Haskell 02: Polymorphism and Functional Programming Paradigms

Soomin Kim

Based on http://www.seas.upenn.edu/~cis194/spring15/lectures/

Contents

Additional Syntax

```
strLength :: String ->
Int
strLength []      = 0
strLength (_:xs) = let
len_rest = strlength xs
in


len_rest + 1
```

- let expressions
  - to define a local variable scoped over an expression

Additional Syntax

```
frob :: String -> Char
frob [] = ‘a’
frob str
  | len > 5    = ‘x’
  | len < 3    = ‘y’
  | otherwise = ‘z’
 where
   len = strLength str
```

- where clauses
  - to define a local variable scoped over multiple guarded branches
  - len cannot be used in the separate top-level pattern
  - where is somewhat more common than let, because using where allows the programmer to get right to the point in defining what a function does

Additional Syntax

- Haskell Layout
  - Haskell is a *whitespace-sensitive* language
  - Haskell uses indentation level to tell where certain regions of code end, and where new statements appear
  - The *layout heralds* are *where*, *let*, *do*, *of*, and *case*.
  - In Haskell, tabs in code are considered with tab stops 8 characters apart, regardless of what your editor might show you

Additional Syntax

```haskell
sumTo20 :: [Int] -> Int
sumTo20 nums = go 0 nums
  where go :: Int -> [Int] -> Int
        go acc [] = acc
        go acc (x:xs)
         | acc >= 20 = acc
         | otherwise = go (acc + x) xs
```

- Accumulators
  - Haskell’s one way to repeat a computation is recursion
  - Sometimes, a problem’s structure doesn’t exactly match Haskell’s structure
  - Structure like this running sum is called an accumulator

Parametric polymorphism

- One important thing about polymorphic function: **the caller gets to pick the types**

- It must work for every possible input type

Parametric polymorphism

```haskell
notEmpty :: [a] -> Bool
notEmpty (_:_) = True
notEmpty []    = False
```

- The notEmpty function does not care what a is
- This “not caring” is what “parametric” in parametric polymorphism means
- All Haskell functions must be parametric in their type parameters

Parametric polymorphism

- A function can’t do one thing when a is Int and a different thing when a is Bool
- Haskell simply provides no facility for writing such an operation
- This property of a language is called *parametricity*
- One consequence of parametricity is *type erasure*
- All the type information can be dropped during compilation

Parametric polymorphism

```
strange :: a -> b


limited :: a -> a
limited x = x
```

- Another consequence of parametricity is that it restricts what polymorphic functions you can write
  - There is no way to write strange
  - There is only one possible definition for limited

Total and partial functions

- *Partial function*
  - When there are certain inputs for which that function will crash
  - functions which have certain inputs that will make them recurse infinitely

- It is good Haskell practice to avoid partial functions as much as possible

- Often partial functions can be replaced by pattern-matching

Recursion patterns

- What sorts of things might we want to do with an [a]?
  - Perform some operation on every element of the list
  - Keep only some elements of the list, and throw others away, based on a test
  - “Summarize” the elements of the list somehow(find their sum, product, maximum, …)
  - …

Recursion patterns

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

- Map
  - Perform some operation on every element of the list
  - Passing functions as inputs to other functions

Recursion patterns

```
p :: a -> Bool
filter :: (a -> Bool) ->
[a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x       = x :
filter p xs
  | otherwise = filter p
xs
```

- Filter
  - Keep only some elements of the list, and throw others away, based on a test
  - The thing to abstract out is the *predicate* used to determine which values to keep

Recursion patterns

```
fold :: (a -> b -> b) ->
b -> [a] -> b
fold f z []      = z
fold f z (x:xs) = f x
(fold f z xs)


foldr f z [a, b, c] == a
`f` (b `f` (c `f` z))
```

- Fold
  - "Summarize" the elements of the list somehow
  - The idea will be to abstract out the parts that vary, aided by the ability to define higher-order functions
  - There is also foldl, folds "from the left"
  - However, you should use foldl' from Data.List instead

Functional Programming

```
(.) :: (b -> c) -> (a ->
b) -> a -> c
(.) f g x = f (g x)

($) :: (a -> b) -> a -> b
f $ x = f x
```

- Functional combinators
  - (.) operator is just function composition
  - ($) operator is useful for avoiding parentheses

Functional Programming

```
duplicate :: [String] ->
[String]
duplicate = map (\x -> x
++ x)
```

- Lambda
  - It is sometimes necessary to create an anonymous function, or lambda expression
  - The backslash binds the variables after it in the expression that follows the ->
  - For anything but the shortest examples, it's better to use a named helper function, though

Currying and Partial Application

```
f :: Int -> Int -> Int
f x y = 2 * x + y

f’ :: Int -> (Int -> Int)
f’ x y = 2 * x + y
```

- All functions in Haskell take only one argument
  - Actually it takes one argument and outputs a function
  - Function arrows *associate to the right*
  - Function application is *left-*associative

Currying and Partial Application

```
\x y z -> ...
\x -> (\y -> (\z -> ...))


f x y z = ...
f = \x -> (\y -> \z -
> ...))
```

- First ones are just syntax sugar for the second ones
- The idea of representing multi-argument functions as one-argument functions returning functions is known as *currying*

Currying and Partial Application

```
currying :: ((a, b) -> c)
-> a -> b -> c
currying f x y = f (x, y)


uncurrying :: (a -> b ->
c) -> (a, b) -> c
uncurrying f (x, y) = f x
y
```

- Standard library defines functions called curry and uncurry, defined like this
- uncurry in particular can be useful when you have a pair and want to apply a function to it

Currying and Partial Application

- Partial application
  - Functions in Haskell are curried makes *partial application* particularly easy
  - The idea of partial application is that we can take a function of multiple arguments and apply it to just *some* of its arguments
  - Every function can be “partially applied” to its first (and only) argument, resulting in a function of the remaining arguments
  - The arguments should be ordered from “least to greatest variation”, that is, arguments which will often be the same should be listed first, and arguments which will often be different should come last

Currying and Partial Application

```
foobar :: [Integer] ->
Integer
foobar []       = 0
foobar (x:xs)
  | x > 3       = (7 * x +
2) + foobar xs
  | otherwise = foobar xs
```

- Wholemeal programming
  - Problem of foobar:
    - doing too much at once
    - working at too low of a level

Currying and Partial Application

```
foobar' :: [Integer] ->
Integer
foobar' = sum . map
((+2) . (*7)) . filter
(>3)
```

- Wholemeal programming
  - We can think about making incremental transformations to the entire input
  - foobar’ as a “pipeline” of three functions
  - map and filter have been partially applied

Currying and Partial Application

```
mumble = (`foldr` []) .
((:).)


grumble = zipWith ($) .
repeat
```

- Point-free Style
  - Style of coding in which we define a function without reference to its argument is known as “point-free” style
  - Both are equivalent to the map function