PhD Transfer Report:
The Modular Compilation of Effects

Laurence Day

Functional Programming Laboratory
University of Nottingham, UK

31^{st} October 2011

Abstract

Compilers are traditionally factorised into smaller components, each dealing with an individual stage of the process, such as parsing, type-checking, code generation and optimisation. The advantage of such an approach is that each component can be designed, implemented, tested and (ideally) proved correct independent of other components. During the course of my first year, I have begun to investigate an alternative method of factorising a compiler, namely according to the effects supported by the source language. More importantly, I seek to develop a compiler in such a way that the addition of a new effect requires minimal extensions to existing code. In this report I present the preliminary results of my research into the code generation stage of the modular compilation of programming languages according to the effects which they support. To this end, I demonstrate an elegant combination of previously developed techniques from the literature to decouple the recursive and syntactic aspects of a source language, and develop a semantics, compiler and virtual machine for such a language which is parameterised over its syntax and the monads which support effectful operations. All prerequisite knowledge to engage with the research material is presented in a background section within the opening chapters of this report. In addition, I present a detailed review of the existing literature concerning compiler modularity and the characterisation of effects, and lay out a preliminary draft for the structure of the doctoral thesis which this research aims towards.

Contents

1 Introduction

"For, usually and fitly, the presence of an introduction is held to imply that there is something of consequence and importance to be introduced."

- Arthur Machen

Consider a simple language comprising integer values and their addition, implemented as an algebraic datatype within the meta-language Haskell. Over such a language, we are able to write functions performing tasks such as pretty-printing and the evaluation of its expressions by pattern matching upon the language constructors. However, it may be necessary to extend the original language with additional functionality, such as the ability to throw and handle exceptions. The extension to the language itself is trivial — simply by adding further constructors — however, the knock-on effect that this has on functions defined over the language is more complex. That is to say, each function must be extended with additional cases to cover the new constructors, and existing cases may need to be altered to reflect a new semantics.

The scenario described above serves to demonstrate a problem ubiquitous with language development, namely that their monolithic representation leaves little room for extensions in the future. This is hardly ideal.

1.1 Making it Modular

"Nothing is particularly hard if you divide it into small jobs."

- Henry Ford

In order to overcome the issue of language extensibility, we need to introduce modularity into our approach. We can define modularity as the degree to which a formal system can be separated into individual components that we can recombine as we wish. In the programming context, this amounts to breaking software down into independent components with clearly defined behaviour which can be combined in a manner specific to an application. The modular approach to programming brings many important benefits, such as the ability to break a large, complex problem down into simpler subproblems, being able to determine a system's correctness via the correctness of its components, and the potential to develop a library of generic components which can be reused in varied application domains.

One particular example of a setting in which the monolithic nature of language implementation is a hindrance is that of compilation. A compiler consists of several stages, each of which relying on (possibly) multiple functions and auxiliary datatypes, with the final compiler constructed from the composition of each stage. The issue of function redefinition in the face of language exten-

sions in such a situation quickly becomes unmanageable. For various reasons this is not ideal, not least because a programmer working on an extension requires familiarity with all aspects of the compiler, as opposed to the one that they are attempting to implement.

The heart of this issue as it relates to programming languages has been dubbed the *expression problem* [**?**], which states that *"...the goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety"*. Techniques providing a solution to the expression problem therefore enable the development of complex systems such as compilers in a modular manner.

Whilst such solutions to the expression problem exist — and will be explained and used in this report — the issue of modular compilation is not a solved problem, and there is no standard approach at present. We are interested in factorising a compiler via a different axis to the parsing, type-checking, code generation tack traditionally taken. Namely, we wish to factorise a compiler via the computational effects supported by the source language, as it is often the introduction of new effects which mandates revision of the semantics of existing functions. We emphasise at this point that we are focussing only on the code generation aspect of the compilation process, as tasks such as syntax analysis are best left to the theory of parsing. By this account, the notion of 'compiler' will appear foreign to those not familiar with functional compilers.

We have observed that the problem appears to be that of adding additional constructors to a language, as the semantics of the language as a whole may change. However, it is usually possible to clearly identify those aspects of a language associated with certain computational effects. The question to ask then is: is it possible to construct a language by the combination of sublanguages, each of which modelling an individual effect? The answer is yes, and the techniques used to both define languages and functions over them in such a manner are key.

Whilst the syntactic structure of a source language presents problems, so too can the methods used in the manifestation of effects in the target language. In Haskell — a purely functional programming language —, effects are enabled through the use of *monads*, a construct adapted from category theory. However, monads too suffer from modularity issues when multiple effects are considered simultaneously. Widely-used techniques exist for overcoming these issues — most importantly the notion of *monad transformer* — however they are not without their own shortcomings. In this report, we will examine monad transformers in detail, identify their flaws and discuss alternative methods of modularising monads.

A critical characteristic of a compiler is that it preserves semantic meaning

when translating from a source to a target language: the compiled program must behave in the same way as the original one regardless of the syntactic differences between them. When using the traditional method of compiler development, proofs of compiler correctness for non-trivial source languages are difficult to produce, although possible. One of our goals is to simplify the process of proving a compiler correct by breaking its proof down into several subproblems which can then be independently verified.

The rest of the report is structured as follows. In section 2, we give brief self-contained introductions to those topics which require familiarity in order to engage with the ideas underlying this research, alongside recommended reading lists for further confidence. In section 3, we provide a review of the literature as it relates to modular compilation at the time of writing. In section 4, we discuss the contributions which have been made in the first year of research, which have been primarily condensed into a symposium paper recently accepted for publication. Finally, in section 5 we outline the chapter titles, projected page lengths and general content direction for the final thesis submission at the end of the period of research, alongside a full set of bibliographic references.

2 Background Theory

"Every philosophy is tinged with the coloring of some secret imaginative background, which never emerges explicitly into its train of reasoning."

- Alfred N. Whitehead

Prior to examining the contributions that have been made in the course of this research programme to date, this section will cover some notation and definitions that the author has had to learn over the previous year in order to engage with the research topic. These are necessarily brief, however further reading lists are provided as an aid to deeper understanding.

2.1 Traditional Semantic Approaches

As discussed in the previous section, the central tenet of the correctness of a compiler is that the meaning of a program is preserved by translation. There are multiple approaches which can be taken when giving a semantics to a language, some of which more suitable than others for certain purposes. In this section, we will give quick introductions to the three most popular styles.

2.1.1 Denotational Semantics

The *denotational semantics* of a language, as originally investigated by Scott and Strachey in the 1960s, 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 share common structure, which we review in section 2.2. Importantly, a denotational semantics must be *compositional*: that is, the meaning of a program phrase can only be constructed via the meaning of its subphrases. To illustrate with an example, consider the following simple language in Haskell:

```
data Arith = Value Int | Add Arith Arith
```

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

```
[[ _ ]]         :: Arith → ℤ
[[ Value n ]] =  n
[[ Add x y ]] =  [[ x ]] + [[ y ]]
```

This approach to semantics presents with some issues, primarily because a denotational semantics cares only for the underlying meaning of an expression and ignores the multiple steps that a function may take. This leads to gaps when

attempting to give a denotation to nonterminating values. Additional structure upon semantic domains is required, which we will introduce in section 2.2.

Nonetheless, denotational semantics is a rich topic, and there is ongoing research into giving denotational semantics to constructs representing notions of concurrency and nondeterminism, amongst others. However, there are modularity issues with this approach that we shall identify when discussing our research contributions in section 4.

Recommended further reading:

- D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*, William C. Brown Publishers, 1988 [25]

2.1.2 Operational Semantics

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

Recall the language `Arith` introduced in section 2.1.1. 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 and a_1, a_2, ... to range over terms in `Arith`:

Structural Operational Semantics Rules

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

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

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

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

Natural Semantics Rules

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

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

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, contrasted with *evaluation* with respect to the denotational semantics of section 2.1.1. We can implement the rules above with ease in Haskell, referring to the overall construct as an *abstract machine*.

However, the extension of a language as described in section 1 results in having to alter such operational rules to account for additional features, such as the introduction of an environment construct.
Recommended further reading:

- H. Huttel, *Transitions and Trees: An Introduction to Structural Operational Semantics*, Cambridge University Press, 2010 [7]

2.1.3 Action Semantics

As an alternative to the two semantic styles described above, Mosses developed action semantics as a way to precisely specify what he deemed to be 'realistic' programming languages, using the notion of *actions* (or computations) as the driving force. Rooted in structural operational semantics and unified algebra, an action semantic description is characterised by both allowing data to be accessed through *yielder* operations, and splitting program behaviour up into *facets* according to the data associated with particular behaviours. For example, the *basic* facet handles control flow, the *declarative* facet deals with binding values to variables, and so forth.

Action semantics is highly accessible to newcomers, in part due to its verbose notation. Consider, for example, the action semantics for an "if-then-else" fragment [32]:

```
execute ⟦ if E then S1 else S2 ⟧ =
    evaluate E then
    | | check it and then execute S1
    | or
    | | check not it and then execute S2
```

The intended interpretation of such a statement is clear from a first reading. Using small combinators such as `check` and `execute`, and yielders such as **it**, the action semantics for such constructs are easy to build up. Descriptions such as these are intended to be modular - for example, if the programming language being specified maintains a store, then the store need not be referenced in the above description as it is handled by a different facet.

The underlying behaviour of the combinators described is expressed using Plotkin's structural operational semantics of section 2.1.2. The downside to this is the same as that of SOS itself: the rules are defined in a non-modular fashion, in stark contrast to the apparent modularity of the faceted approach to action semantics; whereas the equation above does not reference the declarative facet, the rules which power the combinators must. Extensibility, in this case, is poor.

