Funktionale Programmierung Mitschrieb

Finn Ickler

November 12, 2015

„Avoid success at all cost “

Simon Peyton Jones

Contents

List of Listings

Vorlesung 1

```haskell
-- Hello World Haskell
main :: IO ()
main = putStrLn "Chewie, we're home"
```

Code example 1: Hello World

Functional Programming (FP)

A programming language is a medium for expressive ideas (not to get a computer to perform operations). Thus programs must be written for people to read, and only incidentally for machines.

Computational Model in FP : *Reduction*

Replace expressions by their value.
IN FP, expressions are formed by applying functions to values.

1. Function as in maths: $x = y \rightarrow f(x) = f(y)$
2. Functions are values like numbers or text

	FP	Imperative
construction	function application and composition	statement sequencing
execution	reduction (expression evaluation)	state changes
semantics	λ-calculus	denotational

$n \in \mathbb{N}, n \geq 2$ is a prime number $\Leftrightarrow$ the set of non-trivial factors of n is empty.
n is prime $\Leftrightarrow \{m \mid m \in m \in \{2, \ldots, n-1\}, n mod m = 0\} = \{\}$

```
int IsPrime(int n)
{
    int m;
    int found_factor;
    found_factor
    for (m = 2; m <= n -1; m++)
    {
        if (n % m == 0)
        {
            found_factor = 1 ;
            break;
        }
    }
    return !found_factor;
}
```

Code example 2: isPrime in C

```
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 = 42
  print (isPrime n)
```

Code example 3: isPrime in Haskell

```
let xs = [ x+1 | x <- [0..9] ]
:sprint xs = _
length xs
:sprint xs = [_,_,_,_,_,_,_,_,_,_]
```

Code example 4: Lazy Evaluation in der ghci REPL

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 Function composition:

- g (f e)
- Operator "." ("after") : (g.f) e (. = ∘) = g(f (e))
- Alternative "apply" operator $ (lowest precedence, associates to the right), infix 0$): f$$e_1$+ e_2 = f ($e_1 + e_2$)

Vorlesung 2

```
cos 2 * pi
cos (2 * pi)
cos $ 2 * pi
isLetter (head (reverse ("It's a " ++ "Trap")))
(isLetter . head . reverse ) ("It's a" ++ "Trap")
isLetter $ head $ reverse $ "It's a" ++ "Trap"
```

Code example 5: Verschiedene Schreibweise einer Applikation

Prefix application of binary infix operator ⊕

$(\oplus) e_1 e_2 \equiv e_1 \oplus e_2$
(&&) `True False` ≡ `False`

Infix application of binary function f:

e_1 `` `f` `` e_2 ≡ `f` $e_1 e_2$
`` x `elem` xs `` ≡ `x ∈ xs`

User defined operators with characters : `!#%&*+/<=>?@\^|`

```haskell
epsilon :: Double
epsilon = 0.00001
(~=~) :: Double -> Double -> Bool
x ~=~ y = abs (x - y) < epsilon
infix 4 ~=~
```

Code example 6: Eigener ≈ Opperator

Values and Types

Read `::` as "has type"
Any Haskell value e has a type t (`e::t`) that is determined at compile time.
The `::` type assignment is either given explicitly or inferred by the computer

Types

Type	Description	Value
Int	fixed precision integers $(-2^{63} \dots 2^{63}-1)$	`0,1,42`
Integer	arbitrary Precision integers	`0,10^100`
Float,Double	Single/Double precision floating points	`0.1,1e03`
Char	Unicode Character	`'x','\t', '', '\8710'`
Bool	Booleans	`True, False`
()	Unit (single-value type)	`()`

```haskell
2
it :: Integer
42 :: Int
it :: Int
'a'
it :: Char
True
it :: Bool
10^100
it :: Integer
10^100 :: Double
it :: Double
```

