EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programing

Sommersemester 2014, Wintersemester 2015

Author:
Philipp Moers, Update by:
Denis Hirn

Dozent:
Torsten Grust,
Alexander Ulrich

Last updated: Friday 18th December, 2015, 11:22

Abstract

This is just the product of me taking notes on the lecture. Nothing official. If you find mistakes or have got any questions, please feel free to contact me. Cheers!

Contents

≫A programming language is a medium for expressing ideas (not to get a computer to perform operations) and only incidentally for machines to execute.≪

Harold Abelson and Gerald Jay Sussman

Links

Site: `http://db.inf.uni-tuebingen.de/teaching/FunctionalProgrammingSS2014.html`
Ilias: `http://goo.gl/rlqbkK`

Literature

- Lipovača:
  Learn You a Haskell for Great Good
  No Starch Press 2011,
  `http://learnyouahaskell.com`

- O'Sullivan, Steward, Goerzen:
  Real World Haskell
  O'Reilly 2010
  `http://book.realworldhaskell.org`

- Haskell 2010 Report,
  `http://www.haskell.org/onlinereport/haskell2010`

1 Introduction

Computational model in Functional Programming: **reduction** (replace expression to values) In Functional Programming, expressions are formed by applying functions to values.

1. Functions as in math: $x = y \Rightarrow f(x) = f(y)$
2. Functions are values (just like numbers, text ...)

	Functional	Imperative
program construction	function application and composition	statement sequencing
execution	reduction (expression evaluation)	state changes
semantics	lambda calculus	complex (denotational)

Example

$n \in \mathbb{N}, n \geq 2$ is a prime number *if* the set of non-trivial factors is empty:

$$n \text{ is prime} \Leftrightarrow \{\ m \mid m \in \{2, \ldots, n-1\},\ n \mod m = 0\} = \emptyset$$

```haskell
-- Is n a prime number?
isPrime :: Integer -> Bool
isPrime n = factors n == []
  where
    factors :: Integer -> [Integer]
    factors n = [ m | m <- [2..n-1], mod n m == 0 ]


main :: IO ()
main = do
  let n = 43
  print (isPrime n)
```

2 Haskell Ramp-Up

(Read ≡ as “denotes the same value as”)

- Apply f to value e: `f e` (juxtaposition, “apply”, binary operator ␣, Haskell speak: infixL 10 ␣)
- ␣ has max precedence (10): `f` e_1 `+` e_2 ≡ `(f` e_1`)` `+` e_2
- ␣ associates to the left: `g f e` ≡ `(g f) e --(g f) is a function)`
- Function composition:
  - `(g . f) e` ≡ `g (f e) --(. is something like mathematical |∘| ''after'')`
  - Alternative “apply”-operator `$` (lowest precedence, associates to the right, infixR 0 $):
    `g $ f $ e` ≡ `g $ (f $ e)` ≡ `g (f e)`
  - Prefix application of binary infix operator ⊗: (⊗) e_1 e_2 ≡ e_1 ⊗ e_2
  - Infix application of binary function f: e_1 `` `f` `` e_2 ≡ `f` e_1 e_2:
    - `` 1 `elem` [1,2,3] `` `--`$(1 \in \{1, 2, 3\})$
    - `` n `mod` m ``
    - ...
  - User defined operators, built from symbols
    ! # $ % & * + / < = > ? \ ^ |~:.