Furthermore, those programming language features which can be described by action semantics are set in stone, and there is no plan at the time of writing for extension; the facets which Mosses gives are the only ones available. Most notably, continuations cannot be given an action semantic description, limiting its expressive power. Recently, however, there have been attempts to merge action semantics with alternative approaches such as the modular monadic semantics described in section 2.5.1, with mixed success [32].

Recommended further reading:

- P. D. Mosses. *Theory and Practice of Action Semantics*, Springer, 1996 [22]

2.2 Domain Theory

In section 2.1.1 we stated that in some instances, we require additional structure on the semantic domain to provide sensible denotations for particular constructs, such as functions which do not terminate. The notions we require for this structure are part of a field called *domain theory*. In this section, we introduce the key ideas of domain theory as it applies to defining a denotational semantics for a language represented in Haskell.

Consider the following value `loop`:

```
loop :: Bool
loop  = not loop
```

We cannot give a denotation of either `True` or `False` to this value, as it fails to terminate. How, then, are we to assign an object in the semantic domain `Bool` to $[\![$ `loop` $]\!]$? The answer is to introduce an element of 'undefinedness' into our domain, with the understanding that this element somehow represents less information than any object of the original domain. We refer to this element as

$\bot$ — pronounced 'bottom' — and it is this element assigned which is assigned to $[\![$ `loop` $]\!]$ as a denotation.

This notion of carrying less information corresponds to imposing a *partial order* on the semantic domain; namely a relation $\sqsubseteq$ on a domain D obeying the following axioms:

1. Reflexivity: $\forall$ d $\in$ D . d $\sqsubseteq$ d
2. Antisymmetry: $\forall$ d, e $\in$ D . d $\sqsubseteq$ e $\wedge$ e $\sqsubseteq$ d $\Rightarrow$ d = e
3. Transitivity: $\forall$ d, e, f $\in$ D . d $\sqsubseteq$ e $\wedge$ e $\sqsubseteq$ f $\Rightarrow$ d $\sqsubseteq$ f

Assuming that D does not already contain $\bot$, we refer to (D $\cup$ $\{\bot\}$) as the *lifted domain* $D_\bot$, and the partial order must enforce the additional constraint that $\forall$ d $\in$ $D_\bot$. $\bot \sqsubseteq$ d, or that $\bot$ is a least element of the domain. A lifted domain equipped with a partial order is referred to as a partially ordered set (*lifted poset*). Adding one final constraint, we require that the partial order D is *complete*, that is to say that every non-empty subset of D has a supremum, or 'greatest element'. It is precisely these complete partial orders (*cpo's*) that Haskell programs accept as semantic domains for denotations.

Recommended further reading:

- V. Stoltenberg-Hansen, I. Lindström, E. R. Griffor. *Mathematical Theory of Domains*, Cambridge University Press, 1994 [27]

2.3 Category Theory

Category theory is a mathematical tool used to capture the essence of interactions between mathematical objects with common structure. Within our research, 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 of characterising the nature of a category, since we do not require too much detail in this report we simply utilise the following:

A *category* $\mathbf{C}$ is defined to be:

1. A collection $\mathbf{C}_{ob}$ of *objects*, which we denote with metavariables $\mathbf{X}$, $\mathbf{Y}$, ...
2. A collection $\mathbf{C}_{ar}$ of *morphisms* between the objects from C_{ob} (also called *arrows*). For example, $f : \mathbf{X} \to \mathbf{Y}$ is a morphism from objects $\mathbf{X}$ to $\mathbf{Y}$, provided both objects are defined in the set $\mathbf{C}_{ob}$. We denote morphisms with metavariables f, g, ...

3. A pair of functions *dom* and *cod* operating on morphisms from C_{ar}, used to obtain the domain and codomain of a morphism (taking the view of morphisms as functions); e.g. given $f : \mathbf{X} \to \mathbf{Y}$, $dom(f) = \mathbf{X}$, $cod(f) = \mathbf{Y}$.

Further to this, a category must adhere to the following rules:

1. For all objects $\mathbf{C}$, there is an associated morphism $id_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$.
2. For all morphisms $f : \mathbf{B} \to \mathbf{C}$, it must hold that $id_{\boldsymbol{C}} \circ f = f = f \circ id_{\boldsymbol{B}}$.
3. For all morphisms f and g where the domain of g matches the codomain of f, there is an associated morphism $(g \circ f) : dom(f) \to cod(g)$.
4. For all morphisms f, g and h, it must hold that $(f \circ g) \circ h = f \circ (g \circ h)$.

Categories can represent a vast array of different mathematical structures, depending on the choice of $\mathbf{C}_{ob}$ and $\mathbf{C}_{ar}$. For instance, the category **Top** has topological spaces as objects and continuous functions as morphisms, 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.

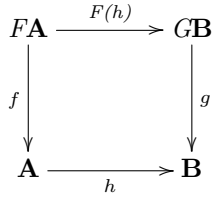
Further, in some categorical instances, particular objects can be denoted as being special in one way or another. We call an object $\mathbf{X} \in \mathbf{C}_{ob}$ *initial* if for all other objects $\mathbf{Y} \in \mathbf{C}_{ob}$ there is a unique morphism $f : \mathbf{X} \to \mathbf{Y}$. Similarly,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

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 C_{ar}. Given two categories $\mathbf{C}$ and $\mathbf{D}$, a functor F between them must obey the following conditions:

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

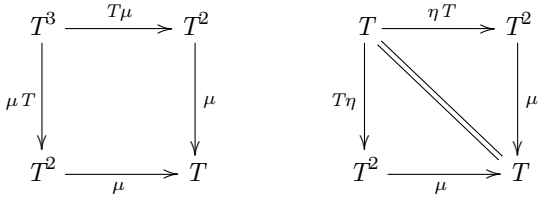
A considerable amount of the research we have done thus far involves the terms *F-algebras* and *catamorphisms*. Although we will be utilising these in a purely

functional context, it is worthwhile understanding the underlying categorical notions for clarity. Given a functor $F : \mathbf{C} \to \mathbf{C}$ over category $\mathbf{C}$ (known as an *endofunctor*), we call a pair $(\mathbf{A}, \alpha)$ where $\mathbf{A} \in \mathbf{C}_{ob}$ and $\alpha : F\mathbf{A} \to \mathbf{A} \in \mathbf{C}_{ar}$ an *F-algebra* on *F*. In the category **F-Alg** with such F-algebras as objects, morphisms between the objects $(\mathbf{A}, f)$ and $(\mathbf{B}, g)$ are defined as the arrow h which makes the following diagram commute:



Mathematically, the above diagram states that we require $\mathbf{F}h \circ g = f \circ h$. The unique morphism h between an *initial algebra* $(\mathbf{A}, in)$ and any another F-algebra $(\mathbf{B}, f)$ is called a *catamorphism on f*, also referred to as a *functorial fold*. The paper included as an appendix gives further details.

One final notion we introduce at this point is the categorical definition of a *monad*. Given a category $\mathbf{C}$, a monad on $\mathbf{C}$ is an endofunctor T equipped with two morphisms between arrows (known as 'natural transformations'), namely $\eta : 1_{\mathbf{C}} \to T$ (where $1_{\mathbf{C}}$ is the identity functor on $\mathbf{C}$) and $\mu : T^2 \to T$. Such a monad must obey the following conditions:



Monads are used to formally characterise computational effects, and it is for this reason precisely that we introduce them. The connection between monads and effects is made in the literature survey of section 3. However, we are primarily interested in the functional programmer's definition of a monad, which differs somewhat, which we investigate in the next section.

Recommended further reading:

- S. Mac Lane. *Categories For The Working Mathematician*, Springer, 1998 [13]
- B. Pierce. *Basic Category Theory for Computer Scientists*, MIT Press, 1991 [23]

2.4 Implementing Categorical Constructs

The ideas which we have explored so far in this research have been implemented using Haskell as a metalanguage. As mentioned, two particular notions which are of particular importance to this work are monads and functors. We now spend some time introducing and explaining these concepts, their departures from their typical categorical implementations and their relation to our work.

2.4.1 Functors and Catamorphisms

We have introduced functors as morphisms between categories, which preserve structure on both the objects and morphisms of the categories themselves. In Haskell, the `Functor` class 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`. 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 x y) = Node (fmap f x) (fmap f y)
```

The above declares that `Tree` is a generic container type with the capability of applying functions to its contents.

A considerable amount of the implementation of our research thus far revolves crucially around the concept of folding over **F**-algebras. In Haskell, the least fixpoint type constructor and generic fold operator are defined as:

```
type Fix f = In (f (Fix f))


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

Observe that `Fix` f ties the recursive 'knot' of a functor f: some may be more familiar with the notation μf. The justification for representating datatypes as least fixpoints — as well as why we utilise `fold` — is made explicit in the appendix [**?**]. It is worth noting that in categorical terminology, `Fix` f is an *initial* datatype.

Recommended further reading:

- E. Meijer. *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*, Springer, 1991 [18]

2.4.2 Monads and Monad Transformers

Haskell, being a pure functional language, does not permit side-effects by definition. However, the vast majority of Haskell programs require some notion of manipulating data in an impure way, be it through requiring exceptions, mutable state, nondeterminism, input/output or a combination of these and others. To introduce such impure computational effects to a pure language, the concept of *monads* was introduced by Wadler via the following class declaration:

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

The `return` operation places a pure value into a (often trivial) impure monadic computation, and the `(>>=)` operation (pronounced *'bind'*) is a way to sequence computations. These operations must satisfy the three *monad laws*:

1. `return a >>= f` $\simeq$ `f a`
2. `m >>= return` $\simeq$ `m`
3. `(m >>= f) >>= g` $\simeq$ `m >>= (λx -> f x >>= g)`

To be more precise, the concept of monad that we use in Haskell corresponds categorically to that of a *strong* monad on a monoidal category, but this is beyond the scope of this report.

