Haskell

Samy Al Bahra

GWU

January 21, 2009

Factorial

```
f n | n ≡ 1 = 1 | otherwise = n * f (n - 1)

f n = product [1 .. n]

f n = scanl (*) 1 [1 ..] !! n
```

```
Prelude> f 100
933262154439441526816992388562667004907159682643816214685929
638952175999932299156089414639761565182862536979208272237582
511852109168640000000000000000000000
```

Satisfiability

```
True --> False = False
_    -->     _ = True

iF f = [ (f p q k, (p, q, k)) | p ∈ v, q ∈ v, k ∈ v]
       where
       v = [True, False]

g m s f = [ m n | n ∈ filter s $ iP f]
vF = and · g fst (λ_ → True)
mF = g snd ((True ≡) · fst)
```

Satisfiability

```
*Main> iF (λp q r → (p ∨ q) --> r --> (r --> p))
[(True,(True,True,True)),(True,(True,True,False)), ...
*Main> mF (λp q r → (p ∨ q) --> r --> (r --> p))
[(True,True,True),(True,True,False),(True,False,True), ...
*Main> vF (λp q r → (p ∨ q) --> r --> (r --> p))
False
```

Sorting

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

```
*Main> qsort [10, 9 .. 1]
[1,2,3,4,5,6,7,8,9,10]
*Main> qsort [20, 19 .. 1]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Features

- Software Engineers
  - Highly composable
  - Lazy evaluation
  - List comprehensions
  - Pattern matching
  - Purely functional
  - Polymorphism
  - and more...
- Additional Utilities
  - Digital signal processing libraries
  - GNU Scientific Library interface
  - GNUplot interface
  - Fixed arbitrary precision reals
  - Abstractions for exact real arithmetic
  - Arbitrary precision integers
  - and more...

Introduction

```
identity :: Bool → Bool
identity p = p
```

Multiple Arguments

```
xor :: Bool → Bool → Bool
xor p q = if (p ≡ q) then False else True
```

Guards

```
xor1 :: Bool → Bool → Bool
xor1 p q | p ≡ q = False
         | otherwise = True
```

Pattern Matching

```
xor2 :: Bool → Bool → Bool
xor2 p q = case (p, q) of
             (True, True) → False
             (False, False) → False
             _ → True
```

Pattern Matching

```
xor3 :: Bool → Bool → Bool
xor3 True True = False
xor3 False False = False
xor3 _ _ = True
```

Infix versus Postfix

```
Prelude> xor True False
True
Prelude> True `xor` False
True
```

Curried Application

- Haskell implements **typed lambda calculus**, every function is a function of 1 argument.
- **xor** is a function that takes a **Bool** and returns a function which takes a **Bool** which in turn evaluates to a **Bool**.

```
Prelude> :t xor
xor :: (Bool → (Bool → Bool))
Prelude> :t xor True
xor :: (Bool → Bool)
Prelude> :t xor True True
xor :: (Bool)
```

Curried Application

- Currying makes function composition easy

```
xorList :: [Bool] → Bool
xorList = foldr1 xor
```

Lambda Abstractions

- Equivalent to "anonymous functions"
- Useful for higher order functions

```
threeOrFive :: [Integer] → [Integer]
threeOrFive = filter (λx → (rem x 3 ≡ 0) ∨ (rem x 5 ≡ 0))

Prelude> threeOrFive [1 .. 10]
[3,5,6,9,10]
Prelude> (λ(_, _, z) → z) (1, 2, 3)
3
```

Algebraic Data Types

- ADTs are foundational to Haskell
- Consist of data constructors and values of other types

```
data Person = Name String String
```

```
*Main> :t Name "Samy" "Bahra"
Name "Samy" "Bahra" :: Person
*Main> :t Name "Samy"
Name "Samy" :: String → Person
*Main> :t Name
Name :: String → String → Person
```

Algebraic Data Types

- Multiple constructors are allowed

```
data Person = Name String String | Social Integer

*Main> :t Social
Social :: Integer → Person
*Main> :t Social 591929191
Social 591929191 :: Person
```

Recursively Defined ADTs

```
data Tree = Node Int Tree Tree | Leaf

*Main> :t Node 3 Leaf Leaf
Node 3 Leaf Leaf :: Tree
*Main> :t Node 3 (Node 4 Leaf Leaf) (Node 5 Leaf Leaf)
Node 3 (Node 4 Leaf Leaf) (Node 5 Leaf Leaf) :: Tree
```

Recursively Defined ADTs

```
data Tree = Node Int Tree Tree | Leaf

search :: Int → Tree → Bool
search _ Leaf = False
search m (Node v left right)
  | m ≡ v = True
  | m < v = search m left
  | m > v = search m right

*Main> let t = Node 3 (Node 2 Leaf Leaf) (Node 5 Leaf (Node 9
*Main> search 3 t
True
*Main> search 4 t
False
*Main> search 9 t
True
```

Parametric Types

