Introduction to Haskell Hacking

Joe Near and **Jean Yang**

IAP 2010: So You've Always Wanted to Learn Haskell?

January 25, 2010

Schemers, welcome to the other side

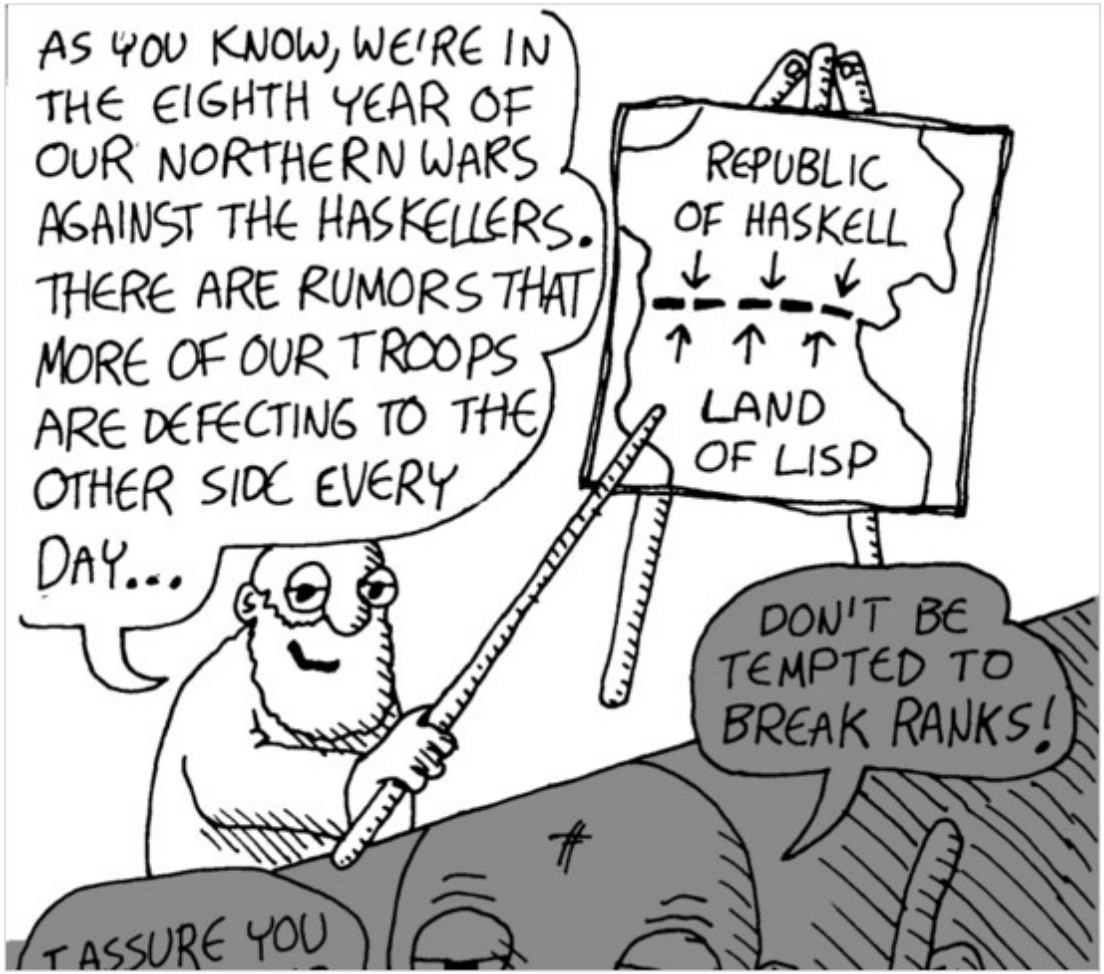


Figure: The long war between the untyped and types communities.