By way of example, the `Maybe` datatype can be instantiated as a monad modelling exceptions. The datatype and monadic operations are defined as follows:

```
data Maybe a = Nothing | Just a

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

The monad laws can be verified for this example relatively simply.

In practice, programs regularly need to employ multiple computational effects. However, each monads traditionally only models a single effect. This quickly leads to the necessity of adding additional effects to a monad, hence the notion of *monad transformer*. Monad transformers are treated at length in the appendix [**?**], however we briefly introduce the concept here:

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

The associated operation `lift` embeds a value in a base monad `m` into the 'higher' monad `t m`. In this fashion, operations associated with effects foreign to a monad `m` can be added to it in a straightforward manner.

Recommended further reading:

- P. Wadler. *The Essence of Functional Programming*, ACM Press, 1992 [30]
- P. Wadler. *Monads for Functional Programming*, Springer, 1995 [31]
- S. Liang, P. Hudak and M. Jones. *Monad Transformers and Modular Interpreters*, ACM Press, 1995 [16]

2.5 Alternative Semantic Approaches

All of the semantic approaches mentioned in section 2.1 suffer from a lack of modularity. When extending a language with additional functionality, the denotational approach may require extensions to the interpretation function to handle new cases, as well as potentially having to reflect a new semantics (when introducing exceptions, for example). Similarly, operational rules may need to be rewritten to dictate the changes — or lack thereof — applicable to newly introduced constructs (adding an environment). And finally, an action semantics is fundamentally non-modular due to the static nature of facets.

However, in section 2.4.2 we were introduced to the idea that an effect can be 'isolated' somewhat via monads and monad transformers. It is worthwhile, then, to investigate their usage in defining a semantics for a language.

2.5.1 Modular Monadic Semantics

Directly addressing the lack of modularity of the traditional denotational semantics of section 2.1.1, Liang, Hudak and Jones developed the notion of modular monadic semantics as a *structured* denotational semantics, revolving around monads and monad transformers as the key descriptive mechanism.

Recall that a monad in Haskell is a type constructor `m` with operations:

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

Using these operations, a denotational semantics can be given to a language which is parameterised by the monad given in the result type. The modular monadic semantics for a language `L` which supports addition can be given thus:

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

Depending on the monad m, the concrete semantics which result differ. For example, observe the return and bind definitions for the two monads representing state and an environment:

```
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, all that remains is to define each aspect of the language in question in such a monadic style. This goes some way to alleviating the issues characteristic of the 'traditional' semantic approaches.

Recommended further reading:

- S. Liang and P. Hudak. *Modular Monadic Semantics*, 1997.

3 Literature Survey

"The difficulty of literature is not to write,
but to write what you mean; not to affect your reader,
but to affect him precisely as you wish."
- Robert L. Stephenson

In this section, we briefly survey a selection of previous work that is most relevant to work on the modular compilation of effects. The papers have been loosely grouped into subcategories to maintain coherence. For each paper, the major contributions are introduced and any particularly interesting or relevant concepts explained. Connections to our research will be made in section 5, which details the planned thesis structure.

3.1 Monads as Computations

In this section, we examine the work credited with both identifying and popularising the idea of modelling computational effects using monads within the category theory and programming communities.

Computational Lambda-Calculus and Monads:
Eugenio Moggi, IEEE Press, 1989 [20]

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 the object $\mathbf{A}$ of *values* of type $\mathtt{A}$ and $\mathtt{T}\mathbf{B}$ of *computations* of type $\mathtt{B}$, where $\mathtt{T}$ is a 'monad' which captures the side-effects that the computation may have. This type of morphism belongs to the *Kleisli category* $\mathbf{C}_\mathtt{T}$ constructed from a base category $\mathbf{C}$, with the identity and composition of morphisms given by the natural transformations associated with $\mathtt{T}$ as defined in section 2.3. For example, in the category **Set**, the monad $\mathtt{T}_\mathbf{ND}$ for nondeterminism is: $\mathtt{T}\mathbf{A} = \mathcal{P}(\mathbf{A})$, $\eta_\mathbf{A}(\mathrm{a}) = \{\mathrm{a}\}$ and $\mu_\mathbf{A}(\mathrm{X}) = \cup\mathrm{X}$.

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 λ_c-*models* over a category $\mathbf{C}$ (where such a model is a strong monad with $\mathtt{T}$-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, ACM Press, 1992 [30]

Following Moggi's discovery that effectful computation can be modelled using monads [20], 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 contrasting these changes to the fact that an impure interpreter would need no such restructuring. Following this, an interpreter for a language `Term` based upon the lambda calculus is introduced in Haskell, and the functional characterisation of monads, as defined in section 2.4.2, is established. It is then shown how the 'standard' interpreter is extended to support a single additional features, ranging from exceptions to nondeterminism. For each extension, the monad `M` and associated operations `unitM` and `bindM` are redefined to support the feature in question, and the lines of the interpreter which are associated with the feature identified and changed appropriately. Further, the changes which need to be made to the interpreter such that it uses call-by-name evaluation — instead of call-by-value — are discussed.

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 :: (a -> b) -> (M a -> M b)` and `joinM :: M (M a) -> M a`, and that these two sets of laws follow from each other. The laws are then used to prove that addition is associative in any monadic interpreter.

The paper goes on to contrast the monadic style of programming used in the interpreter and it's extensions to 'continuation passing style' (CPS). To this end, the continuations monad is introduced and used to structure the interpreter for the unextended language `Term`: it is observed that the resulting interpreter is similar to the interpreter based upon 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 data type: e.g. CPS cannot provide an 'error escape' for a language with exceptions.

Finally, it is noted that some syntactic sugar may go some way to aid 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, Springer, 1992 []

Building upon the work described in the previous paper [30], 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 and defining the associated *monadic* interpreter. 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 previous 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 help define *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 data types ensuring the *single-threadedness* of the array: a crucial condition for updating an array in a 'safe' manner. 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.

3.2 Alternate Computation Characterisations

Whilst our programme of research is heavily reliant on the usage of monads to model computational effects, they are not the only way to do so. In this section, we survey papers which seek to provide alternative characterisations of impure computation.

Computational Effects & Operations: An Overview:
Gordon Plotkin and John Power, Elsevier, 2002 [24]

Whilst the commonly held view of operations associated with computational effects (`inEnv`, `callcc` et al) is that they are derived from the underlying monad `T`, there is a diametric view where the operations themselves are taken as primitive, and a monad is derived using the operations as constraints. It is this latter approach investigated in this paper, centering around the notion of *Lawvere theories* (categorical *equational theories*, the collection of all equations that hold for a particular algebraic structure), moreover countable enriched Lawvere theories – *countable* meaning that the operations and equations form a countable set, *enriched* meaning that the hom-sets of the category are replaced by objects from another category in a 'sensible' manner.

The main notion of the paper is that a description of an effect can be given as a countable Lawvere theory freely generated by the operations expected of

the effect, with a correspondence existing between the theory morphisms and so-called *algebraic* operations. A number of examples are given in the paper, including nondeterminism and partiality amongst others.

In order to combine computational effects given as countable enriched Lawvere theories, one takes the disjoint union of the operations and equations alongside a number of additional equations relating the two classes – either, no equations, in which case we say the combination is the sum of effects, or extra equations which result in either the commutative or distributive combination. A number of categorical definitions explain the general rule alongside additional theorems relating *models* of Lawvere theory combinations. The paper concludes by discussing the structural operational semantic descriptions for Lawvere-theoretic constructs.

Combining Effects: Sum and Tensor: **Martin Hyland, Gordon Plotkin & John Power, 2006** [9]

This paper extends the mathematical theory of combining strong monads — and the computational effects they model — via the notion of Lawvere theories. More precisely, the paper seeks to define binary operations $\circ$, $\otimes$ and $\star$ on strong monads that correspond — respectively — to the 'computationally natural' combination of exceptions with I/O and (probabilistic) nondeterminism, side-effects with the same, and interactive I/O with all effects other than state. To this end, computational effects are reformulated in the context of enriched Lawvere theories, and the above operations are defined over Lawvere theories L and L'. It is observed that the operations $\circ$ and $\star$ correspond to the sum of theories — as previously discussed —and $\otimes$ to their *tensor* or *Kronecker product*.

Despite the paper focussing Lawvere theories enriched over categories such as ω-**Cpo**, the underlying concepts are introduced in the unenriched setting (modelling effects in **Set**), before expanding into ω-**Cpo**. It is worth noting that local state is not discussed in this paper as it doesn't correspond to a 'natural' combination of exceptions and side-effects. Furthermore — and more importantly — continuations are not covered, as the continuations monad has no rank (and so cannot generate a countable theory). Finally, exceptions as discussed in the paper only permit the throwing of exceptions — not their handling — as the `handle` operation is not algebraic.

The paper goes on to detail the generation of Lawvere theories from both the sum and commutative combination of operations and equations defining multiple effects, and the calculation of the tensor product of side-effects with other effects. Unfortunately, the details of the proofs and related discussions are given in categorical detail beyond the author of this report. The paper concludes by discussing the notion of *operation transformers* — liftings of algebraic operations —, which are roughly equivalent to monad morphisms.

