The Different Aspects of Monads and Mixins

Bruno C. d. S. Oliveira

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
bruno@comlab.ox.ac.uk

Abstract

Around twenty years ago two important developments happened in the areas of modularity and reuse in programming languages. On the one hand, Moggi showed how computational effects found in impure languages could be simulated using the notion of *monads* from category theory. Inspired by Moggi's work, Wadler showed how monads could be used to structure (purely functional) programs. On the other hand, work by Cook showed how variations of *mixins* could model different notions of *inheritance* (normally found in object-oriented languages) in simple, elegant and compositional ways, by using traditional techniques of fixed-point theory.

Monads and mixins are helpful to handle different aspects of modularity and reuse in programming languages, yet they have been largely explored independently. In this paper we show that the combination of monads and mixins leads to a simple *aspect-oriented programming* (AOP) style that can be used effectively in purely functional programming languages to write *elegant*, *reusable* and *modular* programs.

1. Introduction

Pioneering work by Moggi [1989, 1991] showed how the category theory notion of *monads* could be used to model computational effects and structure the denotational semantics of programming languages. This inspired Wadler [1992] to apply monads in essentially the same way, but to structure functional programs instead. By using monads, superficially different variations of purely functional programs that model similar impure programs become structured using the same abstractions. This has important benefits in terms of reuse and modularity.

Cook [1989] studied how variations of *mixins* could model different notions of *inheritance*—found in various object-oriented (OO) languages—in simple, elegant and compositional ways, by using traditional techniques of fixed-point theory. He noted that these techniques are not only applicable to OO languages, but they can also be applied, for example, to functional languages. In fact, many of Cook's models of inheritance are essentially nothing more than *simple, beautiful, combinator-style functional programs* that can be translated rather directly into a language like Haskell [Jones, 2003].

Monads and mixins are helpful to handle different aspects of modularity and reuse in programming languages, but so far they have been largely explored independently. A noteworthy exception is the work by Brown and Cook [2007], which studied the application of a simple variety of mixins in combination with monads to the problem of *memoization* [Michie, 1968] in purely functional languages.

The purpose of this paper is to show that the combination of monads and mixins yields significant modularity and reuse benefits and leads to a simple *aspect-oriented programming* [Kiczales et al., 1997] style, which can be used effectively in purely functional programming languages to write *elegant*, *reusable* and *modular* programs. The contributions of this paper are:

- A library of mixin combinators inspired by Cook [1989] and some concepts from AOP.
- A technique consisting of *monadification* [Erwig and Ren, 2004] and "*mixinfication*" of purely functional programs that can be used to write highly modular and reusable programs. This technique is helpful to solve the modularity problems discussed in Section 2.3.
- Several interesting examples illustrating the applications of the technique.

Overview In Section 2 we review Wadler's work on monads for structuring functional programs, discuss the modularity problem that monads solve (in purely functional languages) and some modularity problems that remain unsolved. Section 3 introduces *monad transformers*, which provide a standard way to combine different monads together and that we will make extensive use later in this paper. Section 4 introduces *mixins* and their use to model inheritance; and presents the base mixin library that we are going to use throughout the paper. Section 5 discusses the application of mixins and monads to modularly capture some aspects of interpreters. Section 6 shows another application of mixins for "*scrapping boilerplate*" [Lämmel and Peyton Jones, 2003] of traversal code. In Section 7 we extend the mixin library with some combinators inspired by ideas of AOP and exemplify their use to capture crosscutting concerns like *tracing* or *step counting*. Section 8 discusses our results and related work. Finally Section 9 concludes.

2. Monads and Modularity

This section introduces monads and their use for structuring functional programs, improving the modularity of programs written in purely functional languages. Some remaining modularity problems are also discussed.

2.1 A Modularity Problem in Purely Functional Languages

Wadler [1992, 1993] motivates the use of monads to structure functional programs by discussing a dilemma that occurs in purely functional languages. On the one hand purely functional languages

have the advantage that all the flow of data is explicit, consequently substitution of equals by equals is always valid and reasoning about programs becomes very easy. On the other hand making all flow of data explicit can sometimes be painfully verbose and obscure the essential details of a program. Even worse, this can cause pernicious modularity problems. This is illustrated by Wadler with a simple interpreter for expressions:

```
data Term = Con Int | Div Term Term

eval1              :: Term → Int
eval1 (Con a)      = a
eval1 (Div t u)    = eval1 t `div` eval1 u
```

The simple expressions that can be defined by $Term$ allow the definition of integer expressions and division of two terms. A simple evaluator for expressions is given by $eval_1$. One interesting change to this evaluator is to add division by 0 error checks to make it more robust. This can be achieved with the following program:

```
type Exception = String

eval2 :: Term → Either Exception Int
eval2 (Con a)      = Right a
eval2 (Div t u)    = case (eval2 t, eval2 u) of
  (Left e,  _)          → Left e
  (Right x, Left e)     → Left e
  (Right x, Right y)    →
    if y ≡ 0 then Left "divide by zero"
             else Right (x `div` y)
```

However, this entails entails modifying all the clauses and recursive calls of the program to check and handle errors appropriately. This is in contrast with impure languages where no such restructuring would be necessary and only a relatively local change would be needed.

2.2 Monads to Structure Functional Programs

In order to avoid the excessive plumbing required to carry data around and the global restructurings of programs in purely functional languages, Wadler proposed to structure functional programs using monads, which were pioneered by Moggi [1989, 1991] to simulate computational effects such as *global state*, *exception handling*, *output* and *non-determinism* in a pure setting. In Haskell, monads can be captured by the following type class.

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

The function $return$ lifts a value of type a into a (pure) computation of the same type. The *binding* function $\gg\!=$ provides a way to apply a function of type $a \to m\ b$ to a computation of type $m\ a$. Syntactic sugar for those operations is provided in Haskell through the **do** notation.

A monadic evaluator for expressions can be written as follows:

```
meval1 :: Monad m ⇒ Term → m Int
meval1 (Con a)   = return a
meval1 (Div t u) = do x ← meval1 t
                      y ← meval1 u
                      return (x `div` y)
```

For integer terms we just return the integer denoted by the term; for divisions, we first compute the value of the dividend and bind it to x, then we compute the value of the divisor and bind it to y, and finally we return x `` `div` `` y.

To recover the simple evaluator given by $eval_1$ we can use the identity monad, which provides a trivial instance of monads; the $return$ and $\gg\!=$ operations are, respectively, the identity function and reverse application (modulo the isomorphisms for into and out of the Id type constructor).

```
newtype Id a = Id{runId :: a}

instance Monad Id where
  return x = Id x
  x >>= f  = f $ runId x
```

Now we could define $eval_1$ by instantiating the monad to Id and extracting the value contained in that structure:

```
eval1 :: Term → Int
eval1 = runId ∘ meval1
```

A more compelling example is given by the error monad

```
instance Monad (Either e) where
  return x = Right x
  x >>= f  = case x of
    Left e  → Left e
    Right x → f x

throwError :: String → Either Exception a
throwError = Left
```

which we can use to recover $eval_2$ from the generic monadic version, with a small local change on $meval_1$ to handle division by 0 errors.

```
meval2 :: Term → Either Exception Int
meval2 (Con a)   = return a
meval2 (Div t u) =
  do x ← meval2 t
     y ← meval2 u
     if y ≡ 0 then throwError "divide by zero"
              else return (x `div` y)
```

The benefits of the monadic approach should be clear: a lot of the plumbing needed for particular programs is abstracted in an instance of a monad and can be reused by other programs; and new orthogonal behaviour to the main functionality of the program can be added with much more local changes. This achieves Wadler's goal of structuring programs in purely functional programming languages with essentially the same modularity benefits of impure languages.

2.3 Some Remaining Modularity Problems

Unfortunately, from a modularity and reuse perspective, there are still a few of problems left.

The first problem (common to most impure languages) is that (usually) there is *no true reuse*: in order to adapt a program with some new orthogonal behaviour we normally have to directly modify the original program (even though the change is now much more localized). For example, in order to adapt the monadic evaluator to handle errors, we still have to change part of the last clause of $meval_1$. Moreover, to retain the two programs ($meval_1$ and $meval_2$) around we essentially need to manually copy most of the code and repeat it in both programs; this is the nemesis of modular reuse!

The second problem arises from the fact that monads make computational effects explicit on the types of the functions. This is not a disadvantage on its own, much on the contrary: because effects are explicit on the types, much stronger properties are known about the function. However, it is possible that adapting a program will break existing code—since adding new orthogonal behaviour may imply changing the type of the function.

A final problem is that it is not trivial to mix two different kinds of computational effects in the monadic approach: monads are well-known to be problematic to compose. Nonetheless, there is a fairly standard way to solve this problem through the use of *monad transformers* [Liang et al., 1995], which we will discuss in Section 3.

3. Monad Transformers

This section introduces monad transformers, which will play an important role in this paper.

```
newtype ErrorT e m a = ErrorT{
  runErrorT :: m (Either e a)
}
instance MonadTrans (ErrorT e) where
  lift m = ErrorT $ do {a ← m; return (Right a)}
class Monad m ⇒ MonadError e m | m → e where
  throwError :: e → m a
  catchError :: m a → (e → m a) → m a
instance Monad m ⇒ MonadError e (ErrorT e m) where
  throwError l      = ErrorT $ return (Left l)
  m `catchError` h = ErrorT $ do
    a ← runErrorT m
    case a of
      Left l  → runErrorT (h l)
      Right r → return (Right r)
instance Monad m ⇒ Monad (ErrorT e m) where
  return a = ErrorT $ return (Right a)
  m >>= k  = ErrorT $ do
    a ← runErrorT m
    case a of
      Left l  → return (Left l)
      Right r → runErrorT (k r)
```

Figure 1. Error monad transformer machinery.

3.1 Heavy (But Handy) Machinery for Monad Transformers

The concept of a monad transformer [Liang et al., 1995] provides a way to define programs that use different monads. With monads transformers, different kinds of monads can be layered on top of each other, offering a way to compose the functionality provided by each monad. A monad transformer can be defined with the following type class:

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a
```

The *lift* operation takes some monadic computation $m\ a$, and lifts it to the top-level monad $t\ m$. Of course, we must make $t\ m$ itself an instance of a monad. Monad transformers involve the definition of some machinery. In Figure 1 we show part of the machinery that is required for handling the error monad transformer. The newtype $ErrorT\ e\ m\ a$ defines the type of the transformer; an instance of $MonadTrans$ for $ErrorT\ e$ is also provided. With monad transformers we can have multiple types that are actually error monads. To better capture the notion of error monads a $MonadError\ e\ m$ type class with two methods *throwError* and *catchError* is provided; and an instance of that type class is defined for the error monad transformer type. Finally, we also need an instance of $Monad$ for $ErrorT\ e\ m$. Note that $MonadError$ requires *multiple parameter type classes with functional dependencies* [Jones, 2000]; both $MonadError$ and $MonadTrans$ as well as many other monad transformers can be found in the *Glasgow Haskell Compiler* (GHC) *monad transformer library*.

Suppose that we wanted to combine the functionality of an error monad with a state monad. With monad transformers this is easy. Assuming that all the monad and monad transformer machinery for the state monad is defined, what we would need to do is to essentially define how a error monad transformer behaves when combined with state. However, we shall not bother the reader with all the details required for this to work. For the reader interested in the details, we suggest Liang et al. [1995]. For the purposes of this paper it is enough to know that there are different monad transformer types (such as $ErrorT\ e\ m\ a$ or $StateT\ s\ m\ a$) and that there are type classes like $MonadError$ and $MonadState$ that abstract from particular implementations of error or state monads. We shall introduce the monad transformer types and respective type classes as we need them.

3.2 Using Monad Transformers

Despite the somewhat heavy machinery that is required for monad transformers, their use is fairly easy and intuitive, and the programmer does not need to be aware of most of the intricate details. For example, in order to use a state monad to count the number of divisions, we can use the operations provided by the $MonadState$ type class, without committing to a particular state monad. The state monad transformer type and the $MonadState$ type classes are defined as follows:

```
newtype StateT s m a = StateT{
  runStateT :: s → m (a, s)
}
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

We can now use the get and put operations to count the number of divisions in the evaluator:

```
seval :: MonadState Int m ⇒ Term → m Int
seval (Con a)   = return a
seval (Div t u) = do x ← seval t
                     y ← seval u
                     n ← get
                     put (n + 1)
                     return (x `div` y)
```

Using $seval$ we can easily define a program that given a term returns a pair of integers that represents the value of the term and the number of divisions performed in the process of evaluating the term.

```
eval₃ :: Term → (Int, Int)
eval₃ t = runId $ runStateT (seval t) 0
```

Here we instantiate the monad m in $seval$ to be $StateT\ Int\ Id$ and remove the monadic layers by running $runStateT$ and $runId$; the second argument of $runStateT$ represents the initial number of divisions.

Of course, the greatest advantage of monad transformers comes when we want to use the functionality from different monads together. For example, if we wanted to combine the error checking and division counting functionalities we could write the following program:

```
smeval :: (MonadState Int m, MonadError Exception m) ⇒
          Term → m Int
smeval (Con a)   = return a
smeval (Div t u) =
  do x ← smeval t
     y ← smeval u
     n ← get
     put (n + 1)
     if y ≡ 0 then throwError "divide by zero!"
              else return (x `div` y)
```

In this program both the state monad and the error monad operations are used, so the monad m needs to be both an instance of $MonadState$ and $MonadError$. There are actually a couple of interesting valid candidates for concrete instantiations of m. For example, we could have $m = StateT\ Int\ (ErrorT\ Exception\ Id)$ or $m = ErrorT\ Exception\ (StateT\ Int\ Id)$. We show two programs derived from these two instantiations next:

```
eval₄ :: Term → Either Exception (Int, Int)
eval₄ t = runId $ runErrorT $ runStateT (smeval t) 0
```

$eval_5 :: Term \to (Either\ Exception\ Int, Int)$
$eval_5\ t = runId\ \$\ runStateT\ (runErrorT\ (smeval\ t))\ 0$

These two programs have different behaviours (and this is visible from their types). Both programs will return the result of evaluating the term and the number of divisions if no exception occurs. However, if an exception occurs then $eval_4$ will throw away all the information apart from the exception. In contrast, the program defined by $eval_5$ will also throw an exception, but it will retain the numbers of divisions performed up to that point.

Monad transformers solve the problem of composing the functionality of different monads, but they do not solve the two other modularity problems discussed in Section 2.3.

4. Mixins

In this section we introduce *mixins* and their use to model inheritance. We also present the basic mixin library that we are going to use throughout the paper.

4.1 Mixin Inheritance

Object-oriented languages generally provide powerful reuse mechanisms based on *inheritance*. Cook [1989] studied those mechanisms in detail and proposed a model for inheritance using traditional techniques of fixed-point theory. Borrowing a simple (slightly modified) example from Cook, we illustrate the essential idea next:

$G_1 = \lambda this.[value \mapsto 7, square \mapsto this.value * this.value]$

Here G_1 defines a function that takes a record $this$ as argument and returns another record. For the purposes of this paper, we can think of the two records as having the same structure with fields $value$ and $square$. Functions of this kind, taking parameters with the same structure of the output (but not necessarily records), are essentially what we call $mixins$. G_1 is an instance of a simple type of mixins that Cook calls a *generator*. The record returned by G_1 assigns 7 to $value$ and defines $square$ in terms of $this$. For the reader familiar with object-oriented programming the intention of this program should be clear; $this$ is meant to represent the self-reference to the record (or object). However, $this$ can be any other record and $square$ may or may not be 49. In order to make $this$ the self-reference we can use a fixpoint operation:

$m_1 = fix\ G_1 = [value \mapsto 7, square \mapsto 49]$

In this case, the value of $square$ for m_1 is indeed 49. Now lets assume that we would have another generator G_2 defined as follows:

$G_2 = \lambda this.[value \mapsto 2]$

We can think of the result record has having the same structure as G_1 but being only partly defined (without a definition for $square$). We can combine G_1 with G_2 using some form of composition. For example, if we use what Cook calls the *preferential combination function* (that we represent here by $\boxplus$) and then apply the fixpoint operation we would obtain the following:

$m_2 = fix\ (G_1 \boxplus G_2) = [value \mapsto 2, square \mapsto 4]$

The preferential combination function overrides any fields of the second record that are already defined by the first record and *inherits* the remaining fields. Because $square$ is defined in terms of the self-reference, the value of $square$ will be computed in terms of $G_1 \boxplus G_2$ which will result in 4.

This example shows the essence of the model of mixin inheritance: we can write our programs in terms of self-references and then *compose* those programs to override and reuse existing functionality. In general, to handle all the aspects of inheritance in OO languages, we need to account for the possibility of records having different shapes and we may need combination functions that work by examining the structure of records. However, as we shall see next, we can still enjoy from many of the benefits of inheritance in a functional language like Haskell without considering those more intricate aspects of OO inheritance.

$\mathbf{type}\ Mixin\ s = s \to s \to s$
$(\oplus) :: Mixin\ s \to Mixin\ s \to Mixin\ s$
$f \oplus g = \lambda super\ this \to f\ (g\ super\ this)\ this$
$zero :: Mixin\ s$
$zero\ super\ this = super$
$mixin :: Mixin\ s \to s$
$mixin\ f = \mathbf{let}\ m = mixin\ f\ \mathbf{in}\ f\ m\ m$

Figure 2. Basic mixin combinators.

4.2 A Small Mixin Library

In Figure 2 we define the basis of our mixin library. The type $Mixin$ specifies the type of functions that can be used as mixins. This is just a more general version of *generators*, which also accounts for $super$ references (in OO languages $super$ typically refers to the object that we are inheriting from). The operation $\oplus$ defines mixin composition. It is easy to show that this operation is associative, and that it has the $zero$ mixin as left and right units of $\oplus$, forming a monoid.

$f \oplus zero = f = zero \oplus f$
$(f \oplus g) \oplus h = f \oplus (g \oplus h)$

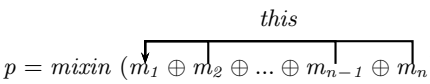
The function $mixin$ is the fixpoint combinator used to "deploy" a mixin (that is, given a mixin, it returns a function with the combined mixin functionality). Provided with these combinators, we could encode Cook's example in Haskell as follows:

$\mathbf{data}\ G = G\{value :: Int, square :: Int\}$
$g_1 :: Mixin\ G$
$g_1\ super\ this =$
  $G\{value = 7, square = value\ this * value\ this\}$
$g_2 :: Mixin\ G$
$g_2\ super\ this = super\{value = 2\}$
$m_1, m_2 :: G$
$m_1 = mixin\ g_1$
$m_2 = mixin\ (g_2 \oplus g_1)$

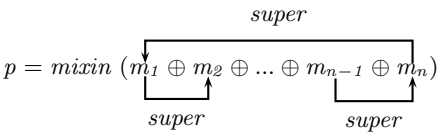
The model of inheritance given by our mixin library is different from that used in the example in Section 4.1 (which did not consider $super$ references), but the end result is the same. Note that the use of $super$ in g_2 has the effect of *inheriting* all the functionality from the $super$ reference, but overriding $value$. This particular model of inheritance does not rely on a composition operation that needs to consider the structure of records and consequently is a good model to be used in Haskell (which does not support operations that inspect the structure of records).

4.3 Visualizing Super and Self References

It is helpful to visualize what happens when we call $super$ and $this$ in a program that has a chain of mixins being composed. Self-references provide the simplest case:



Regardless of which mixin we are at, $this$ always points to the beggining of the mixin chain. For super references the behaviour is a bit more dynamic:



and in the mixin m_1 the *super* reference is pointing to m_2 (the next mixin in the chain); in the m_2 mixin *super* will point to the next mixin in the chain and so on for the other mixins. When the last mixin m_n is reached and there is no other mixin in the chain, *super* just points back to the beggining of the mixin chain (effectively behaving like *this* in that last mixin).

4.4 Relation to Cook's Models of Inheritance

The mixin composition operator $\oplus$ is related to what Cook describes in his thesis as being *boxed application*, which (in our setting) would correspond to[1]:

$$\textbf{type}\ \mathit{Gen}\ s = s \to s$$
$$(\boxtimes) :: \mathit{Mixin}\ s \to \mathit{Gen}\ s \to \mathit{Gen}\ s$$
$$m \boxtimes g = \lambda s \to m\ (g\ s)\ s$$

(Here the type Gen is just the type of generators.) Under Cook's own terminology, $\oplus$ would correspond to *boxed composition*.

Our treatment of the *super* reference is, however, different from Cook's in two respects. The first difference is that the type of the *super* and *self* references is the same and in Cook's models they are different (we also ignore subtyping issues). We should note that this is not a particularly significant difference though, because we could generalize the type of $Mixin$ to something like:

$$\textbf{type}\ \mathit{Mixin}\ s\ t = s \to t \to t$$

and still be able to define all the operations above (though $mixin$ would require that $s = t$). We choose to stick with the simpler type because it is easier to read and we have not encountered that many applications that required the more general type. The second difference is more fundamental: in Cook's and other usual treatments of inheritance the *super* reference does not normally appear closed under the fixpoint and a standard definition would be:

$$\mathit{mixin} :: \mathit{Mixin}\ \mathit{Top}\ s \to s$$
$$\mathit{mixin}\ f = f\ \mathit{top}\ (\mathit{mixin}\ f)$$

where Top represents the supertype of all types and top is the canonical value of Top. The problem with this treatment of *super* is that it assumes the existence of subtyping in the language (as provided, for example, by System $F_{<:}$ [Cardelli et al., 1994]), and Haskell does not support this. We solve this problem by assuming that $top = mixin\ f$.

4.5 Monadic Mixins

The technique that we promote in this paper consists on combining monads with mixins to modularize orthogonal aspects of programs. One application of this technique to *memoization* [Michie, 1968] was explored by Brown and Cook [2007], where they used generators to separate a program from the memoization aspect. We shall use this as a first example of the technique, suitably generalized to fit our mixins.

In Figure 3 we present a simple example of memoization mixins using our library. The functions *fib* and $memo$ define, respectively, mixins for the fibonacci sequence and memoization. The parameters $continue$ and $call$ (used in both *fib* and $memo$) are the two arguments of the mixin. Here we use the convention of naming the super and self references as $continue$ and $call$ (rather than *super* and *this*) because, for functional types, those names seem more intuitive to us: $call$ stands for recursive call; and $continue$ emphasizes the fact that the computation will continue before performing the recursive call. The state monad transformer is used by the memo mixin to read and update the cached values.

Using the $mixin$ function we can convert a mixin into a conventional function. For example if we mixin *fib* as follows

[1] We should note that the observations in this subsection (on the relationship between our combinators and Cook's own models of inheritance) are due to Cook himself and were provided to us in personal communication.

```
fib :: Monad m ⇒ Mixin (Int → m Int)
fib continue call n = case n of
  0 → return 0
  1 → return 1
  _ → do x ← call (n − 1)
         y ← call (n − 2)
         return (x + y)
memo :: MonadState (Map Int Int) m ⇒ Mixin (Int → m Int)
memo continue call x =
  do m ← get
     if member x m then return (m ! x)
                   else do y ← continue x
                           m' ← get
                           put (insert x y m')
                           return y
```

Figure 3. Memoization

```
data Expr where
  Lit      :: Int → Expr
  Var      :: String → Expr
  Plus     :: Expr → Expr → Expr
  Minus    :: Expr → Expr → Expr
  Assign   :: Expr → Expr → Expr
  Sequence :: [Expr] → Expr
  While    :: Expr → Expr → Expr
type Env = [(String, Int)]
```

Figure 4. Datatype and environment type for expressions.

```
nfib :: Monad m ⇒ Int → m Int
nfib = mixin fib
```

then we obtain a function *nfib* that is a monadic version of the fibonacci function. Alternatively, we could have composed $memo$ and *fib* together in the following way

```
mfib :: MonadState (Map Int Int) m ⇒ Int → m Int
mfib = mixin (memo ⊕ fib)
```

to obtain a memoized monadic version of the fibonacci function. A fast fibonacci function could be defined by suitably instantiating the state monad:

```
fastFib :: Int → Int
fastFib = fst ∘ flip runState empty ∘ mfib
```

5. The Modular Aspects of Interpreters

In this section we show how monads and mixins can be used together to build interpreters that can be modularly reused to derive new interpreters with same added orthogonal functionality. The interpreter that we will use in this section is an Haskell translation of an interpreter implemented in ML by Läufer [2003]. We use a monadic style instead of implicit side-effects. Note that the kind of modularity that we are discussing here is *different* from what is discussed by Liang et al. [1995], which is focused on extending interpreters with new language constructs.

5.1 A Classic Monadic Evaluator

In Figure 4 we present a datatype representing for a simple imperative language that can be used to compute numeric expressions. Integer literals and variables can be built using, respectively, the Lit and Var constructors. Simple primitive operations for addition and subtraction are available through the $Plus$ and $Minus$ constructors.

```
eval₁ :: MonadState Env m ⇒ Expr → m Int
eval₁ exp = case exp of
  Lit x              → return x
  Var s              → do e ← get
                          case lookup s e of
                            Just x → return x
                            _      → error msg
  Plus l r           → do x ← eval₁ l
                          y ← eval₁ r
                          return (x + y)
  Minus l r          → do x ← eval₁ l
                          y ← eval₁ r
                          return (x − y)
  Assign (Var x) r   → do e ← get
                          y ← eval₁ r
                          put ((x, y) : e)
                          return y
  Sequence []        → return 0
  Sequence (x : xs)  → eval₁ x ≫ eval₁ (Sequence xs)
  While c b          → do x ← eval₁ c
                          if (x ≡ 0) then return 0
                            else (eval₁ b ≫ eval₁ exp)
  where msg = "Variable not found!"
```

Figure 5. A classic monadic evaluator.

```
eval₁ :: (MonadWriter String m, MonadState Env m) ⇒
         String → Expr → m Int
eval₁ v exp = case exp of
  ...
  Assign (Var x) r | x ≡ v →    -- new case
    do e ← get
       y ← eval₁ v r
       put ((x, y) : e)
       tell (x ++ " = " ++ show y ++ "\n")
       return y
  Assign (Var x) r → ...        -- old case
  ...
  While c b →
    do n ← eval₁ v c
       if (n ≡ 0) then (tell "done\n" ≫ return 0)
         else (tell "repeating\n" ≫ eval₁ v b
                                  ≫ eval₁ v exp)
  ...
```

Figure 6. Modified evaluator with tracing and variable watching.

Mutable assignments to variables can be defined using *Assign* and sequential composition and while loops can be constructed with *Sequence* and *While*. A simple environment type for expressions is given by *Env*.

In Figure 5 we show a classic monadic evaluator for the expressions presented in Figure 4. The state monad transformer is used to pass the environment around and it is also used in the assignment clause to update the value of the variable being assigned. The evaluator is quite standard. Evaluating integer literals returns the integer denoted by the literal. The evaluation of variables looks up the variable from the environment and returns its value; if no value is found an error is raised. The primitive arithmetic operations are evaluated in a similar way: both arguments of the operations are evaluated and the corresponding arithmetic operations are applied to the result of the evaluations. For assignments we need to evaluate the expression being assigned and update the variable with the new value. Sequential composition of an empty list of expressions returns 0, while the sequential composition of a list of expressions is the result of evaluating the expression in the head and the expressions in the tail. Finally, while loops are evaluated similarly to the C programming language, with integers playing the role of booleans: we first evaluate the condition, if that condition is 0 we stop and return 0, otherwise we evaluate the body of the while loop and evaluate the original while loop expression again.

Suppose that, for debbuging reasons, we wanted to watch the assignments of some variable and trace the execution of the while loops. In order to achieve this with the monadic evaluator presented in Figure 5, we would need to directly change the original program and adapt it with the extra functionality. We can see the necessary changes in Figure 6. We only show the parts of the program that need to be modified. The modularity problems discussed in Section 2.3 should be evident. On the one hand we need to modify the type signature of the program in two ways: we need stronger requirements about the monad transformer in use; and we need an extra string argument, which is the variable to be watched. This can of course, break programs that were using the older version of $eval_1$. On the other hand, we also need to change the main body of the program in a couple of places: we need to add a new case for the assignment clause (or, alternatively, modify the existing one with an **if** expression) to add the code to watch the variable v; and we need to decorate the code for evaluating the while loop with tracing code. Note that we also need to change all the recursive calls of the program to account for the extra argument, but this could be avoided using a reader monad.

5.2 A Modular Monadic Evaluator with Mixins

There are significant reuse and modularity problems with the approach in Section 5.1, but these can basically be solved if we combine monads with mixins. In Figure 7 we show the code for a

```
beval :: MonadState Env m ⇒ Mixin (Expr → m Int)
beval continue call exp = case exp of
  Lit x              → return x
  Var s              → do e ← get
                          case lookup s e of
                            Just x → return x
                            _      → error msg
  Plus l r           → do x ← call l
                          y ← call r
                          return (x + y)
  Minus l r          → do x ← call l
                          y ← call r
                          return (x − y)
  Assign (Var x) r   → do e ← get
                          y ← call r
                          put ((x, y) : e)
                          return y
  Sequence []        → return 0
  Sequence (x : xs)  → call x ≫ call (Sequence xs)
  While c b          → do x ← call c
                          if (x ≡ 0) then return 0
                            else (call b ≫ call exp)
  where msg = "Variable not found!"
```

Figure 7. A monadic evaluator with mixins.

```
weval :: (Show a, MonadWriter String m) ⇒
         String → Mixin (Expr → m a)
weval y continue call exp = case exp of
  Assign (Var x) r | x ≡ y →
    do n ← continue exp
       tell (x ++ " = " ++ show n ++ "\n")
       return n
  _ → continue exp
```

Figure 8. The watching variables aspect.

```
teval :: MonadWriter String m ⇒ Mixin (Expr → m Int)
teval continue call exp = case exp of
  While c b →
    do n ← call c
       if (n ≡ 0) then (tell "done\n" ≫ return 0)
         else (tell "repeating\n" ≫ call b ≫ call exp)
  _          → continue exp
```

Figure 9. The tracing loops aspect.

monadic evaluator using mixins. Instead of directly making recursive calls, this evaluator uses the *call* parameter provided by the mixin. Apart from this fairly easy change to the program, we only need to change the type of the function (wrapping *Mixin* around the functional type) and add two arguments for the *continue* and *call* functions provided by the mixin. To recover the basic monadic evaluator presented in Figure 5 we apply the *mixin* function to *beval* as follows:

```
eval₂ :: MonadState Env m ⇒ Expr → m Int
eval₂ = mixin beval
```

In Figure 8 we show how we could modularly define a watching aspect for assignments. This aspect has one extra string argument y, which is the variable to watch. For all the other cases except assignment we *inherit* the functionality from the *continue* function. For the *Assign* constructor we may do something different *overriding* the functionality provided by *continue*. Since we want to watch what happens in the assignments of y we have to compare y with the variable being assigned and, if they represent the same variable, call the *continue* function to execute the assignment code as well as add the extra watching code using the writer monad transformer:

```
newtype WriterT w m a = WriterT{
  runWriterT :: m (a, w)
}
class (Monoid w, Monad m) ⇒
  MonadWriter w m | m → w where
    tell :: w → m ()
```

If y does not match the variable being assigned, then the guard will fail and the execution will fallthrough the default case, just executing the standard assignment code provided by *continue*.

In Figure 9 we show how we could modularly define the code for the tracing while loops using mixins. Like the watching aspect, we handle all cases except *While* through *continue*. For the *While* constructor we make a recursive call using *call*. This has the effect of *completely overriding* all the code for handling while loops. Consequently, we need to essentially repeat the code that we have in *beval*, but this time decorated by some tracing code using the writer monad transformer. We could potentially have tried to achieve a bit more of reuse for the code handling the *While* constructor. However the point here is that, just like in object-oriented languages, we may choose to completely override existing functionality, partly override some functionality adding some extra code, or just inherit the functionality unchanged.

Having created the different mixins for the interpreter, we can combine them using the *mixin* function. For example, we can define the following programs:

```
watchy :: (MonadWriter String m, MonadState Env m) ⇒
          Expr → m Int
watchy = mixin (weval "y" ⊕ beval)

debug :: (MonadWriter String m, MonadState Env m) ⇒
         Expr → m Int
debug = mixin (weval "x" ⊕ weval "y" ⊕
                teval ⊕ weval "r" ⊕ beval)
```

The first program watches a variable "y", while executing the program; the second program watches the variables "x", "y" and "r" while tracing the while loops and executing the program. A suitable C-like program to be used with *watchy* and *debug* would be the following:

```
int x = 2; int y = 3; int r = 0;
while (y){r = r + x; y = y − 1; }
```

which would be represented by the expression:

```
x = Var "x"
y = Var "y"
r = Var "r"

program = Sequence
  [Assign x (Lit 2), Assign y (Lit 3), Assign r (Lit 0),
    While y (Sequence [
      Assign r (Plus r x), Assign y (Minus y (Lit 1))])]
```

By suitably picking state and writer monads to run those programs with, we can write programs that return the string log resulting from the tracing and watching code. For example, $test_1$ and $test_2$ apply *watchy* and *debug* to *program* and return the string built during the execution of the program.

```
test₁ = snd ∘ fst $
  runState (runWriterT (watchy program)) []

test₂ = snd ∘ fst $
  runState (runWriterT (debug program)) []
```

We can see the logs by applying *putStr* to $test_1$ and $test_2$. For $test_1$ we would get the following output:

```
y = 3
y = 2
y = 1
y = 0
```

which shows the different assignments to the variable "y" through the execution of the program. For $test_2$ the following would be obtained:

```
x = 2
y = 3
r = 0
repeating
r = 2
y = 2
repeating
r = 4
y = 1
repeating
r = 6
y = 0
done
```

This string shows all the assignments that occurred through the execution of the program as well as the traces of the while loop.

As we have seen the changes from the traditional monadic version into the version with mixins are fairly small and do not introduce any significant extra burden. Still, the benefits in terms of

```
compos_Expr :: (Expr → Expr) → (Expr → Expr)
compos_Expr call e = case e of
  Lit x          → Lit x
  Var s          → Var s
  Plus e1 e2     → Plus (call e1) (call e2)
  Minus e1 e2    → Minus (call e1) (call e2)
  Assign e1 e2   → Assign (call e1) (call e2)
  Sequence l     → Sequence (map call l)
  While e1 e2    → While (call e1) (call e2)
rename :: Expr → Expr
rename e = case e of
  Var s          → Var ("_" ++ s)
  _              → compos_Expr rename e
```

Figure 10. Renaming and the *compos* operation for expressions.

```
compos_Expr :: Mixin (Expr → Expr)
compos_Expr continue call e = case e of
  Lit x          → Lit x
  Var s          → Var s
  Plus e1 e2     → Plus (call e1) (call e2)
  Minus e1 e2    → Minus (call e1) (call e2)
  Assign e1 e2   → Assign (call e1) (call e2)
  Sequence l     → Sequence (map call l)
  While e1 e2    → While (call e1) (call e2)
renameMix :: Mixin (Expr → Expr)
renameMix continue call e = case e of
  Var s          → Var ("_" ++ s)
  _              → continue e
rename :: Expr → Expr
rename = mixin (renameMix ⊕ compos_Expr)
```

Figure 11. Generalization of *compos* and renaming using mixins.

reuse are very significant. In order to add a new orthogonal piece of the functionality we do not need to alter the original program or manually copy the code and create a new function with the extra functionality. Instead, through the use of mixins, we can just create new mixins that inherit the behaviour of the original program and override just the functionality that needs to be changed. Furthermore, the type of the original program does not need to change, but the mixins around that program can refine the types and adapt themselves to the new functionality. Consequently the combination of monads and mixins solves the problems discussed in Section 2.3.

6. Scrapping Boilerplate With Mixins

A different application of mixins comes from the area of (datatype) generic programming where some approaches [Lämmel and Peyton Jones, 2003, Bringert and Ranta, 2008] have been proposed to "*scrap your boilerplate*" [Lämmel and Peyton Jones, 2003] derived from traversals of large data structures. Common to these approaches is the use of ad-hoc techniques to achieve reuse (or inheritance) of traversal code. We will look at the technique proposed by Bringert and Ranta [2008] and show how mixins can be used to generalize that technique in useful ways.

The key idea of Bringert and Ranta is that many traversals are slight variations of a common scheme that they name *compos*. Bringert and Ranta propose a generalized *compos* operation using of *applicative functors* [McBride and Paterson, 2008], which are a structure that generalizes monads. However, for the purposes of this paper, the use of a simpler version (which does not considers effects) suffices to demonstrate the advantages of using mixins. In Figure 10 we show the simple version of the *compos* operation for the expressions in Figure 5. Essentially, this operation would define the identity traversal if all the occurrences of *call* would be replaced by a recursive call. However, this is not the case and *call* can be something other than just the recursive call to $compos_{Expr}$. We can exploit this to define a *rename* operation that renames all the variables of an expression by appending an "_" to all names. For all the other cases we call $compos_{Expr}$ parametrized with *rename*, which as the effect of inheriting all the code from $compos_{Expr}$ (except for the *Var* case). In essence, this is a simple ad-hoc approach to inheritance.

One problem with the technique proposed by Bringert and Ranta is that we can only inherit from $compos_{Expr}$. Functions that are defined in terms of $compos_{Expr}$ (like *rename*) cannot themselves be inherited because the recursion knot has been closed. One way around this is to use mixins instead and create a renaming mixin. We illustrate the idea in Figure 11. The function $compos_{Expr}$ is suitably generalized to use mixins; and we define a $renameMix$ mixin with essentially the same definition as *rename* except that the call to $compos_{Expr}$ is replaced by a *continue* call. We can easily recover *rename* by applying the *mixin* operation to $renameMix \oplus compos_{Expr}$.

The advantage of making renaming a mixin is that we can now combine it with other mixins. For example, we could define a simple simplifier for expressions as follows:

```
simplify :: Mixin (Expr → Expr)
simplify continue call e = case e of
  Plus (Lit 0) r → continue r
  Plus l (Lit 0) → continue l
  _              → continue e
```

and we could easily combine this with the renaming mixin through mixin composition, obtaining a function that simplifies expressions and renames variables.

```
renSimpl :: Expr → Expr
renSimpl = mixin (simplify ⊕ renameMix ⊕ compos_Expr)
```

7. Crosscutting Aspects

The aspects discussed in Section 5 are tightly coupled with the interpreter functionality in the following sense: they add extra functionality in the context of that interpreter, but would not be in general reusable by other programs. Some aspects are more context independent and can be used by many programs in different domains. In this section we shall see how we can define these crosscutting aspects with our techniques and some additional mixin combinators inspired by AOP.

7.1 Yet More Modularization Oportunities

Lets return to the simple interpreter example by Wadler (shown in Section 2.1). By applying mixins, and using essentially the same approach that we have taken in Section 5, we could easily solve the modularity problems discussed in Section 2.3. We show the resulting code in Figure 12. The *meval* mixin provides the basic monadic evaluator functionality; *countEval* is an aspect that counts the number of divisions; *errorEval* is an aspect that handles divisions by 0 errors; and, finally, the *traceEval* mixin provides some basic tracing functionality. We can combine all the functionality through mixin composition as follows:

```
fullEval :: Term → Either String ((Int, Int), String)
fullEval = unwrap ∘ mix where
  mix = mixin (traceEval ⊕ countEval ⊕
               errorEval ⊕ meval)
```

```
meval :: Monad m ⇒ Mixin (Term → m Int)
meval continue call t = case t of
  Con a   → return a
  Div t u → do x ← call t
               y ← call u
               return (x `div` y)

countEval :: MonadState Int m ⇒ Mixin (Term → m Int)
countEval continue call t = case t of
  Div _ _ → do r ← continue t
               n ← get
               put (n + 1)
               return r
  _       → continue t

errorEval :: MonadError String m ⇒ Mixin (Term → m Int)
errorEval continue call e = case e of
  Div t u →        -- override
    do x ← call t
       y ← call u
       if y ≡ 0 then throwError "divide by zero"
                else return (x `div` y)
  _       → continue e

traceEval :: MonadWriter String m ⇒ Mixin (Term → m Int)
traceEval continue call e =
  do r ← continue e
     tell (line e r)
     return r

line :: (Show a, Show b) ⇒ a → b → String
line t a = "eval (" ++ show t ++ ") <= " ++ show a ++ "\n"
```

Figure 12. Modular mixins for Wadler's interpreters.

```
unwrap = runIdentity ∘ runErrorT ∘
         runWriterT ∘ flip runStateT 0
```

Of course, we could as well combine just some of the mixins or compose them in a different order (which would give us a different program).

It is nice that all the functionality is separated and can be reused independently, but there is still some room for improvement: some of the functionality provided by the aspects is not inherently tied to an evaluator. This is the case for the *counting* and *tracing aspects*, which could be useful in different contexts.

7.2 Crosscutting Aspects with Advices and Pointcuts

The *countEval* mixin is tightly bound to term expressions because we are interested in counting only the number of divisions. Yet, there is nothing inherently dependent on expression terms in a mixin that executes one step of the computation and increments one counter by one. For example, the following mixin

```
count₁ :: MonadState Int m ⇒ Mixin (a → m b)
count₁ continue call x = do r ← continue x
                            n ← get
                            put (n + 1)
                            return r
```

can be used to count steps on any values on some type a. It would be nicer if somehow the counting functionality would be separated from the functionality specific to the evaluator.

In Figure 13, inspired by ideas from AOP, we show a small library of combinators that can be used to provide "*advice*" to existing mixins. The *advises* combinator is used to advise a set of pointcuts with a given mixin. The set of pointcuts is essentially represented as a predicate, of which the *any* and *none* predicates

```
advises :: Mixin (a → b) → (a → Bool) → Mixin (a → b)
advises m p continue call x
  | p x       = m continue call x
  | otherwise = continue x

any :: a → Bool
any = const True

none :: a → Bool
none = const False

(∪) :: (a → Bool) → (a → Bool) → a → Bool
f ∪ g = λx → f x ∨ g x
```

Figure 13. Mixin advice and pointcuts.

represent, respectively, the set of all pointcuts and the empty set; and the ∪ combinator represents the union of two sets of pointcuts. These combinators have a rich set of algebraic properties:

```
m `advises` any = m
m `advises` none = zero
any ∪ p = any = p ∪ any
none ∪ p = p = p ∪ none
p₁ ∪ p₂ = p₂ ∪ p₁
(p₁ ∪ p₂) ∪ p₃ = p₁ ∪ (p₂ ∪ p₃)
```

which have fairly intuitive interpretations. The first property says that if a mixin m advises any pointcut then that is equivalent to m. The second property says that a mixin m advising no pointcuts is equivalent to the *zero* mixin. The third and fourth properties mean that *any* and *none* are, respectively, the zero and neutral elements of ∪. Finally, the last two properties are, respectively, the commutativity and associativity of ∪. Two examples of pointcuts are given by:

```
division :: Term → Bool
division (Div _ _) = True
division _         = False

con :: Term → Bool
con (Con _)        = True
con _              = False
```

The *division* pointcut can be used when we want to trigger a piece of advice on the division constructor; similarly we can use the *con* pointcut to trigger advice on integer terms. For example, to recover the functionality provided by *countEval* we could define the following mixin:

```
divCount₁ :: MonadState Int m ⇒ Mixin (Term → m a)
divCount₁ = count₁ `advises` division
```

The advantage of $divCount_1$ over *countEval* is that the counting mixin functionality is no longer *entangled* with the evaluator code for term expressions. This means that we could use the counting mixin, for example, to create a mixin to count the number of arithmetic operations in the evaluator in Section 5:

```
plus :: Expr → Bool
plus (Plus _ _)    = True
plus _             = False

minus :: Expr → Bool
minus (Minus _ _)  = True
minus _            = False

opCount :: MonadState Int m ⇒ Mixin (Expr → m a)
opCount = count₁ `advises` (plus ∪ minus)
```

Of course, we could also use $count_1$ in programs that are not evaluators. A potential use of the counting mixin would be for spotting functions that are good candidates to memoization. The following function:

```
before :: Monad m ⇒ (b → m a) → Mixin (b → m c)
before mf continue call x = do {mf x; continue x}
after :: Monad m ⇒ (b → c → m a) → Mixin (b → m c)
after mf continue call x =
  do {r ← continue x; mf x r; return r}
```

Figure 14. Monadic advices

```
countFib :: Int → (Int, Int)
countFib = unwrap ∘ mixin (count1 `advises` (<4) ⊕ fib)
  where unwrap = flip runState 0
```

counts all calls to the (naive) fibonacci function that are less than 4. If we run *countFib* a few times with some values we can get some (experimental) evidence of whether or not computation is being repeated. For example, the result of *countFib* 8 would be $(21, 55)$ and the result for *countFib* 15 would be $(610, 1597)$. The first element in the outputed pair is the fibonacci value and the second is the number of recursive calls to input values less than 4. The results given by *countFib* (together with some intuitive understanding of how the fibonacci function works) suggest that computation may be repeated and that the function could benefit from memoization.

7.3 Monadic Advices

For monadic mixins, there are two more combinators in Figure 14 that are quite useful. The *before* and *after* combinators (named after the well-known AOP notions of before and after advices) apply a monadic computation respectively before and after a *continue* call. The argument to *continue* is available to *before*; while both the argument to and the result of calling *continue* are at the disposal of *after*. With *before* and *after* we can create even more fine-grained reusable functionality that does not commit to when *continue* is called. For example, we could define a computation $count_2$ that does not itself call *continue* and does not require the use of mixins.

```
count2 :: MonadState Int m ⇒ a → m ()
count2 _ = do {n ← get; put (n + 1)}
divCount2 :: MonadState Int m ⇒ Mixin (Term → m a)
divCount2 = before count2 `advises` division
```

With *before* we could create some very simple tracing facilities using the primitive function $print :: a \to IO\ ()$

```
condTrace :: Show a ⇒ (a → Bool) → Mixin (a → IO b)
condTrace p = before print `advises` p
trace :: Show a ⇒ Mixin (a → IO b)
trace = condTrace any
```

The *condTrace* is a conditional tracing function that takes a set of pointcuts (or a predicate) and traces all calls that match those pointcuts; the *trace* mixin unconditionally applies the tracing functionality to any pointcut. With *condTrace* we could trace all the calls of the evaluator on divisions using the following program:

```
traceDivEval :: Term → IO Int
traceDivEval = mixin (condTrace division ⊕ meval)
```

With the *after* combinator we could, not only print the arguments, but also the results of the calls:

```
fullTrace :: (Show a, Show b, MonadWriter String m) ⇒
             (a → Bool) → Mixin (a → m b)
fullTrace p = after debug `advises` p where
  debug arg res = tell (
    "Argument : " ++ show arg ++
    "\nResult   : " ++ show res ++ "\n\n")
```

For *fullTrace* we use the writer monad transformer, which we can compose with other monads. With *fullTrace* we could do some conditional tracing on the evaluator from Figure 7. The following program:

```
traceBEval :: Expr → (Int, String)
traceBEval = unwrap ∘ mixin (fullTrace points ⊕ beval)
  where points  = ¬ ∘ (plus ∪ minus)
        unwrap = fst ∘ flip runState [] ∘ runWriterT
```

takes an expression and returns the result of evaluating that expression together with a trace created by *fullTrace*. We make use of pointcuts to ensure that all calls are traced except those made to the arithmetic operators. Note that ¬ is the following function:

```
¬ :: Bool → Bool
¬ True  = False
¬ False = True
```

8. Discussion and Related Work

8.1 Mixins and Inheritance in Functional Programming

Many authors before us [Cook, 1989, McAdam, 1997, Garrigue, 2000, Läufer, 2003, Brown and Cook, 2007] argued about the uses of inheritance in functional programming, employing similar techniques to ours. Unfortunately their work seems to have been largely under-appreciated by the functional programming community. We can think of two reasons for this. Firstly, some of these works [McAdam, 1997, Garrigue, 2000, Läufer, 2003] tend to employ open recursion rather directly to programs, without using mixin abstractions like the ones introduced by us—such as, for example, the *Mixin* type and mixin composition. Without this "*sugar*" the use of open recursion can lead to relatively complex types and definitions and render the techniques impractical. Secondly, these techniques have being mostly employed in impure functional languages like ML and OCaml where side-effects are implicit and monads are not used. While inheritance is certainly valuable in those languages, their need is a bit less pressing than in pure languages because a local change to a program introducing a side-effect is unlikely to change the type of the function. With monads, this is usually not the case and a small local change is likely to break existing code. Therefore, modularization (in purely functional languages) is not only useful for reuse but also for preserving backwards compatibility.

Several other applications of mixins and inheritance to functional programming are discussed in the literature. McAdam [1997] shows how some effects can be simulated (without using monads) using mixins and he presents a type-inference algorithm where the treatment of error messages is modularly defined. Garrigue [2000] employs open recursion to emulate *open functions* in his solution to the expression problem with *polymorphic variants*. Läufer [2003] shows how to apply mixins to interpreters and how to define mutually-recursive functions using mixins. He also argues about the relation of his technique with the OO VISITOR pattern [Gamma et al., 1995]. The "*Scrap your Boilerplate*"-like techniques [Lämmel and Peyton Jones, 2003, Bringert and Ranta, 2008] that appeared recently in the literature fundamentally rely on ad-hoc approaches to inheritance to reuse traversal code.

Brown and Cook [2007] show how to approach the problem of memoization in purely functional languages using *monadic memoization mixins* using a technique that is very close to what we use in this paper. Their technique is more restricted in the sense that only one type of calls is permitted in the mixin programs. Typically, in the base program that call plays the role of the *self* (or *call*) argument, while in the wrapper programs it plays the role of *super* (or *continue*). In our approach both types of calls are available to any mixin, which allows additional flexibility (for example, the use of *call* in a wrapper program, allows us to *override* existing functionality). Of course, Brown and Cook were interested in solving a problem in the much more restricted domain of memoization, while we are focused on the more general problem of modularizing

programs in an AOP-style. We believe that for memoization Brown and Cook's technique is enough.

8.2 Aspect-Oriented Programming

Kiczales et al. [1997]'s aspect-oriented programming aims at modularizing concerns that cut across the components of a software system. In AOP, programmers are able to modularize these crosscutting concerns with locally defined aspects. Typical implementations of AOP (such as, for example, AspectJ [Eclipse-Foundation, 2000-2009]) use *pointcuts* to designate when and where to crosscut using the names of *classes*, *methods*, *modules* or any other entity containing code; *advices* specify what will happen when a pointcut is reached.

The notions of pointcuts and advices inspired the combinators presented in Section 7. However our combinators differ significantly from pointcuts and advices found in most AOP languages in respect to their expressive power, implementation and reasoning properties. Because traditional pointcuts typically refer to *syntactical* entities showing up in a program, we need either a meta-language (that can refer and manipulate the elements that show up in a program); or some form of runtime reflexive capabilities (that retains the meta-information and allows manipulation of programs at runtime). Our notion of pointcuts is based on predicates on the inputs of programs. Advising functions (or other entities) based on their names is simply not possible with our approach. However, as we have seen, we can easily emulate pointcuts on *data constructors*, by creating a simple predicate that tests whether or not that is the constructor of interest. We believe that for functional programs this already provides some significant expressiveness, since data constructors in functional languages play a similar role to methods in object-oriented languages. The advantages of our combinators are that they are *first-class* entities of the language; and they do not endanger modular program understanding and reasoning, which is a significant challenge in other implementations of AOP and a current hot research topic [Aldrich, 2005, Kiczales and Mezini, 2005]. Furthermore, our approach does not rely on any kind of code or bytecode weaving (which is common in many AOP technologies) supporting full separate compilation.

8.3 Aspect Oriented Programming and Monads

The connection between AOP and monads is a recurring theme of discussion since De Meuter [1997] argued about the use of monads as a theoretical foundation for AOP. Monads are clearly a modularization mechanism, but the question is whether they are a modularization mechanism aimed at the separation of cross-cutting concerns. In some sense it can be argued that this is indeed the case as the plumbing necessary by the particular effects can be captured by a monad instance and reused by different programs. However, the plumbing required for particular effects is just a very restricted case of cross-cutting concerns. Hofer and Ostermann [2007] argued recently that "*monads and aspects have to be regarded as quite different mechanisms*" and we have to agree with their conclusion: all the programs in this paper could be modularly implemented in essentially the same way without monads (but still using mixins) in an impure functional language like ML, OCaml or Scala using implicit side-effects. So, the additional modularization of the programs presented in this paper is due to mixins and not monads. In fact, this should not be too surprising in the first place as the main motivation for using monads [Wadler, 1992] was to essentially recover the *same* modularity properties of impure languages (see also Section 2); and typical AOP implementations are aimed at *improving* the modularity of these same impure languages.

Modularity benefits of monads It is generally uncontested that for purely functional languages there is a lot to be gained from using monads, because of their ability to emulate side-effects and structure code. However, an interesting question is whether there are some additional modularity benefits in monads even for impure languages. One potential benefit (pointed out by Wadler [1992]) is that monads are not just restricted to effects that show up in common impure languages like *state*, or *exceptions*. Monads can model other effects like *nondeterminism* or *continuations*, which are not usually natively found in impure languages. Another benefit arises from the use of monad transformers. Programs written with monad transformers are usually more general than similar impure programs because the order on which effects are applied can be controlled by choosing different instantiations of the monad transformer. The choice of different monad transformers can lead to programs with different behaviours, and different types too (see the $eval_4$ and $eval_5$ examples at the end of Section 3.2). When implicit effects are used, the order in which effects are applied is fixed and cannot be controlled.

8.4 Beyond Monads

Although monads are (by far) the most popular abstraction to model effects and structure programs, they are certainly not the only one and there are many other useful abstractions. *Arrows* have been introduced by Hughes [2000] as a generalization of monads that can be useful in many libraries that have interfaces which are fundamentally incompatible with monads. Somewhere in between the notions of arrows and monads stands the notion of *applicative functors* [McBride and Paterson, 2008], which provide a simpler interface than arrows but can still be used in many applications where monads are not usable. *Parametrized monads* and *parametrized Freyd categories* [Atkey, 2006] are two of the latest abstractions proposed to capture computational effects, and they can be used to model some effects that do not quite fit the (unparametrized) versions of the corresponding abstractions.

In this paper we have choosen to just use monads in combination with mixins, since this is the most familiar abstraction to programmers and there is a lot of useful infrastructure already implemented in the Haskell hierarchical libraries. In particular, all the machinery required by monad transformers is already implemented and can be readily used. However, there is nothing fundamental stopping us from using mixins with other kinds of effects. In fact there may be some important benefits in doing so. For example, monads do not to compose well, although monad transformers help significantly in this respect. An interesting alternative that can (sometimes) be used instead of monads and monad transformers are applicative functors, which can always be composed [McBride and Paterson, 2008, Gibbons and Oliveira, 2006]. Also, it is difficult to use two distinct monad transformers of the same kind in the same program (for example, we may want to keep different logs for tracing and watching variables); with other technologies this may be easier to achieve.

9. Conclusions

Monads are widely used in Haskell to model computational effects and structure purely functional programs. Inheritance is a crucial element of all modern object-oriented languages. Both mechanisms have been very successful at modularizing and reusing different aspects of programs, yet their combination has been largely understudied. This paper shows that the combination of the two mechanisms yields substantial modularity and reuse advantages, and can be used to develop programs in a simple and elegant AOP style without compromising any of the beloved properties of pure functional programming.

We hope that this paper has convinced the reader of the uses of inheritance for functional programming. It seems clear to us that there are substantial advantages in the adoption of techniques that exploit inheritance to modularize and reuse functional programs. Moreover, the mixin approach introduced in this paper seems to be

a perfect example of *pure lazy functional programming*: it makes *extensive use of higher-order functions*; it employs an *elegant combinator style with many interesting algebraic properties*; and it is *implemented in a very simple way under a lazy semantics*.

10. Acknowledgements

William Cook explained to us the connection between the mixin composition operation presented in this paper and his own mixin operators. Jeremy Gibbons commented an early version of this paper. We are grateful to both of them.

References

Jonathan Aldrich. Open modules: Modular reasoning about advice. In *LNCS 3586: European Conference on Object-Oriented Programming*, pages 144–168, 2005.

Robert Atkey. Parameterized notions of computation. In *MSFP 2006*, July 2006.

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. *Journal of Functional Programming*, 18 (Special Double Issue 5-6):567–598, 2008.

Daniel Brown and William R. Cook. Monadic memoization mixins. Technical Report TR-07-11, The University of Texas, February 2007.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system f with subtyping. In *Information and Computation*, pages 750–770. Springer-Verlag, 1994.

William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

Wolfgang De Meuter. Monads as a theoretical foundation for aop. In *International Workshop on Aspect-Oriented Programming at ECOOP.*, 1997.

Eclipse-Foundation. Aspectj, 2000-2009. See `http://eclipse.org/aspectj/`.

Martin Erwig and Deling Ren. Monadification of functional programs. *Science of Computer Programming*, 52(1-3):101–129, 2004.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering, Sasaguri, Japan*, 2000.

Jeremy Gibbons and Bruno Oliveira. The essence of the Iterator pattern. In Tarmo Uustalu and Conor McBride, editors, *Mathematically-Structured Functional Programming*, volume 4014 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2006.

Christian Hofer and Klaus Ostermann. On the relation of aspects and monads. In *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, pages 27–33, New York, NY, USA, 2007. ACM.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.

Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ACM: International Conference on Software engineering*, pages 49–58, 2005.

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *LNCS 1241: European Conference on Object-Oriented Programming*, pages 220–242, 1997.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

Konstantin Läufer. What functional programmers can learn from the Visitor pattern. Technical report, Loyola University Chicago, March 2003. URL `http://www.cs.luc.edu/~laufer/papers/mixins03.pdf`.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

Bruce J. McAdam. That about wraps it up — using fix to handle errors without exceptions, and other programming tricks. Technical report, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1997.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 2008.

Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS '89, Proceedings., Fourth Annual Symposium on*, pages 14–23, 1989.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, New York, NY, USA, 1992. ACM.

Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.