Consider quicksort

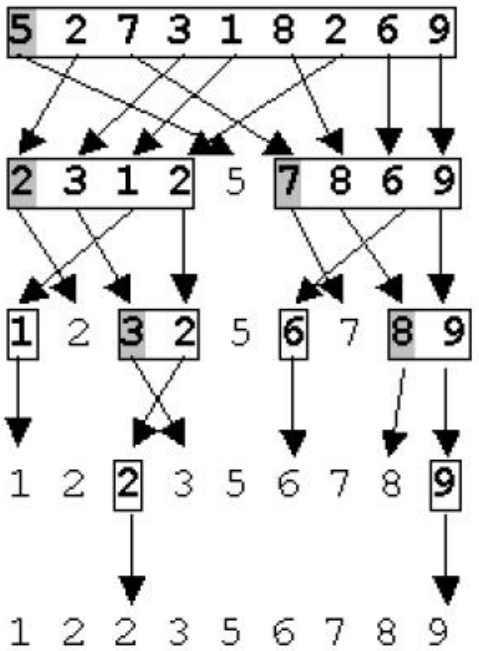


Figure: Graphical representation.

An $O(n \log n)$ divide-and-conquer sorting algorithm.

1. Pick a *pivot*.
2. Reorder the list so all elements around pivot.
3. Recursively sort the sub-lists.

Quicksort in C

```
void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;
  if (lo < hi) {
    l = lo; h = hi; p = a[hi];
    do {
      while ((l < h) && (a[l] <= p)) l = l+1;
      while ((h > l) && (a[h] >= p)) h = h-1;
      if (l < h) {
          t = a[l]; a[l] = a[h]; a[h] = t;
      }
    } while (l < h);
    a[hi] = a[l]; a[l] = p;
    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

Quicksort in Haskell

```
qsort []     = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
  where
    lesser  = [ y | y <- xs, y < p ]
    greater = [ y | y <- xs, y >= p ]
```

Why Haskell matters

Purely functional[1]

- Brevity.
- Easy of understanding.
- Built-in memory management.
- Code re-use.

Strongly, statically typed

- Ease of development.
- Ease of understanding.
- Ease of maintenance.

[1] From an essay by Sebastian Sylvan. (There is also John Hughes's "Why Functional Programming Matters"–a must-read!)

Today's plan

1. What is Haskell?
2. Haskell's type system.
3. Type classes for overloading.
4. Monads for effects.
5. Compiling and running Haskell programs.

Find these slides online at

people.csail.mit.edu/jeanyang/courses/haskell/lect1.pdf

Vocabulary check

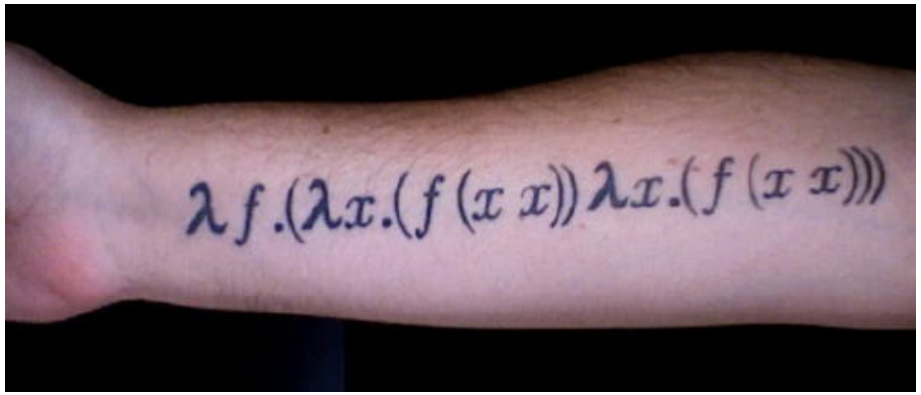


Figure: Y combinator tattoo.

- Functional
- Higher-order functions
- Pure
- Effect
- Static typing
- Polymorphism

The gentle introduction to Haskell

Whitespace sensitive

Purely functional

Idiom: recursion rather than iteration

Function declaration and application.

