JFP **18** (4): 423–436, 2008.
doi:10.1017/S0956796808006758 First published online 18 March 2008 Printed in the United Kingdom

FUNCTIONAL PEARL

Data types à la carte

WOUTER SWIERSTRA

School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, NG8 1BB
(*e-mail:* wss@cs.nott.ac.uk)

Abstract

This paper describes a technique for assembling both data types and functions from isolated individual components. We also explore how the same technology can be used to combine free monads and, as a result, structure Haskell's monolithic IO monad.

1 Introduction

Implementing an evaluator for simple arithmetic expressions in Haskell is entirely straightforward.

$$\begin{aligned}&\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}\\ &\mathit{eval} :: \mathit{Expr} \to \mathit{Int}\\ &\mathit{eval}\ (\mathit{Val}\ x) \quad = x\\ &\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y\end{aligned}$$

Once we have chosen our data type, we are free to define new functions over expressions. For instance, we might want to render an expression as a string:

$$\begin{aligned}&\mathit{render} :: \mathit{Expr} \to \mathit{String}\\ &\mathit{render}\ (\mathit{Val}\ x) \quad = \mathit{show}\ x\\ &\mathit{render}\ (\mathit{Add}\ x\ y) = \texttt{"("} +\!\!+ \mathit{render}\ x +\!\!+ \texttt{" + "} +\!\!+ \mathit{render}\ y +\!\!+ \texttt{")"}\end{aligned}$$

If we want to add new operators to our expression language, such as multiplication, we are on a bit of a sticky wicket. While we could extend our data type for expressions, this will require additional cases for the functions we have defined so far. Phil Wadler (1998) has dubbed this in the *Expression Problem*:

> The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.

As the above example illustrates, Haskell can cope quite nicely with new function definitions; adding new constructors, however, forces us to modify existing code.

In this paper, we will examine one way to address the Expression Problem in Haskell. Using the techniques we present, you can define data types, functions, and even certain monads in a modular fashion.

2 Fixing the expression problem

What should the data type for expressions be? If we fix the constructors in advance, we will run into the same problems as before. Rather than choose any particular constructors, we parameterize the expression data type as follows:

```
data Expr f = In (f (Expr f))
```

You may want to think of the type parameter f as the *signature* of the constructors. Intuitively, the type constructor f takes a type parameter corresponding to the expressions that occur as the subtrees of constructors. The *Expr* data type then ties the recursive knot, replacing the argument of f with *Expr* f.

The *Expr* data type is best understood by studying some examples. For instance, if we wanted expressions that consisted of integers only, we could write:

```
data Val e = Val Int
type IntExpr = Expr Val
```

The only valid expressions would then have the form *In* (*Val* x) for some integer x. The *Val* data type does not use its type parameter e, as the constructor does not have any expressions as subtrees.

Similarly, we might be interested in expressions consisting only of addition:

```
data Add e = Add e e
type AddExpr = Expr Add
```

In contrast to the *Val* constructor, the *Add* constructor does use its type parameter. Addition is a binary operation; correspondingly, the *Add* constructor takes two arguments of type e.

Neither values nor addition are particularly interesting in isolation. The big challenge, of course, is to combine the *ValExpr* and *AddExpr* types somehow.

The key idea is to combine expressions by taking the coproduct of their signatures.

The coproduct of two signatures is straightforward to define in Haskell. It is very similar to the *Either* data type; the only difference is that it does not combine two *base types*, but two *type constructors*.

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

An expression of type *Expr* (*Val* :+: *Add*) is either a value or the sum of two such expressions; it is isomorphic to the original *Expr* data type in the introduction.

Combining data types using the coproduct of their signatures comes at a price. It becomes much more cumbersome to write expressions. Even a simple addition of two numbers becomes an unwholesome jumble of constructors:

```
addExample :: Expr (Val :+: Add)
addExample = In (Inr (Add (In (Inl (Val 118))) (In (Inl (Val 1219)))))
```

Obviously, writing such expressions by hand is simply not an option. Furthermore, if we choose to extend our expression language even further by constructing larger coproducts, we will need to update any values we have written: the injections *Inl*

and *Inr* may no longer be the right injection into the coproduct. Before we deal with these problems, however, we consider the more pressing issue of how to evaluate such expressions.

