Monads for Free!

Andres Löh

Haskell eXchange – 9 October 2013

Well-Typed
The Haskell Consultants

Everything is an (E)DSL

Haskell is great for EDSLs

Deep embeddings

Use data to represent programs:

```haskell
data Expr = Lit Int | Add Expr Expr
```

Deep embeddings

Use data to represent programs:

```
data Expr = Lit Int | Add Expr Expr
```

```
1 + (3 + 4)
```

corresponds to

```
Add (Lit 1) (Add (Lit 3) (Lit 4))
```

One program, many interpretations

```
eval :: Expr → Int
eval (Lit n)        = n
eval (Add e1 e2) = eval e1 + eval e2
```

```
text :: Expr → String
text (Lit n)        = show n
text (Add e1 e2) = "(" ++ text e1 ++ " + " ++ text e2 ++ ")"
```

What if we want to
embed an imperative language?

Example: Interaction

```
Say "Hello".
Say "Who are you?".
Ask for a “ name ”.
Say "Nice to meet you, " ++ name ++ "!".
```

Well-Typed

Example: Interaction

```
Say "Hello".
Say "Who are you?".
Ask for a name.
Say "Nice to meet you, " ++ name ++ "!".
```

Looks monadic!

Example: Interaction

```
do
  say "Hello"
  say "Who are you?"
  name ← ask
  say ("Nice to meet you, " ++ name ++ "!")
```

Example: Interaction

```
do
  say "Hello"
  say "Who are you?"
  name ← ask
  say ("Nice to meet you, " ++ name ++ "!")
```

Trivial to implement directly:

```
say = putStrLn
ask = getLine
```

But can we make a deep embedding?

Interaction interface

```
data Interaction a    -- abstract
instance Monad Interaction
say :: String → Interaction ()
ask :: Interaction String
```

GADTs to the rescue!

Brute-force GADT-based embedding

```
data Interaction :: * → * where
  Say    :: String → Interaction ()
  Ask    :: Interaction String

  Return :: a → Interaction a
  Bind   :: Interaction a → (a → Interaction b) → Interaction b
```

Brute-force GADT-based embedding

```
data Interaction :: * → * where
  Say    :: String → Interaction ()
  Ask    :: Interaction String

  Return :: a → Interaction a
  Bind   :: Interaction a → (a → Interaction b) → Interaction b
```

```
instance Monad Interaction where
  return = Return
  (>>=)  = Bind
```

Brute-force GADT-based embedding

```
data Interaction :: * → * where
  Say    :: String → Interaction ()
  Ask    :: Interaction String

  Return :: a → Interaction a
  Bind   :: Interaction a → (a → Interaction b) → Interaction b
```

```
instance Monad Interaction where
  return = Return
  (>>=)  = Bind
```

```
say = Say
ask = Ask
```

Well-Typed

Interpretation as IO

```
run :: Interaction a → IO a
run (Say msg)  = putStrLn msg
run  Ask       = getLine
run (Return x) = return x
run (Bind m f) = do x ← run m; run (f x)
```

GADTs to the rescue?

Monad laws

Left identity:

$$\text{return } x \gg\!= f \quad \equiv f\ x$$

Right identity:

$$m \gg\!= \text{return} \quad \equiv m$$

Associativity:

$$(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$$

Why?

Expectations

Should these two behave differently?

```
do
  say "Tell me something ..."
  something ← ask
  return something
```

```
do
  say "Tell me something ..."
  ask
```

Well-Typed

Expectations

Or these?

```
do
  let qa question = do say question; ask
  x ← qa "Tell me more ..."
  y ← qa "... and more ..."
  return (x, y)
```

```
do
  say "Tell me more ..."
  x ← ask
  say "... and more ..."
  y ← ask
  return (x, y)
```

Does Interaction adhere to the monad laws?

A close look

```
data Interaction :: * → * where
  Say    :: String → Interaction ()
  Ask    :: Interaction String
  Return :: a → Interaction a
  Bind   :: Interaction a → (a → Interaction b) → Interaction b
```

```
instance Monad Interaction where
  return = Return
  (>>=)  = Bind
```

Wouldn’t it be nice
if we could guarantee the
monad laws by construction?