3.3 Compilation and 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.

Monad Transformers and Modular Interpreters:
Sheng Liang, Paul Hudak and Mark Jones, ACM Press, 1995 [16]

This paper focusses upon notion of using so-called 'building blocks' — corresponding to individual features — which can be combined to form programming language interpreters. The data constructors and supported operations expected of a number of effects are introduced, alongside the monad transformers which are used 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 are given detailing the conditions which `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 way if that feature is removed.

It is observed that if — categorically — a monad cannot compose with all other monads, an associated monad transformer 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 [12]. As such, there is no nondeterminism monad transformer. 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 environments emulated modelled using the state 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 — liftings, mostly involving continuation, are given.

Modular Compilers Based on Monad Transformers:
William L. Harrison and Samuel Kamin, Society Press, 1998 [3]

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 'standard' semantics

of a feature, presented in monadic continuation-passing style, the compilation semantics are obtained by introducing *pass separation*, the introduction of intermediate data structures via 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 'machine language' represented using appropriate combinators (e.g. `popblock` and `push`), 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 respectively — is demonstrated, followed by the combination of the two, with additional equations (*"semantic glue"*) relating the features. The language is then extended to a simple lambda-calculus and compiled similarly. The paper concludes with a discussion of issues which have arisen, such as scoping and recursion.

Compilation as Metacomputation: *Binding Time Separation in Modular Compilers:* **William L. Harrison and Samuel Kamin, Springer, 1998** [5]

This paper builds upon the authors' previous work on constructing modular compilers as discussed above. It is observed 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, for example stack manipulation. This 'staging' of a program can be implemented using monads — namely, a static and a dynamic monad — constructed using monad transformers. The composition of these monads is claimed to give 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 of 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 the staging monads using transformers simplifies the combination of building blocks. Given the staging monads (`Comp` and `Exec`) 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, Doctoral Thesis, 2001** [4]

Harrison's doctoral thesis focusses the problem of modular compilation in the sense discussed in 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 in the previous paper — and as monadic code generators.

The examples and discussion relating to the metacomputational approach correspond strongly with that in the previous paper, including the introduction of the standard semantics $[\![\_]\!]$ and compilation semantics $C[\![\_]\!]$ of a language, alongside theorems given relating the two. The chapter concludes with a lengthy exposition on the metacomputational compilation of both open and closed procedure declarations.

The alternative approach — compilation using monadic code generators (MCGs) — follows, defining an MCG as a function `compile :: Source → m Target` for each feature, parameterised by a monad `m`– this approach being more closely related to that of our own research. The same features used as examples for the metacomputation 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.

Whilst a level of reusability is attained by using RCBBs, this reusability is not unrestrained. Certain conditions must be met in order to combine certain 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, Springer, 2004 [8]

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 and virtual machine operating upon 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. Whilst the introduction of exceptions permits functions to be written via monadic short-hand, for the sake of proof the desugared versions are the ones worked with. A second proof of generalised compiler correctness relating to the 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 compiled handler code — make reference to address locations. This final compiler is proved correct, with the aid of five inductively proved lemmas concerning such issues as the monotonicity of the address allocation function `isFresh`.

Compiling Concurrency Correctly: Cutting out the Middle Man:
Liyang Hu and Graham Hutton, Intellect, 2010 [6]

Traditionally, compiler correctness for concurrent languages is proved via translating from both the source and target languages into an intermediate π-calculus (a formal system giving semantics to concurrent computations) and proving their equivalence via bisimulation. However, this method is overly complicated, requiring several extra layers of formalism. This paper describes a simpler technique. Taking the same approach as the previous paper, a language `Zap` of integer values and addition is defined and - via Agda - given an operational semantics involving *actions* and *labels* which permits a simple notion of nondeterminism by allowing an addition to be 'zapped' to a zero result.

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 proven 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, Springer, 2009 [14]

A large portion of the literature regarding compiler correctness considers a variation on the lambda calculus as the 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 — called `Cminor` — as the source language, and targeting PowerPC assembly code, the formally verified — by the Coq theorem prover — compiler *Compcert* is presented. It is observed that the semantic preservation property of a compiler can be established by by proving the *forward simulation for safe programs* property. This states that the compilation process implies that both the source and target program produce the same set of observable behaviours, with no behaviour classified as 'incorrect'. Also required is a proof that the target language is deterministic. Since Compcert targets assembly code — which is 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 using a processor-specific set of operators. Further stages address issues such as light optimisation (e.g. constant propagation) and register allocation. At each stage, semantic preservation is proved, and detailed semantics of intermediate languages, of which there are several, are given. Noteworthy is the fact that whilst compiler correctness papers often concentrate only on the code generation aspect of compilation, Compcert supports a far larger number of the tasks expected of a realistic compiler.

The paper concludes by discussing the lengths of the Coq verification scripts relative to the program being verified. For programs written in `Cminor`, it is noted that the verification script is often roughly six times the size of the program being verified. The possibility of retargeting Compcert is also discussed, and shown to be possible by retargeting the ARM instruction set.

3.4 Semantics Done Modularly

Throughout this report, we have identified and discussed the fact that traditional semantic approaches are beset by issues relating to their modularity and extensibility. 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, Springer, 1996 [15]

Extending the authors' previous work on monad transformers as described in section 3.2 [16], this paper aims to derive compilers from modular interpreters based on modular monadic semantics. 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*. For instance, function application is defined using the environment monad transformer operations, `rdEnv` and `inEnv`. Following this, focus shifts to constructing the `Compute` monad — which hides the computations of a source program — via monad transformers.

It is observed that since a modular monadic semantics is no more than a structured denotational semantics, the monad laws described in section 2.4.2 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 authors axiomatise the environment monad with four laws — e.g. the fact that inner environments supercede outer environments — and use these to further simplify programs.

The paper goes on to raise the question of whether it is possible to axiomatise other effects in a similar manner, but does not expand upon this. Finally, it is observed that multiple languages can be targeted by comparing the features supported by both monad and the target language, taking the view of a target language as a 'base monad'. For example, considering the source language used as the running example through the paper, the ML 'monad' has a richer set of supported features and therefore requires no additional operations, but if the target language were to be C, some monad transformers would need to be applied in order to enable the missing required operations.

Lifting of Operations in Modular Operational Semantics:
Mauro Jaskelioff, Doctoral Thesis, 2009 [11]

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 approaches taken within both Haskell and category theory, such as the 'shadowing' of operations: wherein two applications of the same monad transformer result in the associated operations of one transformer becoming inaccessible.

Split into two sections — Theory and Applications —, the former introduces all categorical constructs used throughout, and identifes operations as mappings on the *category of monoidal categories* and classifying them as '*H-algebraic*', '*first-order*' or '*algebraic*' depending on their structure. It is then

demonstrated — in categorical detail beyond the scope of the author — how several examples of such operations are lifted in a canonical manner. It is then demonstrated that where several such liftings are possible, they coincide. Key to this is the classification of monoid transformers as '*monoidal*', '*functorial*', '*covariant*' or none of these.

The theory developed is then implemented in Haskell using the *Monatron* library [10]. Monatron differs from the *mtl* in its use of multiple classes of monad transformer. Detailed examples are given, most notably for the exceptions monad. Here, the associated `throw` and `handle` operations are separated and lifted using the appropriate techniques for their algebraic class. Monatron is used extensively in the final two chapters, firstly in developing a modular interpreter for a language which supports processes, conditional arithmetic and exceptions using the *à la carte* technique, and secondly in implementing a modular operational semantics, based upon work by Turi [29].

Semantic Lego:
David A. Espinosa, Doctoral Thesis, 1995 [2]

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 monads is reviewed, alongside an interesting section on alternative ways to 'view' monads than the 'monads equals computation' notion of Moggi [20]. The theory of 'stratification' — the splitting of a monad into levels — is then introduced: for example, the monad T_1`(A) = S` $\rightarrow$ `A` $\times$ `S` is on a 'higher' level than T_2`(A) = A` $\times$ `S`. Stratified levels can then be combined using *stratified monad transformers*, which are classified as *top*, *bottom* or *around* transformers according to their structure. Two examples are given by the exception transformer `F(T)(A) = T(A + X)`, which is a bottom transformer, and the environment transformer `F(T)(A) = Env` $\rightarrow$ `T(A)` which is a top transformer.

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 computation ADT 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. The thesis concludes with a chapter detailing a history of related work in the history of lifting operations, drawing links between previous literature and observations relating to the theory of stratification.

Modular Monadic Action Semantics:
Keith Wansborough and John Hamer, , 1997 [32]

Action semantics and modular monadic semantics (MMS) have been previously introduced in section 2 of this report. Differing in their underlying

formalisms, both styles suffer from issues regarding extensibility and accessibility: an action semantics is fundamentally limited in the number of features it can describe, and MMS uses syntax which can be confusing to the layman. This paper proposes merging the best aspects of both styles, resulting in *modular monadic action semantics* (MMAS): action semantics where combinators are described using MMS as opposed to structural operational semantics.

This change enables the extensibility that action semantics lacks: if a feature cannot be described with action notation (e.g. continuations) then we 'drop down' a layer and implement said feature via the supporting MMS. As MMS is no more than structured denotational semantics, the usual methods for reasoning about programs — e.g. β-reduction — therefore apply. However, unaddressed issues remain: problem areas in denotational semantics restrict some aspects of the facets which MMS seeks to emulate. For example, concurrency and nondeterminism are not representable in MMAS.