3 Evaluation

The first observation we make, is that the types we defined to form the signatures of an *Expr* are both functors.

```
instance Functor Val where
  fmap f (Val x) = Val x
instance Functor Add where
  fmap f (Add e1 e2) = Add (f e1) (f e2)
```

Furthermore, the coproduct of two functors, is itself a functor.

```
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (Inl e1) = Inl (fmap f e1)
  fmap f (Inr e2) = Inr (fmap f e2)
```

These are crucial observations. If f is a functor, we can fold over any value of type *Expr f* as follows:

```
foldExpr :: Functor f ⇒ (f a → a) → Expr f → a
foldExpr f (In t) = f (fmap (foldExpr f) t)
```

This fold generalizes the folds for lists that you may know already. The first argument of the fold is called an *algebra*. An algebra of type $f\ a \to a$ determines how the different constructors of a data type affect the final outcome: it specifies one step of recursion, turning a value of type $f\ a$ into the desired result a. The fold itself uniformly applies these operations to an entire expression.

Using Haskell's type class system, we can define and assemble algebras in a modular fashion. We begin by introducing a separate class corresponding to the algebra we aim to define.

```
class Functor f ⇒ Eval f where
  evalAlgebra :: f Int → Int
```

The result of evaluation should be an integer; this is reflected in our choice of algebra. As we want to evaluate expressions consisting of values and addition, we need to define the following two instances:

```
instance Eval Val where
  evalAlgebra (Val x) = x
instance Eval Add where
  evalAlgebra (Add x y) = x + y
```

These instances correspond exactly to the cases from our original definition of evaluation in the introduction. In the case for addition, the variables x and y are not expressions, but the result of a recursive call.

Last of all, we also need to evaluate composite functors built from coproducts. Defining an algebra for the coproduct $f :+: g$ boils down to defining an algebra for the individual functors f and g.

```
instance (Eval f, Eval g) ⇒ Eval (f :+: g) where
  evalAlgebra (Inl x) = evalAlgebra x
  evalAlgebra (Inr y) = evalAlgebra y
```

With all these ingredients in place, we can finally define evaluation by folding over an expression with the algebra we have defined above.

```
eval :: Eval f ⇒ Expr f → Int
eval expr = foldExpr evalAlgebra expr
```

Using *eval* we can indeed evaluate simple expressions.

```
Main⟩ eval addExample
1337
```

Although we can now define functions over expressions using folds, actually writing expressions such as *addExample*, is still rather impractical to say the least. Fortunately, we can automate most of the overhead introduced by coproducts.

4 Automating injections

The definition of *addExample* illustrates how messy expressions can easily become. In this section, we remedy the situation by introducing smart constructors for addition and values.

As a first attempt, we might try writing:

```
val :: Int → Expr Val
val x = In (Val x)

infixl 6 ⊕

(⊕) :: Expr Add → Expr Add → Expr Add
x ⊕ y = In (Add x y)
```

While this is certainly a step in the right direction, writing *val* $1 \oplus$ *val* 3 will result in a type error. The smart constructor *add* expects two expressions that must themselves solely consist of additions, rather than values.

We need our smart constructors to be more general. We will define smart constructors with the following types:

```
(⊕) :: (Add :≺: f) ⇒ Expr f → Expr f → Expr f
val :: (Val :≺: f) ⇒ Int → Expr f
```

You may want to read the type constraint $Add :\prec: f$ as 'any signature f that supports addition.'

