Functional Programming

Zsók Viktória, Ph.D.

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary
zsv@elte.hu

Overview

Definition

- data structures - store and manipulate collections of data
- list - sequence of elements of the same type elements
- the elements of a list can be of any type
- they are written between `[ ]` brackets
- coma separates the elements
- considered recursive data type

Lists in Clean

- lists in Clean are regarded as linked lists - a chain of boxes referring to each other
- empty list is []
- every list has a type, the type of the contained elements
- no restrictions on the number of elements
- singleton list with one element [False], [[1,2,3]]
- special constructor is [1:[2,3,4]] is equivalent to [1,2,3,4]
  [1,2,3] is equivalent to [1:[2:[3:[]]]]

Defining lists

One of the most important data structures in FP is the list: a sequence of elements of the same type values

```
l1 :: [Int]
l1 = [1, 2, 3, 4, 5]
l2 :: [Bool]
l2 = [True, False, True]
l3 :: [Real→Real]
l3 = [sin, cos, sin]
l4 :: [[Int]]
l4 = [[1, 2, 3], [8, 9]]
l5 :: [a]
l5 = []
l6 :: [Int]
l6 = [1..10]
l7 :: [Int]
l7 = [1..]
```

Defining lists

The elements need not be constants, may be determined by computation

```
[1+5, 2*10, length [1,2,3,4,5]] :: [Int]
```

```
[5<10, x == 8, a && b] :: [Bool]
```

the used expressions must be of the same type

Empty list

- the empty list has polymorphic type, is a list of whatever
- the type is determined from the context
- can be used in an expression whenever a list is needed

`sum []` - empty list of numbers Int or Real
`and []` - empty list of Booleans
`[[], [1,2], [3]]` - empty list of integer numbers
`[[True], [4>1], []]` - empty list of Booleans
`[[[88]], []]` - empty list of lists of numbers
`length []` - empty list of anytype values of type a

Generating lists

```
Start =
    [1..10]       // [1,2,3,4,5,6,7,8,9,10]
    [1,2..10]     // [1,2,3,4,5,6,7,8,9,10]
    [1,0.. -10]   // [1,0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
    [1.. -10]     // []
    [1..0]        // []
    [1..1]        // [1]
    [1,3..4]      // [1,3]
    [1..]         // [1,2,3,4,5,6,7,8,9,10,...
    [1,3..]       // [1,3,5,7,9,11,13,15,...
    [100,80..]    // [100,80,60,40,20,0,-20,-40,...
```

Operations with lists

```
Start =
    hd [1, 2, 3, 4, 5]          // 1
    tl [1, 2, 3, 4, 5]          // [2, 3, 4, 5]
    drop 2 [1, 2, 3, 4, 5]      // [3, 4, 5]
    take 2 [1, 2, 3, 4, 5]      // [1 , 2]
    [1, 2, 3] ++ [6, 7]         // [1, 2, 3, 6, 7]
    reverse [1, 2, 3]           // [3, 2, 1]
    length [1, 2, 3, 4]         // 4
    [1..5]!!2                   // 3
    last [1, 2, 3]              // 3
    init [1, 2, 3]              // [1, 2]
    isMember 2 [1, 2, 3]        // True
    isMember 5 [1, 2, 3]        // False
    flatten [[1,2], [3, 4, 5], [6, 7]]      // [1, 2, 3, 4, 5, 6, 7]
```

Definition of some operations

```
take :: Int [a] → [a]
take n [] = []
take n [x : xs]
| n < 1 = []
| otherwise = [x : take (n-1) xs]

Start = take 2 []                    // []
Start = take 2 [1 .. 10]             // [1,2]
Start = take 0 [1..5]                // []

take 2 [1..5]
n = 2 x = 1 xs = [2,3,4,5]     [1 : take 1 [2,3,4,5]]
n = 1 x = 2 xs = [3,4,5]       [1 : [ 2: take 0 [3,4,5]]]
n = 0 < 1                      [1 : [ 2 : []]] = [1,2]
```

