEPT :<: g, Monad m) =>
 Comp Except g (ErrorT (StateT s m)) where
  compAlg (Throw)     = \_ -> throw
  compAlg (Catch x h) = \m c -> mark (h m c)
                             (x m (unmark c))
-- local compilation scheme
 instance (EXCEPT :<: g, Monad m) =>
  Comp Except g (StateT s (ErrorT m)) where
  compAlg (Throw) \_ -> throw
  compAlg (Catch x h) = \m c -> mark (h m c)
           (save (x m (restore $ unmark c)))
```

An advantage of this technique is that we only need to match on constructors associated with monad transformers that cause semantics to differ. For example, consider the *commutative* monad transformer `ReaderT`:

```
newtype ReaderT w m a = R { runR :: w -> m a }
```

As the name suggests, commutative monad transformers will affect the semantics of a given monad in the same manner whether it is applied before or after any *other* given transformer. If `ReaderT` were to appear between `ErrorT` and `StateT` in the above, we could abstract over this transformer using a generic variable `t` of type `MonadT`, allowing the programmer to focus on the task of defining algebras only for noncommutative orderings. This leaves us with a choice to make. Either; for each noncommutative transformer pair, define two algebra instances (one with, and one without intermediate transformers), or insist that each transformer pair (whether noncommutative or not) is interspersed with an `Identity` commutative transformer in order to cut down the number of algebra instances required. In either case, modularity is somewhat impaired.

More importantly, however, the monadic computation that appears in the carrier of the algebra allows for effectful operations to be manifested by calling its associated methods. The user must be careful to not use any monadic operations when defining a compilation algebra for a particular signature, as we define compilation to be an effect-free mapping between modular source and target languages. Further, this computation cannot be removed from the carrier, as it must be threaded through to subexpressions.

Function-Level Monadic Reification

In order to exclude the possibility of monadic operations being invoked during compilation, we require a way to provide the compilation algebra with information concerning the ordering of monad transformers, *without* explicitly passing around the resulting monad. A solution to this issue is to use GADTs to *reify* a monad, representing it as a sequence of constructors. We capture this notion with the datatype `MTL`, defined as follows:

```
  data ST = IntT | BoolT | ...
  data MTL m where
    Err ::       MTL m -> MTL (ErrorT m)
    Sta :: ST -> MTL m -> MTL (StateT ST m)
...
    Id  ::                 MTL Identity
```

Using the auxiliary datatype `ST` of *state types* to reify monad transformer parameters, an instance of `MTL m` represents the monad `m` by applying the appropriate constructors to `Id`. We note that by defining `MTL` as an ordinary ADT, the set of effects that can be handled is closed, but a modular representation is also possible, at the cost of including the appropriate constraints when defining instances of the resulting datatype. To illustrate, the two monads `LocalM` and `GlobalM` that are defined in the previous section can be reified as follows:

```
local  :: MTL (StateT ST (ErrorT Identity))
local  =  Sta IntT (Err Id)


global :: MTL (ErrorT (StateT ST Identity))
global =  Err (Sta IntT Id)
```

There are two points to be made concerning the above. Firstly, by using `ST` to abstract over the parameter type of state monad transformers, we are highlighting that it is the *structure* of the underlying representation that we are concerned with, as opposed to the actual types involved. Secondly, the ordering of the monad transformers can now be examined at the *function level* by using pattern matching on the data constructors `Sta` and `Err`, .

We can now replace the monadic computation `m ()` in the carrier of the compilation algebra with its reified representation `MTL m`. In doing this, we eliminate the concern that effectful operations may 'leak' into the compilation process by removing the possibility of invoking any monadic operations. This leads to the definition of our second technique:

```
class (Functor f, Functor g) => Comp f g where
  compAlg :: f (MTL m -> Fix g -> Fix g)
               -> MTL m -> Fix g -> Fix g
```

By performing case analysis on the `MTL` argument, we can now define multiple compilation schemes within a *single* compilation algebra instance, as seen in the following:

```
instance (EXCEPT :<: g) => Comp Except g where
  compAlg (Throw) = \_ _ -> throw
  compAlg (Catch x h) = \m c -> case m of
   (Err (Sta s t)) ->
     mark (h m c) (x m (unmark c))
   (Sta s (Err t)) ->
     mark (h m c) (save (x m (restore $ unmark c)))
```

Particularly important here is that the compilation algebra is no longer parameterised by a monad `m`, highlighting the fact that a modular compiler is *informed* by a monad, rather than defined in terms of one. Also interesting is the potential to introduce predicates (of a sort) over instances of the `MTL` datatype in order to bypass intermediate transformers of no interest:

```
globalState :: MTL m -> Bool
globalState (Sta s t) = containsError t
globalState (Err _)   = False
globalState ... = ...
```

Constraint-Level Monadic Proxies

The previous two approaches represent two extremes of the solution spectrum: either put all the information about the monad (this information arguably being primarily of use to the programmer *constructing* new compilation algebras) within which an expression is evaluated into an argument `m ()` and inspect it at the type-level, or reify the monad into a 'list' of constructors `MTL m` and pattern-match upon it at the function-level. Our third approach represents a meeting point between the two, by passing the monad as a type argument to a proxy datatype which cannot access the underlying monad, but is still aware of the context within which it is defined:

```
data Proxy m = Proxy
```

The data constructor `Proxy` can be threaded through to the carrier of a compilation algebra and retain the correct type (in much the same way as the empty list `[]` retains its type), and we are prohibited from invoking monadic operations. The resulting compilation typeclass that makes use of this is defined as follows:

```
class (Functor f, Functor g) => Comp f g where
  compAlg :: f (Proxy m -> Fix g -> Fix g)
             -> Proxy m -> Fix g -> Fix g
```

We believe that this approach is the best of the three, minimising the presence of the monad within the compilation algebra, the boilerplate code required to implement said algebra, and the risk of invoking stateful operations in a context where no such operations should appear. However, each approach has their use, and the choice of which one best suits their needs or taste is ultimately one for the user, in light of the necessity of the compilation algebra to be able to account for all possible interactions. In this sense, the compilation algebra is less modular than its evaluation counterpart, but we believe this to be a necessary consequence of ensuring that expressions are compiled into the 'correct' target instruction set.

5.5 Chapter Summary

In this chapter we have:

- Extended our framework with support for both mutable state and a de Bruijn indexed variant of the lambda calculus, improving the expressive power of a modular source language.

- Shown how the use of generalised algebraic datatypes to model troublesome signature functors and value domains permits certain forms of type constraints to be captured in a clean and modular manner.

- Defined modular variants of a Categorical Abstract Machine and the Krivine machine as suitable targets for our implementation of the lambda calculus. However, as in the previous chapter, we defer all notion of virtual machines and their construction to Chapter 7, where they will be treated in depth.

- Considered the issue of effects that do not commute (such as exceptions and state), which potentially require programs to be compiled in different manners depending on the ordering of the effects, and present three approaches to addressing this.

Chapter 6

Modular Control Structures

At this point, our framework supports multiple source features and targets any language supporting the appropriate instructions. We have seen how troublesome constructors within an individual source signature can have their constraints integrated into the constructor itself by using GADTs, and how the associated evaluators for modular source programs are parametrically polymorphic in the monad that they are evaluated within. Moreover, we have identified the issue of noncommutative effects and the impact that monad transformer ordering can have on the required instruction set of a target program, and provided multiple solutions for this. However, the source programs which we can write thus far are limited, particularly as we have not yet considered the issue of control flow.

In this chapter, we introduce a number of new features to the source syntax implementing cyclic and non-cyclic control flow, further improving the expressive capabilities of source programs. We identify the fact that the presence of such constructs in our source language introduces an entirely new class of incorrect programs, and modify the representation of the syntax of source signatures to solve the issue. We go on to discuss what the presence of cyclicity means for both the compiler itself and the syntax the compiler maps into, and redesign the target representation appropriately.

6.1 Introducing Control Flow

When considering the taxonomy of computational language features, we typically distinguish between two varieties: firstly, effectful features, such as exception handling and mutable state that we have treated in the previous two chapters, and secondly those features which relate to control-flow, such as conditionals and recursion.

In this section, we once again extend the expressive capability of our modular compilation framework, this time with features drawn from the latter category above. We will show that our framework is sufficiently flexible to accommodate this type of feature with minimum effort, and how refining the representation of the target language to use graphs instead of fixpoints unlocks the full breadth of expressibility afforded by these new constructs.

6.2 Re-representing the Source Language

Our goal in this chapter is to be able to compile source languages which contain modular representations of imperative control structures such as loops and conditionals. To do this, we require more care when choosing an appropriate representation of the abstract syntax trees of the source language. Specifically, we claim that the initial algebra representation that we have been working with throughout the previous two chapters is insufficient for this purpose. To illustrate, assume the existence of a loop signature:

```
data For e = For e e
```

The above represents loops with the following intended semantics: the first argument evaluates to an integer value n, and the second argument is then iterated n times. The problem with this representation is that it uses the same 'sort' to represent both expressions and statements. Within a loop we typically expect the first argument to be an expression (i.e. something that evaluates to a value), whereas the second argument is a statement (i.e. something that causes side effects such as variable assignment). Whilst it *is* possible to incorporate the distinct notions of expressions and statements into a single type, simplifying the implementation of both an interpreter and a compiler, their implementations can be inefficient.

Given that we want to compile the source language into code that runs on a stack-based virtual machine, we face the problem of having to clean up the stack after 'executing' an expression whose value is not used. For instance, consider the following example:

```
simpleLoop :: Fix (For :+: Arith)
simpleLoop =  for (val 10) (val 42)
```

The above is a loop that repeats its body 10 times, and where the body of the loop pushes the integer 42 onto the stack. However, since this value is not used after being pushed, the code associated with the body of the loop needs to end with an instruction that removes the topmost element from the stack. We note that this issue does not only appear within the above example, but is a symptom of a more general problem with the current source syntax representation. To illustrate, assume the existence of a signature `State` with `get` and `set` operations defined over a single integer state domain. Then, the following loop simply produces the result of adding ten consecutive numbers, beginning from the current state value:

```
countLoop :: Fix (For :+: State :+: Arith)
countLoop =  for (val 10) (set (get `add` val 1))
```

In this example, `set (...)` must have a semantics that adds a 'result' – a meaningless placeholder value, as the expected result from a setter operation is `()` – to the top of the stack, as it is in the same syntactic category as `val 10`. Here too, we must clean up the stack after each iteration. The only systematic solution to the problem presented by the two examples given above is to distinguish between two *syntactic categories*: statements, with the invariant that their execution leaves the stack unchanged; and expressions, with the invariant that evaluating them puts their result on top of the stack. While these invariants can be, in principle, enforced independent to the representation of the syntax, mistakes are easy to make given the current representation.

6.2.1 Splitting the Source Language

In order to split the source language into different syntactic categories, we make use of Johann and Ghani's initial algebra semantics of GADTs [JG08]. The underlying idea is that each node of a tree type is annotated at the type level with the syntactic category it resides in. To this end, we extend each signature functor with an additional type argument, noting that these augmented signature functors are no longer functors in the Haskell typeclass sense (we shall see why shortly). For example, using Haskell's GADT syntax, we can now redefine `Arith` as follows:

```
data Exp

data Arith e l where

  Val :: Int              -> Arith e Exp

  Add :: e Exp -> e Exp -> Arith e Exp
```

Note that we define an empty datatype `Exp` as a label – or more precisely, an index – for expressions. The `Arith` signature is simple - the addition operator only takes expressions and returns expressions. More interesting is the signature for assigning and dereferencing mutable variables:

```
data Stmt

data State e l where

  Get :: Ref            -> State e Exp

  Set :: Ref -> e Exp -> State e Stmt
```

Note that the `Get` constructor builds an expression, while the `Set` constructor takes an expression and builds a statement. Whilst in Chapter 5.4 we assume the presence of a *single* integer as the state space, and thus require no argument to the `Get` constructor as there was no ambiguity to resolve, in Chapter 6.3.1 we will extend the state space to arbitrarily many mutable references – or variables – which we represent as type `Ref`. For simplicity, we assume that these variables are just strings, i.e.:

```
newtype Ref = Ref String
```

As we have already noted, the above indexed signatures are no longer Haskell functors: instead of mapping types to types, they map functors to functors (and, in turn, natural transformations to natural transformations). In the language of Johann and Ghani, these signatures are akin to *higher-order* functors, and throughout this chapter we shall explore their properties and the recursion schemes they give rise to.

We introduce new type constructors that lift the definitions of `(:+:)` and `Fix` to the higher-order setting by equipping them with an additional type argument in the following manner:

```
data (f ::+ g) (h :: * -> *) e = InlH (f h e)
                               | InrH (g h e)


data FixH f i where
  InH :: f (FixH f) i -> FixH f i
```

As expected, the fixpoint of a higher-order functor is itself a type function of kind $(* \rightarrow *)$ (in other words, a family of types). In the case of the syntax trees for our target language, this family consists of the different syntactic categories we want to represent. Concretely, `FixH (Arith ::+ For) Exp` is the type of expressions over the signature `(Arith ::+ For)`, whilst `FixH (Arith ::+ For) Stmt` is the corresponding type of statements.

We can make use of these higher-order syntax trees to keep track of the types of subexpressions within our source language. To do this, we parameterise `Exp` with an argument indicating the value type of a given expression. For simplicity, here we only consider integer and Boolean expressions:

```
data Exp e

data IntType

data BoolType

type IExp = Exp IntType

type BExp = Exp BoolType
```

To illustrate, the definition of the higher-order representation of `Arith` changes as follows:

```
data Arith e l where
  Val :: Int              -> Arith e IExp
  Add :: e IExp -> e IExp -> Arith e IExp
```

In order to construct Boolean-valued expressions within this setting, we introduce a new signature `Comp` of operators comparing integer expressions:

```
data Comp e l where
  Equ :: e IExp -> e IExp -> Comp e BExp
  Lt  :: e IExp -> e IExp -> Comp e BExp
```

Signatures for control structures and exceptions are defined similarly:

```
data While e l where
  While :: e BExp -> e Stmt -> While e Stmt


data Seq e l where
  Seq :: e Stmt -> e Stmt -> Seq e Stmt


data If e l where
  If :: e BExp -> e Stmt -> e Stmt -> If e Stmt


data Except e l where
  Throw :: Except e Stmt
  Catch :: e Stmt -> e Stmt -> Except e Stmt
```

In the next section, we will see that the machinery needed when defining folds on higher-order fixpoints – and defining higher-order smart constructors – is easily carried over to the setting of higher-order functors.

6.2.2 Higher-Order Folds & Smart Constructors

As mentioned above, higher-order functors map both functors to functors and natural transformations to natural transformations. This characterisation is captured by the following typeclass —

```
class HFunctor f where
  hfmap :: (g :-> h) -> f g :-> f h
```

— where natural transformations are defined as:

```
type f :-> g = forall i. f i -> g i
```

In general, an `HFunctor` should also provide a method of the following type, capturing the requirement that they map functors to functors:

```
Functor g => (a -> b) -> f g a -> f g b
```

However, as in the work of Johann and Ghani [JG08], we do not provide such a method, meaning that our higher-order functors only map type functions to type functions. This generalisation from functors to *type functions* is necessary in order to represent the indexed types required for augmenting expressions with their syntactic categories. For example, given a functor `g`, the parameterised signature `Arith g` is *not* a functor. Technically speaking, what we have defined here are higher-order *endo*functors mapping types to types, with no (non-trivial) mapping of functions to functions (i.e. no `fmap`). In the language of Johann and Ghani, these structures map between functors of kind $| * | \to *$ (wherein $| \mathbb{C} |$ is shorthand for the discrete category derived from the category $\mathbb{C}$ [Pro00], and $*$ refers to any ordinary Haskell type), which is precisely what is needed to represent GADTs.

Instance declarations for `HFunctor` are defined in a straightforward manner, akin to those for `Functor`. For example, we can define the `State` signature as a higher-order functor as follows:

```
instance HFunctor State where
  hfmap f (Get v)   = Get v
  hfmap f (Set v x) = Set v (f x)
```

Using this structure, we can define higher-order folds. Since our signatures are now indexed (as are their fixpoints), so are the algebras that are used to define folds over them. More precisely, given a higher-order functor `f` and a type constructor `c` $:: * \to *$, a higher-order `f`-algebra with carrier `c` is a natural transformation of type `f c :-> c`. Apart from the types, the implementation of higher-order folds is identical to the implementation over typical Haskell functors:

```
foldH :: HFunctor f => (f c :-> c) -> FixH f :-> c
foldH f (InH t) = f (hfmap (foldH f) t)
```

The definition of the subsignature typeclass `(:<:)` is also easily lifted to the higher-order setting:

```
class (sub :: (* -> *) -> * -> *) ::< sup where
  injH :: sub a :-> sup a
```

The instance declarations for `(:<:)` are carried over to the typeclass `(::<)` without surprises, producing the higher-order `inject` function:

```
injectH :: (g ::< f) => g (FixH f) :-> FixH f
injectH = InH . injH
```

As for signature functors, we assume that each constructor of a higher-order signature functor comes equipped with a corresponding smart constructor defined via `injectH`, e.g.:

```
whileH :: (While ::< f) => FixH f BExp
          -> FixH f Stmt -> FixH f Stmt
whileH x y = injectH (While x y)
```

Given these smart constructors, we can write the source program of Figure 6.1, which computes the factorial of the variable `x`. Note that this program makes use of a `mulH` constructor that we have omitted for brevity, however it is part of the `Arith` signature, and trivial to implement.

6.2.3 Well-Kinded Signature Indices

The use of empty data types such as `Stmt` as indices may seem crude at first glance, especially considering that the latest versions of GHC support

```
type FacLang =
 (Seq :+ Arith :+ While :+ State :+ Comp)

fac :: FixH FacLang Stmt
fac =  setH y (valH 1) `seqH`
        whileH (valH 0 `ltH` getH x)
         (setH y (getH y `mulH` getH x) `seqH`
          setH x (getH x `addH` valH (-1)))
  where x = Ref "x"
        y = Ref "y"
```

FIGURE 6.1: A sample program computing factorials.

the promotion of datatypes to the kind [YWC+12] level. Using this new promotion mechanism, we *could* have defined the following datatypes:

```
data Idx = Exp Ty   | Stmt

data Ty  = IntType | BoolType
```

Using GHC's Haskell language extension `DataKinds`, these datatypes are promoted to the kind level, giving us the *type* constructor `Exp` of *kind* (`Ty` $\to$ `Idx`). These types and kinds allow for the definition of more precise kinds for higher-order signatures: that is to say, instead of having kind $(* \to *) \to (* \to *)$, they would have kind $(\texttt{Idx} \to *) \to (\texttt{Idx} \to *)$.

The problem with using this well-kinded representation is that we lose the ability to extend the indices used in our signature functors. For example, we are no longer able to add a language feature that makes use of a 'new' type – say, natural numbers – since the type `Ty` (and thus the corresponding kind via promotion) is closed. Opting instead to use empty datatypes allows us to extend the set of indices by simply defining a corresponding datatype:

```
data NatType
```

The above, for example, now allowing us to index expressions as having the type `NatType` of kind $*$.

Note that the approach of using a typeclass such as `(:<:)` in order to facilitate open definitions (as detailed in Chapter 4.2.2) cannot be used in order to implement an extensible signature index. For this, we would require *kind-classes*, a feature currently not supported in Haskell.

We now go on to consider the semantics for these higher-order modular source signatures, and identify the necessary alterations required in order to both make use of the indices now found within subexpressions, and extend the definitions of the evaluation algebras into a higher-order setting.

6.3 Semantics of Higher-Order Signatures

In this section we demonstrate how the use of higher-order functors when defining source signatures requires a new modular evaluation algebra type-class. Further, we extend the state space to an arbitrarily large key-value mapping, and as such we must reconsider the semantics of modular mutable state before we lift the signature into the higher-order setting.

6.3.1 Revisiting The State Monad

Consider a higher-order representation of `State` that makes use of the index labels discussed in the last section:

```
data State e l where
  Get :: Ref           -> State e IExp
  Set :: Ref -> e IExp -> State e Stmt
```

In Chapter 5.4 we introduced the state monad transformer and `StateMonad` effect typeclass in order to define the semantics for state. The interface for a state *monad* as found in the `mtl` monad transformer library [Gil14] is:

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()

modify :: MonadState s m => (s -> s) -> m ()
modify f = do {x <- get; put (f x)}
```

In order to implement a state space comprising an arbitrarily large number of integer-valued variables, we use the type `Map` of finite mappings. The resulting type of this new state space is:

```
type St = Map Ref Int
```

As before, in the above the mapping could just as easily target a modular `Value` datatype, but for clarity we do not do so here. Functions to read and write individual variables are easily implemented:

```
getRef :: (MonadState St m, MonadPlus m) => Ref -> m Int
getRef v = do s <- get
              case Map.lookup v s of
               Just n  -> return n
               Nothing -> mzero

setRef :: MonadState St m => Ref -> Int -> m ()
setRef v n = modify (Map.insert v n)
```

In the above, note that since a variable may not yet be associated with an integer when referenced (although generally we make the assumption that we only consider closed terms), the `getRef` function provides a means to signal failure (alternatively, a default value can be returned in the event of a lookup failure). For our purposes, we treat failure as an effect in the sense of an exception, and reflect this structure by the `MonadPlus` constraint.

