Arrows for Errors: extending the error monad

Duncan Coutts
duncan@coutts.uklinux.net

August 2002

Abstract

This paper considers an application of arrows. We extend the normal error monad with an extra arrow-style combinator to enable multiple errors from independent computations to be collected. We look at the data-independence property of the extra combinator which allows the extended capabilities. We also demonstrate an applicative notation and give examples of structuring computations to take advantage of the additional combinator.

Background

The reader will be familiar with the use of monadic combinators for structuring functional programs. A common example is the error monad for structuring programs that can raise errors.

```
data ErrorMonad a = Fail String
                  | Return a

bind :: ErrorMonad a -> (a -> ErrorMonad b) -> ErrorMonad b

(Return x) `bind` f = f x
(Fail e)   `bind` f = Fail e

instance Monad ErrorMonad a where
  return = Return
  fail = Fail
  (>>=) = bind
```

Motivation

This standard formulation is certainly useful. We could extend it with a more complex error state and a try/catch function to create a scheme resembling the error handling of a imperative language like C++ or Java.

This kind of error handling is fine when errors are exceptional but is somewhat limited if error handling is intended to be part of the normal execution of the program, as for example in a compiler.

The prime limitation is that we cannot return more than a single error. In a compiler for example we want to report more than the first error, if possible. However, once we can produce more than a single error we have the secondary problem of not producing too many errors. We are all familiar

with compilers that report the first error then get confused and spew out reams of spurious error messages. To abuse a famous quotation: a compiler should detect as many errors as possible but no more.

The key issue is that of dependency. We do not want to report an error that is dependent upon another error, since these would often be false hits.

In a linear algorithm this gains us nothing since all errors raised after the first are dependent upon that first error. However in branching algorithms, such as a depth first traversal over a tree structure, we could raise errors in different branches confident that the errors are independent. So in our compiler example we would like to report errors in independent subtrees of the abstract syntax tree.

What we want is a function for combining two computations that may raise errors. The result should be a computation that can raise errors from both component computations if both computations fail.

The type signature would be:

```
combine :: (a -> b -> c) -> ErrorsMonad a -> ErrorsMonad b -> ErrorsMonad c
```

The problem

It turns out that the desired behaviour is impossible to implement with the monadic bind operator. Intuitively the problem is that the way of combining monadic values is fundamentally sequential, which is why we can use it for IO. There is no way of executing both actions and inspecting the result without executing one first. In this case if the first raises an error, the second will not be executed and so we cannot return both errors.

The function is easy to implement if we allow ourselves direct access to the representation. Note of course that we have changed the representation so it can now hold multiple errors, ie the error state is now a list of error messages.

```
data ErrorsMonad a = Fail [String]
                   | Return a

combine f (Return x) (Return y) = Return (f x y)
combine f (Return x) (Fail  ey) = Fail ey
combine f (Fail  ex) (Return y) = Fail ex
combine f (Fail  ex) (Fail  ey) = Fail (ex ++ ey)
```

This works but is not very satisfactory since it only allows us to combine two actions and it feels insufficiently abstract.

New combinator

We would like to be able to chain actions in a similar way that the monadic bind operator allows:

```
actionA >>= \outputA ->
actionB >>= \outputB ->
actionC
```

We can arrive at such an operator by considering the function for combining two actions, one for three actions and then generalising. The resulting function has the following type:

```
combine :: ErrorsMonad (a -> b) -> ErrorsMonad a -> ErrorsMonad b
```

A more abstract presentation can be obtained if we create a new class of which our error monad will be an instance.

```
class Monad m => CombineMonad m where
  (<&>) :: m (a -> b) -> m a -> m b

infixl 1 <&>
```

Implementing it as an extension to the monad class, rather than as an independent class, allows us to reuse the `return` function for lifting values to actions. As we will see it is useful for an error handling scheme to have both combinators.

Now with this formulation and the appropriate operator fixity we can chain several actions together

```
return triple <&> actionA <&> actionB <&> actionC
```

and by adding an extra derived operator

```
infixl 1 &>

f &> x = return f <&> x
```

we achieve a notation that reminds us of function application.

```
f &> x <&> y <&> z
```

In fact it behaves very much like function application, but with the interpretation that the arguments are evaluated in parallel.

We can now implement our new error monad. The monad functions are unchanged.

```
(Return x) `bind` f = f x
(Fail e)   `bind` f = Fail e

instance Monad ErrorsMonad a where
  return = Return
  fail = Fail . (\e -> [e])
  (>>=) = bind

instance CombineMonad ErrorsMonad where
  Return f <&> Return x = Return (f x)
  Fail  ef <&> Return x = Fail ef
  Return f <&> Fail  ex = Fail ex
  Fail  ef <&> Fail  ex = Fail (ef ++ ex)
```

Data dependence

Notice the similarity of the two operators:

```
(<&>) :: CombineMonad m => m (a -> b) -> m a -> m b
(=<<) ::        Monad m => (a -> m b) -> m a -> m b

where (=<<) = reverse (>>=)
```

Is it obvious why one is data dependent and the other is not? Here is an interpretation: with the combine operator the two parameters are actions, both can be executed immediately. For the bind operator, the first argument is a function returning an action. So we cannot obtain the action without supplying a value of type 'a'. We cannot get a value of type 'a' without executing the 'm a' action. So the bind operator requires an order of execution. With the combine operator on the other hand, since both arguments are actions we cannot do anything with them except execute them. The first action returns a function which can be supplied with the result of the second action, but the action itself cannot depend on the result of the second action. This is the crucial data dependence property.

An analogy is of function evaluation in pure and impure languages. In Java for example function parameters are evaluated in left to right order and they can interact by modifying some shared state. In a referentially transparent language such as Haskell, parameters are completely independent and could be evaluated in any order[1] or indeed in parallel. Another analogy is with with sequencing and parallel operators in a parallel imperative language. The monad bind operator corresponds to '$A; B$' and our new combinator corresponds to '$A \parallel B$'.

It is possible to implement the type signature of our new operator using just the monad bind operator in fact this is exactly what the '`ap`' function from the standard library does. But as we noted earlier it will always serialise the actions. We can see in the following implementation that essentially it is irrelevant in which order the '`f`' and the '`x`' actions are performed as they do not depend on each other[2].

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap f x = do
           f' <- f
           x' <- x
           return (f' x')
```

Derived combinators

We can define derived functions similar to the monad's `(>>)`, `sequence`, `mapM` and `zipWithM` functions.

```
(<&>>) :: CombineMonad m => m a -> m b -> m b
x <&>> y = (\x y -> y) &> x <&> y

combine :: CombineMonad m => [m a] -> m [a]
combine = foldr (\x xs -> (:) &> x <&> xs) (return [])
```

[1] as long as the function is strict

[2] in general the order in which the actions are performed may be observable depending upon how the monad's state is combined by the bind operator. It does not matter in our errors monad example if we consider our collection of errors to be unordered.

```
combine_ :: CombineMonad m => [m a] -> m ()
combine_ = foldr (<&>>) (return ())

mapCM :: CombineMonad m => (a -> m b) -> [a] -> m [b]
mapCM f = combine . map f

mapCM_ :: CombineMonad m => (a -> m b) -> [a] -> m ()
mapCM_ f = combine_ . map f

zipWithCM :: CombineMonad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithCM f xs ys = combine (zipWith f xs ys)
```

Each of these functions can be used in a similar way to their monadic counterparts but with the extra interpretation that all the actions take place in parallel. In the case of our error monad this means that errors from all failing actions are reported.

Notice that we cannot find a counterpart to `foldM` since this threads a state through the actions creating a linear data dependency between the actions. We cannot do any better than `foldM` in terms of parallel error reporting unless we resort to ugly hacks which I will not expand upon here.

Using a combination of the two flavours of operators we can write algorithms which sometimes use branching and sometimes serialise their error handling.

Examples

In a compiler front end we make several passes over an abstract syntax tree. On the assumption that errors detected in independent subtrees are independent we can use our extended error monad. Heres is an example from the semantic checking phase of a compiler for a traditional imperative language.

```
checkExprs :: Env -> [Expr] -> ErrorsMonad [Expr]
checkExprs env = mapCM (checkExpr env)

checkExpr :: Env -> Expr -> ErrorsMonad Expr
checkExpr env = ... checkFunc ...

checkFunc :: Env -> Ident -> [Expr] -> ErrorsMonad Expr
checkFunc env fname exprs
    = checkFuncDeclared env id
   >> checkFuncNumArgs env id (length exprs)
 <&>> (ExprFunc fname &> checkExprs env exprs)
```

Notice that we mix the two forms of combinator to match the dependencies of the different classes of checks. Here we do not bother to check if the function has been called with the right number of arguments if the function has not even been declared. If it has been declared however, we check both for the correct number of arguments and the validity of the arguments themselves.

The next example is similar in that we are mixing the two forms. In this case we use a tuple to do two checks at once. This demonstrates how to insert a parallel check into an existing algorithm written using the 'do' monad syntax.

```
checkBranch :: Env -> Expr -> Stmts -> ErrorsMonad (Expr, Stmts)
checkBranch env expr stmts =
  do
    (expr', stmts') <- pair
                         &> checkExpr env expr
                        <&> checkStmts env stmts
    requireTypeBool expr'
    return (expr', stmts')
```

Further work

It would be good to discover or decide what laws this operator should obey. Some laws will be inherited from the fact that it is an arrow.

I conjecture that it is possible to prove the data independance property directly from the type signature of the operator, in particular without knowing the implementation. That is for all implementations of the `<&>` class method the data independance property holds.

Further reading

This form of combinator was used first by S. Swierstra and L. Duponcheel[SD96] in their parser combinator library. They made use of the data independence property to statically analyse and optimise the parser. This would not have been possible with a monadic formulation since the data dependence means that the control flow is not known until the parser is invoked.

D. Leijen and E. Meijer[LM01] point out an advantage of monadic parser combinators, which is that they can parse context sensitive grammars, whereas the arrow style parser combinators are restricted to context free grammars. Again, this is a direct consequence of the data dependence property. This is the flip side to the advantage of static analysis.

It was this form of combinator, as used by Swierstra and Duponcheel which inspired John Hughes[Hug00] to seek the generalisation he termed arrows.

References

[Hug00] J. Hughes, Generalising Monads to Arrows, 2000

[LM01] D. Leijen and E. Meijer, Parsec: Direct Style Monadic Parser Combinators For The Real World (draft), 2001

[SD96] S. Swierstra and L. Duponcheel, Deterministic, Error Correcting Combinator Parsers, 1996