Functional Mocking

Lars Hupel

February 26th, 2015

lambda
DAλS

What even is "Mocking"?

> "In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways."
>
> Wikipedia: Mock object

What even is “Mocking”?

> “In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways.”
>
> Wikipedia: Mock object

Mocking in Scala

Example: ScalaMock

```
def testTurtle {
  val m = mock[Turtle]

  (m.setPosition _).expects(10.0, 10.0)
  (m.forward _).expects(5.0)
  (m.getPosition _).expects().returning(15.0, 10.0)

  drawLine(m, (10.0, 10.0), (15.0, 10.0))
}
```

Mocking: Why Not?

> "When you write a mockist test, you are testing the outbound calls of the SUT to ensure it talks properly to its suppliers ... Mockist tests are thus more coupled to the implementation of a method. Changing the nature of calls to collaborators usually cause a mockist test to break."
>
> Martin Fowler

Mocking: Why Not?

> “When you write a mockist test, you are testing the outbound calls of the SUT to ensure it talks properly to its suppliers ... Mockist tests are thus more coupled to the implementation of a method. Changing the nature of calls to collaborators usually cause a mockist test to break.”
>
> Martin Fowler

It is pitch black.
You are likely to be eaten by a grue.

Functional Mocking

... there's no such thing.

Functional Mocking

... there's no such thing.

various techniques avoid the need for mocking altogether

Functional Programming

- separation of data and operations
- parametric polymorphism
- higher-order functions
- lightweight interpreters

A Simple Calculator

```
data Expr =
    Literal Int
  | Var String
  | Sum Expr Expr

evaluate :: Map String Int -> Expr -> Maybe Int
```

A Simple Calculator

```
data Expr =
    Literal Int
  | Var String
  | Sum Expr Expr

evaluate :: (String -> Maybe Int) -> Expr -> Maybe Int
```

A Simple Calculator

```haskell
data Expr a =
    Literal a
  | Var String
  | Sum (Expr a) (Expr a)

evaluate :: Num a =>
  (String -> Maybe a) -> Expr a -> Maybe a
```

A (Not So) Simple Calculator

```
data Expr a t =
    Literal a
  | Var t
  | Sum (Expr a t) (Expr a t)

evaluate :: Num a =>
  (t -> Maybe a) -> Expr a t -> Maybe a
```

A (Not So) Simple Calculator

```haskell
data Expr a t =
    Literal a
  | Var t
  | Sum (Expr a t) (Expr a t)

evaluateM :: (Num a, Monad m) =>
  (t -> m a) -> Expr a t -> m a
```

Why This Complexity?

> “Always implement things when you actually need them, never when you just foresee that you need them.”
>
> Ron Jeffries about YAGNI

What Have We Gained?

Abstraction over Num

- no messing around with the values
- caller knows that only the `Num` influences the behaviour

Abstraction over Monad

- uniform data access
  - `Map`
  - database lookup
  - reading from standard input

Even More Abstraction

```haskell
-- get all variables
vars :: Expr a t -> [t]
vars = error ”some traversal”

-- check definedness of all variables
check :: Expr a (Maybe t) -> Maybe (Expr a t)
check = error ”traversing again?!”

-- substitute variables
subst :: (t -> Expr a t) -> Expr a t -> Expr a t
subst = error ”seriously?”
```

Even More Abstraction

```haskell
-- get all variables
vars :: Expr a t -> [t]
vars = Data.Foldable.toList

-- check definedness of all variables
check :: Expr a (Maybe t) -> Maybe (Expr a t)
check = Data.Traversable.sequenceA

-- substitute variables
subst :: (t -> Expr a t) -> Expr a t -> Expr a t
subst = (=<<)
```

Aspect-Oriented Programming

What if we want to log the variable access in `evaluateM`?

YAGNI Revisited

- without early abstraction, many concepts stay hidden
- YAGNI limits thinking
- especially important when building libraries

Interlude: Immutable Data Structures

```
class Person {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

Setter and Getter in Java

```
company.getITDepartment()
         .getHead()
         .setName(”Grace Hopper”);
```

Immutable Data in Haskell

```haskell
data Company =
  Company {
    it :: Department
  , hr :: Department
  }
```

```haskell
data Department =
  Department {
    boss :: Person
  , budget :: Currency
  }
```

```haskell
data Person =
  Person {
    name :: String
  }
```

Updating Immutable Data

```
company {
  it = (it company) {
    boss = (boss (it company)) {
      name = ”Grace Hopper”
    }
  }
}
```

Updating Immutable Data