Modular Structural Operational Semantics: Peter D. Mosses, Elsevier, 2005 [21]

As discussed in section 2.1.2, structural operational semantics (SOS) lacks modularity in a fundamental sense. That is to say, extending a language with additional data constructors corresponding to a feature may result in the existing rules being totally rewritten in order to accommodate the new feature. Attempting to overcome this issue, this paper proposes *modular structural operational semantics* (MSOS), which uses 'label categories' to increase the reusability of the rules written within it.

A structural operational semantics is traditionally underpinned by a *labelled terminal transition system* (LTTS), given as $\langle \Gamma, \mathtt{A}, (\rightarrow), \mathtt{T} \rangle$, where: Γ is a set of *configurations* (states of a transition system), $\mathtt{A}$ is a set of labels, $(\rightarrow) \subseteq \Gamma \times \mathtt{A} \times \Gamma$ is a transition relation, and $\mathtt{T} \subseteq \Gamma$ a set of *terminal* configurations such that $\gamma \xrightarrow{\alpha} \gamma_\alpha$ infers $\gamma \notin \mathtt{T}$. Computations in a LTTS are given by a sequence of transitions where the final configuration $\gamma_n \in \mathtt{T}$. A modular structural operational semantics is described by a *generalised transition system* (GTS) and a bisimulation equivalence. A GTS is given as $\langle \Gamma, \mathbb{A}, (\rightarrow), \mathtt{T} \rangle$, where $\mathbb{A}$ is the *category* with morphisms the set of labels A, such that a LTTS $\langle \Gamma, \mathtt{A}, \rightarrow, \mathtt{T} \rangle$ exists and can be derived from the GTS.

Rules in MSOS are written the same way as in SOS, but modularity is obtained by the structure of the category $\mathbb{A}$. Two transitions γ_i and γ_j are consecutive only if the associated label morphisms are composable in $\mathbb{A}$. If the language contains an environment, two transitions can compose only if their labels are identical, as a transition cannot change an environment. In this case, we take $\mathbb{A}$ to be a *discrete* category. If a language supports a store, two transitions can compose only if the resulting store of γ_i is the same as the starting store of γ_j, in which case $\mathbb{A}$ must be a *pre-order* category. Other

language features impose further constraints on $\mathbb{A}$. The combination of various effects in this manner is done by taking the product of the appropriate label categories.

3.5 Functional Pearls

Within any literature survey, there are relevant papers investigating methods which do not fall into a category such as the ones given above, but are still of importance to the direction of current and future research. In this final section of the literature survey, we examine three such 'functional pearls'.

Monads, Zippers and Views: Virtualizing the Monad Stack:
Tom Schrijvers and Bruno C. D. S. Oliveira, ACM Press, 2011 [26]

As identified in work previously surveyed [11], a number of problems exist when lifting operations through the concrete monad stack: e.g. having to edit the number of calls to `lift` if the stack is extended. This recent 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 'ignores' a prefix of a monad stack using operations $\uparrow$ – a '`lift`' of sorts – and $\downarrow$, an inverse which has no traditional 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 different parts of a computation.

Together these notions allow the development of a masking language which allows parts of a monad stack to either be hidden or have their access restricted. 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 'tag' each layer of a monad stack with a singleton type, allowing operations to be applied to the appropriate level by simply referencing the tag, which enables the usage of two state transformers `State1` and `State2` without using `lift`.

Datatypes à la Carte:
Wouter Swierstra, Cambridge University Press, 2008 [28]

This paper — which forms a crucial part of our own research programme — describes a solution to the expression problem described in section 1, using techniques for constructing data types and functions in a modular manner, 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 data type can be represented as the fixpoint of a coproduct of functors — which represent constructor signatures —, and functions over such data types can be represented as algebras which are folded over said fixpoints.

Terms written in this *à la carte* style are, unfortunately, cumbersome to use due to the number of type constructor tags involved. To bypass this, the paper introduces the notion of a *subtyping relation* on functor coproducts, which underpin *smart constructors*: constrained functions injecting data into modular terms. Following a discussion of how certain well known monads (e.g. identity and `Maybe`) are 'free' monads — effectively trees parameterised over a functor —, the paper goes on to discuss 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 'buttons' for storing and recalling values.

Parametric Compositional Datatypes:
Patrick Bahr and Tom Hvitved, 2011 [1]

The work by Swierstra described above is a particularly elegant solution to the expression problem in Haskell. However, the insights it reveals are of limited applicability outside the setting of this problem. Previous work by the authors of this paper has sought to extend the *à la carte* approach in the form of *compositional data types* (CDTs). The CDT framework supports wider functionality for recursion and mutually recursive datatypes. However, it was observed that CDTs cannot implement variable bindings due to issues concerning α-equivalence.

The above issue is resolved in this paper by merging the *à la carte* approach 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 meta-language: for example, in Haskell a data constructor `Lam String e` can be represented as `Lam (e → e)`. However, this combination of methods cannot support recursion over abstract syntax trees (ASTs), as data constructors defined in this way are not functors. The solution involves changing the representation of signatures as functors — as used in *à la carte* — to difunctors [19], and altering the structure of terms to that of a free monad. The combination of compositional data types and higher-order abstract syntax — alongside the above changes — forms a technique referred to as *parametric compositional data types* (PCDTs).

It is shown how monadic computations can be implemented using both CDTs and PCDTs, before discussing *term algebras*, which are algebras with carrier `Fix f` for some `f`, and *term homomorphisms*, functions of type `∀ a. f a → Context g a`, where `Context` is a free monad. It is then demonstrated how these algebras and homomorphisms compose. The paper concludes by introducing *monadic* term homomorphisms and showing how mutually recursive data

types and generalised algebraic data types can be constructed using PCDTs.

4 Research Contributions To Date

"If you want to live a long life, focus on making contributions."
-Hans Selye

At the conclusion of the first year of this doctoral programme, the work that has been done has been primarily condensed into a paper recently accepted for publication by the Trends in Functional Programming symposium [**?**]. The paper itself is provided as an appendix, however in this section we briefly summarise the main contributions.

4.1 Towards Modular Compilers for Effects

Compilers have traditionally been factorised according to the phases of the compilation, including parsing, type-checking and code generation. In this paper, we approach compilation from a different axis, namely that of individual language features — such as exceptions and concurrency — and seek to design compilers for these features independently. We begin by introducing a simple language of integers and addition, defining its syntax, semantics, compiler and virtual machine, before extending the language with exceptions and identifying the areas which require modification.

Following from this, we investigate the notion of monad transformers in depth and demonstrate their implementation using the exceptions monad transformer, defining all associated methods and lifting these methods through the state monad transformer. We then introduce the *à la carte* technique, and apply it to the extended language, clearly separating the arithmetic and exceptional constructors and defining them as functors. We go on to discuss the reasoning for representing a data type as a fixpoint of functor coproducts, and reformulate the semantics for this language using an 'evaluation algebra' which is folded over the fixpoint which represents our modularised language. It is then observed that expressions written in this style present issues, and introduce the notions of the subtyping relation and smart constructors to overcome this. Sample expressions are evaluated within a number of monads to demonstrate the flexibility of our technique.

Following this, we proceed to implement the compiler for this language in much the same way as we defined the semantics by using a 'compilation algebra'. However, it is noted that the target language must be chosen in advance, although represented as a fixpoint. In this case, we do not yet have the truly modular solution we are working towards. We conclude by defining a virtual machine — by way of an 'execution algebra' — as a modular interpreter which produces suitable monadic state transformers.

We conclude by discussing a number of directions for future work, most im-

portantly regarding the issue of modularity in the target language of compilers constructed using the techniques we have introduced, and alternatives to using monad transformers to represent effectful computations due to their issues as discussed in Jaskelioff's thesis [11]. An important note is that the definition of the syntax and semantics of a language modularised in such a manner is particularly elegant, and the work remains to be done focusses on the compilation and execution of expressions written within modular languages.

5 Thesis Plan

"My general plan is good, though in the detail there may be faults."
- Adam Weishaupt

The work done in the previous year and the TFP paper which resulted will constitute roughly a third of the final thesis. In this section, a preliminary draft of the full structure of the thesis is presented with estimated page counts and a brief discussion of the contents therein.

Introduction

Projected page length: 5pp
In the opening chapter of the thesis, we shall introduce the problem that we seek to solve by way of an extended example, namely the definition of a language and the construction of its syntax, semantics, compiler and virtual machine, before extending the language with an additional constructor and observing the issues encountered. It is likely that the introduction of the paper comprising the appendix will be expanded into this section, as the extended example it focusses on is both simple and informative. We shall discuss the aims of the project, provide a brief summary of the final results (to be expanded upon throughout the thesis), and outline a guide to the layout of the remainder of the thesis.

Compilers

Projected page length: 10–15pp
Following on from the first chapter, this section will contrast the traditional definition of a compiler to that used throughout the rest of the thesis. The issues associated with traditional compilation methods within Haskell — as initially identified in the previous chapter — will be discussed at greater length. We will define the compilers for several example languages supporting several features and observe how the introduction of the monadic style increases readability and code reusability. The expression problem will be discussed, and the notion of factorising a compiler into blocks centred around computational effects will be proposed.

Monads

Projected page length: 15–20pp
After a detailed introduction to the categorical characterisation of monads, Moggi's notion of 'monads as computation' will be introduced in this section alongside a discussion of other ways in which monads have been viewed in the past. The fact that monads within Haskell are categorically strong monads — monads equipped with a *tensorial strength* — will be discussed, and several

examples of such monads given. The question of how to combine monads in order to support multiple effects will be posed, and the problems which arise concerning the composition of monads will be identified. Methods of overcoming these problems: such as taking the coproducts of monads [17] will be discussed, however the notion of monad transformers will only be introduced in the section on modular semantics.