6.3.2 Higher-Order Modular Semantics

Having redefined the modular source language of our compilation framework as a family of types – comprising the types for expressions and statements –, the semantic domain must also be redefined as a type family. Here we define the potential types of the semantic domain in a manner similar to the source language itself (i.e. treating the resulting values as literals, which can in turn be used to define datatypes such as `Arith`):

```
data VNum (e :: * -> *) l where
  Num  :: Int  -> VNum  e IExp

data VBool (e :: * -> *) l where
  Bool :: Bool -> VBool e BExp


data VUnit (e :: * -> *) l where
  Unit ::         VUnit e Stmt
```

A higher-order modular semantics is defined in terms of a natural transformation, i.e. of type `(f c :-> c)`. However, we require *more* structure within this algebra, because the carrier `c` may contain both values (constructed using the above constructors) and effects invoked by any monadic methods available to the source expression. Therefore, the definition of the typeclass for the evaluation algebra is:

```
class (HFunctor f, Monad m) => AlgEv f m v where
  algEv :: f (m :o: FixH v) :-> m :o: FixH v
```

In the above, `(:o:)` denotes the composition of type constructors of kind $* \to *$, and is defined as follows:

```
newtype (f :o: g) i = C { unC :: f (g i) }
```

That is to say, the carrier of the evaluation algebra is the composition of a monad `m` with a fixpoint `FixH v` over a higher-order functor `v` describing the semantic domain. This explicit separation of the carrier into distinct components is a necessary consequence of monads composing in a different manner to fixpoints: the former is achieved via the use of monad transformers, and the latter via coproducts.

However, we make one small modification to the above definition of `algEv`. In its current form, the result type is a composition involving `(:o:)`, which means that any results must be explicitly tagged with the constructor `C`. We choose to avoid this, and use the following definition instead:

```
class (HFunctor f, Monad m) => AlgEv f m v where
  algEv :: f (m :o: FixH v) i -> m (FixH v i)
```

The type of the above algebra is isomorphic to the previous definition: we regain the correct algebra type by composing `algEv` with `C`:

```
C . algEv :: AlgEv f m v => f (m :o: FixH v)
                                 :-> m :o: FixH v
```

We use this composition as the argument to `foldH` in order to define the desired higher-order modular evaluator:

```
eval' :: AlgEv f m v => FixH f :-> m :o: FixH v
eval' =  foldH (C . algEv)
```

Finally, by composing `eval'` with `unC`, we obtain a variant that does *not* make use of the composition operator `(:o:)`:

```
eval :: AlgEv f m v => FixH f i -> m (FixH v i)
eval =  unC . eval'
```

To achieve modularity in the source signature, the typeclass `AlgEv` is trivially lifted over higher-order coproducts:

```
instance (AlgEv f m v, AlgEv g m v) =>
 AlgEv (f +:: g) m v where
  algEv (InlH x) = algEv x
  algEv (InrH y) = algEv y
```

As was the case for our previously defined higher-order functors, we assume the existence of smart constructors for `Num`, `Bool` and `Unit` called `num`, `bool` and `unit`, respectively. However, at this point we also require smart *destructors*, as we wish to pattern match on the result of an evaluation. In order to implement such destructors, the `(::<)` typeclass is extended with a *projection* method of the following type:

```
prjH :: sup a i -> Maybe ((sub a) i)
```

In the event that the supersignature does indeed contain the subsignature in question, `fromJust` ∘ `prjH` is both a left and right inverse of `injH`, coercing a value of the supersignature into a value of the subsignature, and returning `Nothing` if this is not possible. Instance declarations for `(::<)` can be easily extended to implement `prjH`:

```
instance HFunctor f => f ::< f where
  injH = id
  prjH = Just
```

```
instance (HFunctor f, HFunctor g) => f ::< (f ::+ g) where
  injH          = InlH
  prjH (InlH x) = Just x
  prjH (InrH y) = Nothing
```

```
instance (HFunctor f, HFunctor g, HFunctor h, f ::< h)
=> f ::< (g ::+ h) where
  injH          = InrH . injH
  prjH (InlH x) = Nothing
  prjH (InrH y) = prjH y
```

With the use of this new method, we can define the smart destructor for `Num` as follows:

```
getNum :: (VNum ::< f) => FixH f IExp -> Maybe Int
getNum (InH e) = case (prjH e) of
                   Just (Num n) -> Just n
                   _            -> Nothing
```

The corresponding smart destructor `getBool` is defined analogously. With these destructors available to us, we can finally define the higher-order semantics of the source language. Note that the context of instance definitions lists both the monadic constraints *and the value signature*, keeping them open for extension as before. Since the carrier of the evaluator algebra is a composition formed using `(:o:)`, we must pattern match on all subexpressions.

There are two points worth noting about the evaluation algebra instantiations for control structure signatures. Firstly, common to all three instances is the lack of an effect typeclass constraint, as their semantics concerns control flow only, and so *any* monad suffices. Secondly, we note that the body of the semantics for while-loops is recursively defined, making it more operational than denotational in nature. This is an important distinction to make given that the latter must be compositional, however we are not bound to this requirement (other than by a desire for a modular equivalent to a denotational semantics), and therefore keep in mind that the introduction of while-loops removes this property. However, those source signatures that do *not* contain while-loops still satisfy compositionality. Moreover, we highlight at this point that whilst non-termination is an effect unto itself (and one that can be captured using while-statements), we have chosen to limit our source expressions to terminating closed terms.

Now that we have set our framework up anew, we demonstrate its usage by way of interpreting the factorial program defined in Figure 6.1.

6.3.3 Modular Semantics: An Example

Recall the type signature of the modular evaluation function:

```
eval :: AlgEv f m v => FixH f i -> m (FixH v i)
```

This type signature tells us that the result of interpreting a modular source expression will have type `m (FixH v i)` for some appropriate monad `m` and semantic domain `v`. Because it is the monad that allows access to the requisite effectful methods, the signature of a given source program provides information about the requisite candidate monads within which it can be interpreted. Likewise, a suitable semantic domain can also be inferred in this manner. We will demonstrate this idea below.

Recall the type signature of `fac`:

```
type FacLang = (Seq :+ Arith :+ While
                    :+ State :+ Comp)
fac :: FixH FacLang Stmt
```

From the above signature alone, we can sketch a complete picture of the type of monads and semantic domains needed to evaluate this particular program. We do this by accumulating the constraints upon each of the five language features. For clarity, we list the relevant signatures below:

```
instance Monad m
 => AlgEv Seq m v   where ...

instance (Monad m, VNum ::< v)
 => AlgEv Arith m v where ...

instance (Monad m, VBool ::< v, VUnit ::< v)
 => AlgEv While m v where ...

instance (MonadPlus m, MonadState St m,
 VNum ::< v, VUnit ::< v) => AlgEv State m v where ...

instance (Monad m, VNum ::< v, VBool ::< v)
 => AlgEv Comp m v  where ...
```

Upon inspection, we conclude that we can interpret `fac` within the context of *any* monad `m` provided that it is also an instance of the `MonadState St` and `MonadPlus` typeclasses. Likewise, we find that an appropriate semantic domain — abusing terminology somewhat, as we do not require a bottom element since we do not incorporate non-termination — is the fixpoint of a higher-order signature functor `v` that contains *at least* `VNum`, `VBool` and `VUnit`. Note that we have done this without even looking at the body of the source program itself!

For this example, we obtain a suitable monad for our purposes by applying the state monad transformer – with state space `St` – to the `Maybe` monad:

```
type FacMonad = StateT St Maybe
```

The semantic domain is obtained by capturing all value types referenced in the constraints of the required evaluation algebras. Here, we have invoked the `Stmt` syntactic category so as to match the type of `fac`, but in practice any syntactic category can be used):

```
type FacValue = FixH (VNum ::+ VBool ::+ VUnit) Stmt
```

The evaluator for the language of the `fac` program is now obtained by instantiating the modular `eval` function with its modular context (the monad) and modular semantic domain:

```
evalFac :: FixH FacLang Stmt -> FacMonad FacValue
evalFac =  eval
```

We can now define our modular factorial function by evaluating `fac` and running the resulting state computation with the initial state map `[(x, n)]`, recalling that `fac` associates its argument with `x`:

```
runFac :: Int -> Maybe (Value, St)
runFac n = runS (evalL1 fac) (fromList [(x, n)])
```

We test the function with input `10`:

```
-- x = Ref "x"

-- y = Ref "y"


> runFac 10

Just (Unit, fromList [(x, 0), (y, 3628800)])
```

As expected, the variable `y` is bound to the value `3,628,800`. The actual return value of running the program, however, is `Unit`, which is expected as we declared `fac` to be a `Stmt` in its signature.

6.4 Further Refining Modular Compilers

Having treated the source language representation of our new framework in depth, we move on to consider the representation of the target language in light of the presence of the control-flow features introduced. Whilst the target language defined in Chapter 5.2 incorporates a simple notion of control flow in the form of exceptions, the target languages required when compiling these new signatures must necessarily allow for *cyclic* control flow. For instance, the factorial program illustrated in Figure 6.1 is inherently cyclic.

The cyclicity of a source program must be reflected in the code produced by a compiler, and this is typically achieved by making use of a graph structure using explicit jumps and labels. However, in this thesis we will make use of the purely functional representation of graphs proposed by Oliveira and Cook dubbed *structured graphs* [OC12], which provide a representation of term graphs, using an elegant encoding of sharing and cyclicity via *parametric higher-order abstract syntax* [Chl08]. This representation provides a simple interface for constructing graphs in a compositional fashion, at the cost of a more complicated and restrictive interface for their consumption, as we shall see shortly.

