# Compilation à la Carte

Laurence E. Day
Functional Programming Laboratory
University of Nottingham
led@cs.nott.ac.uk

Graham Hutton
Functional Programming Laboratory
University of Nottingham
graham.hutton@nottingham.ac.uk

## Abstract

In previous work, we proposed a new approach to the problem of implementing compilers in a modular manner, by combining earlier work on the development of modular interpreters using monad transformers with the à la carte approach to modular syntax. In this article, we refine and extend our existing framework in a number of directions. In particular, we show how generalised algebraic datatypes can be used to support a more modular approach to typing individual language features, we increase the expressive power of the framework by considering mutable state, variable binding, and the issue of noncommutative effects, and we show how the Zinc Abstract Machine can be adapted to provide a modular universal target machine for our modular compilers.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

***Keywords*** compilation, modularity, monads, catamorphisms

## 1. Introduction

When describing a compiler, the term *modularity* is frequently used to refer to the decoupling of the various stages of the manipulation of a source program, such as lexing, parsing and code generation. However, there is an alternative axis of modularity on which comparatively little work has been done, namely according to the features supported by the source language, of which there are two varieties. First of all, there are effectful features, such as exception handling and mutable state, and secondly those relating to control-flow, such as conditional expressions and recursion schemes.

In previous work [9], we have begun addressing the implementation of programming languages in a modular manner by building upon the work of Liang et al. [22], who showed how to construct interpreters in a modular manner using monad transformers. Combining this approach with the *à la carte* technique of Swierstra [30] permits a framework in which the syntax of individual language features are described as functors, the semantics of features are given as algebras, and fold operators (or catamorphisms) are used to combine the individual components. The result is a modular approach in which the addition of new language features only requires describing these new features and how they interact with the existing features, rather than modifying any existing definitions.

In this article, we refine and extend the modular compiler framework that was developed in our previous work [9]. More specifically, this article makes the following contributions:

- We show how the use of *generalised algebraic datatypes* (GADTs) [28] to model particular signature functors and value domains permits certain forms of type constraints to be captured in a clean and modular manner.
- We extend the framework with support for both mutable state and variable binding (via the lambda calculus), improving the potential expressive power of a modular source language.
- We consider the issue of effects that do not commute (such as exceptions and state), which potentially require programs to be compiled in different manners depending on the ordering of the effects, and present two approaches to addressing this issue.
- We define a modular variant of the Zinc Abstract Machine [12] as a suitable universal target machine upon which to execute the resulting code from our modular compilers.

This article is aimed at functional programmers with a basic knowledge of interpreters, compilers, monads and monad transformers, but we do not assume specialist knowledge of modular interpreters or the à la carte technique. As in our previous work, we use Haskell throughout as both a semantic meta-language and an implementation language. The Haskell code associated with the article is available from the authors' web pages.

The rest of the article is structured as follows. In sections 3 and 4 we briefly recap the theory behind the à la carte technique, and describe the structure of generalised algebraic datatypes and show how they can be used when compiling into a modular target language in section 5. Sections 6 and 7 respectively describe the way in which the lambda calculus and state are introduced to this framework, with the latter leading into a discussion on noncommutative effects in section 8. Two approaches to the issue of compiling programs with respect to varying semantics are proposed in section 9, and we describe the construction of a virtual machine capable of executing the resulting code in section 10. The article concludes with a brief survey of related work in the field of modular compilation, and presents a number of directions for future research.

## 2. Motivation

A central tenet when designing a programming language compiler is that it must be correct. The correctness of a compiler can be captured by stating that the result of compiling an expression in the source language and then executing the resulting code is equivalent to evaluating the expression with respect to its semantic interpretation, as illustrated by the following commuting diagram:

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

Given the close interplay between the various datatypes and functions involved in such a statement of correctness, the compiler designer should seek to ensure that any changes made to the source language `Expr` are minimally disruptive. However, even comparatively minor extensions can require disproportionate modifications. By way of a simple example, consider the source language `Expr` comprising integers and addition, and a function `eval` which maps expressions to their integer values:

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

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

Now suppose that we wish to extend our expression language to support a simple form of exception handling, by adding throw and catch constructors to the `Expr` datatype:

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

In order to accurately reflect the new semantics expected of `Expr`, we must alter the `eval` function to take account of the fact the evaluation may either succeed or fail:

```
eval :: Expr -> Maybe Int
eval (Val n)     = Just n
eval (Add x y)   = case eval x of
                    Nothing -> Nothing
                    Just n  -> case eval y of
                     Nothing -> Nothing
                     Just m  -> Just (n+m)
eval (Throw)     = Nothing
eval (Catch x h) = case eval x of
                    Nothing -> eval h
                    Just n  -> Just n
```

However, as we can see, the required changes to the definition of `eval` are substantial. In particular, the return type has been changed from `Int` to `Maybe Int` to accommodate potential failure, the existing cases for `Val` and `Add` are rewritten to reflect the new semantics, which in the case of addition now requires a cumbersome nested case analysis, and two new cases are added to handle the new `Throw` and `Catch` constructors. This is far from ideal.

This is a common issue, which Wadler terms the *expression problem* [34]. Multiple solutions have been proposed, and in the next section we describe one such technique for solving the expression problem in Haskell: *datatypes à la carte*.

