CPS506 - Comparative Programming Languages

Haskell

Dr. Dave Mason
Department of Computer Science
Ryerson University

Overview

- Paradigms
  - Functional
    - Fully - side effects are restricted to monads
    - Lazy evaluation outside of monads
    - staticly typed
  - Imperative subset
    - command line in some compilers
- Syntax
  - mathematical
  - Infix multi-precedent operators (standard 10 levels, definable)
  - control structures are expressions
  - no special forms except definitions
  - all functions have arity 1, currying
  - indentation matters in file
- Semantics
  - everything is lazy function application
  - everything returns a value, control are parts of expressions
  - parameters are call-by-need
  - richly staticly typed - parametric polymorphism
- Pragmatics
  - native compilers

Data

- "Normal" Values
  - `4*3+5*2`
  - `[1,2,3]`
- Function Values
  - First Class - variables, parameters, returns, lists
  - `let double x = x + 2` - interactive
  - `double x = x + 2` - non-interactive
- Types
  - strongly typed
  - type inference - rarely need to give type
  - get the types right, program probably close to correct

Running Haskell

- ghci is the interactive interpreter
- `ghc` is the compiler
- `man ghc` - 2500 line manual page on Linux/MacOsX
- online User's Guide

Examples

```
let a = 7
let f x = 5
let id x = x
f a
id a
id f a
id id id a
let l = [1,2,3,4]
:t l
map id l
map f l
let adda x = x+a
map adda l
:t a
:t adda
:t id
```

Examples... 2

```
let inc x = x+1
map inc l
:t map inc l
:t (map adda)
let madda = map adda
madda l
let f x y = x - y
let f4 = f 4
map f4 l
4+5*6
f4 5*6
:h
:browse Prelude
:e
double 2000000000
double 2000000000000
```

Examples... 3

```
:t map f l
let g x f = f x
map (g 4) (map f l)
:info (+)
let second x = head ( tail x )
let second x = head $ tail x
let second   = head . tail
map (g 4) . map f $ l
map ($ 4) . map f $ l
let third x = head ( tail ( tail x ) )
let third x = head $ tail $ tail x
map third ["asdf", "qwer", "1234"]
map (\x -> head $ tail $ tail x) ["asdf", "qwer", "123
map (head . tail . tail) ["asdf", "qwer", "1234"]
Open pipe.hs
["asdf", "qwer", "1234"] |> tail |> tail |> head
```

Examples... 4

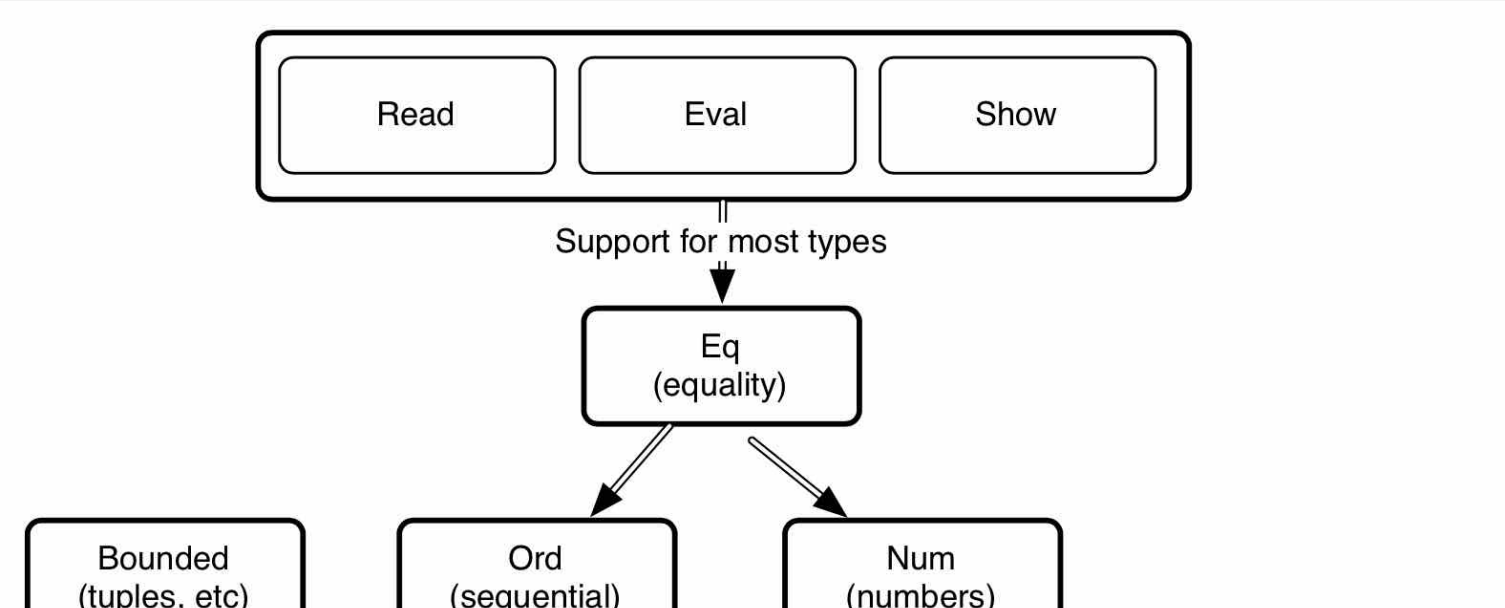
```
:set +t
Open partial.hs
Open map.hs
Open factorial.hs Open fact_with_guard.hs
Open fib.hs Open fib_pair.hs Open fib_tuple.hs Open fi

lists, ranges, list comprehensions
Open lists.hs
Open my_range.hs
filter, foldl, foldr
Open all_even.hs
```

Types

- data
- type
- class - instance

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



Examples... 5

```
Open triplet.hs
Open cards.hs
Open cards-with-show.hs
Open tree.hs
Open tree-read.hs
Open factors.hs
```

Monads

- functions passing state as an argument
- external world is the state for IO monad

```
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

    m >> k           =  m >>= \_ -> k
```

- Open drunken-pirate.hs
- Open drunken-pirate.monad.hs
- Open io.hs

List Monad

```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

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

instance Monad [] where
    (>>=) :: [a] -> (a -> [b]) -> [b]

do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)

[1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >
   (\r -> case r of True -> return (x,y)
                    _    -> fail "")))
```

Open password.hs

Maybe Monad

- `Maybe` is used for conditional computation
- `let div x y = if y/= 0 then Just (x/y) else Nothing`

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

instance Monad Maybe where
    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    (Just x) >>= f = f x
    return = Just
    fail = Nothing
```

Package Manager

- `cabal`
- `hackage`
- `cabal install http-client`
- problem with recent cpp (e.g. clang) - on MacOSX

Unit Testing

- `import Test.HUnit`

Evaluation

- Simplicity
  - size of the grammar
  - complexity of navigating modules/classes
  - groking the type system
- Orthogonality
  - number of special syntax forms
  - number of special datatypes
- Extensibility
  - functional
  - syntactically
  - defining literals
  - overloading