The constraint $sub :\prec: sup$ should only be satisfied if there is some injection from $sub\ a$ to $sup\ a$. Rather than write the injections using *Inr* and *Inl* by hand, the injections will be inferred using this type class.

```
class (Functor sub, Functor sup) ⇒ sub :≺: sup where
  inj :: sub a → sup a
```

The $(:\prec:)$ class only has three instances. These instances are not Haskell 98, as there is some overlap between the second and third instance definition. Later on, we will see why this should not result in any unexpected behavior.

```
instance Functor f ⇒ f :≺: f where
  inj = id
instance (Functor f, Functor g) ⇒ f :≺: (f :+: g) where
  inj = Inl
instance (Functor f, Functor g, Functor h, f :≺: g) ⇒ f :≺: (h :+: g) where
  inj = Inr ∘ inj
```

The first instance states that $(:\prec:)$ is reflexive. The second instance explains how to inject any value of type $f\ a$ to a value of type $(f :+: g)\ a$, regardless of g. The third instance asserts that provided we can inject a value of type $f\ a$ into one of type $g\ a$, we can also inject $f\ a$ into a larger type $(h :+: g)\ a$ by composing the first injection with an additional *Inr*.

We use coproducts in a list-like fashion: the third instance only searches through the right-hand side of coproduct. Although this simplifies the search—we never perform any backtracking—it may fail to find an injection, even if one does exists. For example, the following constraint will not be satisfied:

```
f :≺: ((f :+: g) :+: h)
```

Yet clearly $Inl \circ Inl$ would be a suitable candidate injection. Users should never encounter these limitations, provided their coproducts are not explicitly nested. By declaring the type constructor $(:+:)$ to be right-associative, types such as $f :+: g :+: h$ are parsed in a suitable fashion.

Using this type class, we define our smart constructors as follows:

```
inject :: (g :≺: f) ⇒ g (Expr f) → Expr f
inject = In ∘ inj
val :: (Val :≺: f) ⇒ Int → Expr f
val x  = inject (Val x)
(⊕) :: (Add :≺: f) ⇒ Expr f → Expr f → Expr f
x ⊕ y = inject (Add x y)
```

Now we can easily construct and evaluate expressions:

```
Main⟩ let x :: Expr (Add :+: Val) = val 30000 ⊕ val 1330 ⊕ val 7
Main⟩ eval x
31337
```

The type signature of x is very important! We exploit the type signature to figure out the injection into a coproduct: if we fail to provide the type signature, a compiler has no hope whatsoever of guessing the right injection.

As we mentioned previously, there is some overlap between the instances of the $(:\prec:)$ class. Consider the following example:

```
inVal :: Int → Expr (Val :+: Val)
inVal i = inject (Val i)
```

Which injection should be inferred, *Inl* or *Inr*? There is no reason to prefer one over the other—both choices are justified by the above instance definitions. The functions we present here, however, do not inspect *where* something occurs in a coproduct. Indeed, we can readily check that *eval* (*In* (*Inl* (*Val x*))) and *eval* (*In* (*Inr* (*Val x*))) are equal for all integers x as the instance of the *Eval* class for coproducts does not distinguish between *Inl* and *Inr*. In other words, the result of *eval* will never depend on the choice of injection. Although we need to allow overlapping instances to compile this class, it should only result in unpredictable behavior if you abuse the information you have about the order of the constructors of an expression.

5 Examples

So far we have done quite some work to write code equivalent to the evaluation function defined in introduction. It is now time to reap the rewards of our investment. How much effort is it to add multiplication to our little expression language? We begin by defining a new type and its corresponding functor instance.

```
data Mul x = Mul x x

instance Functor Mul where
  fmap f (Mul x y) = Mul (f x) (f y)
```

Next, we define how to evaluate multiplication and add a smart constructor.

```
instance Eval Mul where
  evalAlgebra (Mul x y) = x * y

infixl 7 ⊗
(⊗) :: (Mul :≺: f) ⇒ Expr f → Expr f → Expr f
x ⊗ y = inject (Mul x y)
```

With these pieces in place, we can evaluate expressions with multiplication:

```
Main⟩ let x :: Expr (Val :+: Add :+: Mul) = val 80 ⊗ val 5 ⊕ val 4
Main⟩ eval x
404
Main⟩ let y :: Expr (Val :+: Mul) = val 6 ⊗ val 7
Main⟩ eval y
42
```

As the second example illustrates, we can also write and evaluate expressions of type *Expr* (*Val* :+: *Mul*), thereby leaving out addition. In fact, once we have a menu

of expression building blocks, we can assemble our own data types *à la carte*. This is not even possible with proposed language extensions for open data types (Löh & Hinze, 2006).

