Functions and Their Types

Prof. Susan Older

21 October 2019

The Truth about Functions

Each Haskell function takes exactly one argument and returns one value.

Examples

1. `isAlphaNum :: Char -> Bool`
   Accepts a `Char` and returns a `Bool`
2. `length :: [a] -> Int`
   Accepts a list and returns an `Int`
3. `sum :: Num a => [a] -> a`
   Accepts a list of numbers and returns a number
4. `replicate :: Int -> a -> [a]`
   Accepts an `Int` and returns a function of type `a -> [a]`
   (more on that later...)

Function Types

Choose any types `t1` and `t2`:

- `t1 -> t2` is a type.
- The values of type `t1 -> t2` are the functions that accept input of type `t1` and return results of type `t2`.

Examples

1. `Int -> (Char,Bool)`
   Functions that accept `Int` and return `(Char, Bool)`
2. `(Int,Char) -> Bool`
   Functions that accept `(Int,Char)` and return `Bool`
3. `Int -> (Char -> Bool)`
   Functions that accept `Int` and return `Char -> Bool` (i.e., functions)
4. `(Int -> Char) -> Bool`
   Functions that accept function of type `Int -> Char` and return `Bool`

Let's See That Again

From the previous slide:

- `Int -> (Char -> Bool)`
  Functions that accept `Int` and return function of type `Char -> Bool`
- `(Int -> Char) -> Bool`
  Functions that accept function of type `Int -> Char` and return `Bool`

Those types are not the same thing:

An analogy, based on money and candy:

- `Money -> (Money -> Candy)`
  Accepts money and returns a vending machine or a gumball machine
- `(Money -> Money) -> Candy`
  Accepts a money changer and returns candy.

So How Many Parameters Does `replicate` Take?

There are two views, both of which are valid:

1. The old way: `replicate :: Int -> a -> [a]`
   - `replicate` takes two arguments and returns a list.
   - For example:
     `replicate 3 "hi"` $\rightsquigarrow$ `["hi","hi","hi"]`
2. The new way: `replicate :: Int -> (a -> [a])`
   - `replicate` takes an `Int` and returns a function `a -> [a]`
   - For example:
     `replicate 3 :: a -> [a]`

     We can use `(replicate 3)` as a function:
     - `(replicate 3) "hi"` $\rightsquigarrow$ `["hi","hi","hi"]`
     - `(replicate 3) 10` $\rightsquigarrow$ `[10,10,10]`
     - `map (replicate 3) [10,20,30,40]` $\rightsquigarrow$ `[[10,10,10],[20,20,20],[30,30,30],[40,40,40]]`

Conventions for Parentheses

Parentheses in types associate to the right:

`Int -> Bool -> Char -> [Float]` is shorthand for `Int -> (Bool -> (Char -> [Float]))`

Parentheses in function applications associate to the left:

`fun 7 False 'A'` is shorthand for `((fun 7) False) 'A'`

Why does this make sense?

- `fun :: Int -> Bool -> Char -> [Float]`
- `fun 7 False 'A' :: [Float]`
- `fun :: Int -> (Bool -> (Char -> [Float]))`
- `fun 7 :: Bool -> (Char -> [Float])`
- `(fun 7) False :: Char -> [Float]`
- `((fun 7) False) 'A' :: [Float]`

Implications of the Parentheses Conventions

The following definitions are all equivalent:

```
mult1 :: Int -> Int -> Int
mult1 x y = x*y
```

```
mult2 :: Int -> (Int -> Int)
mult2 x y = x*y
```

```
mult3 :: Int -> (Int -> Int)
mult3 x = \y -> x*y
```


Partial Application and the Rule of Cancellation

Suppose we have the following:

$$\begin{aligned} f &:: t_1 \texttt{ -> } t_2 \texttt{ -> } \cdots \texttt{ -> } t_n \texttt{ -> } t \\ e_1 &:: t_1 \\ e_2 &:: t_2 \\ &\vdots \\ e_k &:: t_k \quad \text{(where } k < n\text{)} \end{aligned}$$

Then:

$$f\ e_1\ e_2\ \cdots\ e_k :: t_{k+1} \texttt{ -> } t_{k+2} \texttt{ -> } \cdots \texttt{ -> } t_n \texttt{ -> } t$$

The textbook calls this the Rule of Cancellation (see page 244).

The Rule of Cancellation: An Example

Suppose we have the following types:

```
   myFun  :: Bool -> Char -> Float -> (Int,Bool) -> String
   False  :: Bool
     'B'  :: Char
    16.2  :: Float
(7,True)  :: (Int,Bool)
```

- `myFun :: Bool -> Char -> Float -> (Int,Bool) -> String`
- `myFun False :: Char -> Float -> (Int,Bool) -> String`
- `myFun False 'B' :: Float -> (Int,Bool) -> String`
- `myFun False 'B' 16.2 :: (Int,Bool) -> String`
- `myFun False 'B' 16.2 (7,True) :: String`

Examples of Partial Application

Consider the following uses of partial application:

```
map (replicate 3) [5..8]
```

```
map (zip [1..3]) ["go","syracuse","orange","crush"]
```

```
filter (elem 6) [[20..30],[4,6,8], map (*2) [1..5]]
```

What do they have in common?

Sometimes it's useful to pass a partially instantiated function as the argument to another function.

Quiz Questions

Suppose `g :: Bool -> [Int] -> Char -> Float -> Int -> String`.

What are the types for each of the following?

1. `g False [1,6,3] 'a' :: Float -> Int -> String`
2. `g True [10] '9' 3.1 :: Int -> String`
3. `g [5,7] True` — Type error: `[5,7]` does not have type `Bool`

Suppose `h :: (Char -> Bool) -> Int -> Float -> Char`.

What are the types for each of the following?

1. `h isUpper 7 :: Float -> Char`
2. `h 'a' False` — Type error: `'a'` does not have type `Char -> Bool`

Polymorphic Quiz Questions

Suppose `j :: a -> b -> (a,b)`.

What are the types for each of the following?

1. `j False :: b -> (Bool,b)`
2. `j False 'M' :: (Bool,Char)`

Suppose `map :: (a -> b) -> [a] -> [b]`.

What are the types for each of the following?

1. `map isLower :: [Char] -> [Bool]`
2. `map (h isUpper 7) :: [Float] -> [Char]`

Some Higher-Order Functions You Should Get to Know

```
      map :: (a -> b) -> [a] -> [b]
concatMap :: (a -> [b]) -> [a] -> [b]
  zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

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

```
     (.) :: (b -> c) -> (a -> b) -> a -> c
    flip :: (a -> b -> c) -> b -> a -> c
   curry :: ((a,b) -> c) -> a -> b -> c
 uncurry :: (a -> b -> c) -> (a,b) -> c
```


Operator Sections: Partial Application with Operators

Binary operators can be partially applied via operator sections:

- An operator section takes a single argument.
- The operator's argument is treated as if it appears on the "empty" side of the operator:

`("abc"++) "de"` $\rightsquigarrow$ `"abcde"` `(++"abc") "de"` $\rightsquigarrow$ `"deabc"`

`(*2)`	Function that multiplies its argument by two
`(2-)`	Function that subtracts its argument from 2
`(10>)`	Function that determines if 10 is greater than its argument
`(7:)`	Function that places 7 on the front of a list
`(++"xyz")`	Function that appends `"xyz"` to a string
`("xyz"++)`	Function that prepends `"xyz"` to a string