## 3. Modular Syntax Revisited

In the previous section, we saw an example of how extending a datatype `Expr` with new constructors can result in many other definitions needing to be changed as a consequence. The *à la carte* technique [30] allows us to build datatypes such as `Expr` and functions over them in a modular manner. To illustrate the technique, consider the underlying signatures for both the arithmetic and exception handling features of our simple language:

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

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

These signatures are trivially functors in Haskell:

```
instance Functor Arith where
  fmap f (Val n)   = Val n
  fmap f (Add x y) = Add (f x) (f y)

instance Functor Except where
  fmap _ (Throw)     = Throw
  fmap f (Catch x h) = Catch (f x) (f h)
```

For any functor *f*, its induced recursive datatype, `Fix` *f*, is defined as the least fixpoint of *f*, implemented as follows:

```
data Fix f = In (f (Fix f))
```

Now that we have tied the 'recursive knot' of a signature, `Fix Arith` is a language equivalent to the original `Expr` datatype which allowed integer values and addition. In turn, `Fix Except` is a degenerate language in which the only operations that are permitted are the throwing and catching of exceptions. What we require now is a manner in which signatures can be combined. This can be achieved using the notion of *coproduct*:

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

By taking the coproduct of multiple signatures and taking the least fixpoint of the result, we can now define the syntax of expression in a modular manner by combining the two sublanguages:

```
type Expr = Fix (Arith :+: Except)
```

The above type synonym describes a datatype that is equivalent to the extended `Expr` of the previous section, but is obtained via the composition of algebraic descriptions of each constituent feature rather than by augmenting an existing datatype. Building values of such types requires the use of *smart constructors* [30] to insert the appropriate fixpoint and coproduct tags, but the details are straightforward and are omitted here.

Having reminded ourselves how to define language syntax in a modular manner, it remains for us to recall how to define functions over such datatypes in a modular fashion.

## 4. Modular Semantics Revisited

In this section we construct a modular function `eval` that interprets expressions that are constructed using the *à la carte* technique, by exploiting the functorial nature of signatures, the monadic nature of semantics, and the class system of Haskell.

For suitable functors *f*, we are able to define a generic fold operator – or *catamorphism* [23] – to act as the foundation upon which functions are defined over `Fix` *f*:

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

The first argument of `fold` is an *f*-algebra, which provides the behaviour of each constructor associated with a given signature *f*. The goal now is to define a modular evaluation function using `fold`. Such a function will have type `Fix f -> m Int` for some functor `f` that captures the syntax of the language, and some monad `m` that captures the underlying effects of the language. To define functions of this type using `fold`, we introduce the notion of a *class of evaluation algebras*, defined as follows:

```
class (Functor f, Monad m) => Eval f m where
  evAlg :: f (m Int) -> m Int
```

For example, the evaluation algebra for the signature functor for arithmetic can be defined in the following manner:

```
instance Monad m => Eval Arith m where
  evAlg (Val n)   = return n
  evAlg (Add x y) = do n <- x
                       m <- y
                       return (n + m)
```

There are two important points to note about this definition. First of all, it is parametric in an underlying monad `m`, as reflected in the `Monad m` class constraint. Secondly, the semantics of arithmetic is now defined in terms of the `return` and `>>=` operations of this monad (as abbreviated by the use of the `do` notation.)

In a similar manner, the evaluation algebra for exceptions can be defined as follows, in which the class constraint `MonadPlus m` allows us to use the generic `mzero` and `mplus` operations to define the semantics for the two exception handling operations:

```
instance MonadPlus m => Eval Except m where
  evAlg (Throw)     = mzero
  evAlg (Catch x h) = x ‘mplus‘ h
```

Finally, an evaluation algebra can be defined for coproducts in terms of the algebras for the two underlying signatures:

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

Using the above machinery, we can now define a modular interpreter `eval` by simply folding an evaluation algebra:

```
eval :: Eval f m => Fix f -> m Int
eval = fold evAlg
```

This evaluation function can be used with any modularly constructed datatype `Fix` *f*, provided that algebra instances have been defined for each component signature, and the associated monad satisfies all of the required constraints. In this way, we recover the behaviour of the original, nonmodular, evaluation functions. For example, consider the following two values (where `val`, `add` and `throw` are smart constructors that insert the appropriate tags):

```
three :: Fix Arith
three = val 1 ‘add‘ val 2

error :: Fix (Arith :+: Except)
error = val 42 ‘add‘ throw
```

The meaning of these values is then given by our modular evaluation function `eval`. Note that the choice of the underlying monad can be varied, provided that it satisfies the necessary constraints, as shown in the two interpretations of `three` below:

```
> eval three :: Identity Int
I 3

> eval three :: Maybe Int
Just 3

> eval error :: Maybe Int
Nothing
```

We now consider how the à la carte technique can be used to implement a modular compiler which translates between modular source and target languages. As we shall see in the next section, there are a number of difficulties that arise.

## 5. Introducing GADTs

In our original presentation of a modular compilation framework [9], the compilation function that we constructed using the techniques described in the previous sections had type:

```
comp :: Expr -> Code -> Code
```

The function `comp` takes an expression in the source language, `Expr`, and with the help of an accumulator of type `Code`, returns a program in the target language `Code` that can then be executed on a stack-based virtual machine. In the case of our example language supporting arithmetic and exceptions, the target language `Code` can be defined in a modular manner as follows:

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

data ARITH  e = PUSH Int e
              | ADD e

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

data NIL    e = NIL
```

The `PUSH` and `ADD` operations simply push a number onto the stack and add the topmost two numbers, respectively. In turn, `THROW` indicates that an exception has been raised and initiates a search for handler code on the stack, `MARK` pushes the supplied handler code onto the stack to be run in the event of an exception, and `UNMARK` pops such a handler when it is no longer in scope. Finally, `NIL` provides a base case for the code type.

However, there is a problematic issue regarding the `MARK` constructor. While the target language `Code` is constructed in a modular manner, it is explicitly defined as a `type` synonym, fixing the features of the target language. The use of this fixed type in the `MARK` constructor breaks precisely the modularity in the definition of the target language that we are attempting to obtain. Note that we cannot simply replace the use of `Code` in `MARK` by the parameter type `e`, because the type for the compilation function `comp` determines that `e` will ultimately be instantiated to `Code -> Code`, whereas what is required here is simply `Code`. Similarly, due to the modular nature of individual language signatures, were the argument to `MARK` to be polymorphic in the form (`Fix` *f*) a compile-time error would occur because there is no way to give *f* its correct type, as nothing is known about its component signatures. A potential solution is to extend the `EXCEPT` signature as follows:

```
data EXCEPT f e = THROW e
                | MARK (Fix f) e
                | UNMARK e
```

However, making the underlying functor `f` into a parameter in this manner essentially means that every language that uses `EXCEPT` now needs to explicitly refer to the overall functor `f` that captures all the desired language features, which again breaks modularity. One approach to resolving this would be to impose a class constraint on the signature itself, in the following manner:

```
data Functor f =>
   EXCEPT e = THROW e
            | MARK (Fix f) e
            | UNMARK e
```

Unfortunately, this is no longer possible using the algebraic datatypes of Haskell[1]. Our solution is to define those signatures which contain problematic constructors such as MARK as *generalised algebraic datatypes* (GADTs), which permits individual constructors to be typed explicitly, and with their own class constraints. For example, consider the GADT representation of the *nonmodular* variant of Expr as described in section 2:

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

Note that whilst this representation enables a level of type-safety which was previously unavailable (consider the Add constructor, which dictates that only subexpressions which represent an Int can be added together), we primarily utilise GADTs to leverage existential types into our framework. By describing constructors as methods associated with a type, we can now impose constraints on individual constructors without affecting the datatype as a whole.

Using this idea, the signature functor for exception handling in the target language can be redefined as follows:

```
data EXCEPT e where
  THROW  :: e -> EXCEPT e
  MARK   :: Functor f =>
              Fix f -> e -> EXCEPT e
  UNMARK :: e -> EXCEPT e
```

As a result, we have made two significant improvements over the original definition. Firstly, by abstracting over the syntax of the target language we have avoided the need to refer to an explicitly defined type synonym that must be edited whenever the source language is changed. And secondly, we have placed a constraint on the argument *f* without including it in the top-level definition of EXCEPT and without constraining other constructors similarly.

This now suggests that our modular compiler need not target a particular language, but rather any language which meets the appropriate constraints. Key to these constraints is the notion of a *subtyping relation*, captured at the type-level as follows:

```
class (Functor f, Functor g) => f :<: g where
  inj :: f a -> g a
```

The inj method embeds a value within the *subtype* functor f into a value within the *supertype* functor g. For example, the subtyping relation Arith :<: (Arith :+: Except) states that the signature functor for arithmetic is a *component* of the signature functor for both arithmetic and exception handling combined.

The modular counterpart of the compilation function comp will have type Fix f -> Fix g -> Fix g, for signature functors f and g that characterise the syntax of the source and target languages respectively. In order to supply an initial value for the accumulator (the second argument), we require that NIL (which represents the empty code fragment) is a subtype of g. Putting this all together, we define the class of *compilation algebras* as follows:

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

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

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

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

In the above, expressions are compiled in the expected manner, framed in our modular setting. In particular, values are compiled by pushing the associated integer onto the stack, and addition is compiled by compiling the two subexpressions and adding the resulting two values on top of the stack. In turn, a thrown exception is compiled directly into a corresponding throw instruction in the machine, while catch blocks are compiled by marking the stack with the compiled handler code, producing code for the body of the block, and finally unmarking the stack by removing the topmost handler. Note that the use of continuation-passing style in the compilation algebras is key to compiling the Catch operation [9], and also means that concatenation of code is achieved simply by function composition as shown in the case for addition.

In conclusion, we have successfully refactored our modular compiler to produce code for a modular target language. Having seen how GADTs can be used in one aspect of our modular framework, in the next section we extend the modular source language with support for variable binding, and discuss the role that GADTs play in defining its modular semantics.

## 6. Introducing the Lambda Calculus

The ability to abstract over variable names in the body of a function is a near-universal feature in programming languages, and in this section we will introduce variable binding into our modular framework using the untyped lambda calculus of Church [7]. Although variables in lambda terms are often given names in the same way that we would name other variables, there are many alternative ways to model bindings, including such approaches as *higher order abstract syntax* (HOAS) and de Bruijn indices [10], amongst others. In this article, we use a de Bruijn indexed encoding of the lambda calculus. Our reasoning for this is that naming variables in lambda terms can give rise to issues of $\alpha$-equivalence, and requires guarantees of capture-avoiding substitution when applying lambda terms. Alternatively, the HOAS approach uses the binders of the metalanguage to describe the binding structure of the language being implemented, which eliminates the need to ensure that term substitution is defined correctly, but comes at the price of increased dependence on the implementation language.

The underlying signature for the de Bruijn indexed lambda calculus is defined in the following manner:

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

The number associated with an Index constructor represents a variable, and refers to the number of binders in scope before its binding site. In turn, Abs indicates the presence of a binder and Apply represents the substitution of lambda terms, and is passed both a function body and its argument as subexpressions.

However, by choosing not to use the HOAS approach, a problematic issue arises regarding the Apply constructor. When defining a modular semantics for the Lambda signature, the carrier of the evaluation algebra determines that both of its subexpressions will be typed as m Int. The following attempt at defining the evaluation algebra illustrates the underlying problem:

[1] The GHC pragma allowing this, -XDatatypeContexts, was removed in Haskell 2010, being widely considered a misfeature.

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

The definition of Apply cannot be completed in a sensible way, because the semantic domain is not *expressive* enough. In particular, the result of binding the function body f has the primitive type Int which accepts no arguments. Moreover, binding the result of f breaks the abstraction that a function body represents.

Our solution to this issue is to extend the semantic domain with support for closures. To do this, we redefine Value as a GADT:

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

In the above, the Num constructor represents an integer value, and the Clos constructor takes as arguments a list of values (which acts as an environment) and a computation which represents an unevaluated function body. There are two points to note about this definition. Firstly, we would not be capable of representing closures in this way without the Monad m constraint [33], and secondly, this constraint is imposed on the Value parameter m, rather than a parameter specific to a single constructor.

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

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

Intuitively, the env operation provides the list of values that are currently in scope, and the with operation takes both a computation and an associated environment and returns the result of performing substitution. We can now give a semantics to the lambda calculus signature, using the CBVMonad class constraint to allow the use of the env and with operations in the following manner:

```
instance CBVMonad m => Eval Lambda m where
  evAlg (Index i)   = env >>= \e
                    -> return (e !! i)
  evAlg (Abs t)     = env >>= \e
                    -> return (Clos e t)
  evAlg (Apply f x) = f >>= \(Clos ctx t)
                    -> x >>= \c
                    -> with (c:ctx) t
```

In the above, a de Bruijn index is evaluated by looking up the index in the current environment, a lambda abstraction is packaged up with the current environment to form a closure, and substitution of lambda terms is performed by evaluating argument x, adding this value to the environment of the closure which represents the function body, and finally evaluating the function body with respect to the updated environment. Implicit in the above is that all lambda terms defined in a modular manner must be *closed*.

We can now write terms in our modular source language that use variable binding. For example, consider the following example (where apply, abs etc. are the appropriate smart constructors):

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

> eval e :: [Value Identity]
[Num 3]
```

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

We are also capable of defining multiple evaluation schemes for terms in the lambda calculus. A common alternative is *call-by-name*, which does not evaluate arguments before applying them to a function body. The difference between this scheme and the call-by-value scheme which we have just implemented is that environments now contain *computations*, not values. Another class CBNMonad is needed to reflect this change, defined as follows:

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

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

```
instance CBNMonad m => Eval Lambda m where
  evAlg (Index i)   = env >>= \e
                    -> (e !! i)
  evAlg (Abs t)     = env >>= \e
                    -> (Clos e t)
  evAlg (Apply f x) = f >>= \(Clos ctx t)
                    -> with (x:ctx) t
```

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

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

Having successfully implemented variable binding modelled using the lambda calculus in a modular manner, a natural progression is to consider how to introduce the notion of persistent, updatable state to our modular compilation framework.

## 7. Introducing Mutable State

In programming languages, a widely used feature is *mutable* state variables that can change value over time. In this section, we extend the expressive power of a modular source language by introducing the notion of mutable state. As proof of concept we consider a single integer variable, and the syntax associated with such an updatable value is given by the following signature:

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

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

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

The update operations takes a function (Int -> Int) and uses it to alter the state variable. By passing different functions to update, it can be used to define an evaluation algebra for the State signature in the following manner:

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

When evaluating a Get constructor, the update operation is passed the identity function id, which leaves the state value unchanged. This value is then bound to n and embedded into the Value domain. In turn, when evaluating a Set constructor, update is passed an anonymous function overwriting the state value to v before evaluating the subexpression c. Note that the use of update cannot be eliminated from the definition for the Get case because of the presence of the underlying monad m.

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

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

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

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

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

Every monad transformer comes equipped with an *accessor function* which, when called, allows access to the underlying representation. By first evaluating an expression and then applying the accessor functions of all monad transformers comprising the monad within which the expression was evaluated to the result, we obtain a final value, as illustrated in the following:

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

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

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

However, an issue arises when considering the *order* in which certain monad transformers are applied. Consider the following definitions for the StateT transformer:

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

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

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

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

The newtype definition of StateT and associated instance declarations appear to be no different to that of any other monad transformer associated with a particular feature. However, closer inspection shows that the manner in which the state monad transformer is defined gives rise to the issue of *noncommutative effects*, the impact of which we describe in the next section.

## 8. Noncommutative Effects

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

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

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

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

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

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

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

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

These are both sensible results, and depend on how we wish to order the underlying effects. The natural progression at this point is to address the issue of compiling expressions with multiple interpretations, such as demo, in a modular manner. Our modular compiler will currently compile demo to the following code sequence (written using Haskell list notation for simplicity):

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

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

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

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

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

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

## 9. Monadic Parameterisation

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

*Technique 1. Type-Level Monadic Parameterisation*

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

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

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

```
instance (EXCEPT :<: g, Monad m) =>
  Comp Except g (ErrorT (StateT s m)) where
  comAlg (Throw)     = \_   -> throw
  comAlg (Catch x h) = \m c -> mark (h m c)
                          (x m (unmark c))

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

An advantage of this technique is that that we only need to match on constructors associated with monad transformers that cause semantics to differ. To illustrate, consider a *commutative* monad transformer T, i.e. a transformer that can be applied in any order. If T were to appear between ErrorT and StateT in the above, we can abstract over T using a generic variable t of type MonadT, allowing the programmer to focus on the task of defining compilation algebras only for non-commutative orderings.

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

To address this concern, we require a way to provide the compilation algebra with information concerning the ordering of monad transformers, *without* explicitly passing around the resulting monad. Our solution to this issue is to use GADTs to *reify* a monad, representing it as a sequence of constructors. We capture this notion with the datatype MTList, defined as follows:

```
data ST = IntT | BoolT | ...

data MTList m where
  Err :: MTList m
      -> MTList (ErrorT m)
  Sta :: ST -> MTList m
      -> MTList (StateT ST m)
  Id  :: MTList Identity
```

Using the auxiliary datatype ST of *state types* to reify monad transformer parameters, an instance of MTList m represents the monad m by applying the appropriate constructors to Id. To illustrate, the two monads LocalM and GlobalM that are defined in the previous section can be reified as follows:

```
local  :: MTList (StateT s (ErrorT Identity))
local  = Sta IntT (Err Id)

global :: MTList (ErrorT (StateT s Identity))
global = Err (Sta IntT Id)
```

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

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

*Technique 2. Function-Level Monadic Reification*

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

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

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

Particularly important in the above is that the compilation algebra is no longer parameterised by a monad `m`, highlighting the fact that a modular compiler is *informed* by a monad, rather than defined in terms of one. However, the use of `MTList` to reify the context comes at the cost of being unable to abstract over commutative monad transformers, as is possible when using the first technique. The user can still abstract over a *base* monad using this second technique, but must explicitly define any intermediate monad transformers that are applied between a conflicting pair.

Both of these techniques are potential solutions for the issue of modular compilation in the presence of noncommutative effects. We highlight that modular compiler instances exist for all of the other features described up to this point, and can be found in the associated code on the authors' websites. In the next section, we discuss how the techniques and extensions we have discussed up to this point apply to executing modular code produce by a modular compiler on a modular virtual machine.

## 10. A Modular Virtual Machine

In previous work [9], we defined a modular virtual machine in terms of an execution algebra targeting a stateful computation, parameterised by a modular datatype `Stack`, as follows:

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

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

By defining `Stack` as a modular datatype, functions over `Stack` were defined as folds, resulting in significant amounts of boilerplate. To illustrate, consider the modular implementation of stack-based addition that adds the top two numbers on a stack [9]:

```
extract :: Stack -> Integ Stack
extract = fromJust . match in

add :: Monad m => StackTrans m ()
add =  do x <- pop; y <- pop
          case (extract x, extract y) of
          (VAL n _, VAL m _) -> push (n + m)
```

The `extract` operation in the above explicitly describes the type of value which we can pop off of the top of the stack. Such stack-based operations are conceptually simple, however the need to write them in terms of folds and monads leaves much to be desired. In order to eliminate this overhead, in our new library for building modular compilers we simply use the Haskell list type to represent stacks, rather than a modular datatype of modular values.

There are two reasons for this change. Firstly, the stack may be considered to be an auxiliary structure, as the real work is done by the compilation and execution algebras upon the source and target languages, which we have already defined in a modular manner. As we have demonstrated in previous work, it *is* possible to construct a modular stack, however the amount of boilerplate required when extending a (modular) source language with new features quickly becomes prohibitive, and detracts from the main task at hand. Secondly, while the two representations of the stack are equivalent, we feel that it is easier for a user to conceptualise stack-based operations in terms of lists. This, combined with access to the Haskell library functions, improves the overall extensibility of our modular compilation framework.

More important than the choice of representation type for the stack is the fact that the stack *transformer* `StackT` is defined by applying the state transformer to a base monad `m`. In our original presentation, the intent of the monad `m` was to help execute modular code in the presence of noncommutative effects. However, as we have seen in the previous section, by inspecting the ordering of monad transformers when *compiling* a modular source expression, we have solved this issue, and the monad is no longer required.

In section 6, we defined a semantics for lambda terms modelled using de Bruijn indices according to the call-by-value evaluation scheme. To execute such terms on a modular virtual machine, we define a compilation algebra mapping between lambda terms and the instruction set `LAMBDA` associated with the *Zinc Abstract Machine*, a call-by-value variant of the *Krivine machine* [8]. This compilation scheme is captured by the following function $\mathbb{C}$:

$$\begin{array}{rl}
\mathbb{C}[n] = & [\texttt{IND}\ n] \\
\mathbb{C}[\lambda\ t] = & [\texttt{CLS}\ (\mathbb{C}[t] +\!\!+ [\texttt{RET}])] \\
\mathbb{C}[f\,x] = & \mathbb{C}[x] +\!\!+ \mathbb{C}[f] +\!\!+ [\texttt{APP}]
\end{array}$$

The instructions of the Zinc Abstract Machine (`IND`, `CLS`, `RET` and `APP`) usually operate upon a pair of lists, the first representing an environment and second representing a stack. In this article we also make use of a third list, representing a *state stack*.

The resulting modular execution algebra class and associated data structures are defined in the following manner:

```
data VALUE e where
  ...

newtype StateM s a
  = S  { runSM :: s -> (a, s) }

type EnvZ   = [VALUE ()]
type StackZ = [VALUE ()]
type StateZ = [VALUE ()]
type ZAM    = StateM (EnvZ, StateZ, StackZ) ()