Observation

In essence, the monad laws say that every monadic computation has a normal form:

```
do
  x1 ← step1
  x2 ← step2
  ...
  xn ← stepn
  return something
```

Well-Typed

Normalizing interactions

```
data Interaction :: * → * where
  Say    :: String → Interaction ()
  Ask    :: Interaction String
  Return :: a → Interaction a
  Bind   :: Interaction a → (a → Interaction b) → Interaction b
```

Normalizing interactions

```
data Interaction :: * → * where
  Say    :: String → Interaction ()
  Ask    :: Interaction String
  Return :: a → Interaction a
  Bind   :: Interaction a → (a → Interaction b) → Interaction b

say'     :: String → (() → Interaction b) → Interaction b
say' msg = Bind (Say msg)
ask'     :: (String → Interaction b) → Interaction b
ask'     = Bind Ask
```

Well-Typed

Normalizing interactions

```
data Interaction :: * → * where
  Say    :: String → Interaction ()
  Ask    :: Interaction String

  Return :: a → Interaction a
  Bind   :: Interaction a → (a → Interaction b) → Interaction b

  Say′   :: String → (() → Interaction b) → Interaction b

  Ask′   :: (String → Interaction b) → Interaction b
```

Normalizing interactions

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say'   :: String → (() → Interaction b) → Interaction b
  Ask'   :: (String → Interaction b) → Interaction b
```

Normalizing interactions

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

No longer a “proper” GADT:

```
data Interaction a =
    Return a
  | Say′ String (() → Interaction a)
  | Ask′ (String → Interaction a)
```

Still a monad?

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

Well-Typed

Still a monad?

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

```
instance Monad Interaction where
  return = Return
  (>>=) :: Interaction a → (a → Interaction b) → Interaction b
  Return x    >>= f = f x
  Say′ msg k >>= f = Say′ msg ((>>=f) ∘ k)
  Ask′ k      >>= f = Ask′ ((>>=f) ∘ k)
```

Still implementing the interface?

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

Well-Typed

Still implementing the interface?

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

```
say :: String → Interaction ()
say msg = Say′ msg Return
ask :: Interaction String
ask = Ask′ Return
```

Still possible to write an interpreter?

```
data Interaction :: * → * where
  Return :: a → Interaction a

  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

Still possible to write an interpreter?

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

```
run :: Interaction a → IO a
run (Return x    ) = return x
run (Say′ msg k) = putStrLn msg >>= run ∘ k
run (Ask′ k      ) = getLine >>= run ∘ k
```

Another interpreter?

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

Another interpreter?

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say′   :: String → (() → Interaction b) → Interaction b
  Ask′   :: (String → Interaction b) → Interaction b
```

```
simulate :: Interaction a → [String] → [String]
simulate (Return _    )  is     = []
simulate (Say′ msg k)    is     = msg : simulate (k ()) is
simulate (Ask′ k      )  (i : is) = simulate (k i) is
```

Well-Typed

Simulation

```
prog =
  do
    say "Hello"
    say "Who are you?"
    name ← ask
    say ("Nice to meet you, " ++ name ++ "!")
```

```
ghci> simulate prog ["Andres"]
["Hello","Who are you?","Nice to meet you, Andres!"]
```

And the monad laws hold as well!

Can we generalize?

Abstracting from Say′ and Ask′

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Say'   :: String → (() → Interaction b) → Interaction b
  Ask'   :: (String → Interaction b) → Interaction b
```

Abstracting from Say′ and Ask′

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Wrap   :: InteractionOp a → Interaction a

data InteractionOp :: * → * where
  Say′   :: String → (() → Interaction b) → InteractionOp b
  Ask′   :: (String → Interaction b) → InteractionOp b
```

Abstracting from Say′ and Ask′

```
data Interaction :: * → * where
  Return :: a → Interaction a
  Wrap   :: InteractionOp (Interaction a) → Interaction a
data InteractionOp :: * → * where
  Say′   :: String → (() → r) → InteractionOp r
  Ask′   :: (String → r) → InteractionOp r
```

Abstracting from Say′ and Ask′

```
data Free :: (∗ → ∗) → ∗ → ∗ where
  Return :: a → Free f a
  Wrap   :: f (Free f a) → Free f a
data InteractionOp :: ∗ → ∗ where
  Say′   :: String → (() → r) → InteractionOp r
  Ask′   :: (String → r) → InteractionOp r
type Interaction = Free InteractionOp
```

Well-Typed

Free f is a monad whenever f is a functor

```
data Free :: (* → *) → * → * where
  Return :: a → Free f a
  Wrap   :: f (Free f a) → Free f a
```

```
instance Functor f ⇒ Monad (Free f) where
  return :: a → Free f a
  return = Return

  (>>=) :: Free f a → (a → Free f b) → Free f b
  Return x >>= f = f x
  Wrap c   >>= f = Wrap (fmap (>>=f) c)
```

Well-Typed

Is InteractionOp a functor?

```
instance Functor InteractionOp where
  fmap f (Say′ msg k) = Say′ msg (f ∘ k)
  fmap f (Ask′ k    ) = Ask′ (f ∘ k)
```

Still implementing the interface

```
say :: String → Interaction ()
say msg = Wrap (Say′ msg Return)
ask :: Interaction String
ask = Wrap (Ask′ Return)
```

Well-Typed

So given a functor, we get a monad for free?

Const

```haskell
newtype Const a b = Const a
  deriving (Functor, Show)

data Void
```

Const

```
newtype Const a b = Const a
  deriving (Functor, Show)

data Void
```

```
data Free :: (* → *) → * → * where
  Return :: a → Free f a
  Wrap   :: f (Free f a) → Free f a
```

Free (Const Void) $\cong$ a

The identity monad.

Const

```
newtype Const a b = Const a
  deriving (Functor, Show)

data Void
```

```
data Free :: (* → *) → * → * where
  Return :: a → Free f a
  Wrap   :: f (Free f a) → Free f a
```

Free (Const Void) ≅ a

The identity monad.

Free (Const ()) ≅ Maybe a

Id

```haskell
newtype Id a = Id a
  deriving (Functor, Show)
```

Id

```
newtype Id a = Id a
  deriving (Functor, Show)
```

```
data Free :: (∗ → ∗) → ∗ → ∗ where
  Return :: a → Free f a
  Wrap   :: f (Free f a) → Free f a
```

Free Id ≅ Delayed a

```
data Delayed a = Now a | Later (Delayed a)
```

More interesting examples?

(Cooperative) Concurrency

```
data ProcessF :: * → * where
  Atomically :: IO a → (a → r) → ProcessF r
  Fork       :: Process () → r → ProcessF r
```

```
type Process = Free ProcessF
```

```
atomically :: IO a → Process a
atomically m = Wrap (Atomically m Return)

fork :: Process () → Process ()
fork p = Wrap (Fork p (Return ()))
```

Scheduling concurrent operations

```
schedule :: [Process ()] → IO ()
schedule                                  []    = return ()
schedule (Return _                     : ps) = schedule ps
schedule (Wrap (Atomically m k) : ps) = do
                                                  x ← m
                                                  schedule (ps ++ [k x])
schedule (Wrap (Fork p1 p2)        : ps) = schedule (ps ++ [p2, p1])
```

Example

```haskell
example :: Process ()
example = do
  fork (replicateM_ 5 (atomically (putStrLn "Haskell")))
  fork (replicateM_ 6 (atomically (putStrLn "eXchange")))
  atomically (putStrLn "2013")
```

Example

```
example :: Process ()
example = do
  fork (replicateM_ 5 (atomically (putStrLn "Haskell")))
  fork (replicateM_ 6 (atomically (putStrLn "eXchange")))
  atomically (putStrLn "2013")
```

```
ghci> schedule [example]
Haskell
2013
eXchange
Haskell
eXchange
Haskell
eXchange
Haskell
eXchange
eXchange
```

So much more to say . . .

A little bit more

- Free monad transformer
- More efficient representations
- `free` package
- `operational` package (different approach using GADTs)
- More applications: effects, parsing, coroutines, games, ...
- A few interesting examples: `IOspec`, `free-game`, `sunroof`
- Other free structures (e.g. free applicatives)
- Cofree comonads
- ...

Questions?