Adding new functions is not very difficult. As a second example, we show how to render an expression as a string. Instead of writing this as a fold, we give an example of how to write open-ended functions using recursion directly.

We begin by introducing a class, corresponding to the function we want to write. An obvious candidate for this class is:

```
class Render f where
  render :: f (Expr f) → String
```

The type of *render*, however, is not general enough. To see this, consider the instance definition for *Add*. We would like to make recursive calls to the subtrees, which themselves might be values, for instance. The above type for *render*, however, requires that all subtrees of *Add* are themselves additions. Clearly this is undesirable. A better choice for the type of render is:

```
class Render f where
  render :: Render g ⇒ f (Expr g) → String
```

This more general type allows us to make recursive calls to any subexpressions of an addition, even if these subexpressions are not additions themselves.

Assuming we have defined instances of the *Render* class, we can write a function that calls *render* to pretty print an expression.

```
pretty :: Render f ⇒ Expr f → String
pretty (In t) = render t
```

All that remains, is to define the desired instances of the *Render* class. These instances closely resemble the original *render* function defined in the introduction; there should be no surprises here.

```
instance Render Val where
  render (Val i) = show i
instance Render Add where
  render (Add x y) = "(" ++ pretty x ++ " + " ++ pretty y ++ ")"
instance Render Mul where
  render (Mul x y) = "(" ++ pretty x ++ " * " ++ pretty y ++ ")"
instance (Render f, Render g) ⇒ Render (f :+: g) where
  render (Inl x) = render x
  render (Inr y) = render y
```

Sure enough, we can now pretty-print our expressions:

```
Main⟩ let x :: Expr (Val :+: Add :+: Mul) = val 80 ⊗ val 5 ⊕ val 4
Main⟩ pretty x
"((80 * 5) + 4)"
```

Finally, it is interesting to note that the *inj* function of the $(:\prec:)$ class has a partial inverse. We could have defined the $(:\prec:)$ class as follows:

```
class (Functor sub, Functor sup) ⇒ sub :≺: sup where
  inj :: sub a → sup a
  prj :: sup a → Maybe (sub a)
```

The *prj* function is straightforward to define for the three instances of the $(:\prec:)$ class defined above. When writing complex pattern matches on expressions, the *prj* function is particularly useful. For example, we may want to rewrite expressions, distributing multiplication over addition. To do so, we would need to know if one of the children of a *Mul* constructor is an *Add*. Using the *Maybe* monad and *prj* function, we can try to apply the distributive law on the outermost constructors of an expression as follows:

```
match :: (g :≺: f) ⇒ Expr f → Maybe (g (Expr f))
match (In t) = prj t

distr :: (Add :≺: f, Mul :≺: f) ⇒ Expr f → Maybe (Expr f)
distr t = do
  Mul a b ← match t
  Add c d ← match b
  return (a ⊗ c ⊕ a ⊗ d)
```

Using the *distr* function, one can define an algebra to fold over an expression, applying distributivity uniformly wherever possible, rather than just inspecting the outermost constructor.

These examples illustrate how we can add both new functions and new constructors to our types, without having to modify existing code. Interestingly, this approach is not limited to data types: we can also use the same techniques to combine a certain class of monads.

6 Monads for free

Most modern calculators are capable of much more than evaluating simple arithmetic expressions. Besides various other numeric and trigonometric operations, calculators typically have a memory cell storing a single number. Pure functional programming languages, such as Haskell, encapsulate such mutable state using monads. Despite all their virtues, however, monads are notoriously difficult to combine. Can we extend our approach to combine monads using coproducts?

