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:

$$\begin{tikzcd} Expr \arrow[r, "eval"] \arrow[d, "comp"'] & Value \\ Code \arrow[r, "exec"'] & Stack \arrow[u, "extr"'] \end{tikzcd}$$

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.