Modular Syntax

Projected page length: 15–20pp
In this section the *à la carte* technique of Swierstra will be examined in great detail, using the example given in the TFP paper, namely that of decoupling the arithmetic and exceptional components of the `Expr` language. The importance of these signatures being functors will be highlighted, before demonstrating how the recursive knot in such a signature can be tied via the `Fix` operator. The natural extension of taking the fixpoint of a single signature is that of taking coproducts of signatures and taking the fixpoint of these, and the relationship between the original language and this representation will be examined. Smart constructors will be introduced as a method of bypassing the cumbersome tagging process involved in the construction of values within such a language, alongside a discussion of the underlying subtyping relation on functors. Finally, it may prove interesting to discuss the categorical theory behind Haskell datatypes.

Modular Semantics

Projected page length: 20–25pp
In defining a languages' syntax in a modular fashion, we enable the modularisation too of the specification of the language. As an introduction to this section, we provide an overview of the important aspects of denotational semantics such as compositionality and domain. Referring back to the chapter on monads, the fact that a single monad supports a single effect will be stated and the issue of providing semantic support for languages supporting multiple such effects raised. Monad transformers being the solution to this problem, an in-depth guide to monad transformers will be given — in much the same style as discussed in the TFP paper, albeit with multiple instances defined — and the notion of lifting of operations introduced. We define effect classes (e.g. `ErrorMonad` for exceptions) and demonstrate how operations can be lifted through other effect classes via the `lift` operator supported by the `MonadTrans` class. Following from this, we give a demonstration of the separation semantics for a simple source language (ideally arithmetic and exceptions, as in the TFP paper). We explain how separated semantics are capable of supporting effectful operations corresponding to an effect class via explicit constraints, and point out how the parametricity of the underlying monad offers true modularity in the sense we desire. It is important to note that this technique as it stands is based upon the assumption

that the `mtl` library is used; however it may well be the case that Monatron is used in the final implementation.

Modular Compilers

Projected page length: 15–20pp
Contrary to the thesis title, the construction of a compilation function between modularised languages is relatively simple, as it can be viewed as the construction of a modular interpreter between languages that does not involve the manifestation of effects. In this section, we will discuss the techniques applied in the construction of a generalised compiler function — as of the time of writing, via the catamorphism of a compilation algebra, however this notion may change in due time — between a modularised source language towards a modular target language characterised by a signature, improving upon the designation within the TFP paper of a specific `Code` target, requiring knowledge of the language being compiled. Regardless, we will demonstrate the construction of modular compilation blocks for a simple language by way of illustration. We will discuss how the structure of the compiled language datatype affects the approach taken by the virtual machine, which we discuss in the next chapter.

Modular Virtual Machines

Projected page length: 20–25pp
Whereas the modular semantics of a language corresponds to a denotational specification, the virtual machine represents the implementation of a (small-step) operational semantics, and this fact will be introduced at some length alongside an introduction to operational semantics as a specification method. We remind the reader of the fact — introduced in the chapter on compilers — that the modular compiler correctness proofs we seek are represented as an equivalence between the denotational specification and operational semantics of a source language. Referring again back to the introductory chapters of the thesis, we identify the structure of the execution function characterising the virtual machine and generalise the result type to an arbitrary monad. As it stands, the TFP paper defines the virtual machine as a metacomputation, a state transformer predicated on the target language of the compiler. We will demonstrate the implementation of the modular virtual machine for a simple language and draw parallels between results obtained via evaluation and execution on a small set of sample expressions, with formal proofs of the correctness of the system discussed further on in the thesis if the direction of future research permits.

Alternative Approaches

Projected page length: 1–5pp
The structure of the target language for the modular compiler has, thus far, been defined in the style of a finite list of instructions within the *à la carte*

syntax style. Whilst this is convenient for applying the techniques previously developed in the field, it may prove advantageous to investigate alternative data structures and decoupling methods for language syntax. One particular such approach is that of implementing the virtual machine as a *register machine* as opposed to the stack machine approach we have taken in the TFP paper. It is unknown at this stage what methodology changes would result from such a shift.

Extended Case Studies

Projected page length: 25–30pp
Having established the techniques for the modularisation of each important aspect of a language and its compiler, in this section we present multiple languages with several effects associated with each and implement their syntax, semantics, compiler and virtual machine. We will investigate how — as is the case with the currently used techniques — there is an observable difference in the manifestation of effects dependent on the order of lifting, as well as how such constructs as continuations (presuming continuations are considered) alter the intended meaning of a language. It is our intention to conclude this section with a language which supports every effect discussed throughout the thesis as a final example, in much the same style as previous literature.

Modular Proofs

Projected page length: ?pp
Having discussed the compiler correctness property for a modular language and its' compiler, we would like to visibly demonstrate the correctness of the system we have constructed via either equational reasoning or a theorem prover. The author suspects that whilst these modularisation technique will invariably lead to simpler proofs, a pen and paper approach to correctness will simply prove too lengthy an exercise; to this end, we seek to use a system such as Agda or Coq to formulate and prove the correctness of modular compilers.

Related Work

Projected page length: 10–15pp
In this section, we will review in detail all of the previous literature relating to the development of both modular compilation and the notion of monads as a vehicle for computational effects within a pure functional language. Although the literature review section provided in this transfer report represents a significant portion of said background, in the coming three years there is the potential for relevant publications to appear. As such, the literature will be re-reviewed in its' entirety come the entering of the write-up phase in order to frame the thesis in the most recent context.

Summary/Conclusion

Projected page length: 5pp
In this concluding section, a synopsis of all of the major topics will be provided, as well as a reiteration of those concrete contributions of the thesis to the scientific literature. We will consider how any results may affect future research in the field, and propose extensions to the work done in order to further refine the techniques developed.

6 References

References

[1] P. Bahr and T. Hvitved. Parametric Compositional Data Types. University of Copenhagen, June 2011.

[2] D. A. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[3] W. Harrison and S. 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.

[4] W. L. Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[5] W. L. Harrison and S. N. Kamin. Compilation as metacomputation: Binding time separation in modular compilers (extended abstract). In *In 5th Mathematics of Program Construction Conference, MPC2000, Ponte de*, 1998.

[6] L. Hu and G. Hutton. Compiling Concurrency Correctly: Cutting out the Middle Man. In Z. Horvath and V. Zsok, editors, *Trends in Functional Programming volume 10*. Intellect, 2010.

[7] H. Huttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010.

[8] G. Hutton and J. Wright. Compiling Exceptions Correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, pages 211–227. Springer, 2004.

[9] M. Hyland, G. Plotkin, and J. Power. Combining effects: sum and tensor.

[10] M. Jaskelioff. Monatron: An Extensible Monad Transformer Library. In *Implementation and Application of Functional Languages*, 2008.

[11] M. Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.

[12] M. P. Jones and L. Duponcheel. Composing monads. Technical report, 1993.

[13] S. Lane. *Categories For The Working Mathematician*. Graduate texts in mathematics. Springer, 1998.

[14] X. Leroy. A formally verified compiler backend, 2008.

[15] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *In European Symposium on Programming*, pages 219–234. Springer-Verlag, 1996.

[16] S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages. ACM Press*, 1995.

[17] C. Lüth and N. Ghani. Composing monads using coproducts. *SIGPLAN Not.*, 37:133–144, September 2002.

[18] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. pages 124–144. Springer-Verlag, 1991.

[19] E. Meijer and G. 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.

[20] E. Moggi. Computational lambda-calculus and monads. pages 14–23. IEEE Computer Society Press, 1988.

[21] P. D. Mosses. Modular structural operational semantics, 2004.

[22] P. D. Mosses and P. D. Mosses. Theory and practice of action semantics. In *In MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science*, pages 37–61. Springer-Verlag, 1996.

[23] B. Pierce. *Basic category theory for computer scientists*. Foundations of computing. MIT Press, 1991.

[24] G. Plotkin and J. Power. Computational effects and operations: An overview, 2002.

[25] D. Schmidt. *Denotational semantics: a methodology for language development*. Wm. C. Brown, 1988.

[26] T. Schrijvers and B. C. D. S. Oliveira. Monads, zippers and views virtualizing the monad, 2011.

[27] V. Stoltenberg-Hansen, I. Lindström, and E. Griffor. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[28] W. Swierstra. Data Types à la Carte. *Journal of Functional Programming*, 18:423–436, July 2008.

[29] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *In Proc. 12 th LICS Conf*, pages 280–291. IEEE, Computer Society Press, 1997.

[30] P. 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, New York, NY, USA, 1992. ACM.

[31] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[32] K. Wansbrough and J. Hamer. A modular monadic action semantics. In *In Conference on Domain-Specific Languages*, pages 157–170, 1997.

Towards Modular Compilers for Effects

Laurence E. Day and Graham Hutton

Functional Programming Laboratory
School of Computer Science
University of Nottingham, UK

Abstract. Compilers are traditionally factorised into a number of separate phases, such as parsing, type checking, code generation, etc. However, there is another potential factorisation that has received comparatively little attention: the treatment of separate language features, such as mutable state, input/output, exceptions, concurrency and so forth. In this article we focus on the problem of modular compilation, in which the aim is to develop compilers for separate language features independently, which can then be combined as required. We summarise our progress to date, issues that have arisen, and further work.