```
company {
  it = (it c
    boss = (
      name =
    }
  }
}
```

Costate Comonad Coalgebra …?

PLT Borat
@PLT_Borat

Costate Comonad Coalgebra is equivalent of Java's member variable update technology for Haskell dl.dropbox.com/u/7810909/medi…

Reply Retweet Favorite More

Lenses To The Rescue!

The Naive Formulation

```
data Lens a b = Lens {
  get :: a -> b
  set :: a -> b -> a
}
```

Lenses To The Rescue!

The Naive Formulation

```haskell
data Lens a b = Lens {
  get :: a -> b
  set :: a -> b -> a
}
```

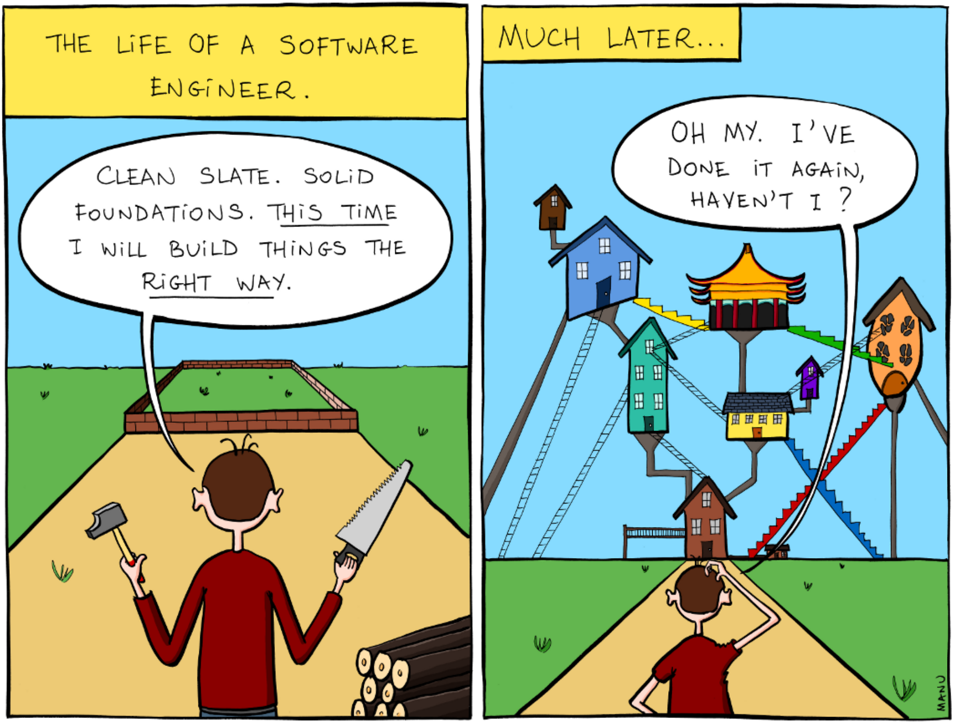
The Advanced Formulation

```haskell
type Lens a b =
  forall f. Functor f => (a -> f a) -> (b -> f b)
```

What Have We Gained?

- Composition for free!

  `set (it . boss . name)` *"Grace Hopper"* `company`
- Mocking for free!
  lenses are ordinary functions, so can be swapped out


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

Calculator Revisited

```
data Expr a t =
    Literal a
  | Var t
  | Sum (Expr a t) (Expr a t)
```

Calculator Revisited

```
data Expr a t =
     Literal a
   | Var t
   | Sum (Expr a t) (Expr a t)
```

```
> :t Sum
Sum :: Expr a t -> Expr a t -> Expr a t
```

Calculator Revisited

```
data Expr a t where
  Literal :: a -> Expr a t
  Var     :: t -> Expr a t
  Sum     :: Expr a t -> Expr a t -> Expr a t
```

Calculator Revisited

```
data Expr a t where
  Literal :: a -> Expr a t
  Var     :: t -> Expr a t
  Sum     :: Expr a t -> Expr a t -> Expr a t
```

- So far: `Expr a t` contains only `a` literals
- type `a` is constant in the whole expression
- What if we want heterogeneous operations?