```
data Tree a = Node a (Tree a) (Tree a) | Leaf

*Main> :t Node "Test" Leaf Leaf
Node "Test" Leaf Leaf :: (Data.String.IsString t) => Tree t
*Main> :t Node 3 Leaf Leaf
Node 3 Leaf Leaf :: (Num t) => Tree t
*Main> :t Node 4.5 Leaf Leaf
Node 4.5 Leaf Leaf :: (Fractional t) => Tree t
```

Class Constraints

```
search :: (Ord a, Eq a) => a → Tree a → Bool
search _ Leaf = False
search m (Node v left right)
  | m ≡ v = True
  | m < v = search m left
  | m > v = search m right
```

Class Definition

```
class RGB a where
  rgb :: a → (Int, Int, Int)
```

Instance Definition

```
data Color = Red | Green | Blue | Yellow | Magenta | Cyan
                deriving (Show, Eq)

instance Num Color where
  Green + Red = Yellow
  Red + Blue = Magenta
  Green + Blue = Cyan
  _ + _ = Red

instance RGB Color where
  rgb Red   = (255, 0, 0)
  rgb Green = (0, 255, 0)
  rgb Blue  = (0, 0, 255)
```

```
*Main> Red + Blue
Pink
*Main> Green + Red
Yellow
*Main> Green + Pink
*** Exception: Prelude.undefined
```

The Tuple

```
data (,) a b = (,) a b
data (,,) a b c = (,,) a b c
data (,,,) a b c d = (,,,) a b c d
...

Prelude> (2,3)
(2,3)
Prelude> (,) 2 3
(2,3)
Prelude> (2,3,4)
(2,3,4)
Prelude> (,,) 2 3 4
(2,3,4)
```

The Tuple

```
Prelude> fst (2,3)
2
Prelude> snd (2,3)
3
Prelude> :t uncurry
uncurry :: (a → b → c) → (a, b) → c
Prelude> uncurry (+) (1, 2)
3
Prelude> :t curry
curry :: ((a, b) → c) → a → b → c
Prelude> curry snd 1 2
2
```

The List

```
data [a] = [] | a : [a] deriving (Eq, Ord)

Prelude> 1 : 2 : 3 : []
[1,2,3]
Prelude> 1 : (2 : (3 : []))
[1,2,3]
Prelude> [1, 2, 3]
[1,2,3]
Prelude> [1, 2, 3] !! 0
1
Prelude> [1, 2, 3] !! 1
2
```

The List

```
Prelude> [1, 2, 3] ++ [1, 2]
[1,2,3,1,2]
Prelude> 0 : [1, 2, 3]
[0,1,2,3]
Prelude> head [1, 2, 3]
1
Prelude> last [1, 2, 3]
3
Prelude> tail [1, 2, 3]
[2,3]
```

The List

```
Prelude> :t map
map :: (a → b) → [a] → [b]
Prelude> map (* 2) [1, 2, 3, 4]
[2,4,6,8]
Prelude> filter (< 2) [1, 2, 3, 4]
[2,4]
```

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

Maybe

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

```
Prelude> lookup 1 [(1, "M"), (2, "A"), (3, "O"), (10, "J")]
Just "M"
Prelude> lookup 10 [(1, "M"), (2, "A"), (3, "O"), (10, "J")]
Just "J"
Prelude> lookup 11 [(1, "M"), (2, "A"), (3, "O"), (10, "J")]
Nothing
```

Monads

```
Prelude> :i Monad
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
```

The Maybe Monad

```
(Just x) >>= k = k x
Nothing  >>= _ = Nothing

(Just _) >> k  = k
Nothing  >> _  = Nothing

return         = Just

fail _         = Nothing
```

The Maybe Monad

```
Prelude> (Just 4) >>= Just . (+ 1) >>= Just . (+ 2)
Just 7
Prelude> (Just 4) >> (Just "Poo") >> (Just "Wow")
Just "Wow"
Prelude> Nothing >> (Just 4) >> (Just "Poo") >> (Just "Wow")
Nothing
Prelude> return 3 :: Maybe Int
Just 3
```

The I/O Monad

```
Prelude> :t putStrLn
putStrLn :: String → IO ()
Prelude> putStrLn "One" >> putStrLn "Two" >> putStrLn "Three"
One
Two
Three
Prelude> :t readFile
readFile :: FilePath → IO String
Prelude> readFile "/etc/hostid" >>= putStrLn
4d1abb81-4950-11cb-acaa-e2a222173061
Prelude> openFile "/etc/hostid" ReadMode >>=
           (λh → hGetContents h >>= putStrLn >> hClose h)
4d1abb81-4950-11cb-acaa-e2a222173061
```

Do Syntax

```
function = do
  h ∈ openFile "/etc/hostid" ReadMode
  c ∈ hGetContents h
  putStrLn c
  hClose h
```

Do Syntax

```
function = do
  h ∈ openFile "/etc/hostid" ReadMode
  hGetContents h >>= putStrLn
  hClose h
```