Functional Programming

Part I

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

Uder, 30.05.2019

Contents

- Basics of functional programming using Haskell
- Haskell development tools
- Writing Haskell programs
- Using Haskell libraries
- Your first Haskell project

What is Functional Programming?

A different approach to programming

Functions and values

rather than

Assignments and addresses

What is Functional Programming?

A different approach to programming

Functions and values

rather than

Assignments and addresses

It will make you a better programmer

Functional vs Imperative Programming: Variables

Functional (Haskell)

```haskell
x :: Int
x = 5
```

Variable `x` has value 5 forever

Functional vs Imperative Programming: Variables

Functional (Haskell)

```
x :: Int
x = 5
```

Variable x has value 5 forever

Imperative (Java / C)

```
int x = 5;
...
x = x+1;
```

Variable x can change its content over time

Functional vs Imperative Programming: Functions

Functional (Haskell)

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

f 42 16 // always 100
```

Return value of a function **only** depends on its inputs

Functional vs Imperative Programming: Functions

Functional (Haskell)

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

f 42 16 // always 100
```

Return value of a function **only** depends on its inputs

Imperative (Java)

```
boolean flag;
static int f (int x, int y) {
  return flag ? 2*x + y , 2*x - y;
}

int z = f (42, 16); // who knows?
```

Return value depends on non-local variable `flag`

Functional vs Imperative Programming: Laziness

Haskell

```
x = expensiveComputation
g anotherExpensiveComputation
```

- The expensive computation will only happen if x is ever used.
- Another expensive computation will only happen if g uses its argument.

Functional vs Imperative Programming: Laziness

Haskell

```
x = expensiveComputation
g anotherExpensiveComputation
```

- The expensive computation will only happen if `x` is ever used.
- Another expensive computation will only happen if `g` uses its argument.

Java

```
int x = expensiveComputation;
g (anotherExpensiveComputation)
```

- Both expensive computations will happen anyway.
- Laziness can be simulated, but it's complex!

Many features that make programs more concise

- Pattern Matching
- Higher-order functions
- Algebraic datatypes
- Polymorphic types
- Parametric overloading
- Type inference
- Monads & friends (for IO, concurrency, ...)
- Comprehensions
- Metaprogramming
- Domain specific languages
- ...

Predefined Types

Every Haskell value has a type

`Bool`	— True :: Bool, False :: Bool
`Char`	— 'x':: Char, '?':: Char, ...
`Double`, `Float`	— 3.14 :: Double
`Integer`	— 4711 :: Integer
`Int`	— machine integers ($\geq$ 30 bits signed integer)
`()`	— the unit type, single value () :: ()
a −> b	— function types
(a, b)	— tuple types
[a]	— list types
String	— "xyz":: String, ...

Functions

Examples.hs

```
dollarRate = 1.3671

-- |convert EUR to USD
usd euros = euros * dollarRate
```

- dollarRate defines a constant
- usd is a function
- Its type Double -> Double is inferred by the Haskell compiler
- To compute, a function call usd arg is replaced by the right hand side of its definition

  usd arg $\longrightarrow$ arg $*$ dollarRate $\longrightarrow$ arg $*$ 1.3671 $\longrightarrow$ ...

Tuples

```haskell
-- example tuples
examplePair :: (Double, Bool) -- Double x Bool
examplePair = (3.14, False)

exampleTriple :: (Bool, Int, String) -- Bool x Int x String
exampleTriple = (False, 42, "Answer")

exampleFunction :: (Bool, Int, String) -> Bool
exampleFunction (b, i, s) = not b && length s < i
```

Summary

- Syntax for tuple type like syntax for tuple values
- Tuples are **immutable**: in fact, **all values are**!
  Once a value is defined it cannot change!

Typing for Tuples

Typing Rule

$$\frac{e_1 :: t_1 \qquad e_2 :: t_2 \qquad \dots \qquad e_n :: t_n}{(e_1, \dots, e_n) :: (t_1, \dots, t_n)}\ \textsc{Tuple}$$

If

- $e_1, \dots, e_n$ are Haskell expressions
- $t_1, \dots, t_n$ are their respective types
- Then the tuple expression $(e_1, \dots, e_n)$ has the tuple type $(t_1, \dots, t_n)$.

Lists

- The "duct tape" of functional programming
- Collections of things of the same type
- For any type a, [a] is the type of lists with elements of type a
  e.g. [Bool] is the type of lists of Bool
- Syntax for list type like syntax for list values
- Lists are **immutable**: once a list value is defined it cannot change!

Constructing lists

The values of type [a] are ...

- either [], the empty list
- or x:xs where x has type a and xs has type [a]
  ":" is pronounced "cons"
- [] and (:) are the **list constructors**

Constructing lists

The values of type [a] are ...

- either [], the empty list
- or x:xs where x has type a and xs has type [a]
  ":" is pronounced "cons"
- [] and (:) are the **list constructors**

Typing Rules for Lists

$$\textsc{Nil}\quad [] :: [t] \qquad\qquad \textsc{Cons}\quad \frac{e_1 :: t \qquad e_2 :: [t]}{(e_1 : e_2) :: [t]}$$

- The empty list can serve as a list of any type t
- If there is some t such that e_1 has type t and e_2 has type $[t]$, then $(e_1 : e_2)$ has type $[t]$.

Typing Lists

Quiz Time

Which of the following expressions have type [Bool]?

```
[ ]
True : [ ]
True : False
False : (False : [ ])
(False : False) : [ ]
(False : []) : [ ]
(True : (False : (True : []))) : (False:[]):[ ]
```

Functions on lists

Definition by **pattern matching**

```
-- double every element of a list of integers
-- doubles [3,6,12] = [6,12,24]
doubles :: [Integer] -> [Integer]
doubles [] = []
doubles (x:xs) = (2 * x) : doubles xs
```

Functions on lists

Definition by **pattern matching**

```
-- double every element of a list of integers
-- doubles [3,6,12] = [6,12,24]
doubles :: [Integer] -> [Integer]
doubles [] = []
doubles (x:xs) = (2 * x) : doubles xs
```

Explanations — patterns

- patterns contain constructors and variables
- patterns are checked in sequence
- constructors are checked against argument value
- variables are bound to the values in corresponding position in the argument
- each variable may occur at most once in a pattern
- wild card pattern _ matches everything, no binding, may occur multiple times

References

- Paper by the original developers of Haskell in the conference on History of Programming Languages (HOPL III): http://dl.acm.org/citation.cfm?id=1238856
- The Haskell home page: `http://www.haskell.org`
- Haskell libraries repository: `https://hackage.haskell.org/`
- Haskell Tool Stack: `https://docs.haskellstack.org/en/stable/README/`

?