Programming with Monadic Effect Hierarchies
BCTCS’15

Michael B. Gale
Computer Laboratory, University of Cambridge

14 September 2015
Can object-oriented programming techniques simplify monadic programming?
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

f is pure
\[ \text{map } f \ (\text{map } g \ xs) = \text{map } (f \circ g) \ xs \]

Proof is by induction on \( xs \).
data Expr = Val Int | Add Expr Expr
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val n) = n
eval (Add l r) = eval l + eval r
data Expr = Val Int | Add Expr Expr | Div Expr Expr
data Expr = Val Int | Add Expr Expr | Div Expr Expr

data Maybe a = Nothing | Just a

safediv :: Int -> Int -> Maybe Int
safediv x 0 = Nothing
safediv x y = Just (x `div` y)
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Add l r) = case eval l of
  Nothing   -> Nothing
  (Just x)  -> case eval r of
    Nothing  -> Nothing
    (Just y) -> Just (x+y)
eval (Div l r) = case eval l of
  Nothing   -> Nothing
  (Just x)  -> case eval r of
    Nothing  -> Nothing
    (Just y) -> x `safediv` y
Monads to the rescue!

Abstracting sequential composition:

class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
Monads to the rescue!

Abstracting sequential composition:

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b

instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
```
Monads to the rescue!

eval :: Expr -> Maybe Int
eval (Val n) = return n
eval (Add l r) = eval l >>= \x ->
                    eval r >>= \y ->
                    return (l+r)
eval (Div l r) = eval l >>= \x ->
                    eval r >>= \y ->
                    l `safediv` r
eval :: Expr -> Maybe Int

eval (Val n) = return n

eval (Add l r) = do x <- eval l
                    y <- eval r
                    return (l + r)

eval (Div l r) = do x <- eval l
                    y <- eval r
                    l `safediv` r
There are many different monads: state, descriptive exceptions, backtracking, non-determinism, IO, ...
There are many different monads: state, descriptive exceptions, backtracking, non-determinism, IO, ...

Sometimes we need more than just one at the same time
Monad transformers

Stack effects by defining return and »= in terms of other monads.
Monad transformers

Stack effects by defining `return` and `»=` in terms of other monads.

```haskell
data StateT s m a = MkStateT (s -> m (a,s))

runStateT :: StateT s m a -> s -> m (a,s)
runStateT (MkStateT m) s = m s

instance Monad m => Monad (StateT s m) where
    return x = MkStateT (\s -> return (x,s))

    (MkStateT m) >>= f = MkStateT (\s -> do
        (r,s') <- m s
        let (MkStateT m') = f r in m' s')
```
Monad transformers are awkward to use

type MyEffStack a = StateT Int Maybe a
type MyEffStack a = StateT Int Maybe a

foo :: MyEffStack Int
foo = do
    n <- get
    if n > 42 then lift Nothing
    else return n
Monad transformers are awkward to use

```haskell
type MyEffStack a = StateT Int Maybe a

foo :: MyEffStack Int
foo = do
  n <- get
  if n > 42 then lift Nothing
  else return n

GHCi> runStateT foo 8
Just 8
GHCi> runStateT foo 43
Nothing
```
Monad transformers are awkward to use

Running a computation gets worse the more layers there are in the monad stack:

```haskell
runMachine code =
    execState (runReaderT (runStateT runCode []) code) []
```
Monad transformers are awkward to use

Running a computation gets worse the more layers there are in the monad stack:

```haskell
runMachine code =
    execState (runReaderT (runStateT runCode []) code) []
```

We may also need multiple lifts to lift through more than one layer:

```haskell
lift (lift get)
```
flip :: Monad m => StateT Bool m ()
flip = get >>= \b -> put (not b)

fresh :: Monad m => StateT Int m Int
fresh = get >>= \n -> put (n + 1) >> return n

foo :: StateT Int (StateT Bool Identity) Int
foo = do
  lift flip
  fresh
“Can we think of monad stacks as objects and hide them behind object-oriented abstractions?”
Idea

Use dynamic dispatch to
- call lift
- and runStateT (or similar)
when necessary
■ We need an object system
We need an object system

Some unpublished work by Kiselyov and Lämmel: “Haskell’s overlooked object system” (2005)
- We need an object system
- Some unpublished work by Kiselyov and Lämmel: “Haskell’s overlooked object system” (2005)
- Let’s roll our own, purely-functional system
- We need an object system
- Some unpublished work by Kiselyov and Lämmel: “Haskell’s overlooked object system” (2005)
- Let’s roll our own, purely-functional system
- Formal systems exist – e.g. Cardelli and Pierce
What even is OO?
What even is OO?


1. Inheritance
2. Object
3. Class
4. Encapsulation
5. Method
Example 1: Counters

state Counter where
    data count :: Int

    next :: Int
    next = do
        r <- this.!count
        count <: (r+1)
        return r

    c :: Counter
    c = new 0

main = print (result (c.!next))
Example 1: Counters

state Renamer : Counter where

  label :: Tree Int -> Tree Int

  label (Leaf _) = do
      n <- this.!next
      return (Leaf n)

  label (Node l r) = do
      l' <- this.!label l
      r' <- this.!label r
      return (Node l' r')

r :: Renamer
r = new 0

main = print (result (r.!label atree))
Example 2: Abstract machines

state Stack a where

  data stack :: [a]

  push :: a -> ()
push x = do
      st <- this.!stack
      stack <-: (x : st)

  pop :: a
  pop = do
      (x:xs) <- this.!stack
      stack <-: xs
      return x
Example 2: Abstract machines

state AbstractMachine : Stack Int where
    data memory :: Map Word32 Word32
    data accum :: Int

write :: Word32 -> Word32 -> ()
write addr val = ...

read :: Word32 -> Word32
read addr = ...

exec :: Instr -> ()
exec (PUSH v) = this.!push v
exec (LOAD a) = do
    v <- this.!read a
    accum <: v
abstract state Expr where
  eval :: Int

state Val : Expr where
  data val :: Int

  eval = do
    r <- this.!val
    return r

v :: Val
v = new 23

main = print (result (v.!eval))
Example 3: Modularity

v :: Val
v = new 23

e :: Expr
e = downcast v

main = print (result (e.eval))
Example 3: Modularity

state Add : Expr where
  data left  :: Expr
  data right :: Expr

  eval = do
    x <- this.!left.!eval
    y <- this.!right.!eval

    return (x+y)

a :: Add
a = new (e,e)

GHCi> result (a.!eval)
46
“How can I do this?”
import Language.MSH
“How can I do this?”

import Language.MSH *

(* and enable about 10 existing Haskell language extensions)
import Language.MSH

[state]

state Counter where
    data count :: Int

next :: Int
next = do
    r <- this.!count
    count <: (r+1)
    return r

]


c :: Counter
c = new 0

main = print (result (c.!next))
“Where do I get it?”

Code is available on GitHub:
https://github.com/mbg/monadic-state-hierarchies
“How does it work?”
“How does it work?”

Draft paper with all the details:
State classes

State classes are desugared into

- A $\Delta$-object
State classes are desugared into

- A $\Delta$-object
- A data type for the state
State classes

State classes are desugared into

- A $\Delta$-object
- A data type for the state
- A specialised $StateT$ for the class’s state
State classes are desugared into

- A $\Delta$-object
- A data type for the state
- A specialised $StateT$ for the class’s state
- A type class, representing the class’s interface
State classes are desugared into

- A $\Delta$-object
- A data type for the state
- A specialised $StateT$ for the class’s state
- A type class, representing the class’s interface
- Type class instances for the interface of the current class as well as all parents
State classes are desugared into

- A Δ-object
- A data type for the state
- A specialised `StateT` for the class’s state
- A type class, representing the class’s interface
- Type class instances for the interface of the current class as well as all parents
- Instances for miscellaneous type classes
Well-known technique for dynamic dispatch:

data Object =
    forall sub.
    ObjectLike sub ObjectM =>
    MkObject ObjectState sub
Well-known technique for dynamic dispatch:

\[
data \text{ Object } = \\
\forall \text{ sub. } \\
\text{ ObjectLike sub ObjectM } \Rightarrow \\
\text{ MkObject ObjectState sub}
\]

sub is existentially quantified – the type class constraint ensures sub-classes have at least the same interface.

But: rightly considered an anti-pattern
But: only works for abstract (base) classes.
Non-abstract (base) classes:

data Object = MkObjectSelf ObjectState
             | forall sub.
             ObjectLike sub ObjectM =>
             MkObject ObjectState sub

Object is also an instance of ObjectLike.
Non-abstract, non-final, non-base classes:

data SubObject =
    MkSubObjectSelf SubObjectState
| forall sub.
    SubObjectLike sub SubObjectM =>
    MkSubObjectSubObjectState sub
| forall sub.
    SubObjectLike sub SubObjectM =>
    MkSubObjectMid Object SubObjectState sub
| SubObjectEnd Object SubObjectState

Objects are functional zippers!
obj :: ChildChild
obj = MkChildChildChildEnd (MkChildEnd ...) data

view :: Child
view = downcast obj
=> MkChildMid
  => MkBaseSelf ...
childData
  (MkChildChildChildSelf data)
We want to reuse names as much as possible:

- For every method, getter, and setter there is an “internal” and an “external” version
- “external” calls require an object and construct a monad stack from it, before making an “internal” call
- “internal” call is the actual implementation and requires an existing monad stack
- “internal” and “external” versions are bundled up into selectors
Selectors

- .! composes selectors (as well as objects and selectors)
- this and assignment use “internal” calls
- Calling a method on an object uses the “external” calls
What next?

- We have built an object system which lets us write OO-style code if we need it
What next?

- We have built an object system which lets us write OO-style code if we need it
- Functional and object-oriented code can be interleaved
What next?

- We have built an object system which lets us write OO-style code if we need it
- Functional and object-oriented code can be interleaved
- Each state class corresponds to a state monad transformer
What next?

- We have built an object system which lets us write OO-style code if we need it
- Functional and object-oriented code can be interleaved
- Each state class corresponds to a state monad transformer
- Each $\Delta$-object sets up its own layer in the monad stack
We have built an object system which lets us write OO-style code if we need it.

Functional and object-oriented code can be interleaved.

Each state class corresponds to a state monad transformer.

Each $\Delta$-object sets up its own layer in the monad stack.

$\Delta$-objects don’t care what monad subsequent $\Delta$-objects add to the stack.
Let's have other flavours of classes which correspond to monads other than the state monad.
Example hierarchy

```
state ParserState
  msg : [ParseMessage]
  addError : String → ()
  addWarn : String → ()

exception Parser
  addFatal : String → ()
  parse : String → AST
```
Return values

- External method calls return a value and an updated object (*result* and *object* combinators)
Return values

- External method calls return a value and an updated object (*result* and *object* combinators)
- For the state monad, the result is always the return type of a function
Return values

- External method calls return a value and an updated object (*result* and *object* combinators)
- For the state monad, the result is always the return type of a function
- For other monads, such as exceptions, that may not be the case
Return values

- External method calls return a value and an updated object (*result* and *object* combinators)
- For the state monad, the result is always the return type of a function
- For other monads, such as exceptions, that may not be the case
- First approach: can’t override a method if it the effect of the defining class is different
Return values

- External method calls return a value and an updated object \((result\) and \(object\) combinators\)
- For the state monad, the result is always the return type of a function
- For other monads, such as exceptions, that may not be the case
- First approach: can’t override a method if the effect of the defining class is different
- Second approach: can override, but must return the original type
Conclusions

- Pure object-system in Haskell – no new language features needed
Conclusions

- Pure object-system in Haskell – no new language features needed
- Library takes care of all the boilerplate
Conclusions

- Pure object-system in Haskell – no new language features needed
- Library takes care of all the boilerplate
- Stacks of state monad transformers are easier to work with
Conclusions

- Pure object-system in Haskell – no new language features needed
- Library takes care of all the boilerplate
- Stacks of state monad transformers are easier to work with
- Also allows us to write modular programs
Conclusions

- Pure object-system in Haskell – no new language features needed
- Library takes care of all the boilerplate
- Stacks of state monad transformers are easier to work with
- Also allows us to write modular programs
- Encoding should generalise to other monads, thereby also generalising object-oriented programming
- Email: michael.gale@cl.cam.ac.uk
- Twitter: @michaelbgale
- Web: http://www.cl.cam.ac.uk/~mbg28