```
fac n = if n == 0 then 1 else n * fac (n-1)
```

Idiom: map and fold with higher-order functions

Use of Prelude list function and anonymous λ-function.

```
inc1 lst = map (\x -> x + 1) lst
```

Pure: all effects captured in monads

```
main :: IO ()
main = putStrLn "Hello world!"
```

Strongly, statically typed with polymorphism

Simple types

```
  5 :: Integer
'a' :: Char
```

Function types

```
inc :: Integer -> Integer
inc x = x + 1
```

Polymorphic types

```
length          :: [a] -> Integer
length []       =  0
length (x:xs)   =  1 + length xs
```

Lazily evaluated

Some definitions

- **Call by name.** Args directly substituted into function body.
- **Call by need.** Memoized version of call by name.
- **Haskell's lazy evaluation.** Store *thunks* in heap and evaluate only when necessary.

A cool consequence: infinite data structures[2]

```
fib = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

```
zip (x:xs) (y:ys)       = (x,y) : zip xs ys
zip  xs     ys          = []
```

[2] Can also do this with `delay` in Scheme.

Pretty fast for a high-level language

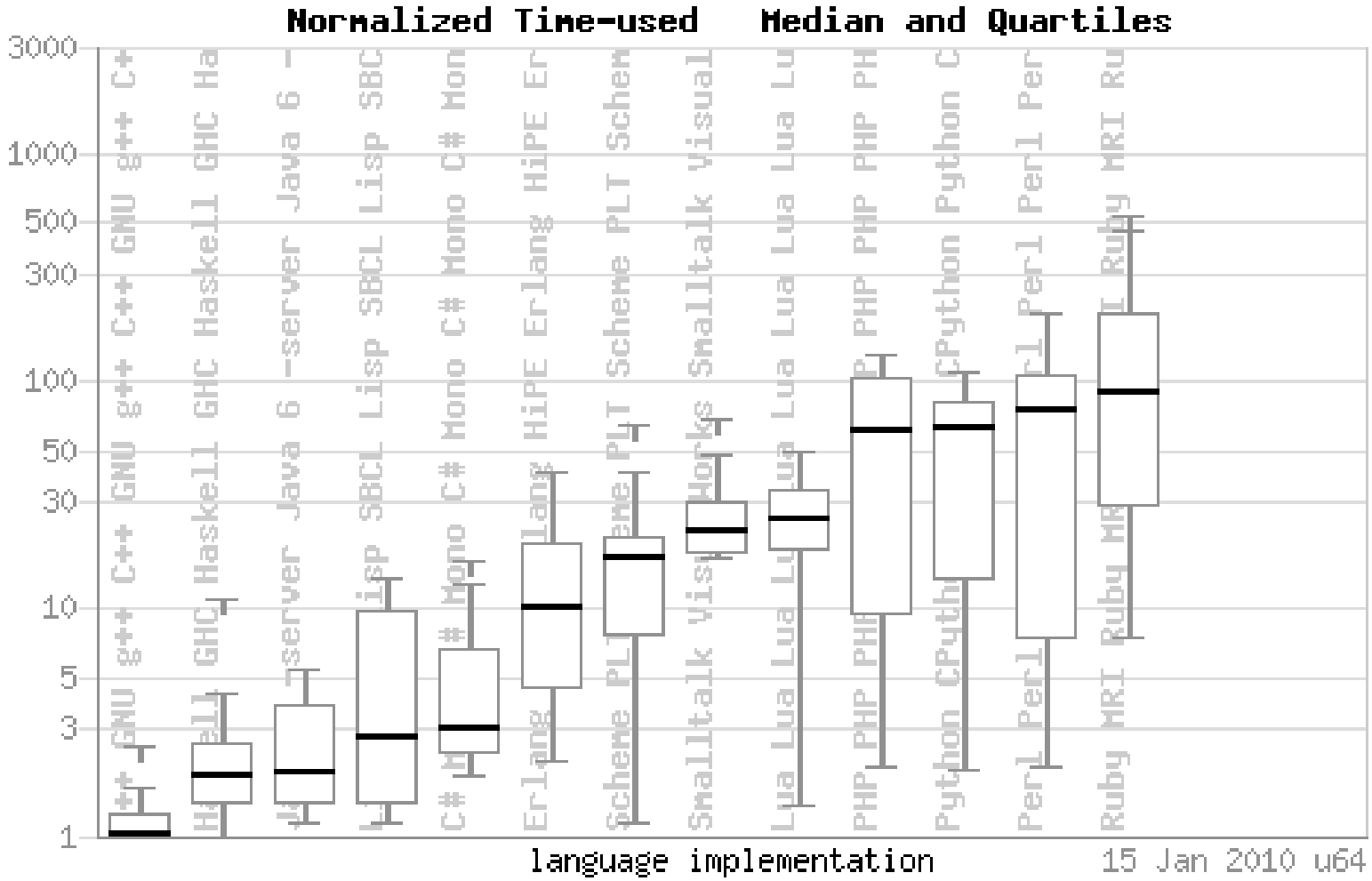


Figure: Numbers from the Debian language shootout benchmarks.

