(Simulating) Effects in Domain-Specific Languages

IOHK Global Summit Lisbon 2018

Andres Löh

2018-01-17 —

Well-Typed
The Haskell Consultants

Messages of the talk

- Use **deep embeddings** so that you can define multiple interpretations, and in particular simulate all your programs in different contexts.
- Keep in mind that you can use both **initial** and **final** style – whatever works easier for you.
- Define **meaningful application-specific interfaces** that express what you are actually doing.
- Keep simulation in mind when designing your interfaces. Do not expose **unnecessary implementation details**.

Flavours of Haskell EDSLs

Hutton's Razor[1]

A language with just two operations:

```
lit :: Integer -> Expr          -- integer literal
add :: Expr -> Expr -> Expr     -- addition
```

[1] named after Graham Hutton

Hutton's Razor[1]

A language with just two operations:

```
lit :: Integer -> Expr         -- integer literal
add :: Expr -> Expr -> Expr    -- addition
```

One of the most simple EDSLs one can define.

[1] named after Graham Hutton

Well-Typed

Hutton's Razor[1]

A language with just two operations:

```
lit :: Integer -> Expr          -- integer literal
add :: Expr -> Expr -> Expr     -- addition
```

One of the most simple EDSLs one can define.

Example program:

```
term :: Expr
term = lit 1 `add` (lit 3 `add` lit 4)
```

[1] named after Graham Hutton

Shallow embedding

The embedded program directly denotes its interpretation:

```
type Expr = Integer

lit = id
add = (+)
```

Shallow embedding

The embedded program directly denotes its interpretation:

```
type Expr = Integer
lit = id
add = (+)
```

```
GHCi> term
8
```

Shallow embedding – evaluation

Pro:

- Very direct and simple.
- Usually quite performant.
- Easy to add new language constructs.

Con:

- Tied to a single interpretation / semantics.
- Interface of the EDSL is implicit.
- No analysis of the program possible.
- No proper abstraction.

Deep embedding

The embedded program represents itself (and thereby all possible interpretations):

```
data Expr =
    Lit Integer
  | Add Expr Expr
```

```
lit = Lit
add = Add
```

Interpretation: evaluation

```haskell
eval :: Expr -> Integer
eval (Lit i)     = i
eval (Add e1 e2) = eval e1 + eval e2
```

Interpretation: evaluation

```
eval :: Expr -> Integer
eval (Lit i)     = i
eval (Add e1 e2) = eval e1 + eval e2
```

```
GHCi> eval term
8
```

Interpretation: textual representation

```
text :: Expr -> String
text (Lit i)     = show i
text (Add e1 e2) =
  "(" <> text e1 <> "+" <> text e2 <> ")"
```

Well-Typed

Interpretation: textual representation

```haskell
text :: Expr -> String
text (Lit i)     = show i
text (Add e1 e2) =
  "(" <> text e1 <> "+" <> text e2 <> ")"
```

```
GHCi> text term
"(1+(3+4))"
```

Interpretation: instructions for a stack machine

```
data Instr =
    PUSH Integer
  | ADD
```

```
compile :: Expr -> [Instr]
compile (Lit i)     = [PUSH i]
compile (Add e1 e2) =
  compile e1 ++ compile e2 ++ [ADD]
```

Interpretation: instructions for a stack machine

```haskell
data Instr =
    PUSH Integer
  | ADD
```

```haskell
compile :: Expr -> [Instr]
compile (Lit i)     = [PUSH i]
compile (Add e1 e2) =
  compile e1 ++ compile e2 ++ [ADD]
```

```
GHCi> compile term
[PUSH 1, PUSH 3, PUSH 4, ADD, ADD]
```

Deep embedding – evaluation

Pro:

- Easy to define several interpretations.
- Easy to perform analysis and transformations of the program.
- Interface is explicit (via constructors of datatype).
- Hard to refer to a particular abstraction.

Con:

- Sometimes trickier in terms of performance (e.g. sharing).
- Harder to add new language constructs.

Using a type class instead

```
class IsExpr e where
  lit :: Integer -> e
  add :: e -> e -> e
```

Using a type class instead

```haskell
class IsExpr e where
  lit :: Integer -> e
  add :: e -> e -> e
```

```haskell
term :: IsExpr e => e
term = lit 1 `add` (lit 3 `add` lit 4)
```

Using a type class instead

```
class IsExpr e where
  lit :: Integer -> e
  add :: e -> e -> e
```

```
term :: IsExpr e => e
term = lit 1 `add` (lit 3 `add` lit 4)
```

Slightly different interface:

```
add :: Expr -> Expr -> Expr
add :: IsExpr e => e -> e -> e
```

Interpretation: evaluation

```haskell
instance IsExpr Integer where
  lit :: Integer -> Integer
  lit = id
  add :: Integer -> Integer -> Integer
  add = (+)
```

Interpretation: evaluation

```haskell
instance IsExpr Integer where
  lit :: Integer -> Integer
  lit  =  id
  add :: Integer -> Integer -> Integer
  add  =  (+)
```

```
GHCi> term :: Integer
8
```

Interpretation: textual representation

```haskell
instance IsExpr String where
  lit :: Integer -> String
  lit i = show i
  add :: String -> String -> String
  add e1 e2 =
    "(" <> e1 <> "+" <> e2 <> ")"
```

Well-Typed

Interpretation: textual representation

```haskell
instance IsExpr String where
  lit :: Integer -> String
  lit i = show i
  add :: String -> String -> String
  add e1 e2 =
    "(" <> e1 <> "+" <> e2 <> ")"
```

Note: No recursive calls.

Interpretation: textual representation

```
instance IsExpr String where
  lit :: Integer -> String
  lit i = show i
  add :: String -> String -> String
  add e1 e2 =
    "(" <> e1 <> "+" <> e2 <> ")"
```

Note: No recursive calls.

```
GHCi> term :: Eval
"(1+(3+4))"
```

Interpretation: instructions for a stack machine

```
instance IsExpr [Instr] where
  lit :: Integer -> [Instr]
  lit i = [PUSH i]
  add :: [Instr] -> [Instr] -> [Instr]
  add e1 e2 =
    e1 ++ e2 ++ [ADD]
```

```
GHCi> term :: [Instr]
[PUSH 1, PUSH 3, PUSH 4, ADD, ADD]
```

Well-Typed

Probably better: use a `newtype`

```haskell
newtype Eval = EvalC {unEval :: Integer}
  deriving Show
```

```haskell
instance IsExpr Eval where
  lit :: Integer -> Eval
  lit =  coerce

  add :: Eval -> Eval -> Eval
  add =  coerce ((+) :: Integer -> Integer -> Integer)
```

```
GHCi> term :: Eval
EvalC {unEval = 8}
```

Comparison

“Initial” style:

```
data Expr =
    Lit Integer
  | Add Expr Expr
```

“Final” style:

```
class IsExpr e where
  lit :: Integer -> e
  add :: e -> e -> e
```

Both are deep embeddings, with slightly different advantages and disadvantages.

From final to initial

```haskell
instance IsExpr Expr where
  lit = Lit
  add = Add
```

From initial to final

```haskell
from :: IsExpr e => Expr -> e
from (Lit i)     = lit i
from (Add e1 e2) = add (from e1) (from e2)
```

Well-Typed

Summary

We’ll focus on **deep embeddings**, because we want multiple interpretations of our programs, in particular:

- “real-world” execution,
- simulated execution.

We’ll still consider both initial and final style.

Adding effects

New interface

```haskell
data Var
data Expr
data Imp a
instance Monad Imp
new :: Imp Var
set :: Var -> Expr -> Imp ()
say :: Var -> Imp ()
var :: Var -> Expr
lit :: Integer -> Expr        -- as before
add :: Expr -> Expr -> Expr   -- as before
```

Example program

```
fib = do
  x <- new
  y <- new
  z <- new
  set x (lit 1)
  set y (lit 1)
  forever $ do
    say x
    set z (var x)
    set x (var y)
    set y (add (var z) (var y))
```

A deep embedding?

```haskell
data Expr =
    Lit Integer
  | Add Expr Expr
  | Var Var

data Imp :: * -> * where
  New      :: Imp Var
  Set      :: Var -> Expr -> Imp ()
  Say      :: Var -> Imp ()

data Var
```

A deep embedding?

```
data Expr =
     Lit Integer
   | Add Expr Expr
   | Var Var

data Imp :: * -> * where
  New       :: Imp Var
  Set       :: Var -> Expr -> Imp ()
  Say       :: Var -> Imp ()

data Var
```

Two problems:

- What about `instance Monad Imp` ?
- What about `Var` ?

A deep embedding?

```
data Expr =
    Lit Integer
  | Add Expr Expr
  | Var Var

data Imp :: * -> * where
  New        :: Imp Var
  Set        :: Var -> Expr -> Imp ()
  Say        :: Var -> Imp ()

  Return_Imp :: a -> Imp a
  Bind_Imp   :: Imp a -> (a -> Imp b) -> Imp b

data Var
```

Instances

```
instance Monad Imp where
  return = Return_Imp
  (>>=)  = Bind_Imp
instance Applicative Imp where
  pure   = return
  (<*>)  = ap
instance Functor Imp where
  fmap   = liftM
```

Instances

```
instance Monad Imp where
  return = Return_Imp
  (>>=)  = Bind_Imp
instance Applicative Imp where
  pure   = return
  (<*>)  = ap
instance Functor Imp where
  fmap   = liftM
```

Laws are not fulfilled on the syntactic level, therefore there is a proof obligation for each interpretation.

Or switch to a proper **free monad**.

Attempting an interpretation

Idea:

- Interpret `Imp` programs as `IO` actions.
- Represent variables as `IORef` s.

Attempting an interpretation

Idea:

- Interpret `Imp` programs as `IO` actions.
- Represent variables as `IORef` s.

However, this immediately fails:

```
execIO :: Imp a -> IO a
execIO New = newIORef 0  -- :: IO (IORef Integer), not IO Var
...
```

Because:

```
New :: Imp Var -> Imp a
```

Attempting an interpretation

Idea:

- Interpret `Imp` programs as `IO` actions.
- Represent variables as `IORef` s.

However, this immediately fails:

```
exec_IO :: Imp a -> IO a
exec_IO New = newIORef 0  -- :: IO (IORef Integer), not IO Var
...
```

Because:

```
New :: Imp Var -> Imp a
```

We must have the freedom to choose what `Var` is interpreted as.

Abstracting from `Var`

```haskell
data Expr var =
     Lit Integer
   | Add (Expr var) (Expr var)
   | Var var

data Imp :: * -> * -> * where
  New        :: Imp var var
  Set        :: var -> Expr var -> Imp var ()
  Say        :: var -> Imp var ()

  Return_Imp :: a -> Imp var a
  Bind_Imp   :: Imp var a -> (a -> Imp var b) -> Imp var b
```

Interpretation: execution

```
execIO :: Imp (IORef Integer) a -> IO a
execIO New            = newIORef 0
execIO (Set v e)      = do
  x <- evalIO e
  writeIORef v x
execIO (Say v)        = do
  x <- readIORef v
  print x
execIO (ReturnImp x) = return x
execIO (BindImp m k) = execIO m >>= execIO . k
```

```
evalIO :: Expr (IORef Integer) -> IO Integer
evalIO (Lit i)     = return i
evalIO (Add e1 e2) = liftM2 (+) (evalIO e1) (evalIO e2)
evalIO (Var v)     = readIORef v
```

How to simulate without IO?

```
newtype Counter = Counter {getCounter :: Integer}
  deriving (Show, Num, Eq, Ord)

data Sim a where
  Fresh     :: Sim Counter
  Insert    :: Counter -> Integer -> Sim ()
  Lookup    :: Counter -> Sim Integer
  Message   :: String -> Sim ()

  Return_Sim :: a -> Sim a
  Bind_Sim   :: Sim a -> (a -> Sim b) -> Sim b

instance Monad Sim where
  return = Return_Sim
  (>>=)  = Bind_Sim
```

How to simulate without IO?

```
newtype Counter = Counter {getCounter :: Integer}
  deriving (Show, Num, Eq, Ord)

data Sim a where
  Fresh     :: Sim Counter
  Insert    :: Counter -> Integer -> Sim ()
  Lookup    :: Counter -> Sim Integer
  Message   :: String -> Sim ()

  ReturnSim :: a -> Sim a
  BindSim   :: Sim a -> (a -> Sim b) -> Sim b

instance Monad Sim where
  return = ReturnSim
  (>>=)  = BindSim
```

This is just **Phase One**, but `Sim` is quite a bit more low-level than `Imp` .

Well-Typed

Interpretation: simulation of programs

```
execSim :: Imp Counter a -> Sim a
execSim New            = do
  v <- Fresh
  Insert v 0
  return v
execSim (Set v e)      = do
  x <- evalSim e
  Insert v x
execSim (Say v)        = do
  x <- Lookup v
  Message (show x)
execSim (ReturnImp x) = return x
execSim (BindImp m k) = execSim m >>= execSim . k
```

Well-Typed

Interpretation: simulation of expressions

```
evalSim :: Expr Counter -> Sim Integer
evalSim (Lit i)      = return i
evalSim (Add e1 e2) = liftM2 (+) (evalSim e1) (evalSim e2)
evalSim (Var v)      = Lookup v
```

Phase Two

One option is:

```haskell
type SimResult = Stream (Of String) (State SimState)
```

Phase Two

One option is:

```
type SimResult = Stream (Of String) (State SimState)
```

A `Stream` (from the `streaming` package) is a way to interleave items and effects:

```
data Stream f m r =
     Step   !(f (Stream f m r))
   | Effect (m (Stream f m r))
   | Return r
data Of a b = !a :> b
```

```
instance (Functor f, Monad m) => Monad (Stream f m)
instance Functor f => MonadTrans (Stream f)
...
```

Well-Typed

Phase Two

One option is:

```
type SimResult = Stream (Of String) (State SimState)
```

```
data SimState = SimState
  { _ctr :: Counter
  , _env :: Map Counter Integer
  }
```

```
ctr :: Lens' SimState Counter
env :: Lens' SimState (Map Counter Integer)
```

Phase One to Phase Two

```
runSim :: Sim a -> SimResult a
runSim Fresh          = lift $ do
  v <- use ctr
  ctr += 1
  env %= insert v 0
  return v
runSim (Insert v x)  = lift $ env %= insert v x
runSim (Lookup v)    = lift $ (! v) <$> use env
runSim (Message m)   = yield m
runSim (Return_Sim x) = return x
runSim (Bind_Sim m k) = runSim m >>= runSim . k
```

Well-Typed

Actual simulation

```
testFib :: [String]
testFib =
  evalState
    (S.toList_ (S.take 5 (runSim (exec_Sim fib))))
    (SimState {_ctr = 0, _env = empty})
```

```
GHCi> testFib
["1", "1", "2", "3", "5"]
```

Final style?

```haskell
class
  (IsExpr e v, Monad i) => IsImp i e v | i -> e v where
  new :: i v
  set :: v -> e -> i ()
  say :: v -> i ()

class IsExpr e v | e -> v where
  var :: v -> e
  lit :: Integer -> e
  add :: e -> e -> e
```

Well-Typed

A few observations

Abstraction over `Var` was important

- Fixing such a resource type to a concrete choice can easily limit us to one (or just a few interpretations).
- Be careful with anything like handles, database connections, variables, stores, or any abstract types where the interface itself ties you into specific monads.

No `IO`

- Avoid direct use of `IO` or `MonadIO` in your domain-specific code at all costs.
- Try to capture the operations you actually need in the interface directly. It has the added side effect that it becomes clearer what exactly the code is allowed and supposed to do.

Few large interfaces

- We only used actual monad transformers at the lowest level, when implementing `Sim`.
- How exactly we implemented `Sim` is irrelevant to the rest of the program (monad transformers, extensible effects, monolithic monad, ...).
- Too much granularity in interfaces often makes things more complicated rather than less. Only split if there are real use cases where you need one without the other, or where the interfaces really are at different levels.

Effect-free interfaces are still best

- Nothing beats the simplicity of the first scenario.
- Even if interpretation require effects, the DSL not necessarily does.
- Write code as pure functions if you can.