In general, the coproduct of two monads is fairly complicated (Lüth & Ghani, 2002). We choose to restrict ourself to monads of the following form:

```
data Term f a =
    Pure a
  | Impure (f (Term f a))
```

These monads consist of either pure values or an impure effect, constructed using f. When f is a functor, *Term* f is a monad. This is illustrated by the following two instance definitions.

```
instance Functor f ⇒ Functor (Term f) where
  fmap f (Pure x)   = Pure (f x)
  fmap f (Impure t) = Impure (fmap (fmap f) t)
instance Functor f ⇒ Monad (Term f) where
  return x          = Pure x
  (Pure x) >>= f    = f x
  (Impure t) >>= f  = Impure (fmap (>>= f) t)
```

These monads are known as *free monads* (Awodey, 2006).

Several monads you may already be familiar with are free monads. Consider the following types:

```
data Zero a
data One a = One
data Const e a = Const e
```

Now *Term Zero* is the identity monad; *Term One* corresponds to the *Maybe* monad; and *Term* (*Const e*) is the error monad. Most monads, however, are not free monads. Notable examples of monads that are not free include the list monad and the state monad.

In general, a structure is called *free* when it is left-adjoint to a forgetful functor. In this specific instance, the *Term* data type is a higher-order functor that maps a functor f to the monad *Term* f; this is illustrated by the above two instance definitions. This *Term* functor is left-adjoint to the forgetful functor from monads to their underlying functors.

All left-adjoint functors preserve coproducts. In particular, computing the coproduct of two free monads reduces to computing the coproduct of their underlying functors, which is exactly what we achieved in Section 2. Throughout this section, we will exploit this property to define monads modularly.

Although the state monad is not a free monad, we can use the *Term* data type to represent a language of stateful computations. We can incrementally construct these terms and interpret them as computations in the state monad.

We will consider simple calculators that are equipped with three buttons for modifying the memory:

Recall The memory can be accessed using the recall button. Pressing the recall button returns the current number stored in memory.

Increment You can add an integer to the number currently stored in memory using the $M+$ button. To avoid confusion with the coproduct, we will refer to this button as *Incr*.

Clear Finally, the memory can be reset to zero using a *Clear* button.

We will implement the first two operations, leaving *Clear* as an exercise.

Once again, we define types *Incr* and *Recall* corresponding to the operations we wish to introduce. The *Incr* constructor takes two arguments: the integer with which to increment the memory, and the rest of the computation. The *Recall* constructor takes a single, functional argument that expects to receive the contents of the memory cell. Given the contents, it will continue with the rest of the computation. Both these types are obviously functors.

```
data Incr t = Incr Int t
data Recall t = Recall (Int → t)
```

To facilitate writing such terms, we introduce another series of smart constructors, analogous to the smart constructors we have seen for expressions.

```
inject :: (g :<: f) ⇒ g (Term f a) → Term f a
inject = Impure ∘ inj
incr :: (Incr :<: f) ⇒ Int → Term f ()
incr i = inject (Incr i (Pure ()))
recall :: (Recall :<: f) ⇒ Term f Int
recall = inject (Recall Pure)
```

Using Haskell's **do**-notation, we can construct complex terms quite succinctly. For instance, the *tick* term below increments the number stored in memory and returns its previous value.

```
tick :: Term (Recall :+: Incr) Int
tick = do y ← recall
          incr 1
          return y
```

Note that we could equally well have given *tick* the following, more general type:

```
(Recall :<: f, Incr :<: f) ⇒ Term f Int
```

There is a clear choice here. We could choose to let *tick* work in any *Term* that supports these two operations; or we could want to explicitly state that *tick* should only work in the *Term* (*Recall* :+: *Incr*) monad.

In order to write functions over terms, we define the following fold:

```
foldTerm :: Functor f ⇒ (a → b) → (f b → b) → Term f a → b
foldTerm pure imp (Pure x)   = pure x
foldTerm pure imp (Impure t) = imp (fmap (foldTerm pure imp) t)
```

The first argument, *pure*, is applied to pure values; the case for impure terms closely resembles the fold over expressions.

