Embedded Domain Specific Languages in Idris
Lecture 3: State, Side Effects and Resources

Edwin Brady (ecb10@st-andrews.ac.uk)
University of St Andrews, Scotland, UK
@edwinbrady

SSGEP, Oxford, 9th July 2015

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

Effects in IDRIS

Effectful programs

```
data EffM : (m : Type -> Type) -> (res : Type) ->
            (in_effects : List EFFECT) ->
            (out_effects : res -> List EFFECT) ->
            Type
```

Effects in Idris

Effectful programs

```
data EffM : (m : Type -> Type) -> (res : Type) ->
            (in_effects : List EFFECT) ->
            (out_effects : res -> List EFFECT) ->
            Type
```

Composing programs

```
(>>=) : EffM m res es es' ->
        ((x : res) -> EffM m b (es' x) es'') ->
        EffM m b es es''
```

Examples

```
get   :        EffM m t  [STATE t] (\x : t  => [STATE t])
put   : t -> EffM m () [STATE t] (\x : () => [STATE t])

putM : t' -> EffM m () [STATE t]
                                   (\x : () => [STATE t'])
```

Examples

```
get   :        Eff t [STATE t]
put   : t -> Eff () [STATE t]

putM : t' -> Eff () [STATE t] [STATE t']
```

Examples

Combining effects

```
inc : Eff () [STDIO, STATE Nat]
inc = do x <- get
         putStrLn ("Old value " ++ show x)
         put (x + 1)
```

Examples

Labelling effects

```
sumStates : Eff Nat ['xval ::: STATE Nat,
                     'yval ::: STATE Nat]
sumStates = do x <- 'xval :- get
               y <- 'yval :- get
               return (x + y)
```

Running Effectful Programs

```
run : Applicative m => {auto env : Env m xs} ->
      (prog : EffM m a xs xs') -> m a
runPure : {auto env : Env id xs} ->
          (prog : Eff id a xs xs') -> a
```

Idris

Demonstration: An Effectful Evaluator

Effect Signatures

```
Effect : Type
Effect = (t : Type) -> (res : Type) ->
                       (res' : t -> Type) -> Type
```

```
data State : Effect where
     Get :      State a  a (const a)
     Put : b -> State () a (const b)
```

```
STATE : Type -> EFFECT
STATE t = MkEff t State
```

Effect Signatures

```
Effect : Type
Effect = (t : Type) -> (res : Type) ->
                       (res' : t -> Type) -> Type
```

```
data State : Effect where
     Get :      sig State a  a
     Put : b -> sig State () a b
```

```
STATE : Type -> EFFECT
STATE t = MkEff t State
```

Effect Signatures

```
data StdIO : Effect where
     PutStr : String -> sig StdIO ()
     GetStr :           sig StdIO String
     PutCh  : Char ->   sig StdIO ()
     GetCh  :           sig StdIO Char
```

```
STDIO : EFFECT
STDIO = MkEff () StdIO
```

Effect Handlers

Handlers

```
class Handler (e : Effect) (m : Type -> Type) where
  covering
  handle : (r : res) -> (eff : e t res resk) ->
           (k : ((x : t) -> resk x -> m a)) -> m a
```

Example Instances

```
instance Handler State m
instance Handler StdIO IO
instance Handler StdIO (List String ->
                            (a, List String))
```

Effect Handlers

State

```
instance Handler State m where
      handle st Get     k = k st st
      handle st (Put n) k = k () n
```

Effect Handlers

State

```
instance Handler State m where
     handle st Get     k = k st st
     handle st (Put n) k = k () n
```

StdIO

```
instance Handler StdIO IO where
    handle () (PutStr s) k = do putStr s; k () ()
    handle () GetStr     k = do x <- getLine; k x ()
    handle () (PutCh c)  k = do putChar c; k () ()
    handle () GetCh      k = do x <- getChar; k x ()
```

Demonstration: Dependent Effects

Summary

Why are we interested in dependent types?

- *Safety*
  - Programs checked against precise specifications

Summary

Why are we interested in dependent types?

- *Safety*
  - Programs checked against precise specifications
- *Expressivity*
  - Better, more descriptive APIs
  - *Type directed* development

Summary

Why are we interested in dependent types?

- *Safety*
  - Programs checked against precise specifications
- *Expressivity*
  - Better, more descriptive APIs
  - *Type directed* development
  - Type system should be *helping*, not telling you off!

Summary

Why are we interested in dependent types?

- *Safety*
  - Programs checked against precise specifications
- *Expressivity*
  - Better, more descriptive APIs
  - *Type directed* development
  - Type system should be *helping*, not telling you off!
- *Genericity*
  - e.g. program generation

Summary

Why are we interested in dependent types?

- *Safety*
  - Programs checked against precise specifications
- *Expressivity*
  - Better, more descriptive APIs
  - *Type directed* development
  - Type system should be *helping*, not telling you off!
- *Genericity*
  - e.g. program generation
- *Efficiency*
  - More precise type information should help the compiler
  - *Partial evaluation*, *erasure*.

Summary — applications

Dependent types are an active research topic, and we're having lots of fun. Some things we've been working on:

- Concurrency (e.g. verify absence of deadlock)
- Network transport protocols
- Packet formats
- Type-safe web applications
- Scientific programming
- ...