  ```
  > :t even
  even :: Integral a => a -> Bool
  ```

Calculator Revisited

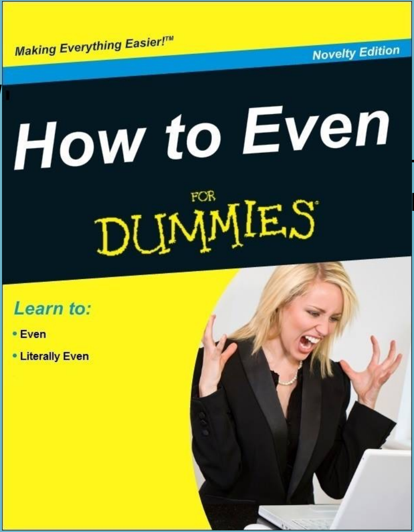
```
data Expr a where
  Literal :: a -> Expr a
  Sum     :: Expr a -> Expr a -> Expr a
```

Calculator Revisited

```
data Expr a where
  Literal :: a -> Expr a
  Sum     :: Num a => Expr a -> Expr a -> Expr a
  Even    :: Integral a => Expr a -> Expr Bool
```

Calculator Revisited

```
data Expr a w
  Literal ::
  Sum     ::                            -> Expr a
  Even    ::                            pr Bool
```



Calculator Revisited

```haskell
data Expr a where
  Literal :: a -> Expr a
  Sum     :: Num a => Expr a -> Expr a -> Expr a
  Even    :: Integral a => Expr a -> Expr Bool
```

Calculator Revisited

```
data Expr a where
  Literal :: a -> Expr a
  Sum     :: Num a => Expr a -> Expr a -> Expr a
  Even    :: Integral a => Expr a -> Expr Bool
  Cast    :: (a -> b) -> Expr a -> Expr b
```

A Fancy Calculator

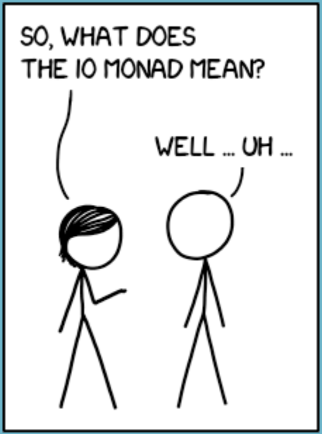
- we now have a datatype which represents (some) arithmetic operations
- Apart from evaluating, what can we do with it?

A Fancy Calculator

- we now have a datatype which represents (some) arithmetic operations
- Apart from evaluating, what can we do with it?
    - print
    - count operations
    - optimize

A Fancy Calculator

- we now have a datatype which represents (some) arithmetic operations
- Apart from evaluating, what can we do with it?
  - print
  - count operations
  - optimize



MY HOBBY:
ASKING SEASONED HASKELL DEVELOPERS ABOUT THE MEANING OF IO

The Essence of IO

- a representation of a computation
- ... which interacts with the world

The Essence of IO

- a representation of a computation
- … which interacts with the world

The Essence of IO

- a representation of a computation
- ... which interacts with the world
- in Haskell: may contain all sorts of effects
- in GHC: opaque, non-inspectable

The Essence of IO

- a representation of a computation
- ... which interacts with the world
- in Haskell: may contain all sorts of effects
- in GHC: opaque, non-inspectable
- but: a better world is possible

IO as a DSL

- calculator: datatype with one constructor per operation
- terminal application: datatype with one constructor per operation?
  - read from standard input
  - write to standard output

IO as a DSL

- calculator: datatype with one constructor per operation
- terminal application: datatype with one constructor per operation?
  - read from standard input
  - write to standard output
  - open file
  - read from file
  - ...

IO as a DSL

- calculator: datatype with one constructor per operation
- terminal application: datatype with one constructor per operation?
  - read from standard input
  - write to standard output
  - open file
  - read from file
  - ...

A Datatype for Terminal IO

```
data Terminal a where
  ReadLine :: Terminal String
  WriteLine :: String -> Terminal ()
```

Simulating IO

```
type IO a = PauseT (State RealWorld) a

data RealWorld =
  RealWorld {
    workDir :: FilePath
  , files :: Map File Text
  , isPermitted :: FilePath -> IOMode -> Bool
  , handles :: Map Handle HandleData
  , nextHandle :: Integer
  , user :: User
  , mvars :: Map Integer MValue
  , nextMVar :: Integer
  , writeHooks :: [Handle -> Text -> IO ()]
  }
```

Conclusion

- FP provides a set of techniques for abstraction over evaluation
- Use them!

Conclusion

- FP provides a set of techniques for abstraction over evaluation
- Use them!

> “Premature evaluation is the root of all evil.”

Q & A

larsr_h larsrh

Image Credits

- Manu Cornet, `http://www.bonkersworld.net/building-software/`
- Randall Munroe, `https://xkcd.com/1312/`
- Thomas Kluyver, Kyle Kelley, Brian E. Granger, `https://github.com/ipython/xkcd-font`