Functional Programing

Sommersemester 2014, Wintersemester 2015

Author:
Philipp Moers, Update by:
Denis Hirn

Dozent:
Torsten Grust,
Alexander Ulrich

Last updated: Friday 18th December, 2015, 16:40

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

1 Introduction **4**

2 Haskell Ramp-Up **5**

3 Values and Types **6**

4 Base Types **6**

5 Type Constructors **6**

6 Currying **6**

7 Defining Values (and thus functions) **7**

7.1 Guards 7
7.2 Local Definitions 8
7.3 Lists 8
7.4 Pattern Matching 9

8 Algebraic Data Types **10**

≫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
    ! # $ % & * + / ¡ = ¿ ? \ˆ |∼:.

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.

4 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	()

5 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`

6 Currying

- *Recall*: e_1 `++` e_2 $\equiv$ `(++)` e_1 e_2
- `(++)` e_1 e_2 $\equiv$ `((++)` e_1`)` e_2
- 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 -> (\dots (a_n -> b)))
- Enables **Partial Application**

7 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 \dots x_n = e
- A Haskell program is a set of bindings.
- Good style: give type assignments for top-level (global) bindings:

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

7.1 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 $\boxed{|}$:

```
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.

```
-- Compute n!
fac :: Integer -> Integer
fac n | n <= 1    = 1
      | otherwise = n * fac (n - 1)

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

7.2 Local Definitions

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

```
f₁ x₁ x₂ ... xₙ | q₁ = e₁
                | q₂ = e₂
                ...
                | qₘ = eₘ
    where
        g₁ = ...
        g₂ = ...
        ...
        gₒ
```

```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 g₁ = ...
    g₂ = ...
    ...
    gₒ
in e
```

7.3 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:[]))` $\equiv$ `3:2:1:[]` $\equiv$ `[3,2,1]` $\equiv$ `3:[2,1]`
- Law: $\forall$ xs :: [a] : (xs $\neq$ [])
  `head xs : tail xs` == xs

7.4 Pattern Matching

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

```
f :: a₁ -> ... aₙ -> b
f p₁1 ... p₁k = e₁
f p₂1 ... p₂k = e₂
...
f pₙ1 ... pₙk = eₖ
```

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	

```haskell
-- 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])
```

```
-- Finite prefix of a list
take' :: Integer -> [a] -> [a]
take' 0 _      = []
take' _ []     = []
take' n (x:xs) = x:take' (n-1) xs

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

```
-- Mergesort list xs, respecting ordering (<<<)

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 (<<<) l1(x:xs) l2(y:ys)
      | x <<< y    = x:merge (<<<) xs l2
      | otherwise = y:merge (<<<) l1 ys

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

8 Algebraic Data Types

(also known as **Sum-of-Product-Types**)

- *Recall*: `[]` and `(:)` are the **values constructors** for **type constructor** [a].
- Can define entirely new type T and its constructors K_i:

  ```
  data T a₁ a₂ ... aₙ = K₁ b₁₁ ... b₁ₙ₁
                        K₂ b₂₁ ... b₂ₙ₂
                        ...
                        Kᵣ bᵣ₁ ... bᵣₙᵣ
  ```

  b_{ij} types mentioning the type vars $a_1 \ldots a_n$

- Defines type constructor T and r value constructors:
  $\texttt{K}_i$ `::` $\texttt{b}_{i_1}$ `->` $\texttt{b}_{i_2}$ `-> ...` $\texttt{b}_{i_n}$ `-> T` $\texttt{a}_1$ `...` $\texttt{a}_n$
- Compare `[] :: [a]` and `(:) :: a -> [a] -> [a]`
- **Sum Type** (n=0, $n_i = 0$)

```haskell
-- Demonstrate algebraic data types: sum type

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq, Show, Ord, Enum, Bounded)

-- Is this day on a weekend?
weekend :: Weekday -> Bool
weekend Sat = True
weekend Sun = True
weekend _   = False


-- The classic rock/paper/scissor game

data Move = Rock | Paper | Scissor
  deriving (Eq)

data Outcome = Lose | Tie | Win
  deriving (Show)

-- Outcome of a game round (us vs. them)
outcome :: Move -> Move -> Outcome
outcome Rock    Scissor = Win
outcome Paper   Rock    = Win
outcome Scissor Paper   = Win
outcome us      them
  | us == them = Tie
  | otherwise  = Lose


main :: IO ()
main = do
  print $ Mon == Sun
  print $ Thu < Sat
  print [Mon .. Fri]
  print (minBound :: Weekday)
  print $ outcome Rock Rock
```

