Haskell (pure and lazy, yet functional)

Andrey Kotlarski

9.I.2012

Outline

Quick overview

The language from bird's-eye

Pros and cons

Resources

Fun

History and stuff

- Unification of efforts in lazy functional programming
- A lot of theory underneath
- Academy driven, cutting edge research
- Evolving standard
- Glasgow Haskell Compiler being the canonical implementation
- Avoid Success at All Costs

History and stuff

- Unification of efforts in lazy functional programming
- A lot of theory underneath
- Academy driven, cutting edge research
- Evolving standard
- Glasgow Haskell Compiler being the canonical implementation
- Avoid Success at All Costs

I fear that Haskell is doomed to succeed.
– C.A.R. Hoare

Theoretical base

- (Typed) λ -calculus
- Category theory
- Hindley-Milner(-Damas) type inference

Technical merits

- Purely functional
- Lazy (non-strict)
- Polymorphic strong static typing

Technical merits

- Purely functional
- Lazy (non-strict)
- Polymorphic strong static typing
- Elegant (sort of), math inspired syntax

Pure functional?

- Program is a tree of nested expresions
- Functions are the base building unit
- No side effects by default
    - like in mathematic functions

Functions

- Pattern matching

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

- Curring
  - Function of N arguments is actually an application of N 1-argument functions

```
map (5 +) [1..10]
```

- Composition

```
map (negate . sum . tail) [[1..5],[3..6],[1..7]]
```

Lazy?

- Evaluation order
- Thunks
  - Delayed computations

```
int* take(int amount, int collection[])
{...}
```

  - Don't compute anything until/unless required

```
take 10 $ map (5 +) [1..]
```

Strong static typing?

- Each expression has a type known at compile time
  - so do functions
- Our types determine a theorem and compiling is a proof of its correctness within the Haskell world
  - common theme for such advanced type systems
- Polymorphic types

  Prelude> :t filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

Algebraic data types

- Union of possible values or value constructors

```
data Bool = False | True

data Car = Car {model :: String
               , year :: Int
               , burnTime :: Int
               } deriving (Show)
```

- Type parameters

```
data Maybe a = Nothing | Just a

data Tree a = EmptyTree | Node a (Tree a)
                              (Tree a)
            deriving (Show, Read, Eq)
```

Typeclasses

- Interfaces sort of
- If it quacks like a duck, it's a duck

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

instance (Eq m) => Eq (Maybe m) where

    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

I/O vs Purity

- The IO Monad
- Reverse words

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

- Cool one-liner

```
main = interact $ unlines .
    filter ((>200) . length) . lines
```

Functors

- Don't confuse with C++ ;-)
- Iterable?
- Lift ordinary function to operate on boxed value

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map

instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

Applicative

- Beefed up functors
- Sequence of several boxed actions

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b

instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

- pure f <*> x ≡ fmap f x

Monoids

- Associative binary function + identity value
- Accumulate a boxed value from several boxes

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty

instance Monoid [a] where
    mempty = []
    mappend = (++)
```

Monads

- Beefed up applicatives

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

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg

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

Benefits

- The pervasive type system gives a lot of information to the compiler
  - many types (pun intended) of bugs are prevented at compile time
  - much room for automatic optimizations
    - Data Parallel Haskell
  - secure and formally verifiable programs
- Side effects are not the norm and are explicitly specified and controlled
  - easier to reason about
  - better concurrency state
    - how many languages have a working STM implementation?

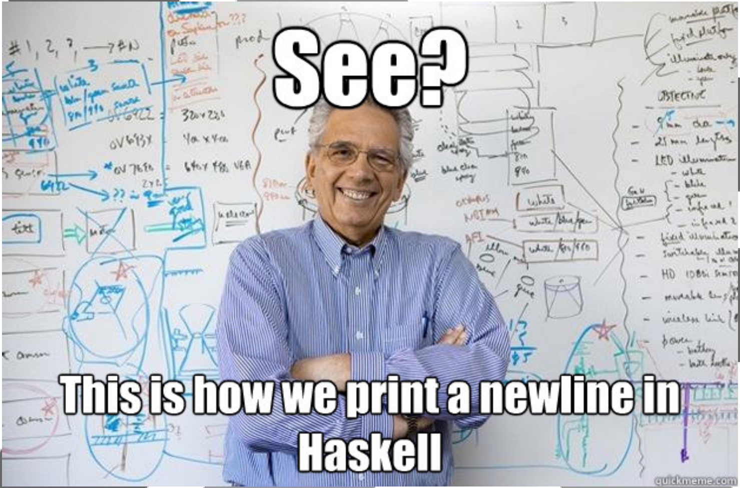
Problems

- There are cases where static typing may not be natural
- For huge systems, you may paint yourself in the corner if having somehow wrong base
- Laziness makes order of evaluation non-obvious
  - trouble with performance bottlenecks identification
  - memory spikes

Links & books

- Official site
- Learn You a Haskell for Great Good!
- The Haskell Programmer's Guide to the IO Monad - Don't Panic.
- Real World Haskell
- Great list of tutorials
- Recent interview with Simon Peyton-Jones

Why so serious?



- The Evolution of a Haskell Programmer