6.4.1 From Fixpoints To Graphs

The idea of structured graphs is to represent term graphs – graphs wherein vertices denote subterms – via mutually recursive let-bindings. The definition of structured graphs that we make use of extends the definition of the least-fixpoint construct by including two additional constructors, `Var` and `Mu` for representing variables and mutually recursive bindings:

```
data GraphT f v = Var v
                | Mu ([v] -> [GraphT f v])
                | InG (f (GraphT f v))
```

The newly-added parameter `v` defines the type for the metavariables in the graph. We are already familiar with the notion of the `InG` constructor, as it is equivalent to the `In` constructor for fixpoints. The `Mu` constructor represents binders using higher-order abstract syntax (HOAS). In order to enable mutually recursive bindings, we define `Mu` as a function taking a list of metavariables and returning a list of associated term graphs. The simplest way to explain the intended semantics of `Mu` is to show how it corresponds to the `let`-binding notation of Haskell. Specifically, a `let`-binding that takes the form –

```
let x1 = b1; x2 = b2; ...; xn = bn in b
```

– is represented as a structured graph as follows:

```
Mu (\[_, x1, x2, ..., xn] -> [b, b1, b2, ..., bn])
```

More specifically, the function associated with the `Mu` constructor takes the list of bound metavariables as arguments, and returns a list of the same length such that the i^{th} element of that list is bound to the i^{th} metavariable from the input list. The first element of the return list, `b`, is the entry point of the graph. In addition, we choose to represent the metavariable list passed to `Mu` as an *irrefutable pattern*:

```
Mu (\~(_ : x1 : x2 : ... : xn : _)
   -> [b, b1, b2, ..., bn])
```

In brief, we have made the list of metavariables into a lazy pattern, meaning that all matches immediately succeed. Such patterns are only matched against – and in our case, the metavariables in question looked up – if a variable contained within is needed on the right-hand side of the function, creating a more general and less strict setting for modelling cyclic computation. In this thesis, we shall only use two special cases of the `Mu` constructor, namely non-recursive let bindings and fixpoints over a single argument:

```
letx :: GraphT f v -> (v -> GraphT f v) -> GraphT f v
letx g f = Mu (\~(_ : x : _) -> [f x, g])


mu :: (v -> GraphT f v) -> GraphT f v
mu f = Mu (\~(x : _) -> [f x])
```

Using these combinators, within our framework we represent non-recursive `let`-bindings of the form `(let x = b in s)` as `(letx b (`λ`x` $\rightarrow$ `s)`, and a least-fixpoint `Fix f` as `mu f`.

As mentioned at the beginning of this section, structured graphs make use of a restricted form of HOAS called *parametric* HOAS. When constructing structured graphs, the type `v` of metavariables is left polymorphic. To ensure this, structured graphs of type `GraphT` are wrapped in the newtype `Graph`, which enforces the parametric polymorphism of `v`:

```
newtype Graph f = MkGraph (forall v. GraphT f v)
```

The parametricity of `v` ensures that it is only used to define binders within the graph. Moreover, parametricity is used when defining recursion schemes on structured graphs. The recursion schemes we use here operate similarly to the usual `fold` operator on fixpoints. One particularly general recursion scheme over graphs is defined below:

```
gfold :: Functor f => (t -> c) -> (([t] -> [c]) -> c)
                  -> (f c -> c) -> Graph f -> c
gfold v l f (MkGraph g) = trans g where
  trans (Var x)  = v x
  trans (Mu g)   = l (map trans . g)
  trans (InG fa) = f (fmap trans fa)
```

In contrast to `fold`, `gfold` takes two additional arguments corresponding to the actions to be taken for the `Var` and `Mu` constructors. The first argument, of type `(t` $\rightarrow$ `c)`, is used to transform metavariables into the result type `c`. The second argument, of type `([t]` $\rightarrow$ `[c])` $\rightarrow$ `c`, is used to interpret mutually recursive binders, where the right-hand side of each binding has already been transformed.

In analogy to the fixpoint type constructor `Fix`, smart constructors simplify graph construction. We transcribe `inject` from fixpoints to graphs below:

```
injectG :: (f :<: g) => f (GraphT g a) -> GraphT g a
injectG = InG . inj
```

Using `injectG`, we obtain smart constructors such as the following:

```
pushG :: (ARITH :<: f) => Int -> GraphT f v -> GraphT f v
pushG n c = injectG (PUSH n c)
```

Note that the structured graphs that we have defined above make use of standard functors as opposed to the higher-order variant introduced in this chapter (i.e. there is only one syntactic category used within the code), even though building graph types based on higher-order functors *is* possible, as shown by Oliveira and Löh [OL13]. However for the purpose of representing control-flow graphs, this is not necessary. The potential for a more richly typed target language has many applications, such as using types to encode invariants about the current state of a stack (see, for example, McKinna and Wright [MW06]), but this is an orthogonal issue and will not be discussed further here.

6.4.2 Compiling To Structured Graphs

At this point we can refactor the carrier of the modular compilation algebra typeclass to reflect the usage of structured graphs instead of least fixpoints:

```
class HFunctor f => AlgCoG' f g where
  algCoG' :: f (GraphT g v -> GraphT g v)
             -> GraphT g v -> GraphT g v
```

It is important to note that as well as shifting to a graph structure to represent target code, the syntax of the various source language features are now represented as higher-order functors in order to include information about syntactic categories. However, because the target language remains untyped – and is represented via standard functors – we use the type constructor `K` to forget the type information from the source language:

```
newtype K a i = K { unK :: a }


class HFunctor f => AlgCoG'' f g where
  algCoG'' :: f (K (GraphT g v -> GraphT g v))
             :-> K (GraphT g v -> GraphT g v)
```

Finally, we repeat the type manipulations of Chapter 6.3.2 to eliminate the type constructor `K` in the result type, producing the following (final) definition of the higher-order compilation algebra typeclass:

```
class HFunctor f => AlgCoG f g where
  algCoG :: f (K (GraphT g v -> GraphT g v)) i
               -> GraphT g v -> GraphT g v
```

To obtain an algebra of the *correct* type, we compose `algCoG` with `K`:

```
K . algCoG :: f (K (GraphT g v -> GraphT g v))
             :-> K (GraphT g v -> GraphT g v)
```

As before, this typeclass is easily lifted to coproducts:

```
instance (AlgCoG f g, AlgCoG h g) =>
 AlgCoG (f +:: h) g where
  algCoG (InlH x) = algCoG x
  algCoG (InrH y) = algCoG y
```

Note that we have been using the `GraphT` type constructor, instead of the encapsulated `Graph` variant. As a rule, the manipulations involving `GraphT` are passed on to `MkGraph` to construct a graph of type `Graph g`. The type of `MkGraph` – a rank-2 polymorphic constructor – ensures that the underlying graph is indeed polymorphic in the type `v` of metavariables:

```
comp :: (AlgCoG f g, HALT :<: g) => FixH f i -> Graph g
comp e = MkGraph (unK (foldH (K . algCoG) e) haltG)
```

Here we use `unK` to turn the result of the fold, which is of type `K (GraphT g v → GraphT g v) i`, into a function of type `(GraphT g v → GraphT g v)`. This function is then applied to a singleton `HALT` instruction which serves as the final code continuation. The implementation of the new compilation algebra instance for arithmetic is analogous to the version defined via standard least-fixpoints in Chapter 5.2:

```
instance (ARITH :<: g) => AlgCoG Arith g where
  algCoG (Val n)           c = pushG n         |> c
  algCoG (Add (K x) (K y)) c = x |> y |> addG |> c
```

The `(|>)` constructor used above is simply a graph-specific variant of the function application operator `($)`. Furthermore, because we are compiling into a graph structure, we make use of the smart constructors for graphs, and because the carrier of the algebra is wrapped in the type constructor `K`, we pattern match argument subexpressions against `K`.

When defining the algebra instance for the higher-order exceptions signature, we can now exploit the sharing capabilities afforded to us by the newfound target graph structure:

```
instance (EXCEPT :<: g) => AlgCoG Except g where
  algCoG (Throw)             c = throwG
  algCoG (Catch (K x) (K h)) c = letx c (\v ->
    markG (h |> Var v) |> x |> unmarkG |> Var v)
```

Instead of placing the continuation `c` directly into the generated code (thereby duplicating `c` and wasting memory on an uninvoked thunk), we choose to bind `c` to the metavariable `v` and refer to *this* instead of `c`, eliminating the risk of code duplication. The same approach is used to compile *if*-statements, with the aid of a conditional jump instruction:

```
data COND e = JPC e e
```

The `JPC` instruction removes the topmost element from the stack and inspects it, executing its first argument (and skipping the second) if this value is `True`, and skipping the first argument and executing the second argument otherwise. In a manner similar to `MARK`, `JPC` has the potential for code duplication when joining the execution paths of the conditional. We avoid this duplication using `letx` as before:

```
instance (COND :<: g) => AlgCoG If g where
  algCoG (If (K b) (K p) (K q)) c = letx c (\c ->
    b |> jpcG (p |> Var v) |> q |> Var v)
```

We also make significant usage of this new graph structure when compiling loops, as we must construct cycles in the target code. For this, we make use of the `mu` combinator defined earlier:

```
instance (COND :<: g) => AlgCoG While g where
  algCoG (While (K b) (K lb)) c =
    mu (\v -> b |> jpcG (lb |> Var v) |> c)
```

Next, we define the algebras for both the `State` and `Comp` signatures. These require corresponding target instructions:

```
data STATE e = GET Ref e | SET Ref e
data COMP  e = EQ e      | LT e
```

The semantics of the instructions for the `STATE` signature above are identical to the variant presented in Chapter 5.4, albeit operating over a larger state space. The instructions of the `COMP` signature take the topmost two integers from the stack, and replace them with the appropriate Boolean value. The instance declarations themselves are defined thus:

```
instance (STATE :<: g) => AlgCoG State g where
  algCoG (Get v)           c = getG v      |> c
  algCoG (Set v (K e))     c = e |> setG v |> c

instance (COMP :<: g) => AlgCoG Comp g where
  algCoG (Equ (K x) (K y)) c = x |> y |> eqG |> c
  algCoG (Lt  (K x) (K y))  c = x |> y |> ltG |> c
```

Finally, the algebra instance for compiling sequential composition:

```
instance AlgCoG Seq g where
  algCoG (Seq (K x) (K y)) c = x |> y |> c
```

To see the compiler thus defined in action, we apply it to the example source program from Figure 6.1. Firstly, we specialise the source and target languages in the manner described in Chapter 6.3.3. We note that here the constraints which must be adhered to are fewer, as we need not consider monadic effects. We only consider target language requirements:

```
type FacLangG = (ARITH :+: COND :+: STATE
                       :+: COMP :+: HALT)
compFacG :: FixH FacLang i -> Graph FacLangG
compFacG =  comp
```

The type system of Haskell will ensure that the target language contains the necessary instruction set to compile the source language. If, for example, we were to forget to include the `COMP` signature as a component of the target language `FacLangG`, the typechecker would produce:

```
No instance for (COMP :<: HALT)
 arising from a use of ‘comp’.
```

Applying `compFacG` to `fac` returns the following (pretty-printed) graph:

```
> compFacG fac

PUSH 1; SET y; [v1 -> PUSH 0; GET x; LT;

 JPC (GET y; GET x; MUL; SET y; GET x;

 PUSH (-1); ADD; SET x; v1); HALT]
```

The code inside the square brackets corresponds to bindings in the graph structure constructed using `Mu`. As expected, the output code graph has a single cycle, corresponding to the loop of the source program.

