EECS 776

Functional Programming and Domain Specific Languages

Professor Gill

The University of Kansas

August 29th 2016

KU

Comprehensions

Comprehensions in Haskell are like set comprehensions in math.

$$\{ x \mid x \leftarrow [1..10], \text{odd } x \}$$

```
GHCi> [ x | x <- [1..10], odd x ]
[1,3,5,7,9]
```

What might this do?

```
GHCi> [ x * x | x <- [1..10], odd x ]
????
```

Primes

```
GHCi> let factors n = [ x | x <- [2..n-1], n `mod` x == 0 ]
GHCi> factors 12
[2,3,4,6]
```

Now we can compute is a number is prime.

```
GHCi> let isPrime n = length (factors n) == 0
GHCi> isPrime 11
True
GHCi> isPrime 12
False
```

We can also compute with groups of primes.

Primes (2)

```
GHCi> let factors n = [ x | x <- [2..n-1], n `mod` x == 0 ]
GHCi> let isPrime n = length (factors n) == 0
```

We can also compute with groups of primes.

```
GHCi> filter isPrime [2..100]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
GHCi> map isPrime [2..12]
[True,True,False,True,False,True,False,False,False,True,False]
GHCi> [ if isPrime x then 'x' else '.' | x <- [2..50]]
"xx.x.x...x.x...x.x...x.....x.x.....x...x.x...x.....x.....x.x.....x...x.x....."
```

Primes improved

```
GHCi> let isPrime n = length (factors n) == 0
GHCi> :set +s
GHCi> isPrime 12345678
False
(8.98 secs, 2569789760 bytes)
```

How can we optimize this? The **null** function checks to see if a list is empty, so we use this instead of checking the length.

```
GHCi> let isPrime' n = null (factors n)
GHCi> isPrime' 12345678
False
(0.01 secs, 2096496 bytes)
```

We turn off the timing with :unset.

```
GHCi> :unset +s
```

Testing primes improved

We want to check if **isPrime** and **isPrime'** are equal.

```
GHCi> let isPrime n = length (factors n) == 0
GHCi> let isPrime' n = null (factors n)
```

So how do we compare a function?

A function is equal if for all inputs, the result is always the same.

```
GHCi> and [ isPrime' x == isPrime x | x <- [2..100]]
True
```

This is not comprehensive, but better than no tests. We will see how to do automatic random test-case generation later.

KU

Problem: Haskell sessions are lost on exit

We can take our declarations, and put them into a file. Then load the file.

```
module Primes where

factors n = [ x | x <- [2..n-1], n `mod` x == 0 ]
isPrime n = length (factors n) == 0
isPrime' n = null (factors n)
```

Note how the **let** has been dropped. This is because we only have declarations.

Back at the prompt, we can load this module.

```
Prelude> :l Primes
[1 of 1] Compiling Primes           ( Primes.hs, interpreted )
Ok, modules loaded: Primes.
*Primes> isPrime 12
False
```

Small Haskell Program

```haskell
-- This is a small Haskell program
module Main where

import System.Environment

main :: IO ()
main = do args <- getArgs
          printArgs args

-- printArgs print to stdout the input list,
-- one line at a time.
printArgs :: [String] -> IO ()
printArgs (arg:args) = do putStrLn arg
                          printArgs args
printArgs []         = return ()
```

```
$ ghc --make Main.hs
[1 of 1] Compiling Main ( Main.hs, Main.o )
Linking Main ...
$ ./Main Hello World
Hello
World
```

Types

Types are the distinguishing feature of Haskell-like languages

- What are types?

```
42 :: Int
```

- What is type-checking and type-inference?
  - Type-checking is checking if the types are **self-consistent**
  - Type-inference is checking **without being told** what the types are

Most modern languages have some form of type-checking, some have type-inference

Robin Milner

Robin was an outstanding and well-rounded computer scientist

- Machine-assisted proof construction (LCF)
- **Design of typed programming languages** (ML)

"Well-typed programs don't go wrong."

- Models of concurrent computation (CCS, π -calculus)

He was awarded the Turing Award in 1991

KU

Type systems in modern languages

Java - static typing

```
public int example(int x,double y) {
  String z = "Hello";
  ...
}
```

Statically typed languages are dependable but rigid

JavaScript - dynamic typing

```
function example(x,y) {
  var z = "Hello";
  ...
}
```

Dynamically typed languages are flexible but unreliable

The type system in Haskell

In Haskell, you can give the types of the values ...

```
sphereArea :: Double -> Double
sphereArea r = 4 * pi * r^2
```

... or let Haskell infer it ...

```
sphereArea r = 4 * pi * r^2
```

The type says "take a **Double**, return a **Double**"

So **r** is a **Double**, and **4 * pi * r^2** is a **Double**

```
Prelude> :l Example.hs
*Main> sphereArea 5
314.1592653589793
```

The type system in Haskell (GHCi)

You can also give the type in GHCi ...

```
Prelude> let sphereArea :: Double -> Double ; sphereArea r = 4 * pi * r^2
Prelude> :t areaOfSphere
areaOfSphere :: Double -> Double
```

... or let GHCi infer it ...

```
Prelude> let sphereArea r = 4 * pi * r^2
Prelude> :t
????
```

Type inference

Parametric polymorphism is a sweet spot on the typing landscape.

- Static typing,
- with Polymorphic values (give you dynamic-like typing when you need it)

The type inference in Haskell is really powerful.

It is considered good form (and documentation) to write some types, and let Haskell figure the rest out.

Haskell is not guessing the types, it is inferring them.

An inferred type is a high form of truth, and inference is a crowning achievement of centuries of mathematics.

Caveat: In order to be work within this powerful system, many primitives

in Haskell have non-obvious types. There is always a reason why.

KU

Everything has a type

Everything has a type, and GHCi can tell you, using **`:t`**.

Basic characters have type **`Char`**.

```
Prelude> 'c'
'c'
Prelude> 'c' :: Char
'c'
Prelude> :t 'c'
'c' :: Char
```

Strings have type **`[Char]`**, which means many chars. Strings are literally lists of characters.

```
Prelude> :t "Hello"
"Hello" :: [Char]
```

Type of a Number

```
Prelude> 1 :: Int
1
```

This is a C-style 32 or 64 bit number. (The Haskell spec says at least 29 bits + sign bit.)

```
Prelude> 1.0 :: Double
1
Prelude> 1.0 :: Float
1
```

`Double` and **`Float`** are 64 bit and 32 bit floating point numbers.

```
Prelude> 1 :: Integer
1
```

`Integer` has an arbitrary precision.

Type-inference of a Number

```
Prelude> :t 1
1 :: Num a => a
```

What can this mean? There is clearly more than meets the eye.

You can always use the **::** notation to fix a number as an **Int**, **Float**, etc.

Let us see some other examples, get back to basics, and come back to this.

Type-inference of a Function

```
Prelude> let f x = x
Prelude> :t f
???
```

What can you know about **x**. Nothing at all?

Literally, the type of **f** is ∀t . t →t. Haskell assumes the ∀ in this example.

```
Prelude> :t f
f :: t -> t
```

f takes anything, and returns (the same) anything.

Terminology: **t** is polymorphic, and **f** is a polymorphic function.

In the type syntax, polymorphic arguments are lower case.

Type-inference of a Function (2)

If we are more specific about arguments or results, the function will have a more specific type to reflect this.

```
Prelude> let f x = (x :: Int)
Prelude> :t f
f :: Int -> Int
```

Alternatively (Uses an extension **`ScopedTypeVariables`** ; originally not considered good form):

```
Prelude> :set -XScopedTypeVariables
Prelude> let f (x :: Int) = x
Prelude> :t f
f :: Int -> Int
```

Key observation: the original polymorphic function is the most general version of the function.

Types of key arithmetic functions

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

This means

- **(+)** takes two **a** values,
- and returns an **a** value,
- and **a** is a Num-thing.
- "**Num a =>**" means this is my constraint.
- "**a ->**" means this is what I pass as an argument.

Now, addition does add two numbers, to give a number.