Functional Programming

Types

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

Predefined Types

- `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 () :: ()
- function types
- tuples and lists
- `String`: "xyz":: String, ...
- ...

Tuples

```
1 -- example tuples
2 examplePair :: (Double, Bool) -- Double x Bool
3 examplePair = (3.14, False)
4
5 exampleTriple :: (Bool, Int, String) -- Bool x Int x String
6 exampleTriple = (False, 42, "Answer")
7
8 exampleFunction :: (Bool, Int, String) -> Bool
9 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

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

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

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 xs
  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

$$\textsc{Nil}\quad [] :: [t]$$

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

- 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

Which of the following expressions have type [Bool]?

```
1  []
2  True : [ ]
3  True:False
4  False:(False:[ ])
5  (False:False):[ ]
6  (False:[]):[ ]
7  (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]	—	bracketed notation

Functions on lists

Definition by **pattern matching**

```
1 -- function over lists, examples for list patterns
2 summerize :: [String] -> String
3 summerize [] = "None"
4 summerize [x] = "Only " ++ x
5 summerize [x,y] = "Two things: " ++ x ++ " and " ++ y
6 summerize [_,_,_] = "Three things: ???"
7 summerize _ = "Several things." -- wild card pattern
```

Functions on lists

Definition by **pattern matching**

```
1 -- function over lists, examples for list patterns
2 summerize :: [String] -> String
3 summerize [] = "None"
4 summerize [x] = "Only " ++ x
5 summerize [x,y] = "Two things: " ++ x ++ " and " ++ y
6 summerize [_,_,_] = "Three things: ???"
7 summerize _ = "Several things." -- wild card pattern
```

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

Pattern matching on lists

Explanations — expressions

- (++) **list concatenation**
- (++) associates to right

Primitive recursion on lists

Common example: double every element in a list of numbers

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

Primitive recursion on lists

Common example: double every element in a list of numbers

```
1 -- doubles [3,6,12] = [6,12,24]
2 doubles :: [Integer] -> [Integer]
3 doubles [] = undefined
4 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: double every element in a list of numbers

```
1 -- doubles [3,6,12] = [6,12,24]
2 doubles :: [Integer] -> [Integer]
3 doubles [] = undefined
4 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

map: Apply Function to Every Element of a List

Definition

```
1 -- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
2 map :: (a -> b) -> [a] -> [b]
3 map f [] = undefined
4 map f (x:xs) = undefined
```

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

map: Apply Function to Every Element of a List

Definition

```
1 -- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
2 map :: (a -> b) -> [a] -> [b]
3 map f [] = undefined
4 map f (x:xs) = undefined
```

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

Define doubles in terms of map

map: Apply Function to Every Element of a List

Definition

```
1 -- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
2 map :: (a -> b) -> [a] -> [b]
3 map f [] = undefined
4 map f (x:xs) = undefined
```

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

Define doubles in terms of map

```
1 doubles xs = map double xs
2
3 double :: Integer -> Integer
4 double x = undefined
```

The function filter

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

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

Definition

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

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

Questions?