6.5 Modifying Existing Language Features

Recall that the compilation of exceptions makes use of the instructions `MARK` and `UNMARK`, which place handler code onto the stack and removes it respectively. This scheme is very general, allowing exceptions to be compiled within a language when the context of a *throw*-statement is not statically known. However, if we assume that we are working within the context of the language features considered in this thesis, then we know for each occurrence of a throw-statement which exception handler is associated with it. It is a simple matter to define an alternative compilation scheme that exploits this property, using the following target signature:

```
data EXCEPT' e = THROW' e | UNMARK' e | MARK' e
```

Note in the above that `MARK'` only has *one* argument, as we do not pass the handler code to it. Instead, the handler is passed directly to the `THROW'` instruction. We cannot compile the source signature `Except` into `EXCEPT'` using the `AlgCoG` compilation algebra class, as this latest formulation requires that we extend the carrier with the handler currently in scope:

```
type Triple a = a -> a -> a


class AlgCoG' f g where
  algCoG' :: f (K (Triple (GraphT g v))) l
                -> Triple (GraphT g v)
```

To run a modular compilation function that invokes this variant of compiling exceptions, we must supply an initial exception handler. For simplicity, we opt for the smart constructor `haltG`:

```
comp' :: (HFunctor f, AlgCoG' f g, HALT :<: g)
  => FixH f i -> Graph g
comp' e = MkGraph (unK (foldH (K . algCoG') e) haltG haltG)
```

Thankfully, we do not have to redefine the compilation algebra instances for those language features unaffected by this change, as we can map directly from the original algebra typeclass `AlgCoG` into the modified class `AlgCoG'`, with only the `Except` source signature itself requiring redefinition:

```
instance AlgCoG f g => AlgCoG' f g where
  algCoG' x h c =
    algCoG (hfmap (K . (\f -> f h) . unK) x) c


instance (EXCEPT' :<: g) => AlgCoG' Except g where
  algCoG' (Throw)              h c = throwG' h
  algCoG' (Catch (K x) (K h')) h c =
   letx c (\v -> markG' |> letx (h' h |> Var v)
       (\vh -> x (Var vh) |> unmarkG' |> Var v))
```

One last point to note here is that whilst we have focussed on the adaptations to the types involved in the switch from fixpoints to graphs, existing concerns such as correctly compiling source expressions in light of noncommutative effects still hold. To this end, the techniques demonstrated in Chapter 5.4.2 for monadic parameterisation are equally applicable here.

6.6 Chapter Summary

The final component required for a complete definition of our modular compilation framework is the semantics of the modular target languages themselves: modular variants of virtual machines. Before we begin discussing these, however, we give a summary of the work presented above.

In this chapter, we:

- Eliminated a class of ill-typed imperative programs by using a typed representation of source signatures using Johann and Ghani's fixpoint representation of generalised algebraic datatypes.
- Refactored the target languages of our modular framework to make use of Oliveira and Cook's structured graph representation.
- Demonstrated that the additional structure provided by structured graphs allows us to compile *non*-cyclic control structures such as conditionals in a way that eliminates code duplication.
- Showed that the graph representation also allows for the sensible compilation of cyclic control structures such as while-loops.
- Showed how the compilation schemes for individual features can be redesigned, and that if necessary existing compilation typeclasses can be mapped into modified variants.

Chapter 7

Modular Virtual Machines

7.1 Virtual Machines

The final semantic component of our modular compilation framework is that of a virtual machine which executes a semantics of the target language. For our purposes, this chapter is primarily concerned with demonstrating that the same techniques of modular function construction can be used to define modular semantics for both source and target languages, be they represented via least-fixpoints or structured graphs.

In Chapter 2, we presented two variants of a CPS-style nonmodular execution function. The first, which considered only arithmetic, has type `Code` $\rightarrow$ `Stack` $\rightarrow$ `Stack` for some appropriate datatype `Stack`, whilst the second has type `Code` $\rightarrow$ `Stack` $\rightarrow$ `Maybe Stack` to account for the new possibility of an uncaught exception. Generalising from this, our modular variant might well have type `Code` → `Stack` → `m Stack` for an arbitrary monad `m`. We observe that since `Stack` → `m Stack` is a state transformer, a naive first implementation may have type:

```
type StackT m a = StateT Stack m a


class (Monad m, Functor f) => Exec f m where
  exAlg :: f (StackT m ()) -> StackT m ()
```

In the above, `StackT` is a type synonym for a state transformer parameterised over the aforementioned `Stack` datatype. Within the execution algebra itself, we instantiate the metavariable of `StackT` to the void result type `()`, indicating that following execution we are interested in the value of the `Stack` rather than any potential result value.

7.1.1 In Defence of Non-Modular Stacks

At this point, we make the design choice of using a non-modular representation – i.e. standard Haskell lists – of the stack for the purposes of clarity. The purpose of this subsection is to demonstrate why we do this, by considering the alternative and assuming that we have implemented the stack of a virtual machine operating over arithmetic as the following:

```
data Integer e    = Intgr Int e

data Null    e    = Null

type ModularStack = Fix (Integer :+: Null)
```

The intuitive instantiation of the naive execution algebra above for the `ARITH` functor which uses this stack representation is:

```
instance (Monad m) => Exec ARITH m where

  exAlg (PUSH n st) = pushint n >> st

  exAlg (ADD st)    = addstack  >> st
```

The intended meaning of the above should be evident from a first reading, but contains subtleties due to the fact that the algebra carrier is defined as a state transformer. Specifically, the `pushint` and `addstack` operations produce state transformers which are anonymously composed (recall that the result type is always `()`) with the continuation state transformer `st`.

Whilst the above appears straightforward, consider the underlying implementation. In contrast with the usage of least-fixpoints for the representation of the target language of our modular compilation algebra – wherein we are only interested in *building up* final results from an initial continuation – when using the same structure for a stack we are interested in the potential to inspect values at specific locations. As such, we must define modular variants of combinators we take for granted when working with typical lists:

```
class Functor f => ModularList f where
  modHead :: f (Fix f) -> Fix f
  modTail :: f (Fix f) -> Fix f
```

We note two points in the above: that taking the head of a modular list returns a least fixpoint - this is due to the fact that modular stack constructors are also parameterised by their continuations. As such, when taking the head of a list, we simply prepend the result to an empty stack. For the sake of brevity, we will not define the instantiations of the `Integer` and `Null` functors for this class, as they are trivial to implement. Suffice it to say, the amount of such boilerplate required quickly becomes prohibitive, exceeding the size of the program implementing the modular execution algebra itself. The smart constructors for both injecting an integer value into a stack and updating the stack transformer computation appropriately are defined as follows:

```
intgr n st = inject $ Intgr n st
pushval n = update (\st -> intgr n st) >> return ()
```

In contrast, the definition of `addstack` is complicated by the inability to efficiently pattern-match on our modular representation of a stack:

```
addstack st = let (Just n) = getInt (modHead st)
                  st'      = modTail st
                  (Just m) = getInt (modHead st')
                  st''     = modTail st''
              in  intgr (n + m) st''
```

The above assumes that we have access to a smart destructor `getInt` similar in nature to `getNum` as defined in Chapter 6.3.2.

At this point, we declare that whilst defining a modular execution algebra using modular auxiliary datatypes such as stacks *is* indeed possible, we do not consider doing so instructive, as the code required to implement the required combinators both detracts from the main aims of the chapter and is – in our opinion – not worthwhile in terms of utility gained. As such, for the rest of this chapter we will treat `Stack` and other such constructs as standard Haskell lists of typical ADTs.

7.2 Executing Structured Graphs

In light of the above subsection, we will be operating over a non-modular version of a virtual machine. In conjunction with a stack representation, we also require access to the key-value map `St` originally introduced in

Chapter 6.3.1. Thus, we refer to the collection of auxiliary datatypes which make up the representation of a virtual machine as the *configuration*:

```
type St    = Map Ref Int

type Stack = [Elem]

type Conf  = (Stack, St)
```

We could well inculcate `St` into the definition of `Stack`, pushing the values associated with variables onto the stack as soon as they are introduced, and searching through the stack to update them whenever necessary. However, such an update operation would be O(`n`) in the length of the stack, rather than O(`1`) when using `St`. As such, we opt for the above tuple representation instead.

Having decided that key-value pairs are recorded externally to the stack, we must decide what *can* appear on the stack. We define the datatype of stack elements `Elem` as follows:

```
data Elem = VAL Int | VALB Bool

          | HAN'    | HAN (Conf -> Conf) | STATE Ref
```

There are two points to note about the definition. Firstly, the execution of exception-handling constructs introduces mutual recursion between the definitions of the stack and the virtual machine configuration via the argument to the `HAN` constructor. Secondly, we will see in the next subsection how the (currently unused) constructor `HAN'` can be used to factor out this mutual recursion by compiling exceptions with respect to an alternative compilation scheme.

Now that we have chosen the representation of the virtual machine, we must decide upon the definition of the execution function over structured graphs. Given this graph structure, we can describe the aspects related to cyclicity and sharing – as identified in Chapter 6.4.1 – separate to the core semantics, which can be viewed simply as a tree structure. Recall the type signature of the most general type of fold applicable to structured graphs:

```
gfold :: Functor f => (t -> c) -> (([t] -> [c]) -> c)
                   -> (f c -> c) -> Graph f -> c
```

We note that the first two arguments of `gfold` do not fit into the general structure of the modular functions we have seen up to this point, and further highlight that these arguments correspond to the treatment of sharing and cyclicity respectively. Consider the following typeclass definition of the execution algebra, using `Conf` → `Conf` as the semantic domain:

```
class Functor f => Exec f where
  exAlg :: f (Conf -> Conf) -> Conf -> Conf
```

The above algebra fits into `gfold` as its third argument, handling the definition of the core semantics. For the other two cases (i.e. where the argument to `gfold` is a `Var` or `Mu` constructor) we *unravel* the graph structure and fold over the resulting tree using `exAlg`. This recursion scheme is the underlying intuition of the `cfold` combinator [OC12]:

```
cfold :: Functor f => (f t -> t) -> Graph t -> t
cfold = gfold id (head . fix)
   where fix :: (a -> a) -> a
         fix f = let r = f r in r
```

This *cyclic fold* combinator turns each `Mu` in a graph into a fixpoint computation, which corresponds to the intuition above. Given this combinator, we define the modular execution function of the virtual machine as follows:

```
exec :: (Functor f, Exec f) => Graph f -> Conf -> Conf
exec = cfold exAlg
```

The observant will note that we have not yet made any reference to the execution of modular lambda-calculus terms as introduced in Chapter 5.3.3. The reason for this is that we have not yet described either the semantics of the target machines we seek to emulate, or the auxiliary data structures required to implement them, however both points will be addressed in Chapter 7.4.

