Dependently Typed Functional Programming with Idris

Lecture 3: Effect Management

Edwin Brady
University of St Andrews

ecb10@st-andrews.ac.uk
@edwinbrady

An Effectful Problem (Haskell)

Evaluator

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

An Effectful Problem (Haskell)

Evaluator

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

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

An Effectful Problem (Haskell)

Evaluator with variables

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

An Effectful Problem (Haskell)

Evaluator with variables

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

type Env = [(String, Int)]
```

An Effectful Problem (Haskell)

Evaluator with variables

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

type Env = [(String, Int)]

eval :: Expr -> ReaderT Env Maybe Int
```

An Effectful Problem (Haskell)

Evaluator with variables

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

type Env = [(String, Int)]

eval :: Expr -> ReaderT Env Maybe Int
eval (Val n)    = return n
eval (Add x y) = liftM2 (+) (eval x) (eval y)
eval (Var x)    = do env <- ask
                     val <- lift (lookup x env)
                     return val
```

An Effectful Problem (Haskell)

Evaluator with variables and random numbers

```
data Expr = Val Int | Add Expr Expr
          | Var String
          | Random Int
```

An Effectful Problem (Haskell)

Evaluator with variables and random numbers

```
data Expr = Val Int | Add Expr Expr
          | Var String
          | Random Int

eval :: RandomGen g =>
        Expr -> RandT g (ReaderT Env Maybe) Int
```

An Effectful Problem (Haskell)

Evaluator with variables and random numbers

```
data Expr = Val Int | Add Expr Expr
          | Var String
          | Random Int

eval :: RandomGen g =>
        Expr -> RandT g (ReaderT Env Maybe) Int
...
eval (Var x) = do env <- lift ask
                  val <- lift (lift (lookup x env))
                  return val
eval (Random x) = do val <- getRandomR (0, x)
                     return val
```

An Effectful Problem (Haskell)

Challenge — write the following:

```
dropReader :: RandomGen g =>
              RandT g Maybe a ->
              RandT g (ReaderT Env Maybe) a
```

```
commute :: RandomGen g =>
           ReaderT (RandT g Maybe) a ->
           RandT g (ReaderT Env Maybe) a
```

An Effectful Problem (Idris)

Instead, we could capture everything in one evaluation monad:

Eval monad

```
EvalState : Type
EvalState = (Int, List (String, Int))

data Eval a
   = MkEval (EvalState -> Maybe (a, EvalState))
```

An Effectful Problem (Idris)

Instead, we could capture everything in one evaluation monad:

Eval monad

```
EvalState : Type
EvalState = (Int, List (String, Int))

data Eval a
   = MkEval (EvalState -> Maybe (a, EvalState))
```

We make `Eval` an instance of `Monad` (for do notation) and `Applicative` (for idiom brackets)

An Effectful Problem (Idris)

Eval operations

```
rndInt : Int -> Int -> Eval Int
get    : Eval EvalState
put    : EvalState -> Eval ()
```

An Effectful Problem (Idris)

Evaluator

```
eval : Expr -> Eval Int
eval (Val i) = return i
eval (Var x) = do (seed, env) <- get
                  lift (lookup x env)
eval (Add x y) = [| eval x + eval y |]
eval (Random upper) = do val <- rndInt 0 upper
                         return val
```

Embedded DSLs to the rescue!

Neither solution is satisfying!

- Composing monads with transformers becomes hard to manage
  - Order matters, but our effects are largely independent
- Building one special purpose monad limits reuse

Instead:

- We will build an *extensible* embedded domain specific language (EDSL) to capture *algebraic effects*.

The `Effect` EDSL

The rest of this lecture is about an EDSL, `Effect`. It is in three parts:

- How to *use* effects
- How to *implement* new effects
- How `Effect` works

Using Effects

Effectful programs

```
EffM : (m : Type -> Type) ->
       List EFF -> List EFF -> Type -> Type
Eff  : (Type -> Type) -> List EFF -> Type -> Type

run     : Applicative m =>
          Env m xs -> EffM m xs xs' a -> m a
runPure : Env id xs -> EffM id xs xs' a -> a
```

Using Effects

Some Effects

```
STATE      : Type -> EFF
EXCEPTION  : Type -> EFF
STDIO      : EFF
FILEIO     : Type -> EFF
RND        : EFF
```

Using Effects

Some Effects

```
STATE     : Type -> EFF
EXCEPTION : Type -> EFF
STDIO     : EFF
FILEIO    : Type -> EFF
RND       : EFF
```

Examples

```
get : Eff m [STATE x] x
putM : y -> EffM m [STATE x] [STATE y] ()

raise : a -> Eff m [EXCEPTION a] b

putStr : String -> Eff IO [STDIO] ()
```

sicsa*

Demonstration: Effects

You will need to include the `effects` package:

```
idris -p effects
```