EECS 776

Functional Programming and Domain Specific Languages

Professor Gill

The University of Kansas

Feb 7th 2020

KU

Recap: Types

Types are shorthand descriptions of things

```
42 :: Int
```

- Type-checking vs. type-inference?
  - Type-checking is checking if the types are self-consistent
  - Type-inference is checking without being told what the types are
- Haskell supports both. In practice:
  - Most types are inferred by the compiler
  - Types given by Haskell users (you!) are a mild form of documentation
- Question to ask: **Does adding types add clarity?**

Primitives in Haskell

All primitive types start with an **upper case** letter

`Int, Integer, Float, Double, Bool, Char`

- **`Int`** - Signed value, the size of the machine **`int`**
- **`Integer`** - Arbitrary precision signed value
- **`Float`** - IEEE 32-bit floating point number
- **`Double`** - IEEE 64-bit floating point number
- **`Bool`** - result of a comparison, **`True`** or **`False`**.
- **`Char`** - a single character

There are ways of defining new types

Structure in Haskell

There are two built-in structures

- Lists - arbitrary length, every element has the same type

```
[1,2,3]::[Int]
```

- Tuples - specific length, every element can have a different type

```
(1,'c',1.0)::(Int,Char,
Float)
```

Both are used extensively in programs
(There are also ways of defining new structures)

Lists

Lists are conceptually linked-lists. You can build them directly, or build a list out of a smaller list.

```
GHCi> let xs = [1,2,3]
GHCi> xs
[1,2,3]
GHCi> length xs
3
```

The type is written **[type]**, and every element must be of that type.
What is the type of these bindings?

```
GHCi> let xs = [1,2,3::Float]
GHCi> let xs = [1] :: [Int]
GHCi> let xs = [1..100 :: Float]
GHCi> let xs = []
```

More Lists

Lists (and most structures) can be nested

```
GHCi> let xs = [[1,2,3::Float], [5,6,7]]
GHCi> :t xs
[[Float]]
GHCi> let xs = [[[['a','b','c']]]]
GHCi> :t xs
[[[[Char]]]]
```

Lists can not mix types

```
GHCi> let xs = [True,'c']
<interactive>:15:16:
    Couldn't match expected type `Bool' with actual type
`Char'
GHCi> let xs = [1,2,[3,4]]
<more bad things happening>
```

There are many operations over list

```
length :: [a] -> Int                -- length of a list
(++)   :: [a] -> [a] -> [a]         -- append two lists
null   :: [a] -> Bool               -- is a list empty
head   :: [a] -> a                  -- take the first element of
a list
tail   :: [a] -> [a]                -- take the rest of a list
(:)    :: a -> [a] -> [a]           -- add a value to front of
list
(!!)   :: [a] -> Int -> a           -- get n-th element
```

Remember:

- -> is used for function types (requiring arguments)
- lower-case names are polymorphic (can be anything)
- No structure is ever damaged by any function
- Instead, new structures are created

Strings are Lists of Char

```
GHCi> let str = "Hello EECS 776"
GHCi> :t str
str :: [Char]
```

This means that strings can use the list operators
There is a short-cut name for strings, because they are so common

```
type String = [Char]
```

`String` and **`[Char]`** are interchangeable.

```
GHCi> let str = "Hello EECS 776" :: String
GHCi> :t str
str :: String
GHCi> :t str ++ []
str :: [Char]
```

Tuples

Tuples are a specific length, and every element can have a different type

- There are 2-tuples to (at least) 15-tuples
- GHC supports up to 62
- If you are using more than (say) a 5-tuple, then you are using Haskell wrong
- They are intended, like in math, for small local groupings

```
GHCi> let xs = (1,"Hello",'c')
GHCi> :t xs
(Integer,[Char],Char)
GHCi> let xs = (1,"Hello")
GHCi> :t xs
(Integer,[Char])
GHCi>
```

- There is a zero-tuple, called unit
- There is no one-tuple (can you work out why?)

Tuples types

The type is written **`( type, type, ..., type )`**, mirroring the tuple value.

```
GHCi> let xs = ()              :: ()
GHCi> let xs = (1,"Hello")     :: (Integer,String)
GHCi> let xs = (1,2,3)         :: (Integer,Float,Double)
GHCi>
```

Tuples can also be nested, with themselves or lists (or any type).

```
GHCi> :t ('c',("Hello",()))
('c',("Hello",())) :: (Char, ([Char], ()))
GHCi> :t [('c',"Hello"),('d',"World")]
[('c',"Hello"),('d',"World")] :: [(Char, [Char])]
GHCi> :t (['c'],[True,False])
(['c'],[True,False]) :: ([Char], [Bool])
```

Passing tuples to functions

You've seen tuples before, in C / Java / C++.

```
GHCi> let add3 (a,b,c) = a + b + c
GHCi> add3 (1,2,3)
6
```

What is the type of **add3**?

```
Prelude> :t add3
add3 :: Num a => (a, a, a) -> a
```

Returning functions from tuples

Tuples are a way of grouping together arguments.

- You can pass multiple arguments to a function in C (Java, C++, etc).
- Why can you not return multiple results in C?

You can return multiple results in Haskell.

```
GHCi> let near x = (x - 1, x + 1)
GHCi> near 42
(41,43)
```

Haskell functions can take anything as arguments (including structures), and return anything as results (also including structures)

Tuples functions

```
??? :: (a,b) -> a
??? :: (a,b) -> b
```

What do these do?

```
fst :: (a,b) -> a
fst (a,b) = a

snd :: (a,b) -> b
snd (a,b) = b
```

- Type level - there is an idea called "Theorems for free" - the theorem of **`fst`** comes from its type
- Value level - this way of taking tuples to bits is called pattern matching

Common usage for tuples

- **`(Double,Double)`** is a 2-D coordinate, vector, etc.
- **`(Double,Double,Double)`** is a 3-D coordinate.

For example, a translation can be written

```
scaleBy2 :: (Double,Double) -> (Double,Double)
scaleBy2 (a,b) = (a*2,b*2)
```

Further, a circular region can be defined using

```
circle :: (Double,Double) -> Bool
circle (x,y) = x^2 + y^2 <= 1
```

We will see a Domain Specific Language that builds on this idea later.

Bring back lists

Remember lists, and list operations? First, look at the types

```
length :: [a] -> Int                -- length of a list
(++)   :: [a] -> [a] -> [a]         -- append two lists
null   :: [a] -> Bool               -- is a list empty
head   :: [a] -> a                  -- take the first element of
a list
tail   :: [a] -> [a]                -- take the rest of a list
(:)    :: a -> [a] -> [a]           -- add a value to front of
list
```

What might a function of this type do? **`? :: [[a]] -> [a]`**

```
concat :: [[a]] -> [a]              -- flatten a list
```

What might this function of this type do? **`? :: [(a,b)] -> ([a],[b])`**

```
unzip :: [(a,b)] -> ([a],[b])  -- split a list of pairs
into a pair of lists
```

Summary of types

Base types

`Int, Integer, Float, Double, Bool, Char`

Structural types

`[...], (), (...,...), (...,...,...), ...`

What about functions?

`... -> ...`

- Functions **are** values, in the way `4` or `[1,2,3]` are values
- As values, they have a type that describes them
- Functions, like structural types, can be nested