

Arrows for Errors: extending the error monad

Duncan Coutts

AFP, August 2002

Background

- Error handling is often implemented using an error monad

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

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

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

Problem

- Would like to return more than one error
- Don't want to return too many errors!

“A compiler should detect as many errors as possible but no more”

- We only want independent errors

Usually we do not want errors that are dependent on another error since they may be “false hits”

What We Want

- To report independent errors

We would like to try two actions and get both errors if both fail. The type signature might be:

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

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

However it is impossible to implement this behavior using a monad, ie using only bind ($>>=$)

What We Want (cont...)

- Easy to implement with direct access to the representation

```
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 is still unsatisfactory

- Can only combine two actions
- Not abstract enough

A New Combinator

- We would like to be able to chain actions like we can with `bind (>>=)`

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

- We can discover the right form by considering functions for combining two or three actions and then generalising.

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

A New Class

- A more abstract view:

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

- This type allows actions to be chained
- We will use an infix notation (presented shortly)

Implementation

- Monad class methods are not changed, except for a trivial change due to the new error state

```
fail = Fail . (\e -> [e])
```

- New operator implementation:

```
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)
```

Functional Notation

- If we make the `<&>` operator left associative we can chain actions conveniently

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

- An extra operator gives us a notation that reminds us of normal function application

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

Example:

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

Comparison

- Consider the types of the different 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 operator has a data dependency between its operands and the other does not?

Comparison (cont...)

$$\begin{array}{l} (\langle \& \rangle) \quad :: \quad \text{CombineMonad } m \Rightarrow m \overbrace{(a \rightarrow b)}^1 \rightarrow m a \rightarrow m b \\ (= \langle \langle) \quad :: \quad \text{Monad } m \Rightarrow \underbrace{(a \rightarrow m b)}_2 \rightarrow m a \rightarrow m b \end{array}$$

1. is an action. We cannot give it any information before executing the action.
2. is a function. We must give it more information before obtaining an action.

Comparison (cont...)

A couple of analogies to give us an intuition

- Argument evaluation which allows or disallows interaction through side effects

eg Java “ $f(g(), x)$ ” vs. Haskell “ $f\ g\ x$ ”

In Java, evaluating g may mutate x . In Haskell (if f is strict) we can evaluate g and x in any order or in parallel.

- Sequential or parallel composition in an imperative language

Monadic composition is sequential composition “ $A; B$ ”. Our new operator is more like parallel composition “ $A \parallel B$ ”.

Digression

It is possible to implement the type of our new operator using `bind`. This is exactly what the library function `ap` does:

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

Note that `f` and `x` are independent, so the order in which they are done is (mostly) irrelevant. As we noted earlier however, the actions are (un)necessarily serialised.

Derived Functions

We can define equivalents to the monadic functions `sequence`, `mapM` and `zipWithM` but not `foldM`. Why not?

```
combine :: CombineMonad m => [m a] -> m [a]
combine = foldr (liftCM2 (:)) (return [])
```

```
mapCM :: CombineMonad m => (a -> m b) -> [a] -> m [b]
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)
```

Example

```
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)
```

Example (cont...)

We mix the two styles to control what gets checked in parallel.

```
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')
```

What's Next?

- A better name!

- Laws

Which arrow laws apply here?

- Conjecture about type proof

Is it possible to prove the data independence property from the type signature without knowing the implementation?

- parallel IO?