class Functor f => Execute f where
  exAlg :: f ZAM -> ZAM
```

In the above, the type `VALUE` of data that is manipulated on the stack is defined as a GADT. We omit the definition of the constructors here, as we shall see them in use shortly. In turn, the state transformer `StateM` is defined in a similar manner to the state monad transformer, but without the presence of a monad `m`.

Further, the type synonym `ZAM` is defined as a stateful computation parameterised by a triple of lists defined over `VALUE`, and finally we define the execution algebra class, targeting the modular representation of the Zinc Abstract Machine.

In previous work, the instructions that are executed upon a virtual machine have had intuitive operational transitions. For example, the `PUSH` operation pushes a value onto the top of a stack. However, the operational transitions associated with the instructions of the Zinc Abstract Machine are more complicated, and are defined by a transition relation on *(Code,Env,Stack)* triples:

| *Code* | | *Env* | *Stack* | | *Code* | *Env* | *Stack* |
|---|---|---|---|---|---|---|---|
| `IND` *n*; | *c* | **e** | s | → | *c* | **e** | (**e** !! *n*); s |
| `CLS` *k*; | *c* | **e** | s | → | *c* | **e** | [*k*, **e**]; s |
| `APP`; | *c* | **e** | v; [*d*, **f**]; s | → | *d* | v; **f** | c; e; s |
| `RET`; | *c* | **e** | v; *d*; **f**; s | → | *d* | **f** | v; s |

In the above, the notation [*c*, **e**] is shorthand for a closure consisting of a code fragment *c* and associated environment **e**. Instantiating the resulting execution algebra is straightforward:

```
instance Execute LAMBDA where
  exAlg (IND i c)  = S $ \(e, s, stk)  ->
    runSM c (e, s, (e !! i):stk)
  exAlg (CLS k c) = S $ \(e, s, stk)   ->
    runSM c (e, s, (CLO k e):stk)
  exAlg (RET c)    = S $ \(e, s, stk)  ->
    case stk of (v:(CLO k e'):stk')    ->
     runSM k' (v:e', s, ((CTN c e):stk'))
     where k' = fold exAlg k
  exAlg (APP c)     = S $ \(e, s, stk) ->
    case stk of (v:(CTN c' e'):stk')   ->
     runSM c' (e', s, v:stk')
```

It is worth noting that had the representation of the stack types not been changed to lists, the amount of boilerplate required to implement the necessary operations upon the stacks would dwarf the definition of the above algebra.

One final, important point must be addressed. By changing the carrier of the execution algebra to `ZAM`, new algebra instances must be defined for existing features. As we have seen previously, the virtual machine only requires the use of a single stack when executing terms supporting arithmetic and error handling, with the exception of the newly-introduced `SAVE` and `RESTORE` operations that make use of the state stack. These 'new' instances of the execution algebra are defined in the following manner:

```
instance Execute ARITH where
  exAlg (PUSH n c)    = S $ \(e, s, stk) ->
   runSM c (e, s, (NUM n):stk)
  exAlg (ADD c)       = S $ \(e, s, stk) ->
   case stk of ((NUM n):(NUM m):stk')    ->
    runSM c (e, s, ((NUM (n + m)):stk'))

instance Execute EXCEPT where
  exAlg (THROW c)     = S $ \(e, s, stk) ->
    case dropWhile (not . isHAN) stk of
     ((HAN h):stk') -> runSM h' (e, s, stk')
    where h' = fold exAlg h
  exAlg (MARK h c)    = S $ \(e, s, stk) ->
   runSM c (e, s, (HAN h):stk)
  exAlg (UNMARK c)    = S $ \(e, s, stk) ->
    case dropWhile (not . isHAN) stk of
     ((HAN _):stk') -> runSM c (e, s, stk')
  exAlg (SAVE c)      = S $ \(e, s, stk) ->
    case s of ((NUM n):_) ->
     runSM c (e, s, (REC n):stk)
  exAlg (RESTORE c)   = S $ \(e, s, stk) ->
    case dropWhile (not . isREC) stk of
    ((REC n):stk') ->
     runSM c (e, ((NUM n):s), stk')

instance Execute STATE where
  exAlg (GET c)   = S $ \(e, s, stk) ->
   case s of z@((NUM n):_) ->
    runSM c (e, z, (NUM n):stk)
  exAlg (SET n c) = S $ \(e, s, stk) ->
   runSM c (e, ((NUM n):s), stk)
```

Our modular virtual machine is now defined by folding the algebra over an initial triple of empty stacks:

```
exec  :: Execute f => Fix f -> ZAM
exec f = S $ \_ ->
  runSM (fold exAlg f) ([], [], [])
```

In section 6, we saw an example of the idea that lambda terms can be interpreted according to more than one evaluation scheme. In particular, we showed how both the call-by-value and call-by-name schemes are implemented. The Zinc Abstract Machine executes lambda terms in a manner corresponding to call-by-value, however its instruction set can be used in the definition of an alternative compilation scheme $\mathbb{K}$, equivalent to the Krivine machine [8], which is defined recursively as follows:

$$\begin{aligned} \mathbb{K}[n] &= \quad [\texttt{IND}\ n, \texttt{APP}] \\ \mathbb{K}[\lambda\ t] &= \quad [\texttt{POP}] +\!\!+\ \mathbb{K}[t] \\ \mathbb{K}[f\,x] &= \quad [\texttt{CLS}(\mathbb{K}[x])] +\!\!+\ \mathbb{K}[f] \end{aligned}$$

The only instruction that we have not yet specified the behaviour for is `POP`, and this simply moves the topmost value of the stack onto the environment stack. The resulting execution algebra that executes lambda terms with respect to call-by-name is defined over the instruction set `LAMBDA'` (to avoid constructor naming conflicts, as seen in section 6) in the following manner:

```
instance Execute LAMBDA' where
  exAlg (IND' i c) = S $ \(e, s, stk)     ->
    runSM c (e, s, (e !! i):stk)
  exAlg (CLS' k c) = S $ \(e, s, stk)     ->
    runSM c (e, s, (CLO k e):stk)
  exAlg (APP' c)   = S $ \(e, s, stk)     ->
    case stk of (v:(CTN c' e'):stk')      ->
     runSM c' (e', s, v:stk')
  exAlg (POP c)    = S $ \(e, s, (v:stk)) ->
    runSM c (e, v:s, stk)
```

## 11. Related Work

In this section we briefly review a range of previous work that is related to our approach to the implementation of compilers in a modular manner. We consider a number of categories of related work, each described in a separate subsection.

*Modular Interpreters and Monad Transformers.* Whilst the work we present here is primarily focussed on the process of compilation between modular languages, a key source of inspiration has been the work of Liang, Hudak and Jones [22]. This article introduced the idea of defining the syntax of a language in a modular manner, but did not consider how this approach could be exploited further to define the meaning of programs using fold operators. It also discussed the issue of combining different effects by combining their underlying monads, giving rise to the notion of base monads, monad transformers, and noncommutative effects.

*Modular Compilers Based on Monad Transformers.* There is a long line of previous work on generating compilers starting from interpreters by using the technique of partial evaluation [19, 21, 35].

Harrison and Kamin showed how a modular compiler can be developed by applying partial evaluation to a modular interpreter that is structured using the monadic approach summarised above. In particular, by writing the interpreter in continuation-passing style [26] and then partially evaluating, we can obtain a compiler and associated virtual machine. However, the use of monad transformers in this work was primarily for the purpose of introducing intermediate data structures in the virtual machine, whereas in our setting these are used explicitly to model individual language features. Moreover, Harrison's work did not consider the issue of noncommutative effects, or the role that types can play in structuring and informing the development of a modular compiler.

*Compilation as Metacomputation.* Extending their previous work, Harrison and Kamin identified that metacomputations (computations producing computations) naturally arise in the compilation process [14]. Metacomputations can be classified into two distinct varieties capturing different aspects of the behaviour of a program: *static*, such as code generation and optimisation, and *dynamic*, such as stack and state manipulation. This 'staging' of a program can readily be implemented using monads constructed using transformers. The notions of static and dynamic metacomputations correspond, respectively, with the compilation and execution algebras that we implement using the à la carte technique.

*Modular Compilers & Their Correctness Proofs.* Harrison's PhD thesis consolidates the work of the two items above by describing the construction and verification of *reusable compiler building blocks* (RCBBs) for various features of a source language. Two distinct approaches to RCBBs are proposed, namely as metacomputations and monadic code generators. The approach to the latter is similar to that taken in this article, in the sense that for each individual feature a function `compile :: Source -> m Target` is defined. The correctness of a compiler can then be verified by proving relations between the standard and compilation semantics of any given RCBB. The thesis concludes by describing limitations on the combinations of RCBBs for a non-trivial language in much the same way as the usage of monad transformers in our own work dictates the need to consider multiple compilation schemes.

*Monatron.* The standard monad transformer library in Haskell [13] suffers from the problem of requiring a quadratic number of instance declarations to lift monadic operations through each possible monad transformer. Jaskelioff's Monatron library [16] solves this problem by lifting operations through monad transformers in a uniform manner, underpinned by ideas from Plotkin and Power's algebraic theory of effects [15, 25]. As an application, Jaskelioff's PhD thesis [17] describes how Monatron can be used to both define a modular interpreter for a multi-feature language and to implement a modular operational semantics for the language [18]. It would be interesting to consider how this approach could be adapted to the development of modular compilers.

*Eff.* The recently developed language *Eff* of Bauer and Pretnar [4] presents another approach to handling noncommutative effects, based upon the algebraic theory of effects. Whereas in our approach we explicitly define each monad and monad transformer, in the algebraic approach one specifies the operations that are provided and their desired properties, rather than how they are actually implemented. In this manner, the issue of combining behaviours becomes one of combining mathematical specifications, rather than combining concrete implementations. These ideas are realised in Eff via the notion of *handlers*, which describe how particular effects are implemented, and can be applied by the user in different orders to manifest different semantics of noncommutative effects. A Haskell library based upon the ideas that underlie the Eff language has been implemented by Visscher [32].

*Compositional Datatypes.* Recent work by Bahr and Hvitved [2, 3] extends the *à la carte* technique in a range of directions, including alternative recursion schemes, generic programming, datatypes with holes (zippers) and mutual recursion. Building upon this work, it was then shown how languages with variable binding can also be implemented in the same setting, using Chlipala's parametric higher-order abstract syntax (PHOAS) [6]. However, the use of PHOAS requires higher-order folds [23], which adds significantly to the overall complexity of the approach, and Bahr's work has not yet considered the issue of non-commutative effects, or the problem of developing modular compilers. As noted in the next section, it will be interesting to consider how Bahr's extensions to the à la carte technique can be combined with our own to to support multiple axes of modularity in the implementation of compilers.

## 12. Conclusions

In this article we have described a number of extensions and improvements to a framework for defining compilers that are modular with respect to the various features provided by a source language. In particular, we have demonstrated how generalised algebraic datatypes allow for increased flexibility and type-safety when targeting modular languages, how variable binding and mutable state can be handled in our modular framework, and how the issue of noncommutative effects such as exceptions and state can be resolved by pattern matching on the structure of the underlying monad transformers either at the type or the value level.

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

*Additional Computational Features.* We wish to investigate the extension of our framework with support for additional features, in particular other forms of control flow such as recursion and continuations. In the case of recursion, we expect to benefit from Bahr's work in this area [2]. Furthermore, it would be interesting to see how one might exploit the algebraic theory of effects to give a principled understanding of how easy it may be to integrate a new feature based upon the new operations that it provides.