7.3 Modifying Language Features

Recall from Chapter 4.3 that exception handlers are compiled via the two stack instructions `MARK` and `UNMARK`, which respectively place a handler onto the stack and remove it. This compilation scheme is very general, as it allows for the compilation of exceptions for a language where the context of a *throw*-statement is not statically known. However, if we know precisely which exception handler is associated with each occurrence of *throw* – which is the case in the presence of all of the features that we have introduced within this thesis–, we can define an alternative compilation scheme that exploits this property. Consider the following representation of the `EXCEPT` signature functor:

```
data EXCEPT' e = THROW' e | MARK' e | UNMARK' e
```

Note that `MARK'` only has *one* argument, as it is not provided with handler code. Instead, the handler code is given to the `THROW'` instruction. The execution semantics of the new instructions follow:

```
instance Exec EXCEPT' where
  exAlg (THROW' c) (k, s)
    = unwind' c (k, s)
  exAlg (MARK' c) (k, s)
```

```
      = c (HAN' : k, s)
    exAlg (UNMARK' c) (e : HAN' : k, s)
      = c (e : k, s)


  unwind' c (HAN' : k, s) = c (k, s)
  unwind' c (_ : k, s)    = unwind' c (k, s)
  unwind' c ([], s)       = ([], s)
```

We cannot compile the source signature `Except` into this modified target signature `EXCEPT'` using the original compilation algebra typeclass `Comp`. To do this, we require a slight variation of the compilation algebra:

```
  type Trio a = a -> a -> a


  class (Functor f, Functor g) => AlgCoGE f g where
    algCoGE :: f (K (Trio (GraphT g v))) l
                  -> Trio (GraphT g v)
```

The additional `GraphT g v` argument in the carrier represents the exception handler that is currently in scope. For simplicity, we pick `HALT` as the initial exception handler:

```
  comp' :: (AlgCoGE f, HALT :<: g) => FixH f i -> Graph g
  comp' e = MkGraph (unK (foldH (K . algCoGE) e) haltG haltG)
```

However, we do not have to redefine compilation algebras for features that are not affected by this change, as we can embed the existing definitions from `AlgCoG` into `AlgCoGE`. The only feature requiring redefinition is `Except`, as it is directly affected:

```
instance AlgCoG f g => AlgCoGE f g where
  algCoG' x h c
    = algCoG (hfmap (K . (\f -> f h) . unK) x) c

instance (EXCEPT' :<: g) => AlgCoGE Except g where
  algCoG' (Throw)               h c = throwG' h
  algCoG' (Catch (K x) (K h')) h c
    = letx c (\v  -> markG' |> letx (h' h   |> Var v)
                  (\vh -> x (Var vh) |> unmarkG' |> Var v))
```

We now turn our attention to describing how terms within both modular variants of the lambda calculus – as introduced in Chapter 6 – fit into this execution framework.

7.4 The Operational Semantics of the λ-Calculi

As mentioned in Chapter 5.3.3, a more operational explanation of the modular representations and compilation schemes implementing the call-by-value and call-by-name recursion schemes is required. In this section, we define their operational semantics and execution algebra implementations.

Recalling the compilation schemes $\mathbb{C}(\mathtt{t})$ for the Categorical Abstract Machine and $\mathbb{K}(\mathtt{t})$ for the Krivine machine from Chapter 5.3.3, tables 7.1 and 7.2 below describe the operational semantics of both machines in terms of transformations over a tuple consisting of an environment and a heap:

Code	**Env**	**Heap**		**Code**	**Env**	**Heap**
`IND` n; c	$\mathbf{e}$	$\mathtt{h}$	$\rightarrow$	c	$\mathbf{e}$	$(\mathbf{e} \,!!\, n)$; $\mathtt{h}$
`CLS` k; c	$\mathbf{e}$	$\mathtt{h}$	$\rightarrow$	c	$\mathbf{e}$	$[k, \mathbf{e}]$; $\mathtt{h}$
`APP`; c	$\mathbf{e}$	$\mathtt{v}$; $[d, \mathbf{f}]$; $\mathtt{h}$	$\rightarrow$	d	$\mathtt{v}$; $\mathbf{f}$	c; $\mathbf{e}$; $\mathtt{h}$
`RET`; c	$\mathbf{e}$	$\mathtt{v}$; d; $\mathbf{f}$; $\mathtt{h}$	$\rightarrow$	d	$\mathbf{f}$	$\mathtt{v}$; $\mathtt{h}$

TABLE 7.1: Call-by-Value Operational Semantics for the Categorical Abstract Machine

The notation $[c, \mathbf{e}]$ used by both of these tables is shorthand for a closure consisting of a code fragment c and current environment $\mathbf{e}$. Looking at the instructions for the CAM in Table 7.1, we highlight the fact that variable lookup corresponds simply to indexing into the environment via the de-Bruijn index (corresponding to the number of binders that must be 'jumped' over to reach the appropriate binding site), and function application is performed by reducing both the function to a closure and the argument to a value, and then calling the code fragment of the closure with the associated environment updated with the argument value.

When considering the instructions for the Krivine machine of Table 7.2, the heap represents the spine of the lambda-term being executed, and we note in particular that β-reduction is performed by calling `GRAB`.

Code	**Env**	**Heap**		**Code**	**Env**	**Heap**
`ACS` 0; c	$[d, \mathbf{f}]; \mathbf{e}$	h	$\rightarrow$	d	$\mathbf{f}$	h
`ACS` $(n+1)$; c	$[d, \mathbf{f}]; \mathbf{e}$	h	$\rightarrow$	`ACS` n; c	$\mathbf{e}$	h
`GRAB`; c	$\mathbf{e}$	$[d, \mathbf{f}]$; h	$\rightarrow$	c	$[d, \mathbf{f}]; \mathbf{e}$	h
`PSH` c'; c	$\mathbf{e}$	h	$\rightarrow$	c	$\mathbf{e}$	$[c'; \mathbf{e}]$; h

TABLE 7.2: Call-by-Name Operational Semantics for the Krivine Machine

The transitions in the above tables directly inform the definitions of the datatypes used by our virtual machine to implement both schemes:

```
type Heap = [Thunk]

type Env  = [Thunk]

data Thunk = Clsr (LConf -> LConf) [Thunk]

           | Thk [Thunk]

           | Cnt (LConf -> LConf)

type LConf = (Stack, St, Env, Heap)
```

```
class ExecLC f where

     exAlg' :: f (LConf -> LConf) -> LConf -> LConf
```

We state at this point that we are defining the auxiliary datatypes for the implementation of the lambda-calculi separate to the stack and variable map defined for all of the other features we have discussed up until this point. Whilst they could be inculcated into the existing structures – in much the same way as the variable map could be merged into the stack – we make this decision as a separation of concerns, and to clarify the semantics. This new typeclass `ExecLC` accounts for the fact that the lambda calculus makes exclusive use of `Heap` and `Env`.

As an aside, whilst terms in the lambda calculus do not make use of either `Stack` or `St`, we include them both in the updated configuration `LConf` above so that they can be accessed by other features when being executed, the details of which we shall see shortly. The instances of `ExecLC` implementing both evaluation schemes for the lambda calculus follow:

```
instance ExecLC LAMBDAV where

  exAlg' (IND i c) (s, m, e, h)

    = c (s, m, e, (e !! i) : h)

  exAlg' (CLS k c) (s, m, e, h)

    = c (s, m, e, ((Clsr k e):h))
  exAlg' (APP c) (s, m, e, (v:(Clsr d f):h))
    = d (s, m, (v:f), ((Cnt c):(Thk e):h))
  exAlg' (RET c) (s, m, e, (v:(Cnt d):(Thk f):h))
    = d (s, m, f, (v:h))


instance ExecLC LAMBDAN where
  exAlg' (ACS n c) (s, m, e, h)
    = let (Clsr d f) = (e !! n)
      in d (s, m, f, h)
  exAlg' (GRAB c) (s, m, e, clsr:h)
    = c (s, m, clsr:e, h)
  exAlg (PSH c' c) (s, m, e, h)
    = c (s, m, e, (Clsr c' e):h)
```

Given that we have already defined a number of execution algebras over an existing configuration, instead of redefining them over the richer configuration associated with the execution of lambda-calculus terms, we repeat the technique used in Chapter 7.3 of embedding one typeclass into another:

```
instance Exec f => ExecLC f where
  execAlgLC c (s, m, _, _) =
    embedLC (execAlg (stripLC <$> c) (s, m))
```

```
stripLC :: (LConf -> LConf) -> Conf -> Conf

stripLC f (s, t) = let (x, y, _, _) = f (s, t, [], [])

                    in (x, y)


embedLC :: Conf -> LConf

embedLC (s, t) = (s, t, [], [])
```

The above instance minimises the need to write duplicate code wherein the only changes required add dummy variables to the configuration.

7.5 Chapter Summary

The end of this chapter represents the conclusion of the presentation of our modular compilation framework. Whilst we discuss the impact and future research potential of this work as a whole in the next (and final) chapter, the salient points of *this* chapter are:

- We justify our representation of the current configuration of a modular virtual machine as a tuple of non-modular lists as being both simpler to comprehend and as a reduction of necessary boilerplate.

- We show how features that are compiled via distinct schemes may necessitate distinct execution algebra carriers, and furthermore, demonstrate how embeddings between execution algebras can eliminate the need to rewrite code that is unaffected .

- We present the operational semantics of two implementations of the lambda calculus – namely the Categorical Abstract Machine and Krivine's machine – in table form, and then implement both as instances of a new execution algebra with an extended carrier, defining an embedding between the two.

Chapter 8

Discussion & Conclusion

To conclude, this final chapter presents a general overview of the thesis and a discussion on both its motivation and to what extent it has succeeded in its original aims, as well as a number of directions for future work.

8.1 Retrospection

The desire to construct compilers that are provably correct in a systemic manner has existed since the 1950s. This thesis is itself the third in a trilogy under Graham Hutton; the first being that of Joel Wright on compiling and reasoning about exceptions and interrupts [Wri05], published in 2005, and the second being that of Liyang Hu, treating the compilation of concurrency in a correct manner via a verified software transactional memory model [Hu12], published in 2012.

My own work on this topic began in 2010, with my original goal being to develop a compiler over a modularised variant of a language based upon Hutton's Razor, as presented in Chapter 4. The intention was that each constituent feature could be proved correct in the style of Wright, and the combination of these proofs would itself constitute a proof for the compound language. However, it was quickly realised that the truly interesting material lay in the interactions between features themselves, not necessarily the proofs that they were being compiled in a sane manner. With that said, such a proof technique would go a long way towards motivating the usage of the ideas we have presented as an alternative to the deep embedding of DSLs such as Functional MetaPost [Hob15]. As it stands, the framework we have developed is currently best suited to exploring the requirements placed on source expressions for experimental DSLs.

