Introduction to Functional Programming

Zsók Viktória, Ph.D.

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

Overview

1. Introduction
   - Why FP? - motivation
2. Content
3. Defining functions
   - Guards and patterns
   - Recursive functions

Motivation

Functional programming:

- allows programs to be written clearly, concisely
- has a high level of abstraction
- supports reusable software components
- encourages the use of formal verification
- permits rapid prototyping
- has inherent parallel features

What is functional programming?

- the closest programming style to mathematical writing, thinking
- which one should be the first programming language?

- the basic element of the computation is the function
- basically function compositions are applied
- running a program is called evaluation

Syntax

The syntax of a programming language is the set of rules applied to describe a problem.

f(a) => `f a`
f(a,b) + cd => `f a b + c * d`
f(g(b)) => `f (g b)`
f(a)g(b) => `f a * g b`

FP outline

Course content:

- how to write functions - definition, guards, recursive functions
- datastructures - lists, tuple, record, array, class
- higher order functions - map, fold, iterate
- types - simple types, algebraic types, composite types, trees
- algorithms - search, sort, traverse

History

- Lisp - list processor, in early 60s John McCarthy
- operates on lists, functions can be arguments to other functions
- type checking, ability to check programs before running them
- ML, Miranda, Haskell, Clean
- lazy functional programming

Writing functional programs is FUN

- to motivate you to write functional programs
- to get involved in working with FP
- the Clean compiler can be downloaded from:
  `http://wiki.clean.cs.ru.nl/Clean`
  have FUN

`examples.icl, examples.prj`

```
module examples
import StdEnv
Start = 42 // 42
```

Clean - Start

- Some start expressions:

```
Start = 4*6+8

Start = sqrt 2.0

Start = sin x

Start = sum [1..10]
```

- constants pi = 3.1415926

Program evaluation

- reduction steps
- redex
- normal form

```
f x = (x + 8) * x

Start = f 2
```

```
Start
→ f 2
→ (2 + 8) * 2
→ 10 * 2
→ 20
```

Reduction steps, redex

- the process of evaluation is called *reduction*
- replacing a part of expression which matches a function definition is called *reduction step*
- *redex* = reducible expression
- when a function contains no redexes is called *normal form*

Lazy and eager evaluation

- lazy = the expression is not evaluated until is not needed
- opposite is eager evaluation = all arguments are evaluated before the function's result
- Clean is pure, lazy functional language
- advantages of lazy evaluation: infinite lists, less evaluations

Standard functions

- `StdEnv` - contains all
- the name of your own functions should start with letter then zero or more letters, digits, symbols
- upper and lower case allowed but treated differently
- funny symbols, built-in function names can not be used

Some predefined operators / functions on numbers

- integers 18, 0, -23 and floating-point numbers 1.5, 0.0, 4.765, 1.2e3 1200.0
- addition +, subtraction -, multiplication *, division /
- for Int some standard functions abs, gcd, sign
- for Real sqrt, sin, exp
- for Bool type True, False (George Boole eng.math. 1815-1864)
- boolean operators
  >, <=, == (equal), <> (not equal), && (and), || (or)
- comments // or /* ... */

Getting started

Simple examples of Clean functions:

```
inc x = x + 1
double x = x + x
quadruple x = double (double x)
factorial n = prod [1 .. n]
```

Their usage:

```
Start = 3+10*2 // 23
Start = sqrt 3.0 // 1.73...
Start = quadruple 2 // 8
Start = factorial 5 // 120
```

Definitions by cases

The cases are guarded by Boolean expressions:

```
abs1 x
| x < 0 = ¬x
| otherwise = x
Start = abs1 -4    // two cases, the result is 4

// otherwise can be omitted
abs2 x
| x < 0 = ¬x
= x
Start = abs2 4    // 4

// more then two guards or cases
signof x
| x > 0 = 1
| x == 0 = 0
| x < 0 = -1
Start =  signof -8  // -1
```

Definitions by recursion

Examples of recursive functions:

```
fac n
| n == 0 = 1
| n > 0 = n * fac (n - 1)

Start = fac 5 // 120
```

```
power x n
| n == 0 = 1
| n > 0 = x * power x (n - 1)

Start = power 2 5 // 32
```

Summary

- function evaluations, evaluation strategies, expressions
- basic types, operations, `Start` expressions
- definition of function by guards, basic recursions