Functional Programming

Types

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2022/23

Contents

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 () :: ()
- `A -> B` — function types
- `(A, B)`, `[A]` — tuple and list types
- `String` — "xyz":: **String**, ...
- ... —

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

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

- $e_1, \dots, e_n$ 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 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, pronounced "nil"
- 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, pronounced "nil"
- 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**]?

```
[]
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]	—	bracketed notation

Functions on lists

Definition by **pattern matching**

```haskell
-- function over lists, examples for list patterns
summerize :: [String] -> String
summerize [] = "None"
summerize [x] = "Only " ++ x
summerize [x,y] = "Two things: " ++ x ++ " and " ++ y
summerize [_,_,_] = "Three things: ???"
summerize (x:xs) = "First " ++ x ++ " and then " ++ concat xs
summerize _ = "Several things." -- wild card pattern
```

Functions on lists

Definition by **pattern matching**

```
-- function over lists, examples for list patterns
summerize :: [String] -> String
summerize [] = "None"
summerize [x] = "Only " ++ x
summerize [x,y] = "Two things: " ++ x ++ " and " ++ y
summerize [_,_,_] = "Three things: ???"
summerize (x:xs) = "First " ++ x ++ " and then " ++ concat xs
summerize _ = "Several things." -- wild card pattern
```

Explanations — patterns

- patterns can occur in place of formal parameters, on the left side of function definitions
- 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

- (++) :: [a] −> [a] −> [a] **list concatenation**
- (++) associates to right
- **concat** :: [[a]] −> [a] **concatenate a list of lists**

Primitive recursion on lists

Common example: double every element in a list of numbers

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

Primitive recursion on lists

Common example: double every element in a list of numbers

```
-- 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: double every element in a list of numbers

```haskell
-- 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

map: Apply Function to Every Element of a List

Definition

```
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
map :: (a -> b) -> [a] -> [b]
map f [] = undefined
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

```
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
map :: (a -> b) -> [a] -> [b]
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

map: Apply Function to Every Element of a List

Definition

```
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
map :: (a -> b) -> [a] -> [b]
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?

?