Keywords: Modularity, Haskell, Compilation, Monads

1 Introduction

The general concept of *modularity* can be defined as the degree to which the components of a system may be separated and recombined. In the context of computer programming, this amounts to the desire to separate the components of a software system into independent parts whose behaviour is clearly specified, and can be combined in different ways for different applications. Modularity brings many important benefits, including the ability to break down larger problems into smaller problems, to establish the correctness of a system in terms of the correctness of its components, and to develop general purpose components that are reusable in different application domains.

In this article we focus on the problem of implementing programming languages themselves in a modular manner. In their seminal article, Liang, Hudak and Jones [9] showed how to implement programming language interpreters in a modular manner, using the notion of monad transformers. In contrast, progress in the area of modular compilers has been more limited, and at present there is no standard approach to this problem. In this article we report on our progress to date on the problem of implementing modular compilers. In particular, the paper makes the following contributions. We show how:

– Modular *syntax* for a language can be defined using the *à la carte* approach to extensible data types developed by Swierstra [15];

- Modular *semantics* for a language can be defined by combining the *à la carte* and modular interpreters techniques, extending the work of Jaskelioff [8];
- Modular *compilers* can be viewed as modular interpreters that produce code corresponding to an operational semantics of the source program;
- Modular *machines* that execute the resulting code can be viewed as modular interpreters that produce suitable state transformers.

We illustrate our techniques using a simple expression language with four features, namely integers, addition, a single exceptional value, and a catch operator for this value. The article is aimed at functional programmers with a basic knowledge of interpreters, compilers and monads, but we do not assume specialist knowledge of monad transformers, modular interpreters, or the *à la carte* technique. We use Haskell throughout as both a semantic meta-language and an implementation language, as this makes the concepts more accessible as well as executable, and eliminates the gap between theory and practice. The Haskell code associated with the article is available from the authors' web pages.

2 Setting the Scene

In this section we set the scene for the rest of the paper by introducing the problem that we are trying to solve. In particular, we begin with a small arithmetic language for which we define four components: syntax, semantics, compiler and virtual machine. We then extend the language with a simple effect in the form of exceptions, and observe how these four components must be changed in light of the new effect. As we shall see, such extensions cut across all aspects and require the modification of existing code in each case.

2.1 A Simple Compiler

Consider a simple language `Expr` comprising integer values and binary addition, for which we can evaluate expressions to an integer value:

```
data Expr      =  Val Value | Add Expr Expr

type Value     =  Int

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

Evaluation of expressions in this manner corresponds to giving a denotational semantics to the `Expr` datatype [14]. Alternatively, expressions can be compiled into a sequence of low-level instructions to be operated upon by a virtual machine, the behaviour of which corresponds to a (small-step) operational semantics [4]. 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 compiler is defined in terms of an auxiliary function `comp'` that takes an additional `Code` argument that plays the role of an accumulator, which avoids the use of append (`++`) and leads to simpler proofs [5]. We execute the resulting `Code` on a virtual machine that operates using a `Stack`:

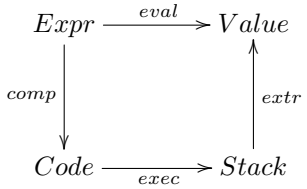
```
type Stack           =  [Item]

data Item            =  INT Value

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 compiler can now be captured by stating that the result of evaluating an expression is the same as first compiling, then executing, and finally extracting the result value from the top of the `Stack` (using an auxiliary function `extr`), which can be expressed in diagrammatic form as follows:



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's syntax, semantics, compiler and virtual machine as a result of this extension. First of all, we extend the `Expr` datatype with two new constructors:

```
data Expr = ... | Throw | Catch Expr Expr
```

The `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.

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. In turn, we must rewrite the semantics of values and addition accordingly, and define appropriate semantics for throwing and catching.

```
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 [12, 16, 17]. In particular, we utilise the basic operations of the `Maybe` monad, namely `return`, which converts a pure value into an impure result, `(>>=)`, used to sequence computations, `mzero`, corresponding to failure, and `mplus`, for sequential choice.

Finally, in order to compile exceptions we must introduce new operations in the virtual machine and extend the compiler accordingly [6]:

```
data Op              =  ... | THROW | MARK Code | UNMARK

comp                 :: Expr -> Code
comp e               =  comp' e []

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` fail, and `UNMARK` indicates that no uncaught exceptions were encountered and hence the record of the handler code can be removed. Note that the accumulator 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.

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

```
data Item            =  ... | HAND 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 (HAND h : s)
exec' (UNMARK : c) s =  let (v : HAND _ : s') = s in
                         exec' c (v : s')
```

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

```
unwind               :: Stack -> Maybe Stack
unwind []            =  mzero
unwind (INT _ : s)   =  unwind s
unwind (HAND h : s)  =  exec' h s
```

2.3 The Problem

As we have seen with the simple example in the previous section, extending the language with a new effect results in many changes to existing code. 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 the functions involved.

The need to modify and extend existing code for each 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. Most importantly, changing code that has already been designed, implemented, tested and (ideally) proved correct is bad practice from a software engineering point of view [18]. Moreover, the need to change existing code requires access to the source code, and demands familiarity with the workings of all aspects of the language rather than just the feature being added. In the remainder of this paper we will present our work to date on addressing the above problems.

3 Modular Effects

In the previous section, we saw one example of the idea that computational effects can be modelled using monads. Each monad normally corresponds to a

single effect, and because most languages involve 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* [9].

In Haskell, monad transformers have the following definition:

```
class MonadTrans t where
  lift :: Monad m => 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 also required to satisfy a number of laws, but we omit the details here. Associated with every monad transformer is the operation `lift`, used to convert from values in the base monad `m` to the new monad `t m`. By way of example, the following table summarises five commonly utilised computational effects, their monad transformer types, and the implementations of these types:

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 [9], 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 transformers, we will examine the transformer for exceptions in more detail. Its type constructor is declared as follows:

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

Note that `ErrorT Identity` is simply the `Maybe` monad. It is now straightforward to declare `ErrorT` as a member of the `Monad` and `MonadTrans` classes:

```
instance Monad m => Monad (ErrorT m) where
  return      :: a -> ErrorT m a
  return a    =  E $ return (Just a)

  (>>=)       :: ErrorT m a -> (a -> ErrorT m b) -> ErrorT m b
  (E m) >>= f =  E $ do v <- m
                        case v of
                          Nothing -> return Nothing
                          Just a  -> run (f a)
```

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

In addition to the general monadic operations, we would 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 specify this by having these operations supported by 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       :: ErrorT m a
  throw       =  E $ return Nothing

  catch       :: ErrorT m a -> ErrorT m a -> ErrorT m a
  x `catch` h =  E $ do v <- run x
                        case v of
                          Nothing -> run h
                          Just a  -> return v
```

We can also declare monad transformers as members of effect classes other than their own. Indeed, this is the primary purpose of the `lift` operation. For example, we can extend `StateT` to support exceptions as follows:

```
instance ErrorMonad m => ErrorMonad (StateT s m) where
  throw       :: StateT s m a
  throw       =  lift . throw

  catch       :: StateT s m a -> StateT s m a -> StateT s m a
  x `catch` h =  S $ \s -> run x s `catch` run 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 the necessary liftings being handled automatically.

Returning to our earlier remark that some transformers do not commute, the semantics resulting from lifting in this manner need not be unique for a set of transformers. For example, consider a monad supporting both exceptions and state. Depending on the order in which this monad is constructed, we may or may not have access to the state after an exception is thrown, as reflected in the types `s -> (Maybe a, s)` and `s -> Maybe (a, s)`. Semantic differences such

as these are not uncommon when combining effects, and reflect the fact that the order in which effects are performed makes an observable difference.

Now that we have reviewed how to handle effects in a modular way, let us see how to modularise the syntax of a language.

4 Modular Syntax and Semantics

We have seen that adding extra constructors to a datatype required the modification of existing code. In this section, we review the modular approach to datatypes and functions over them put forward by Swierstra [15], known as *datatypes à la carte*, and show how it can be used to obtain modular syntax and semantics for the language `Expr` previously described.

4.1 Datatypes à La Carte

The underlying structure of an algebraic datatype such as `Expr` can be captured by a constructor signature. We define *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. We can easily declare `Arith` and `Except` as functors in Haskell:

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

instance Functor Arith where
  fmap               :: (a -> b) -> Arith a -> Arith b
  fmap f (Val n)     =  Val n
  fmap f (Add x y)   =  Add (f x) (f y)

instance Functor Except where
  fmap               :: (a -> b) -> Except a -> Except b
  fmap f 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 can be implemented as follows [11]:

```
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 later on in this section how these languages can be combined.

Given a functor `f`, it is convenient to use a fold operator (sometimes called a *catamorphism*) [10] in order to define functions over `Fix f` [15]:

```
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 called an `f`-algebra, and 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 the functorial and recursive characteristics of `Fix` to process recursive values.

The aim now is to take advantage of the above machinery to define a semantics for our expression language in a modular fashion. Such semantics will have type `Fix f -> m Value` for some functor `f` and monad `m`; we could also abstract over the value type, but for simplicity we do not consider this here. To define functions of this type using `fold`, we require an appropriate *evaluation algebra*, which notion we capture by the following class declaration:

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

Using this notion, it is now straightforward to define algebras that correspond to the semantics for both the arithmetic and exception components:

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

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

There are three important points to note about the above declarations. 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 many different 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`. The latter two points generalise the work of Jaskelioff [8] from a fixed monad to an arbitrary

monad 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 :: (Monad m, Eval f m) => Fix f -> m Value
eval =  fold evalAlg
```

Note that this function is both modular in the syntax of the 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 (disjoint sum) [9]. 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           :: (a -> b) -> (f :+: g) a -> (f :+: g) b
  fmap f (Inl x) =  Inl (fmap f x)
  fmap g (Inr y) =  Inr (fmap g y)
```

It is then straightforward to obtain a coproduct of evaluation algebras:

```
instance (Eval f m, Eval g m) => Eval (f :+: g) m where
  evalAlg         :: (f :+: g) (m Value) -> m Value
  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. For example, if we wished the concrete expression `1 + 2` to have type `Fix (Arith :+: Except)`, 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 may require the modification of existing tags, which breaks modularity. And finally, `Fix (f :+: g)` and `Fix (g :+: f)` are isomorphic as languages, but require equivalent values to be tagged in different ways. The next two sections review how Swierstra resolves these problems [15], and shows how this can be used to obtain modular syntax and semantics for our language.

4.2 Smart Constructors

We need 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, in which the function `inj` injects a value from a subtype into a supertype:

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

It is now straightforward to define instance declarations to ensure that `f` is a subtype of any coproduct containing `f`, but we omit the details here. Using the notion of subtyping, we can define an injection function,

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

which then allows us to define *smart constructors* which bypass the need to tag values when embedding them in expressions:

```
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)

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

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

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

4.3 Putting It All Together

We have now achieved our goal of being able to define modular language syntax. Using the 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 =  throw `catch` throw

ex3 :: Fix (Arith :+: Except)
ex3 =  throw `catch` (val 1337 `catch` throw)
```

The types of these expressions can be generalised using the subtyping relation, but for simplicity we have given fixed types above. In turn, the meaning of such expressions is given by our modular semantics:

```
> eval ex1 :: Value
> 42

> eval ex2 :: Maybe Value
> Nothing

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

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

```
> eval ex1 :: Maybe Value
> Just 42

> eval ex2 :: [Value]
> []
```

5 Modular Compilers

With the techniques we have described, we can now construct a modular compiler for our expression language. First of all, we define the `Code` datatype in a modular manner as the coproduct of signature functors corresponding to the arithmetic and exceptional operations of the virtual machine:

```
type Code     = Fix (ARITH :+: EXCEPT :+: EMPTY)

data ARITH  e = PUSH Int e | ADD e

data EXCEPT e = THROW e | MARK Code e | UNMARK e

data EMPTY  e = NULL
```

There are two points to note about the above definitions. First of all, rather than defining the `Op` type as a fixpoint (where `Code` is a list of operations), we have combined the two types into a single type defined using `Fix` in order to allow code to be processed using the generic `fold`; note that `EMPTY` now plays the role of the empty list. Secondly, the first argument to `MARK` has explicit type `Code` rather than general type `e`, which is undesirable as this goes against the idea of treating code in a modular manner. However, this simplifies the definition of the virtual machine and we will return to this point in the conclusion.

The desired type for our compiler is `Fix f -> (Code -> Code)` for some signature functor `f` characterising the syntax of the source language. To define such a compiler using the generic `fold` operator, we require an appropriate *compilation algebra*, which notion we define as follows:

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

In contrast with evaluation algebras, no underlying monads are utilised in the above definition, because the compilation process itself does not involve the manifestation of effects. We can now define algebras for both the arithmetic and exceptional aspects of the compiler in the following manner:

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

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

In a similar manner to the evaluation algebras defined in section 4.1, 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 `Add` case corresponds to function composition. The smart constructors `pushc`, `addc` and so on are defined in the obvious manner:

```
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, it is now straightforward to define a general compilation function of the desired type by folding a compilation algebra, supplied with an initial accumulator `empty`:

```
comp    :: Comp f => Fix f -> Code
comp e  =  comp' e empty

comp'   :: Comp f => Fix f -> (Code -> Code)
comp' e =  fold compAlg e

empty   :: Code
empty   =  inject NULL
```

For example, applying `comp` to the expression `ex3` from the previous section results in the following `Code`, in which we have removed the fixpoint and coproduct tags `In`, `Inl` and `Inr` for readability:

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

6 Towards Modular Machines

The final component of our development is to construct a modular virtual machine for executing code produced by the modular compiler. Defining the underlying `Stack` datatype in a modular manner is straightforward:

```
type Stack     = Fix (Integer :+: Handler :+: EMPTY)

data Integer e = VAL Int e

data Handler e = HAND Code e
```

As we saw in section 2.1, the virtual machine for arithmetic had type `Code -> Stack -> Stack`, while in section 2.2, the extension to exceptions required modifying the type to `Code -> Stack -> Maybe Stack`. Generalising from these examples, we seek to define a modular execution function of type `Code -> Stack -> m Stack` for an arbitrary monad m. We observe that `Stack -> m Stack` is a state transformer, and define the following abbreviation:

```
type StackTrans m a  = StateT Stack m a
```

Using this abbreviation, we now seek to define a general purpose execution function of type `Fix f -> StackTrans m ()` for some signature functor `f` characterising the syntax of the code, and where `()` represents a void result type. In a similar manner to evaluation and compilation algebras that we introduced previously, this leads to the following notion of an *execution algebra*,

```
class (Monad m, Functor f) => Exec f m where
  execAlg :: f (StackTrans m ()) -> StackTrans m ()
```

for which we define the following three instances:

```
instance Monad m => Exec ARITH m where
  execAlg :: ARITH (StackTrans m ()) -> StackTrans m ()
  execAlg (PUSH n st) =  pushs n >> st
  execAlg (ADD st)    =  adds >> st

instance ErrorMonad m => Exec EXCEPT m where
  execAlg :: EXCEPT (StackTrans m ()) -> StackTrans m ()
```

```
  execAlg (THROW _)   =  unwinds
  execAlg (MARK h st) =  marks h >> st
  execAlg (UNMARK st) =  unmarks >> st

instance Monad m => Exec EMPTY m where
  execAlg :: EMPTY (StackTrans m ()) -> StackTrans m ()
  execAlg (Null)      =  stop
```

The intention is that `pushs`, `adds`, etc. are the implementations of the semantics for the corresponding operations of the machine, and `>>` is the standard monadic operation that sequences two effectful computations and ignores their result values (which in this case are void). We have preliminary implementations of each of these operations but these appear more complex than necessary, and we are in the process of trying to define these in a more elegant, structured manner.

Folding an execution algebra produces the general execution function:

```
exec :: (Monad m, Exec f m) => Fix f -> StackTrans m ()
exec = fold execAlg
```

7 Summary and Conclusion

In this article we reported on our work to date on the problem of implementing compilers in a modular manner with respect to different computational effects that may be supported by the source language. In particular, we showed how modular syntax and semantics for a simple source language can be achieved by combining the *à la carte* approach to extensible datatypes with the monad transformers approach to modular interpreters, and outlined how a modular compiler and virtual machine can be achieved using the same technology.

However, this is by no means the end of the story, and much remains to be done. We briefly outline a number of directions for further work below.

Challenges: supporting a more modular code type in the virtual machine, as our current version uses a fixed `Code` type rather than a generic fixpoint type to simplify the implementation; and improving the implementation of the virtual machine operations, by developing a modular approach to case analysis.

Extensions: considering other effects, such as mutable state, continuations and languages with binding constructs, for example using a recent generalisation of the *à la carte* technique for syntax with binders [2]; formalising the idea that some effects may be 'compiled away' and hence are not required in the virtual machine, such as the `Maybe` monad for our simple language; exploring the extent to which defining compilers in a modular manner admits modular, and hopefully simpler, proofs regarding their correctness; and considering other aspects of the compilation process such as parsing and type-checking.

Other approaches: investigating how the more principled approach to lifting monadic operations developed by Jaskelioff [7] and the modular approach to operational semantics of Mosses [13] can be exploited in the context of modular

compilers; the relationship to Harrison's work [3]; considering the compilation to register machines, rather than stack machines; and exploring how dependent types may be utilised in our development (a preliminary implementation of this paper in Coq has recently been produced by Acerbi [1]).

Acknowledgements

We would like to thank Mauro Jaskelioff, Neil Sculthorpe, the participants of BCTCS 2011 in Birmingham and our anonymous referees for useful comments and suggestions; and Matteo Acerbi for implementing our work in Coq.

References

1. M. Acerbi. Personal Communication, May 2011.
2. P. Bahr and T. Hvitved. Parametric Compositional Data Types. University of Copenhagen, June 2011.
3. W. L. Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.
4. H. Huttel. *Transitions and Trees: An Introduction to Structured Operational Semantics*. Cambridge University Press, April 2010.
5. G. Hutton. *Programming in Haskell*. Cambridge University Press, Jan. 2007.
6. G. Hutton and J. Wright. Compiling Exceptions Correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, pages 211–227. Springer, 2004.
7. M. Jaskelioff. Monatron: An Extensible Monad Transformer Library. In *Implementation and Application of Functional Languages*, 2008.
8. M. Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.
9. S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages. ACM Press*, 1995.
10. E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. pages 124–144. Springer-Verlag, 1991.
11. E. Meijer and G. 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.
12. E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93:55–92, 1989.
13. P. D. Mosses. Modular structural operational semantics, 2004.
14. D. A. Schmidt. *Denotational Semantics: A Methodology For Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.
15. W. Swierstra. Data Types à la Carte. *Journal of Functional Programming*, 18:423–436, July 2008.
16. P. Wadler. Comprehending Monads. In *Proc. ACM Conference on Lisp and Functional Programming*, 1990.
17. P. Wadler. Monads for Functional Programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*. Springer–Verlag, 1992.
18. P. Wadler. The Expression Problem. Available online at: `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.