*Attribute Grammars.* The Utrecht University Attribute Grammar Compiler (UUAGC) [29] is a Haskell preprocessor which simplifies the construction of catamorphisms over tree-like structures, whilst taking advantage of *inherited* attributes that are passed down the tree, such as environments, and *synthesised* attributes that are passed up the tree, such as computed results. Moreover, the ability of the UUAGC to define the constructors of a datatype in multiple distinct locations and define its attributes and semantics separately provides an alternative approach to modular programming. We intend to examine the extent to which the UUAGC system is suitable for generating modular compilers.

*Indexed Type Families.* In Haskell, the indexed type family extension [5], which permits ad-hoc overloading of datatypes, may prove useful in explicitly declaring a link between the signature functors of a source and target language for a particular effect. For example, in section 5 we defined an evaluation algebra that mapped terms that are constructed from the source functor `Arith` into terms constructed from the target functor `ARITH`. At present, we are capable of compiling into *any* target language, provided it supports the requisite signatures. Declaring a type family with a functional dependency in order to define a mapping from a source language `Fix` *f* to a target language `Fix` (`Target` *f*) would ensure that the target language is minimal, and remove the requirement that the user define the target language in advance.

*Automatic Context Inference.* A recent observation [31] is that it may be possible to use the ordering of signature functors in the type of an expression to automatically infer the monadic context within which we wish to evaluate it. For example, from a term with signature `(Arith :+: Except :+: State)`, we might infer that it is to be evaluated in a monad that is built up from the identity monad corresponding to `Arith` by first applying the exception monad transformer, and then applying the state monad transformer. Such an interpretation may be useful as the default behaviour, which the user could override if they wished.

*Modular Correctness Proofs.* If one constructs compilers in a modular manner, it is natural to ask if the proof of correctness of such a compiler can also be structured in a modular manner. A suitable starting point for such an exploration would be Acerbi's encoding [1] of our previous work on modular compilers in the Coq theorem prover, combined with the use of *metatheory à la carte* (MTC) [11], a recent Coq library designed for reasoning about modular definitions using Mendler-style catamorphisms [24].

*Alternative Target Languages.* At present, we compile into a stack-based target language. It would also be useful to consider how our framework can be adapted to other forms of target language, in particular register-based languages such as LLVM [20], which is used as the target language for many imperative language compilers, or logic-based languages such as System F [27], a variant of which is used as the target language for GHC.

## Acknowledgments

We would like to thank Conor McBride for suggesting how to implement the lambda-calculus as an evaluation algebra by using a monadic context in the value type, and Wouter Swierstra, Ohad Kammar and Michael Gale for their useful comments and suggestions. We are also grateful to the Software Technology Group in Utrecht for their hospitality during a two-week visit.

## References

[1] M. Acerbi. Personal Communication, May 2011.

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

[3] P. Bahr and T. Hvitved. Compositional Data Types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, WGP '11, pages 83–94, New York, NY, USA, Sept. 2011. ACM.

[4] A. Bauer and M. Pretnar. Programming with Algebraic Effects and Handlers. *CoRR*, abs/1203.1539, 2012.

[5] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 1–13, New York, NY, USA, 2005. ACM.

[6] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 143–156. ACM, 2008.

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

[8] P. L. Curien. The $\lambda\rho$-Calculus: An Abstract Framework For Environment Machines. *Rapport de Recherche LIENS-88-10, Ecole Normale Supérieure*, Paris, France, 1988.

[9] L. E. Day and G. Hutton. Towards Modular Compilers For Effects. In *Proceedings of the 12th international conference on Trends in Functional Programming*, TFP'11, pages 49–64, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] N. G. de Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Studies in Logic and the Foundations of Mathematics*, 133:375–388, 1994.

[11] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Metatheory à La Carte. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM.

[12] S. Diehl, P. Hartel, and P. Sestoft. Abstract Machines For Programming Language Implementation, 2000.

[13] A. Gill. mtl: Monad classes, using functional dependencies. `http://hackage.haskell.org/package/mtl-2.1.2`, June 2012.

[14] W. L. Harrison and S. N. Kamin. Compilation as Metacomputation: Binding Time Separation in Modular Compilers. In *In 5th Mathematics of Program Construction Conference, MPC 2000, Ponte de*, 1998.

[15] M. Hyland, G. Plotkin, and J. Power. Combining effects: sum and tensor, 2003.

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

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

[18] M. Jaskelioff, N. Ghani, and G. Hutton. Modularity and Implementation of Mathematical Operational Semantics. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, Reykjavik, Iceland, July 2008.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[20] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[21] P. Lee. *Realistic Compiler Generation*. MIT Press, 1989.

[22] 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.

[23] 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.

[24] N. P. Mendler. Inductive Types and Type Constraints in the Second-Order Lambda Calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991.

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

[26] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

[27] J. C. Reynolds. Towards A Theory Of Type Structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, UK, 1974. Springer-Verlag.

[28] T. Sheard. Languages of the future. In *In OOPSLA 04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119. ACM Press, 2004.

[29] S. D. Swierstra, P. R. A. Alcocer, J. Saraiva, D. Swierstra, P. Azero, and J. Saraiva. Designing and Implementing Combinator Languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 150–206. Springer-Verlag, 1999.

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

[31] W. Swierstra. Personal Communication, March 2013.

[32] S. Visscher. The Control.Effects library for haskell, May 2012.

[33] P. Wadler. Monads for Functional Programming. In M. Broy, editor, *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*. Springer–Verlag, 1992.

[34] P. Wadler. The Expression Problem. Available online at: `http://tinyurl.com/wadler-ep`, 1998.

[35] M. Wand. Deriving target code as a representation of continuation semantics. *ACM TOPLAS*, 4(3):496–517, July 1982.