Definition of some operations

```
drop :: Int [a] → [a]
drop n [] = []
drop n [x : xs]
| n < 1 = [x : xs]
| otherwise = drop (n-1) xs

Start = drop 5 [1,2,3]              // []
Start = drop ([1..5]!!2) [1..5]     // [4,5]
Start = drop 0 [1..5]               // [1,2,3,4,5]

drop 2 [1..5]
n = 2 x = 1 xs = [2,3,4,5]      drop 1 [2,3,4,5]
n = 1 x = 2 xs = [3,4,5]        drop 0 [3,4,5]
n = 0 < 1                       [3,4,5]
```

Compare take and drop definition

```
take :: Int [a] → [a]
take n [] = []                          // for [] result is []
take n [x : xs]
| n < 1 = []                            // returns empty list
| otherwise = [x : take (n-1) xs]       // collecting heads

drop :: Int [a] → [a]
drop n [] = []                          // for [] result is []
drop n [x : xs]
| n < 1 = [x : xs]                      // returns last step's input
| otherwise = drop (n-1) xs             // ignoring heads
```

Definition of some operations

```
reverse :: [a] → [a]
reverse [] = []
reverse [x : xs] = reverse xs ++ [x]

Start = reverse [1,3..10]          // [9,7,5,3,1]
Start = reverse []                 // []
Start = reverse [5,4 .. -5]        // [-5,-4,-3,-2,-1,0,1,2,3,4,5]

reverse [1..3]
(reverse [2,3]) ++ [1]
(reverse [3]) ++ [2] ++ [1]
(reverse []) ++ [3] ++ [2] ++ [1]
[] ++ [3] ++ [2] ++ [1] = [3,2,1]
```

Definition of some operations

```
isMember::a [a] → Bool | Eq a
isMember x []       = False
isMember x [hd:tl] = hd==x || isMember x tl

Start = isMember 0 []                  // False
Start = isMember -1 [1..10]            // False
Start = isMember ([1..5]!!1) [1..5]    // True

flatten :: [[a]] → [a]
flatten []      = []
flatten [h:t]   = h ++ flatten t

Start = flatten [[1,2], [5], [6,7,9]]  // [1,2,5,6,7,9]
```

Definitions by patterns

Various patterns can be used:

```
// some list patterns - a list of exactly 3 values
triplesum :: [Int] → Int
triplesum [x, y, z] = x + y + z
Start = triplesum [1,2,4] // 7 [1,2,3,4] error

head :: [Int] → Int      // not defined for [] list
head [x : y] = x
Start = head [1..5] // 1

tail :: [Int] → [Int]   // not defined for [] list
tail [x : y] = y
Start = tail [1..5] // [2,3,4,5]

// omitting values
f :: Int Int → Int
f _ x = x
Start = f 4 5 // 5
```

Definitions by patterns

```
// patterns with list constructor - a list with min 2 elements
g :: [Int] → Int
g [x, y : z] =  x + y
Start = g [1, 2, 3, 4, 5] // 3

// patterns + recursively applied functions
lastof [x] = x
lastof [x : y] = lastof y
Start = lastof [1..10] // 10
```

Definitions by recursion 2

```
// recursive functions on lists - calling built-in functions
sum1 x
| x == [] = 0
| otherwise = hd x + sum1 (tl x)

// recursive functions on lists - using [head : tail] pattern
sum2 [] = 0
sum2 [first : rest] = first + sum2 rest
Start = sum1 [1..5] // 15 the same for sum2

// recursive function with any element pattern
length1 [] = 0
length1 [_ : rest] = 1 + length1 rest
Start = length1 [1..10] // 10
```

Warm-up exercises

Evaluate the following expressions:

1. `(take 3 [1..10]) ++ (drop 3 [1..10])`
2. `length (flatten [[1,2], [3], [4, 5, 6, 7], [8, 9]])`
3. `isMember (length [1..5]) [7..10]`
4. `[1..5] ++ [0] ++ reverse [1..5]`

Solutions

1. `(take 3 [1..10]) ++ (drop 3 [1..10])`

2. `length (flatten [[1,2], [3], [4, 5, 6, 7], [8, 9]])`

3. `isMember (length [1..5]) [7..10]`

4. `[1..5] ++ [0] ++ reverse [1..5]`

1. `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
2. `9`
3. `False`
4. `[1, 2, 3, 4, 5, 0, 5, 4, 3, 2, 1]`