Programming in Haskell

Patrick Hurst, sipb-iap-haskell@mit.edu

January 21, 2013

About Haskell

- Background
- Language paradigms
    - Lazy
    - Functional
    - Strong, inferred, static types

It makes you think!

Other reasons

- If it compiles, it's probably right
- Hoogle, Hackage, cabal
- Good library support: QuickCheck, Parsec, HXT, snap
- Fast!
- Very friendly community: irc.freeenode.org/#haskell, haskell-cafe@haskell.org

Getting set up

- Haskell Platform: http://hackage.haskell.org/platform/
- Don't use OS package manager
- vi/emacs modes

Getting started

- Simple math works the same, e.g. `(2*3)+5*(7-228)`
- ‘div‘ vs (/):

```
Prelude> 14 / 5
2.8
Prelude> 14 ‘div‘ 5 --or div 14 5
2l
```

- Negative number syntax is tricky
- Assignment in ghci requires ’let’

Lists

- Lists are linked lists, like in LISP.
  `[1,2,3,4] == 1 : (2 : (3 : (4 : [])))`
- Functions for working with lists:
  `head, tail, init, last, (!!)`
- Lists are homogenous, but you don't want homogenous lists
- Ranges: `[1..10]`, `[1..]`, `['A'..'Z']`. Be careful with floating point!

Tuples

- 'pairs', 'triplets' of data that don't need their own type
- Heterogenous, but constant length
- No accessors aside from ; `fst :: (a,b) -> a` and `snd :: (a,b) -> b`

Functions

- No commas or parens:

```
Prelude> head [2,3,4]
2
Prelude> take 3 [1,2,3,4,5,6,7]
[1,2,3]
```

- Example:

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

- Pattern-matching:

```
not True = False
not False = True
and True True = True
and False _ = False
```

- Case statements
- Let and where bindings, useful for 'local' functions

Types

- Everything has a type.
  `''Hello'' :: [Char], True :: Bool`
- Numeric literals are 'special': `1 :: Num a => a`
- Function types look like `(++) :: [a] -> [a] -> [a]`.
- Common types: `Char, Bool, String, Int, Integer, ()`
- No casting, no subtyping.

Data types

- `data Person = Person String Int`
- `name (Person n _) = n`
-
  ```
  data Person = Person { name :: String, age :: Int }
  ```

Parameterized types

- What's the type of []? `1 : []`, `True : []`, `‘‘hi’’ : []` all legal
- `head :: [a] -> a`
- Reasoning from types
  - `f :: [a] -> Int`
  - `g :: a -> b -> a`

Combining it all

Parameterized data types!

- `data Tree a = Node a` Branch a (Tree a) (Tree a)—
- `treeMap :: (a -> b) -> Tree a -> Tree b`

IO

- IO cannot be accidentally mixed with non-IO!
- `getLine :: IO String`
- `putStrLn :: String -> IO ()`
- `main :: IO ()` is what makes the actions happen.

Control flow

- No for or while loops! Use recursion instead.

```
fib n | n < 1 = 0
      | n = 1 = 1
      | otherwise = fib (n-1) + fib (n-2)
```

- Alternatively,

```
fib n | n < 1 = 0
      | otherwise = fib' n 0 1
   where fib' 0 _ b = b
         fib' n a b = fib' (n-1) b (a+b)
```

- Function arguments as state

Modules

- Defining
  - `import Data.List`
  - `import Data.List (intersperse)`
  - `import Data.List as L`
- Creating:

  ```
  module MyModule (myFunc, anotherFunc) where
  ```

Control flow and IO

- Writing a simple get-the-number program in guess.hs
- Recurse instead of while loops