All together now

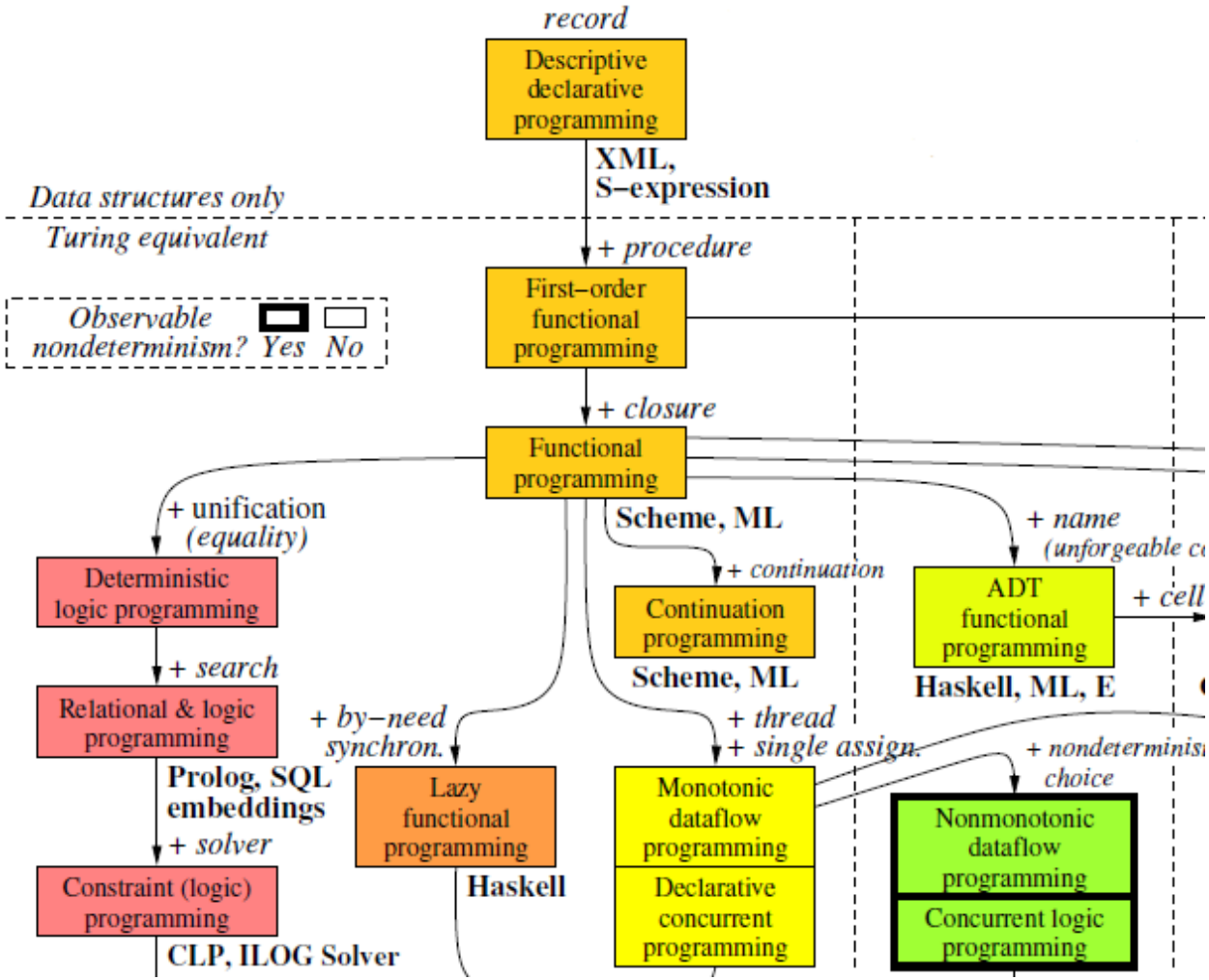


Figure: From Peter Van Roy's programming paradigms chart.

Back to quicksort

Our original program

```
qsort []     = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
  where
    lesser  = [ y | y <- xs, y < p ]
    greater = [ y | y <- xs, y >= p ]
```

Alternatively with list comprehensions

```
qsort []     = []
qsort (x:xs) =
  qsort (filter (< x) xs) ++
  [x] ++
  qsort (filter (>= x) xs)
```

Types and typing restrictions

Haskell's type system

Pedantic version

A restriction of System F_ω (polymorphic λ-calculus) to rank-1 polymorphic types with a version of Hindley-Milner type inference. Type inference and checking are both decidable.

Simpler version

- We only have quantifiers at the outermost level of types.
- There are some restrictions on when we can infer a fully polymorphic type.

User-defined data types

Predefined and user-defined

```
data Bool = False | True
```

```
data Color = Red | Green | Blue | Indigo | Violet
```

Polymorphic definitions

```
data Point a = Pt a a
```

Type synonyms

```
type String             = [Char]
type Name               = String
data Address            = None | Addr String
```

Pattern matching

As-patterns

```
f (x:xs) = x:xs
f s@(x:xs) = x:s
```

Wild-cards

```
head (x:_)  = x
tail (_:xs) = xs
```

Case expressions

```
take m ys = case (m,ys) of
              (0,_)      -> []
              (_,[])     -> []
              (n,x:xs)   -> x : take (n-1) xs
```

Monomorphism restriction

Cannot overload a function without an explicit type signature[3].

```
f1 x = show x          -- Allowed.
```

```
f2 = \x -> show x      -- Not allowed.
```