To execute our terms, we must still define a suitable algebra to pass to the *foldTerm* function. It is not immediately obvious what the type of our algebra should be. Clearly, we will need to keep track of the state of our memory cell. To avoid any confusion with other integer values, we introduce a separate data type

that represents the contents of the memory cell:

```
newtype Mem = Mem Int
```

To interpret our terms in the state monad, we aim to define a *run* function with the following type:

```
run :: ... ⇒ Term f a → Mem → (a, Mem)
```

The *run* function should take a term and initial state of the memory, and execute the term, returning a result value of type a and the final state of the memory cell. As we wish to define *run* as a fold, this determines the type of our algebra and motivates the following class definition:

```
class Functor f ⇒ Run f where
  runAlgebra :: f (Mem → (a, Mem)) → (Mem → (a, Mem))
```

We can now write suitable instances for *Incr*, *Recall*, and coproducts.

```
instance Run Incr where
  runAlgebra (Incr k r) (Mem i) = r (Mem (i + k))
instance Run Recall where
  runAlgebra (Recall r) (Mem i) = r i (Mem i)
instance (Run f, Run g) ⇒ Run (f :+: g) where
  runAlgebra (Inl r) = runAlgebra r
  runAlgebra (Inr r) = runAlgebra r
```

In the case for *Incr* we increment the memory cell and continue recursively; for *Recall* we lookup the value stored in memory, but leave the state of the memory unchanged; the instance definition for coproducts should be familiar.

Using the fold over terms and the above algebra, we define the *run* function. In the base case, we simply tuple the memory and value being returned—in a similar fashion to the *return* of the state monad. For the *Impure* case, we use the *runAlgebra* we have defined above.

```
run :: Run f ⇒ Term f a → Mem → (a, Mem)
run = foldTerm (,) runAlgebra
```

Using *run* we can execute our *tick* function as follows:

```
Main⟩ run tick (Mem 4)
(4, Mem 5)
```

We could have written out functions *incr* and *recall* directly in the state monad $Mem \to (a, Mem)$. What have we gained by the extra indirection introduced by the *Term* data type? Looking at the *type* of our terms, we can now say something about their behavior. For example, any term of type *Term Recall Int* will never modify the state of the memory cell; dually, any term of type *Term Incr a* will produce the same result, regardless of the initial state of the memory cell. If we had just written functions in the state monad directly, we could not have distinguish these special kinds of computations.

As was the case for expressions, we can extend our terms with new operations, allowing us to assemble monads modularly. It is important to emphasize that this technique for combining monads does not generalize all Haskell's monads—free monads are a special case that we can deal with quite nicely.

7 Applications

For all its beauty, Haskell does have its less appealing aspects. In particular, the IO monad has evolved into a 'sin bin' that encapsulates every kind of side effect from *addFinalizer* to *zeroMemory*. With the technology presented in the previous sections, we can be much more specific about what kind of effects certain expressions may have.

Consider the following two data types, describing two classes of IO operation from the Haskell prelude:

```
data Teletype a =
    GetChar (Char → a)
  | PutChar Char a
data FileSystem a =
    ReadFile FilePath (String → a)
  | WriteFile FilePath String a
```

We can execute terms constructed using these types by calling the corresponding primitive functions from the Haskell Prelude. To do so, we define a function *exec* that takes pure terms to their corresponding impure programs.

```
exec :: Exec f ⇒ Term f a → IO a
exec = foldTerm return execAlgebra
```

The *execAlgebra* merely gives the correspondence between our constructors and the Prelude. Note that we qualify the IO functions imported from the Prelude to avoid name clashes.

```
class Functor f ⇒ Exec f where
  execAlgebra :: f (IO a) → IO a
instance Exec Teletype where
  execAlgebra (GetChar f)    = Prelude.getChar >>= f
  execAlgebra (PutChar c io) = Prelude.putChar c >> io
```