- Add `deriving (c, c, ... c)` to data declaration to define canonical operations:

c	operations
Eq	equality (==, /=)
Show	printing (show)
Ord	ordering (<, <=, max)
Enum	enumeration
Bounded	minBound, maxBound

- **Product Types** (r=1)

```haskell
-- Demonstrate algebraic data types: product type

data Sequence a = S Int [a]
  deriving (Eq, Show)


fromList :: [a] -> Sequence a
fromList xs = S (length xs) xs

(+++) :: Sequence a -> Sequence a -> Sequence a
S lx xs +++ S ly ys = S (lx + ly) (xs ++ ys)

len :: Sequence a -> Int
len (S lx _) = lx


main :: IO ()
main = do
  print $ fromList [0..9]
  print $ len (fromList ['a'..'m'] +++ fromList ['n'..'z'])
```

- **Sum-of-Product-Types**

```haskell
data Maybe a = Just a | Nothing
data Either a b = Left a | Right b
data List a = Nil | Cons a (List a)
```

```haskell
-- Our own formulation of cons lists
data List a = Nil
            | Cons a (List a)
  deriving (Show)

-- Haskell's builtin type [a] and List a are isomorphic:
--          toList . fromList = id
--    and   fromList . toList = id
toList :: [a] -> List a
toList []     = Nil
toList (x:xs) = Cons x (toList xs)

fromList :: List a -> [a]
fromList Nil         = []
fromList (Cons x xs) = x:fromList xs

-- The family of well-known list functions (combinators) can be
-- reformulated for List a
mapList :: (a -> b) -> List a -> List b
mapList f Nil         = Nil
mapList f (Cons x xs) = Cons (f x) (mapList f xs)

filterList :: (a -> Bool) -> List a -> List a
filterList p Nil                     = Nil
filterList p (Cons x xs) | p x       = Cons x (filterList p xs)
                         | otherwise = filterList p xs

liftList :: ([a] -> [b]) -> List a -> List b
liftList f = toList . f . fromList

mapList' :: (a -> b) -> List a -> List b
mapList' f = liftList (map f)

filterList' :: (a -> Bool) -> List a -> List a
filterList' p = liftList (filter p)

main :: IO ()
main = print $ fromList $ filterList' (> 3) $ mapList' (+1) $ toList [1..5]
```

```haskell
-- Use the isomorphism between [a] and List a
-- to save work when defining functions over List a:
--
--                    fromList
--         List a ------------→ [a]
--           |                    |
--         g |                    | f
--           ↓                    ↓
--         List b ←------------ [b]
```

```haskell
-- Abstract syntax tree for arithmetic expressions of literals
data Exp a = Lit a
           | Add (Exp a) (Exp a)
           | Sub (Exp a) (Exp a)
           | Mul (Exp a) (Exp a)
  deriving (Show)

ex1 :: Exp Integer
ex1 = Add (Mul (Lit 5) (Lit 8)) (Lit 2)


evaluate :: Num a => Exp a -> a
evaluate (Lit n)     = n
evaluate (Add e1 e2) = evaluate e1 + evaluate e2
evaluate (Mul e1 e2) = evaluate e1 * evaluate e2
evaluate (Sub e1 e2) = evaluate e1 - evaluate e2


main :: IO ()
main = print $ evaluate ex1
```