Universität Tübingen

SS 2014

Torsten Grust, Alexander Ulrich

Functional Programming

Notes from

Philipp Moers
<soziflip funnycharacter gmail dot com>

Last updated: Monday 21st April, 2014, 18:03

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

0.1. Links

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

0.2. 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 *iff* the set of non-trivial factors is empty:

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

Listing 1.1: isPrime.hs

```
isPrime :: Integer -> Bool
isPrime n = factors == []
  where
    factors = [ m | m <- [2..n-1], n `mod` m == 0 ]


main :: IO ()
main = do
  let n = 42
  print (isPrime n)
```

2. Haskell Ramp-Up

(Read $\equiv$ 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 e₁ + e₂` $\equiv$ `(f e₁) + e₂`
- ␣ associates to the left: `g f e` $\equiv$ `(g f) e`
  ('`(g f)`' is a function)
- Function composition:
  - `( g . f ) e` $\equiv$ `g (f e)`
    (. is something like mathematical $\circ$ 'after')
  - Alternative 'apply'-operator `$` (lowest precedence, associates to the right, infixR ∅ $):
    `g $ f $ e` $\equiv$ `g $ (f $ e)` $\equiv$ `g (f e)`
  - Prefix application of binary infix operator $\otimes$: `(⊗) e₁ e₂` $\equiv$ `e₁ ⊗ e₂`
  - Infix application of binary function f: `` e₁ `f` e₂ `` $\equiv$ `f e₁ e₂`:
    * `` 1 `elem` [1,2,3] `` ($1 \in \{1,2,3\}$)
    * `` n `mod` m ``
    * ...
  - User defined operators, built from symbols
    ! # $ % & * + / < = > ? \^ |~:.

3. Values and Types

Any Haskell expression e has a type t (`e :: t`) that is determined at compile time. The **type assigmnent ::** is either given explicitly or inferred by the compiler.

3.1. Base Types

Type	Description	values
Int	fixed-prec. integer	0, 1, (-42)
Integer	arbitrary prec. integer	10^100
Float, Double	single/double floating point (IEEE)	0.1, 1e02
Char	Unicode character	'x', '\t', '△', '\8710'
Bool	Boolean	True, False
()	Unit	()

3.2. Type Constructors

- Build new types from existing types
- Let a, b ... denote arbitrary types (**type variables**)

Type	Description	values
(a, b)	pairs of values of type a, b	(1, True) :: (Int, Bool)
(a_1, a_2, ... a_n)	n-tuples	
[a]	list of values of type a	[True, False] :: [Bool], []::[a]
Maybe a	optional value of type a	Just 42 :: Maybe Int Nothing :: Maybe a
Either a b	choice	Left 'x' :: Either Char b Right pi :: Either a Double
IO a	I/O actions that return a value of type a	print 42 :: IO ()
a -> b	functions from a to b	isLetter :: Char -> Bool

3.3. Currying

- Recall: `e₁ ++ e₂` ≡ `(++) e₁ e₂`
- `(++) e₁ e₂` ≡ `((++) e₁) e₂`
- Function application happens one argument at a time.
  (**Currying**, Haskell B. Curry)
- Type of n-ary function is
  a_1 -> a_2 -> ... a_n -> b
- Type fun -> associates to the right, read above type as
  a_1 -> (a_2 -> (... (a_n -> b)))
- Enables **Partial Application**

3.4. Defining Values (and thus functions)

- `=` binds names to values. Names must not start with A-Z (Haskell style: camelCase)

- Define constant (0-ary function) c. Value of c is value of expression e.
  `c = e`

- Define n-ary function f with arguments x_i. f may occur in e.
  `f` x_1 x_2 $\ldots x_n$ `= e`

- A Haskell program is a set of bindings.

- Good style: give type assigmnents for top-level (global) bindings:

```
f :: a_1 -> a_2 -> b
f x_1 x_2 = e
```

3.4.1. Guards

Guards are conditional expressions (something like 'switch' in Java). They are a lot more readable and more powerful than `if ...then ...else ...`.

Guards are introduced by `|`:

```
f x_1 x_2 ... x_n
  | q_1       = e_1
  | q_2       = e_2
  ...
  | q_m       = e_m
[ | otherwise    = e_m+1 ]
```

Guards (q_i) are expressions of type Bool, evaluated top to bottom.