The instance definitions for *FileSystem* and coproducts have been omitted; they are entirely unremarkable. Provided we define smart constructors as before, we can write pseudo-IO programs without any syntactic overhead beyond the obligatory type signature:

```
cat :: FilePath → Term (Teletype :+: FileSystem) ()
cat fp = do
  contents ← readFile fp
  mapM putChar contents
  return ()
```

Now the type of *cat* tells us exactly what kind of effects it uses: a much healthier situation than a single monolithic IO monad. For example, our types guarantee that executing a term in the *Term Teletype* monad will not overwrite any files on our hard disk. The types of our terms actually have something to say about their behavior! An additional advantage of this two-step approach is that the terms we write are pure Haskell values—information we can exploit if we are interested in debugging or reasoning about effectful functions (Swierstra & Altenkirch, 2007).

8 Discussion

There are many interesting topics that I have not covered. While we have encountered the fold over a data type, I have not mentioned the *unfold*. Furthermore, we have not dealt with polymorphic data types, such as lists or trees. Such data types can also be written using the techniques described above. Rather unsurprisingly, this requires a shift from functors to bifunctors.

This approach does have its limitations. Although GADTs and nested data types can also be expressed as initial algebras (Johann & Ghani, 2007; Johann & Ghani, 2008), doing so requires a higher-order representation of data types that can be a bit cumbersome to program with in Haskell. Furthermore, modular functions that take different types of modular arguments and return modular data types will require multi-parameter type classes and several other extensions to Haskell 98. It would be interesting to explore the limits of this approach further.

Much of the code presented here is part of the functional programming folklore. The fixed-points of functors and their corresponding folds have been introduced to the functional programming community more than fifteen years ago (Meijer *et al.*, 1991). More recently, Tim Sheard (2001) has proposed using fixed-points of functors to write modular code. Free monads are well understood in category theory, but are much less widespread in the functional programming community. The $(:\prec:)$ class is an obvious generalization of existing work on modular interpreters (Liang *et al.*, 1995). Yet, amazingly, no one has ever put all these pieces together.

I am sure there are many, many other ways to achieve results very similar to ones presented here. Haskell's type class system, along with its various dubious extensions, is open to all kinds of abuse. I doubt, however, there is a simpler, more tasteful solution.

Acknowledgments

Most of this work is the result of many entertaining and educational discussions with my colleagues at the University of Nottingham, for which I would like to express my sincere gratitude. Mauro Jaskelioff, Conor McBride, and Nicolas Oury deserve a particular mention for their ideas and encouragement. Thorsten Altenkirch, George Giorgidze, and Andres Löh all provided valuable feedback on a draft version of this paper. Last but not least, I would like to thank an anonymous reviewer for the helpful comments I received.

References

Awodey, S. (2006) *Category Theory*. Oxford Logic Guides, vol. 49. Oxford: Oxford University Press.

Johann, P. & Ghani, N. (2007) Initial algebra semantics is enough! *Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science, vol. 4583. Springer.

Johann, P. & Ghani, N. (2008) Foundations for structured programming with GADTs. In *Conference record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, pp. 297–308.

Liang, S., Hudak, P. & Jones, M. (1995) Monad transformers and modular interpreters. In *Conference record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, pp. 333–343.

Löh, A. & Hinze, R. (2006) Open data types and open functions. *Princ. Prac. Declarative Program. Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. Venice, Italy, pp. 133–144.

Lüth, C. & Ghani, N. (2002) Composing monads using coproducts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. Pittsburgh, PA, pp. 133–144.

Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture*.

Sheard, T. (2001) Generic unification via two-level types and parameterized modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. Florence, Italy, pp. 86–97.

Swierstra, W. & Altenkirch, T. (2007) Beauty in the beast: A functional semantics of the awkward squad. In *Proceedings of the ACM SIGPLAN Haskell Workshop*. Freiburg, Germany, pp. 25–36.

Wadler, P. (1998). The Expression Problem. Accessed at http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt