Functional Programming

Types

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2017-2018

Predefined Types

- `Bool` (True, False)
- `Char` ('x', '?', ...)
- `Double`, `Float`
- `Integer`
- `Int` — machine integers ($\geq$ 30 bits signed integer)
- () — the unit type, single value ()
- function types
- tuples and lists
- `String` ("xyz", ...)
- ...

Tuples

```
-- 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** : once a tuple value is defined it cannot change!

Lists

- The "duct tape" of functional programming
- Collections of things of the same type
- For any type `x`, `[x]` is the type of lists of `x`s
  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"

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"

Quiz

Which of the following expressions have type `[Bool]`?

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

List shorthands

Equivalent ways of writing a list

- `1:(2:(3:[ ]))` — standard, fully parenthesized
- `1:2:3:[ ]` — `(:)` associates to the right
- `[1,2,3]`

Functions on lists

Definition by **pattern matching**

```
-- function over lists - examples
summerize :: [String] -> String
summerize []  = "None"
summerize [x] = "Only " ++ x
summerize [x,y] = "Two things: " ++ x ++ " and " ++ y
summerize [_,_,_] = "Three things: ???"
summerize _   = "Several things."   -- wild card pattern
```

Functions on lists

Definition by **pattern matching**

```
-- function over lists - examples
summerize :: [String] -> String
summerize []  = "None"
summerize [x] = "Only " ++ x
summerize [x,y] = "Two things: " ++ x ++ " and " ++ y
summerize [_,_,_] = "Three things: ???"
summerize _   = "Several things."   -- wild card pattern
```

Explanations — expressions

- (++) **list concatenation**
- (++) associates to right because it's more efficient
  - `[1,2,3,4,5] ++ ([6,7,8,9] ++ [])` — 10 copy operations
  - `([1,2,3,4,5] ++ [6,7,8,9]) ++ []` — 14 copy operations, because `[1,2,3,4,5]` is copied twice

Pattern matching on lists

Explanations — patterns

- patterns are checked in sequence
- variables in patterns 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

Primitive recursion on lists

Common example

```
-- doubles [3,6,12] = [6,12,24]
doubles :: [Integer] -> [Integer]
doubles []     = undefined
doubles (x:xs) = undefined
```

Primitive recursion on lists

Common example

```
-- doubles [3,6,12] = [6,12,24]
doubles :: [Integer] -> [Integer]
doubles []     = undefined
doubles (x:xs) = undefined
```

BUT

Would not write it in this way — it's a common pattern that we'll define in a library function

Primitive recursion on lists

Common example

```
-- doubles [3,6,12] = [6,12,24]
doubles :: [Integer] -> [Integer]
doubles []     = undefined
doubles (x:xs) = undefined
```

BUT

Would not write it in this way — it's a common pattern that we'll define in a library function

- `undefined` is a value of any type
- evaluating it yields a run-time error

The function map

Definition

```
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
map f []     = undefined
map f (x:xs) = undefined
```

(map is in the standard Prelude - no need to define it)

The function map

Definition

```
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
map f []     = undefined
map f (x:xs) = undefined
```

(map is in the standard Prelude - no need to define it)

Define doubles in terms of map

The function map

Definition

```
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
map f []     = undefined
map f (x:xs) = undefined
```

(map is in the standard Prelude - no need to define it)

Define doubles in terms of map

```
doubles xs = map double xs

double :: Integer -> Integer
double x = undefined
```

The function filter

Produce a list by removing all elements which do not have a certain property from a given list:

```
filter odd [1,2,3,4,5] == [1,3,5]
```

Definition

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []       = undefined
filter p (x:xs) = undefined
```

(filter is in the standard Prelude - no need to define it)

Questions?