Listing 3.1: factorial.hs

```
-- Compute n!
fac :: Integer -> Integer
fac n = if n <= 1 then 1 else n * fac (n - 1)

-- A reformulation using guards
fac' :: Integer -> Integer
fac' n | n <= 1    = 1
       | otherwise = n * fac' (n - 1)
```

```

main :: IO ()
main = print $ (fac 10, fac’ 10)
```

3.4.2. Local Definitions

1. **Where bindings**: local definitions visible in the entire rhs of a definition.

```
f_1 x_1 x_2 ... x_n | q_1 = e_1
                    | q_2 = e_2
                    ...
                    | q_m = e_m
          where
              g_1 = ...
              g_2 = ...
              ...
              g_o
```

Listing 3.2: power.hs

```haskell
-- Efficient power computation, basic idea: x^2k = (x
   ^2)^k

power :: Double -> Integer -> Double
power x k | k == 1     = x
          | even k     = power (x * x) (halve k)
          | otherwise = x * power (x * x) (halve k)
  where
    even n  = n `mod` 2 == 0
    halve n = n `div` 2

main :: IO ()
main = print $ power 2 16
```

2. **Let expressions**: local definitions visible inside one expression.

```
let g_1 = ...
    g_2 = ...
    ...
    g_o
in e
```

3.4.3. Lists

- Recursive definitions:
  1. `[]` is a list (nil), type [] :: [a]
  2. `x:xs` is a list, if x :: a, xs :: [a]
     (x is head, xs is tail)
- Notation: `3:(2:(1:[]))` ≡ `3:2:1:[]` ≡ `[3,2,1]` ≡ `3:[2,1]`
- Law: ∀ xs :: [a] : (xs ≠ [])
  `head xs : tail xs` == xs

3.4.4. Pattern Matching

- *The* idiomatic Haskell way to define a function by cases:

```
f :: a_1 -> ... a_n -> b
f p_11 ... p_1k = e_1
f p_21 ... p_2k = e_2
...
f p_n1 ... p_nk = e_k
```

Pattern	Matches If	Bindings in e_r
constant c	$x_i == c$	
variable v	always	$v \equiv x_i$
wildcard _	always	
tuple $(p_1, \ldots p_m)$	components of x_i match patterns p	
[]	$x_i ==$ []	
$(p_1 : p_2)$	head x_i matches p_1, tail x_i matches p_2	

Listing 3.3: tally.hs

```
-- Equivalent definitions of sum (over lists of integers)

-- (1) Conditional expression
sum' :: [Integer] -> Integer
sum' xs = if xs == [] then 0 else head xs + sum' (tail xs)

-- (2) Guards
sum'' :: [Integer] -> Integer
sum'' xs | xs == []  = 0
         | otherwise = head xs + sum'' (tail xs)

-- (3) Pattern matching
sum''' :: [Integer] -> Integer
sum''' []     = 0
sum''' (x:xs) = x + sum''' xs


main :: IO ()
main = print $ (sum' [1..100], sum'' [1..100], sum'''
   [1..100])
```

Listing 3.4: take.hs

```
-- Finite prefix of a list
```

```haskell
take' :: Integer -> [a] -> [a]
take' 0 _       = []
take' _ []      = []
take' n (x:xs) = x : take' (n - 1) xs

main :: IO ()
main = print $ take' 20 [1,3..]
```

Listing 3.5: mergesort.hs

```haskell
-- Mergesort

mergeSort :: (a -> a -> Bool) -> [a] -> [a]
mergeSort (<<<) []  = []
mergeSort (<<<) [x] = [x]
mergeSort (<<<) xs  = merge (<<<) (mergeSort (<<<) ls) (mergeSort (<<<) rs)
  where
    (ls, rs) = splitAt (length xs `div` 2) xs

    merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
    merge (<<<) []     ys         = ys
    merge (<<<) xs     []         = xs
    merge (<<<) (x:xs) (y:ys)
      | x <<< y   = x:merge (<<<) xs (y:ys)
      | otherwise = y:merge (<<<) (x:xs) ys

main :: IO ()
main = print $ mergeSort (>) [1,3..19]
```