The discovery that a program containing a given set of features may be compiled into different instruction sets (Chapter 5) depending on its intended semantics was a direct consequence of the shortcomings of the monad transformer approach of Haskell. Notably, the first two techniques which we present to permit the compilation of noncommutative effects were borne from a desire to produce a working solution from the monad transformer approach itself, and the subsequent realisation that we were simply pattern-matching upon the type constructors of the transformer stack. These techniques are themselves subject to modularity concerns (concerning the presence of commutative intermediate transformers), but are intended as an alleviation, not a cure.

Attempts to construct source programs which could act as running examples for our framework highlighted the need to introduce syntax associated with control-flow (Chapter 6). By doing so, the desire for syntactic categories within source syntax and a target representation capable of supporting cyclicity arose, and hence justified the presence of higher-order source functors and structured graphs.

Finally, the work on constructing modular virtual machines (Chapter 7) is intended primarily as a mechanism for establishing the equivalence between terms that been evaluated, and compiled and executed. Whilst the presence of the virtual machine was originally attributed to the desire to construct inductive proofs of correctness à la Wright, the algebras for the lambda calculus prove interesting reading. The initial approach we took to virtual machines – that they are simply computations producing state transformers – proved useful in that we have adapted it into the justification for non-modular auxiliary datatypes. Further, this initial approach led us to the decision that monads should not appear within virtual machine carriers, as we use them solely to manifest and describe effectful operations.

The thesis as a whole concerns the piecemeal definition and construction of functions that manipulate syntax trees, with particular emphasis on the relationships between syntax drawn from distinct features. We have merged together the seminal work on modular monadic semantics [LHJ95b], the datatypes à la carte technique for extensible datatypes (alongside showing how this technique can support multi-sortedness), and Haskell's rich static type system to form a novel framework allowing for a denotational semantics, operational semantics or syntactic transformation to be defined for a language in an easily-composable manner. By introducing the languages in question as representing the features that can be invoked by a programming language, we have been able to scale up from trivial examples (arithmetic) to treatments of features with fully-fledged associated fields of research behind them (the lambda calculus).

Our final result is a set of features which, when combined, corresponds to a substantially expressive source language. Associated with this language is a fully defined, intuitive semantics that can be inspected on a per-feature basis. In addition, we present a schema for transforming programs from the source language into a target language consisting of stack-based instructions. However, we are not fixed in this choice of target: we have, for example, been able to successfully target the graph representation of the Hoopl dataflow analysis library [RDPJ10] in the same manner. To our knowledge, this technique of transforming syntax piecemeal alongside the ability to produce different results according to the presence and ordering of certain features is unique.

8.2 Potential Future Research Directions

To reiterate the claim we made at the end of Chapter 3, the compilation of EDSLs in a modular manner is a particularly rich source topic, and whilst we are satisfied that the work presented by this thesis constitutes a novel and powerful set of techniques for manipulating both the syntax and semantics of the source and target languages associated with a given EDSL, there is no shortage of ways in which such a framework can be improved upon. In particular, the question of what exactly would be required for this approach to 'scale up' to a production quality DSL compiler (or a general compiler) is worth considering, requiring as it were a far more formal specification of the range of types and constructors available for source, target and execution algebras. In this subsection, we list a number of additional avenues that we feel are worthy of future consideration.

Additional Computational Features: extending our framework with support for additional features is a project that holds significant potential, as there will always be ‘something else’ that might make an ideal candidate for integration. In particular, we would be interested in seeing the introduction of other forms of control flow such as continuations, explicit parallelism and I/O, as well as any domain-specific features unique to the DSL we wish to introduce (most likely to present as combinations of the former with those we have introduced in this thesis). Furthermore, the recursion schemes used to define the compiler and the language semantics can be extended: more structured recursion schemes derived from tree automata [Bah12, BD13] and attribute grammars [VS12] offer more freedom to *replace* parts of a given modular definition as opposed to only being able to *extend* them. In the same direction goes the work of Kimmell et al. [KKA05] and Frisby et al. [WKFA] which introduce algebra combinators such as `switch` and `sequence` to compose algebras. Finally, it would be interesting to investigate how one might exploit the algebraic theory of effects to give a principled understanding of the complexity involved in integrating a new feature based upon the effectful methods it provides.

Attribute Grammars: The Utrecht University Attribute Grammar Compiler (UUAGC) [SAS$^+$99] is a Haskell preprocessor which simplifies the construction of catamorphisms over tree-like structures. Moreover, the UUAGC supports open data types and functions, providing an excellent foundation for modular programming. Future investigation into the extent to which the UUAGC system is suitable for further highlighting the independence of individual signature functors and their associated semantics appears promising. There are also a number of standalone attribute grammar systems with particular focus on extensible language implementation such as LISA [MU05], JastAdd [EH07] and Silver [VWBGK08].

Modular Syntax: As identified in Chapter 6.2.1, the usage of higher-order functors to represent indexed datatypes and families of mutually recursive datatypes stems from Johann and Ghani [JG08], however Yakushev et al. [YHLJ09] also applied this technique to generic programming. Axelsson [Axe12] introduces a different approach to modular well-typed definitions of syntax and semantics: said work develops an applicative encoding of syntax making heavy use of type indexing to describe the signature of individual language features.

Indexed Type Families: In Haskell, the indexed type family extension [CKJM05], which permits ad-hoc overloading of datatypes, may prove useful in explicitly declaring a link between the signature functors of a source and target language for a particular effect. For example, in Chapter 4.3 we defined an evaluation algebra mapping terms constructed from the source functor `Arith` into terms constructed from the target functor `ARITH`. At present, we are capable of compiling into *any* target language, provided it supports the requisite signatures. Declaring a type family with a functional dependency in order to define a mapping from a source language `FixH f i` to a target language `Graph (Target f)` may ensure the target language is minimal, removing the requirement that the user pre-defines the target language.

Automatic Context Inference: An observation [1] that arose during my research is that it may be possible to use the ordering of signature functors in the type of a source expression to automatically infer the monadic context within which we wish to evaluate it. For example, given a term with signature `(Arith :+ Except :+ State)`, one might infer that it is to be evaluated within a monad built up from the identity monad – corresponding to `Arith` – by first applying the exception transformer, and then applying the state monad transformer. Such an interpretation could prove to be useful as a method of providing a default behaviour, which a user can override if they wish. With that said, further differentiation would likely be required if, for example, a state-carrying signature appeared twice (perhaps both before and after an exception-based signature).

Alternative Target Languages: As presented in this thesis, we compile into a stack-based language. It would be useful to consider how our framework can be adapted to other forms of target language, in particular register-based languages such as LLVM [LA04], which can be used as the target language for many imperative language compilers or logic-based languages such as System F [Rey74], a variant of which is used by GHC.

[1] Personal Communication: Wouter Swierstra, March 2013

Dataflow Analysis and Optimisations: A natural extension of our work is the implementation of dataflow analysis and optimisations in a modular style. A good starting point for extending our work in this direction is *Hoopl*, the *Higher Order Optimisation Library* of Ramsey et al. [RDPJ10]. Hoopl is a Haskell library that allows compiler representations to define dataflow analyses and implement optimising transformations that are informed by said analyses. Modular implementations of optimising transformations can be achieved using the same techniques as presented in this thesis. However, as mentioned in the previous subsection, we have observed that dataflow analysis and the underlying lattice structures can be defined in a modular manner for – at least – standard textbook analyses.

Testing and Reasoning: An important property of a compiler is its trustworthiness. Does it perform only semantics preserving transformations? Establishing such trustworthiness in a modular fashion remains a considerably challenge. However, using the same techniques as presented here, automatic test case generation (i.e. generation of input programs and initial configurations) can be implemented in a modular fashion. Rigorous and machine-checked correctness proofs, however, require new reasoning techniques that work in a modular setting. There is growing interest in formalising programming language metatheory in a modular fashion [DdSOS13, DKSO13, SS13]. However, the process in of building modular proofs of compiler correctness includes several additional difficulties.

Such proofs must be modular along both the source and target languages as well as its computational effects. As the work of Delaware et al. [DdSOS13] shows, modular reasoning about effects already becomes a considerable obstacle for type soundness proofs.

Bibliography

[AAvS94] Martin Alt, Uwe Aßmann, and Hans van Someren. CoSy: Compiler Phase Embedding with the CoSy Compiler Model. In Peter A. Fritzson, editor, *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 278–293. Springer Berlin Heidelberg, 1994.

[AC71] F. E. Allen and J. Cocke. A Catalogue of Optimizing Transformations. In *Design and Optimization of Compilers*, pages 1–30, 1971.

[AC76] F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Commun. ACM*, 19(3), March 1976.

[All70] Frances E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, 1970.

[App97] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques.* Cambridge University Press, New York, NY, USA, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Axe12] Emil Axelsson. A Generic Abstract Syntax Model for Embedded Languages. *SIGPLAN Not.*, 47(9):323–334, 2012.

[Bah12] Patrick Bahr. Modular Tree Automata. In *Proceedings of the 11th International Conference on Mathematics of Program Construction*, MPC'12, pages 263–299. Springer-Verlag, 2012.

[BBB+57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN Automatic Coding System. In *Papers Presented at the Western Joint Computer Conference: Techniques for Reliability*, pages 188–198, 1957.

[BD13] Patrick Bahr and Laurence E. Day. Programming Macro Tree Transducers. In *Proceedings of the 9th Workshop on Generic Programming*, WGP '13, pages 61–72. ACM, 2013.

[BH11] Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, pages 83–94. ACM, 2011.

[BK01] Nick Benton and Andrew Kennedy. Exceptional Syntax. *J. Funct. Program.*, 11(4):395–410, July 2001.

[Bra] Edwin Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144.

[Bra13] Edwin Brady. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *J. Funct. Program.*, pages 552–593, 2013.

[CCM85] G. Cousineau, P-L. Curien, and M. Mauny. The Categorical Abstract Machine. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. 1985.

[CF94] Robert Cartwright and Matthias Felleisen. Extensible Denotational Language Specifications. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, TACS '94, pages 244–272. Springer-Verlag, 1994.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph.

ACM Trans. Program. Lang. Syst., 13(4):451–490, October 1991.

[Chl08] Adam Chlipala. Parametric Higher-order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, 2008.

[Chu32] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, pages 346–366, 1932.

[Chu36] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[Chu40] Alonzo Church. A Formulation of the Simple Theory of Types. *J. Symbolic Logic*, 5(2):56–68, 06 1940.

[CKJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated Types with Class. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 1–13. ACM, 2005.

[Cur30] H. B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52(3):pp. 509–536, 1930.

[Cur91] P.-L. Curien. An Abstract Framework for Environment Machines. *Theoretical Computer Science*, 82(2):389 – 402, 1991.

[dB72] N.G de Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[DB14] Laurence E. Day and Patrick Bahr. Pick'n'Fix: Capturing Control Flow in Modular Compilers. *Pre-Proceedings of 15th Symposium on Trends in Functional Programming*, 2014.

[DdSOS13] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-Theory À La Carte. *SIGPLAN Not.*, 48(1):207–218, January 2013.

[DF80] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst.*, 2(2):191–202, April 1980.

[DH11] Laurence E. Day and Graham Hutton. Towards Modular Compilers for Effects. *Proceedings of the 12th Symposium on Trends in Functional Programming*, 7193:49–64, 2011.

[DH13] Laurence E. Day and Graham Hutton. Compilation À La Carte. *Proceedings of the 25th International Symposium* *on the Implementation and Application of Functional Languages*, pages 13–24, 2013.

[DKSO13] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular Monadic Meta-Theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 319–330. ACM, 2013.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd System: Modular Extensible Compiler Construction. *Sci. Comput. Program.*, 69(1-3):14–26, December 2007.

[Esp95] David A. Espinosa. *Semantic Lego*. PhD thesis, New York, NY, USA, 1995. UMI Order No. GAX95-33546.

[Gil14] Andy Gill. mtl-2.2.1: the Monad Transformer Library. Hackage, https://hackage.haskell.org/package/mtl, 2014.

[Har01] William Lawrence Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, 2001.

[HH09] Liyang Hu and Graham Hutton. Compiling Concurrency Correctly: Cutting out the Middle Man. In *Trends in Functional Programming '09*, pages 17–32, 2009.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 1–55, 2007.

[HK98a] William Harrison and Samuel N. Kamin. Modular Compilers Based on Monad Transformers. In *In Proceedings of the IEEE International Conference on Computer Languages*, pages 122–131. Society Press, 1998.

[HK98b] William L. Harrison and Samuel N. Kamin. Compilation as Metacomputation: Binding Time Separation in Modular Compilers. In *In 5th Mathematics of Program Construction Conference, MPC2000*, 1998.

[Hob15] John Hobby. funcmp-1.8: Functional MetaPost. Hackage, https://hackage.haskell.org/package/funcmp, 2015.

[Hop87] Grace Murray Hopper. The Education of a Computer. *Annals of the History of Computing*, 9(3):271–281, July 1987.

[Hu12] Liyang Hu. *Compiling Concurrency Correctly: Verifying Software Transactional Memory*. PhD thesis, Nottingham University, 2012.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[Hut10] Hans Huttel. *Transitions and Trees: An Introduction to Structured Operational Semantics.* Cambridge University Press, April 2010.

[HW04] Graham Hutton and Joel Wright. Compiling Exceptions Correctly. In *In Proceedings of the 7th International Conference on Mathematics of Program Construction*, pages 211–227. Springer, 2004.

[Jas09] Mauro Jaskelioff. *Lifting of Operations in Modular Monadic Semantics.* PhD thesis, Nottingham University, 2009.

[Jas11] Mauro Jaskelioff. Monatron: An Extensible Monad Transformer Library. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL'08, pages 233–248. Springer-Verlag, 2011.

[JD93] Mark P. Jones and Luc Duponcheel. Composing Monads. Technical report, Yale University, 1993.

[JG08] Patricia Johann and Neil Ghani. Foundations for Structured Programming with GADTs. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–308, 2008.

[Joh87] Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages* *and Computer Architecture, volume 274 of LNCS*, pages 154–173. Springer-Verlag, 1987.

[Jon01] Simon Peyton Jones. Tackling The Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In *Engineering Theories of Software Construction*, pages 47–96. Press, 2001.

[Kah87] G. Kahn. Natural Semantics. In *STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer Berlin Heidelberg, 1987.

[KKA05] Garrin Kimmell, Ed Komp, and Perry Alexander. Building Compilers by Combining Algebras. *IEEE International Conference on the Engineering of Computer-Based Systems*, pages 331–338, 2005.

[KR35] S.C. Kleene and J.B. Rosser. The Inconsistency of Certain Formal Logics. *Annals of Mathematics*, 36(3), 1935.

[KSS13] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible Effects: An Alternative to Monad Transformers. *SIGPLAN Not.*, 48(12):59–70, September 2013.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code*

Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, 2004.

[Ler90] Xavier Leroy. The ZINC Experiment: An Economical Implementation Of The ML Language. Technical Report 117, INRIA, 1990.

[Ler09] Xavier Leroy. A Formally Verified Compiler Back-End. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[Lev06] Paul Blain Levy. Call-by-Push-Value: Decomposing Call-by-Value and Call-by-Name. *Higher Order Symbol. Comput.*, 19(4):377–414, December 2006.

[LH96] Sheng Liang and Paul Hudak. Modular Denotational Semantics for Compiler Construction. In *In European Symposium on Programming*, pages 219–234. Springer-Verlag, 1996.

[LH06] Andres Löh and Ralf Hinze. Open Data Types and Open Functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 133–144, New York, NY, USA, 2006. ACM.

[LHJ95a] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22Nd*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95, pages 333–343, 1995.

[LHJ95b] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages. ACM Press*, 1995.

[Lia98] Sheng Liang. *Modular Monadic Semantics and Compilation.* PhD thesis, New Haven, CT, USA, 1998. AAI9835276.

[Mö93] Hanspeter Mössenböck. Oberon0 — A Case Study. In *Object-Oriented Programming*, pages 153–213. 1993.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. pages 124–144. Springer-Verlag, 1991.

[MH95] Erik Meijer and Graham Hutton. Bananas In Space: Extending Fold and Unfold To Exponential Types. In *Proceedings of the 7th SIGPLAN-SIGARCH-WG2.8 International Conference on Functional Programming and Computer Architecture.* ACM Press, La Jolla, California, June 1995.

[Mog89] E. Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE Press, 1989.

[Mos89] Peter D. Mosses. Unified Algebras and Action Semantics. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '89, pages 17–35, London, UK, UK, 1989. Springer-Verlag.

[Mos96] Peter D. Mosses. Theory and Practice of Action Semantics. In *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*, MFCS '96, pages 37–61, London, UK, UK, 1996. Springer-Verlag.

[MU05] Marjan Mernik and Viljem Umer. Incremental Programming Language Development. *Comput. Lang. Syst. Struct.*, 31(1):1–16, April 2005.

[MW06] James Mckinna and Joel Wright. A Type-Correct, Stack-Safe, Provably Correct, Expression Compiler in Epigram. In *Journal of Functional Programming*, 2006.

[NMRW02] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language And Tools For Analysis And Transformation Of C Programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002.

[OC12] Bruno C.d.S. Oliveira and William R. Cook. Functional Programming with Structured Graphs. *SIGPLAN Not.*, 47(9):77–88, September 2012.

[OL13] Bruno C. d. S. Oliveira and Andres Löh. Abstract Syntax Graphs for Domain Specific Languages. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, pages 87–96. ACM, 2013.

[PE] F. Pfenning and C. Elliot. Higher-order Abstract Syntax. *SIGPLAN Not.*, 23(7):199–208.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press, 1991.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, University of Aarhus, 1981.

[PP01] Gordon Plotkin and John Power. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2001.

[PP03] Gordon Plotkin and John Power. Algebraic Operations and Generic Effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[PP04] Gordon Plotkin and John Power. Computational Effects and Operations: An Overview. *Electron. Notes Theor. Comput. Sci.*, 73:149–163, October 2004.

[PP08] Gordon Plotkin and Matija Pretnar. A Logic for Algebraic Effects. In *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, LICS '08, pages 118–129. IEEE Computer Society, 2008.

[PP09] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer Berlin Heidelberg, 2009.

[Pro00] Fabienne Prosmans. Derived Categories for Functional Analysis. *Publ. Res. Inst. Math. Sci.*, 36(1):19–83, March 2000.

[RDPJ10] Norman Ramsey, João Dias, and Simon Peyton Jones. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, pages 121–134, 2010.

[Rey74] John C. Reynolds. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408–423. Springer-Verlag, 1974.

[SAS$^+$99] S. Doaitse Swierstra, Pablo R. Azero Alcocer, Joao Saraiva, Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 150–206. Springer-Verlag, 1999.

[See06] Sean Seefried. *Language Extension via Dynamically Extensible Compilers*. PhD thesis, 2006.

[SHLG94] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical Theory of Domains*. Cambridge University Press, New York, NY, USA, 1994.

[SO11] Tom Schrijvers and Bruno C.d.S. Oliveira. Monads, Zippers and Views: Virtualizing the Monad Stack. *SIGPLAN Not.*, 46(9):32–44, September 2011.

[SS71] Dana Scott and Christopher Strachey. Toward a Mathematical Semantics for Computer Languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[SS13] Christopher Schwaab and Jeremy G. Siek. Modular Type Safety Proofs in Agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 3–12. ACM, 2013.

[Swi08] Wouter Swierstra. Data Types À La Carte. *Journal of Functional Programming*, 18:423–436, July 2008.

[TP97] Daniele Turi and Gordon Plotkin. Towards a Mathematical Operational Semantics. In *In Proc. 12th LICS Conf.*, pages 280–291. IEEE, Computer Society Press, 1997.

[Vie12] Marcos O. Viera. *First Class Syntax, Semantics, and their Composition*. PhD thesis, 2012.

[VS12] Marcos Viera and Doaitse Swierstra. Attribute Grammar Macros. In *Programming Languages*, Lecture Notes in Computer Science, pages 150–164. 2012.

[VWBGK08] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An Extensible Attribute Grammar System. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116, April 2008.

[Wad92] Philip Wadler. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14. ACM, 1992.

[Wad95] Philip Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52. Springer-Verlag, 1995.

[Wad98] Philip Wadler. The Expression Problem. Available online at: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, 1998.

[WH97] Keith Wansbrough and John Hamer. A Modular Monadic Action Semantics. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, DSL'97, Berkeley, CA, USA, 1997. USENIX Association.

[WKFA] Philip Weaver, Garrin Kimmell, Nicolas Frisby, and Perry Alexander. Constructing Language Processors with Algebra Combinators. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 155–164. ACM.

[Wri05] Joel Wright. *Compiling And Reasoning About Exceptions And Interrupts*. PhD thesis, Nottingham University, 2005.

[YHLJ09] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic Programming with Fixed

Points for Mutually Recursive Datatypes. *SIGPLAN Not.*, 44(9):233–244, August 2009.

[YWC$^+$12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell A Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66. ACM, 2012.