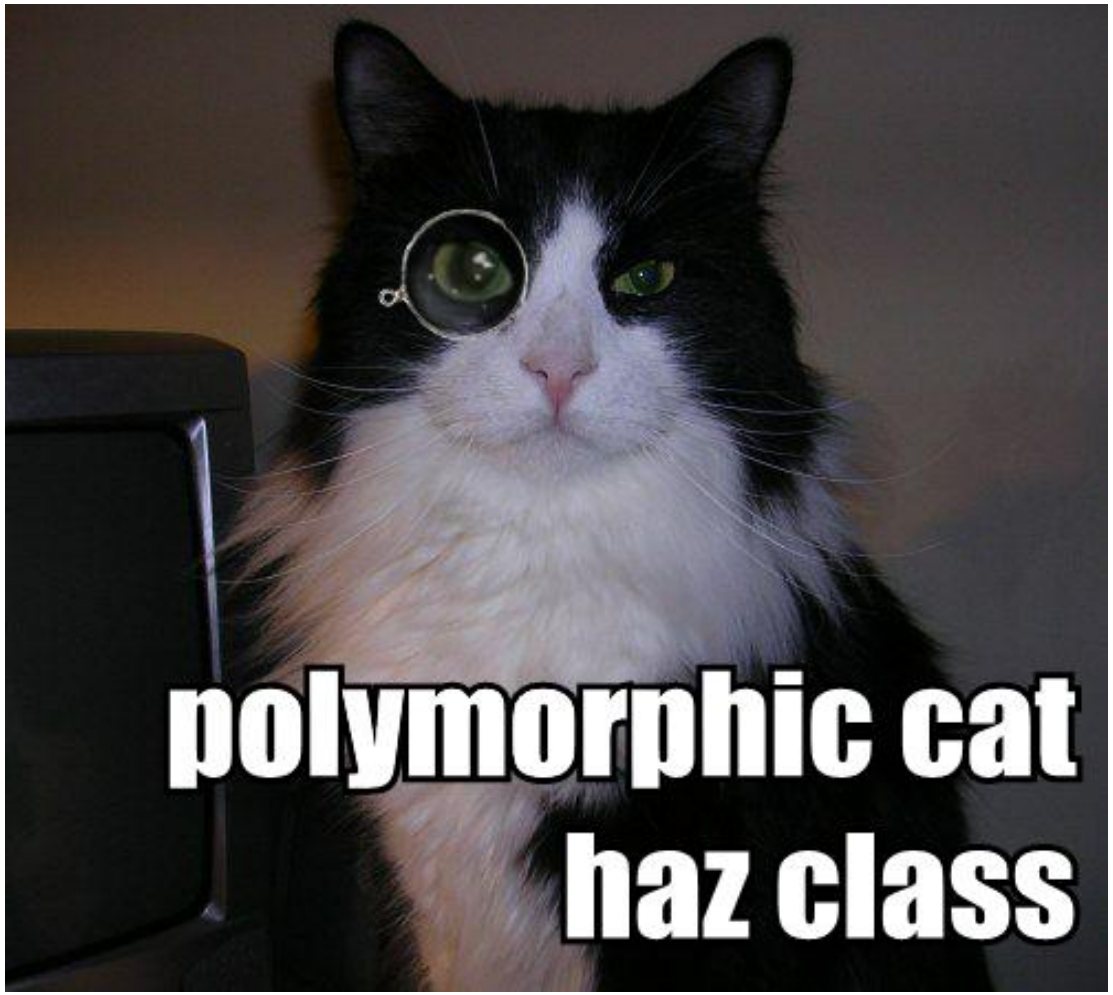
```
f3 :: (Show a) => a -> String
f3 = \x -> show x      -- Allowed.
```

```
f4 = show              -- Not allowed.
```

```
f5 :: (Show a) => a -> String
f5 = show
```

[3] *But Haskell has a flag for everything!* Can turn off restriction with flag `-XNoMonomorphismRestriction`.

Ad-hoc polymorphism with type classes



qsort: not parametrically polymorphic

```
qsort []     = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
  where
    lesser  = [ y | y <- xs, y < p ]
    greater = [ y | y <- xs, y >= p ]
```

Does `qsort` have type `[a] -> [a]`?

No–need types a for which operations $<$ and $\geq$ are defined!

Type classes give us *bounded* parametric polymorphism

Haskell's `Ord` class

```
class (Eq a) => Ord a where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

`qsort`'s type signature

```
qsort :: (Ord a) => [a] -> [a]
```

Type classes by example: the Eq class

(Reduced) class definition

```
class Eq a where
  (==)                    :: a -> a -> Bool
```

Defining instances

```
instance Eq Integer where
  x == y                  =  x `integerEq` y
```

```
instance (Eq a) => Eq (Tree a) where
  Leaf a          == Leaf b          =  a == b
  (Branch l1 r1) == (Branch l2 r2)  =  (l1==l2) && (r1
      ==r2)
  _               == _               =  False
```

Useful Haskell type classes

Eq

```
class Eq a where
  (==), (/=)              :: a -> a -> Bool
  x /= y                  =  not (x == y)
```

Read and Show

```
class Read a where
    ...
    read          :: String -> a
    ...
class Show a where
    ...
    show          :: a -> String
    ...
```

So we said Haskell is purely functional...

A functional programming pattern for handling state

Store state in a value

```
data S = S { intS :: Integer, strS :: String }
```

Pass this value around

```
evaluate :: (S, Exp) -> (S, Exp)
evaluate (state, exp) =
  if state.intS == 0
    then {-- Do something. --}
    else {-- Do something else. --}
```

Monads are an abstraction for storing state

```
evaluate :: S Exp -> S Exp
```

Monads

Monads for all effectful computation

- Language support for carrying around state explicitly.
- Requires definitions for how to initialize a value in this state and how to compute new values in the context of the state.
  - How to "lift" something into the monad (`return`).
  - How to sequence operations within the monad (>>=, or bind).

Monads defined as a type class

```
infixl 1  >>, >>=
class  Monad m  where
    (>>=)            :: m a -> (a -> m b) -> m b
    (>>)             :: m a -> m b -> m b
    return           :: a -> m a
    fail             :: String -> m a
```

Monad use example: Maybe

Maybe type

```
data Maybe a = Nothing | Just a
```

Unnecessary casing

```
case ... of
  Nothing -> Nothing
  Just x  -> case ... of
```

Monad definition

```
instance Monad Maybe where
    return          = Just
    fail            = Nothing
    Nothing  >>= f  = Nothing
    (Just x) >>= f  = f x
```

Syntactic sugar: do

do for sequencing monadic operations

```
do e1 ; e2        =        e1 >> e2
do p <- e1; e2    =        e1 >>= \p -> e2
```

Example with Maybe

```
data MailPref = HTML | Plain
data MailSystem = ...

getMailPrefs :: MailSystem -> String -> Maybe MailPref
getMailPrefs sys name =
  do let nameDB = fullNameDB sys
         nickDB = nickNameDB sys
         prefDB = prefsDB sys
  addr <- (lookup name nameDB) `mplus` (lookup name
      nickDB)
  lookup addr prefDB
```

Practical monad use: IO

Interacting with the command line

Haskell has the following built-ins:

```
getChar :: IO Char
putChar :: Char -> IO ()
```

We can write the following function:

```
getLine       :: IO String
getLine       =  do c <- getChar
                    if c == '\n'
                       then return ""
                       else do l <- getLine
                               return (c:l)
```

Haskell's IO library

File processing, exception handling, and more.

Compiling with the Glasgow Haskell Compiler (GHC)

Compiling "Hello world"

```
$ ghc --make hello.hs}
[1 of 1] Compiling Main                ( hello.hs, hello.o )
Linking hello ...
```

Fancier compilation

GHC's `make` will track dependencies for you.

```
ghc -isrc --make -main-is Main src/Main.hs -o Main -hidir
out -odir out -o ./simple}
```

GHCi

GHC's interactive interpreter–allows you to load modules, evaluate expressions, and check types.

Important logistical issues

Program entry point

All Haskell programs need a `main` function which has type `IO ()` ("IO unit").

```
main :: IO ()
main = putStrLn "Hello world!"
```

Compiling modules

- GHC's `make` searches for module `M` in the file `M.hs`.
- `make` searches for module `Dir1.Dir2.M` in the file `Dir1/Dir2/M.hs`.

A Haskell program

```
{- A Haskell file. -}
module Main                    -- Module name
  (main)                       -- Exported function
where

{- Namespace imports. -}
import List (find)             -- Selective import
import SomeLibrary.M as M      -- Aliased import
import SomeLibrary.OtherM      -- A plain old import

-- | main, a top-level function.
main :: IO ()
main = putStrLn strToPrint
where
  -- | A valued defined in the scope of main.
  strToPrint = "Hello world!"
```

Have fun!

Figure: Oleg Kiselyov as a λ-cat.

Until tomorrow...

Tomorrow

- Discussions of practical programing in Haskell.
- Looking at larger Haskell programs.

Questions?

{ jeanyang, jnear }@csail.mit.edu.