Type Constructors

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

Type Constructor	Description	Values
(a,b)	pairs of values of types a and b	`(1,True) :: (Int, Bool)`
$(a_1, a_2, \ldots, a_n)$	n-Types	`2,False :: (Int, Bool)`
`[a]`	list of values of type a	`[] :: [a]`
`Maybe` a	optional value of type a	`Just 42 Maybe Integer` `Nothing :: Maybe a`
`Either` a b	Choice between values of Type a and b	`Left 'x' :: Either Char b` `Right pi :: Either a Double`
`IO` a	I/O action that returns a value of type a (can habe side effects)	`print 42 :: IO` () `getChar :: IO Char`
a `->` b	function from type a to b	`isLetter :: Char -> Bool`

```
(1, '1', 1.0)
it :: (Integer, Char, Double)
[1, '1', 1.0]
it :: Fehler
[0.1,1.0,0.01]
it :: [Double]
[]
it :: [t]
"Yoda"
it :: [Char]
['Y', 'o', 'd', 'a']
"Yoda"
[Just 0, Nothing, Just 2]
it :: [Maybe Integer]
[Left True, Right 'a']
it :: [Either Bool Char]
print 'x'
it :: ()
getChar
*
it :: Char
:t getChar
getChar :: Io Char
:t fst
fst :: (a,b) -> a
:t snd
snd :: (a,b) -> b
:t head
head :: [a] -> a
:t (++)
(++) :: [a] -> [a] -> [a]
```

Currying

- Recall:
  1. e_1 `++` $e_2 \equiv$ `(++)` $e_1 e_2$
  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: : a_1 `->` a_2 `. . .` `->` a_n `->` `b`
- Type constructor `->` associates to the right thus read the type as: a_1 `->` (a_2 `->` a_3 (`. . .` `->` (a_n `->` `b`)...))
- Enables partial application: "Give me a value of type a_1, I'll give you a (n-1)-ary function of type a_2 `->` a_3 `->` `. . .` `->` a_n `->` `b`

```
"Chew" ++ "bacca"
"Chewbacca"
(++) "Chew" "bacca"
"Chewbacca"
((++) "Chew") "bacca"
"Chewbacca"
:t (++) "Chew"
"Chew" :: [Char] -> [Char]
let chew = (++) "Chew"
chew "bacca"
"Chewbacca"
let double (*) 2
double 21
42
```

Vorlesung 3

Defining Values (and thus: Functions)

- = binds names to values, names must not start with A-Z (Haskell style: camelCase)
- Define constant (0-ary) c, value of c is that of expression:
  c = e
- Define n-ary function, arguments x_i and f may occur in e (no "letrec" needed)
  f x_1 $x_2 \ldots x_n = e$
- Hskell programm = set of top-level bindings (order immaterial, no rebinding)

- Good style: give type assignment for top-level bindings:
  `f :: a1 -> a2 -> b`
  `f` x_1 x_2 `= e`

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

fac2 n | n <= 1     = 1
       | otherwise = n * fac2 (n - 1)

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

Code example 7: fac in Haskell

- Guards (introduced by |).

  `f` x_1 x_2 $\ldots$ x_n
  $|q_1$ `=` e_1
  $|q_2$ `=` e_n

```
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 :: Integer -> Bool -- Nicht typisch
    even n  = n `mod` 2 == 0
    halve n = n `div` 2

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

Code example 8: Power in Haskell

- q_i (expressions of type Bool) evaluated top to bottom, first True guards "wins"
  $$\text{fac } n = \begin{cases} 1 & if n \geq 1 \\ n \cdot \text{fac(n} - 1) & else \end{cases}$$

Lokale Definitionen

1. `where` - binding : Local definitions visible in the entire right-hand-side (rhs) of a definition

$$
\begin{array}{l}
\texttt{f}\ x_1\ \ x_2\ \ \ldots\ \ x_n \\
\quad |\, q_1\ =\ e_1 \\
\quad |\, q_2\ =\ e_n \\
\ \texttt{where} \\
\qquad g_1\ \ \ldots\ =\ b_1 \\
\qquad g_i\ \ \ldots\ =\ b_i
\end{array}
$$

2. `let` - expression Local definitions visible inside an expression:

$$
\begin{array}{l}
\texttt{let}\ g_1\ \ \ldots\ =\ b_1 \\
\qquad g_2\ \ \ldots\ =\ b_1 \\
\texttt{in e}
\end{array}
$$

Haskells 2-dimensionale Syntax (Layout) (Forumbeitrag)

Hallo zusammen,
in der dritten Vorlesung hatte ich erwähnt, dass Haskells Syntax darauf verzichtet, Blöcke (von Definitionen) mittels Sonderzeichen abzugrenzen und zu strukturieren. Andere Programmiersprachen bedienen sich hier typischerweise Zeichen wie , und ;.
Haskell baut hingegen auf das sog. Layout, eine Art 2-dimensionaler Syntax. Wer schon einmal Python und seine Konventionen zur Einrückung von Blöcken hinter for und if kennengelernt hat, wird hier Parallelen sehen. Die Regelungen zu Layout lauten wie folgt und werden vom Haskell-Compiler während der Parsing-Phase angewandt:

- The first token **after** a `where`/`let` and the **first token of a top-level definition** define the upper-left corner of a box.
- The first token left of the box closes the box (offside rule).
- Insert a *{* before the box.
- Insert a *}* after the box.
- Insert a *;* before each line that starts at left box border.

Die Anwendung dieser Regeln auf dieses Beispielprogramm:

```
let y   = a * b
    f x = (x + y) / y
in  f c + f d
```

führt zur Identifikation der folgenden Box:

```
let|y   = a * b
   |f x = (x + y) / y
   └──────────

in f c + f d
```

Das Token in in der letzten Zeile steht links von der Boxgrenze im Abseits (siehe die offside rule). Der Parser führt nun die Zeichen , und ; ein und verarbeitet das Programm so, als ob der Programmierer diese Zeichen explizit angegeben hätte. (Haskell kann alternativ übrigens auch in dieser sog. expliziten Syntax geschrieben werden — das ist aber sehr unüblich, hat negativen Einfluss aufs Karma und ist vor allem für den Einsatz in automatischen Programmgeneratoren gedacht.)
Die explizite Form des obigen Programmes lautet (nach den drei letzten Regeln):

```
let {y   = a * b
    ;f x = (x + y) / y}
in   f c + f d
```

Damit ist die Bedeutung des Programmes eindeutig und es ist klar, dass bspw. nicht das folgende gemeint war (in dieser alternativen Lesart ist das Token f aus der zweiten in die erste Zeile "gerutscht"):

```
let y = a * b f
    x = (x + y) / y
in  f c + f d
```

Aus diesen Layout-Regeln ergeben sich recht einfache Richtlinien für das Einrücken in Haskell-Programmen:

- Die Zeilen einer Definition auf dem Top-Level beginnen jeweils ganz links (Spalte 1) im Quelltext.
- Lokale where / let-Definitionen werden um mindestens ein Whitespace (typisch: 2 oder 4 Spaces oder 1 Tab) eingerückt.
- Es gibt in Haskell ein weiteres Keyword (do, wird später thematisiert), das den gleichen Regeln wie where / let folgt.

Beste Grüße,
—Torsten Grust

Lists([a])

- Recursive definition:
  1. `[]` ist a list (nil), type `[] :: [a]`
  2. `x : xs` (head, tail) is a list, if `x :: a`, and `xs :: [a]`.
     `cons: (:) :: a -> [a] -> [a] -> [a]` with `infixr : 5`

- Notation: `3:(2:1:[])` ≡ `3:2:1:[]` ≡ `[3,2,1]`

```
[]
it :: [t]
[1]
it :: [Integer]
[1,2,3]
it :: [Integer]
['z']
"z"
it :: [Char]
['z','x']
"zx"
it :: [Char]
[] == ""
True
it :: Bool
[[1],[2,3]]
it :: [[Integer]]
[[1],[2,3],[]]
[[1],[2,3]]
it :: [[Integer]]
False:[]
[False]
it :: [Bool]
(False:[]):[]
it ::[[Bool]]
:t [(<),(<=),(>)]
[(<),(<=),(>)] :: Ord a => [a -> a-> Bool]
[(1,"one"),(2,"two"),(3,"three")]
it :: [(Integer,[Char])]
:t head
head :: [a] -> a
:t tail :: [a] -> [a]
head "It's a trap"
'I'
it :: Char
tail  "It's a trap"
"t's a trap"
it :: [Char]
reverse "Never odd or even"
"neve ro ddo reveN"
it :: [Char]
```

- Law ∀`xs`≠`[]`: `head xs : tail` = `xs`

```
:i String
type String = [Char]
```

Type Synonyms

- Introduce your own type synonyms. (type names : *U*ppercase) `type` t_1 `=` t_2

```
type Bits = [Integer]

type Predicate a = a -> Bool

bits :: Integer -> Bits
bits n | n == 0    =[0]
       | otherwise = (n `mod` 2) : bits (n `div`2)

isEven :: Predicate Integer
isEven n = head (bits n) == 0

main :: IO ()
main = print $ isEven 35
```

Sequence (lists of enumerable elements)

- `[x..y]` ≡ `[x,x+1,x+2,...,y]`

  ```
  ['a'..'z']
  "abcdefghijklmnopqrstuvwxyz"
  ```

- `x,s..y`≡ `[x,x+i,x+(2*i),...,y] where i = x-s`

  ```
  [1,3..20]
  [1,3,5,7,9,11,13,15,17,19]
  [2,4..20]
  [2,4,6,8,10,12,14,16,18,20]
  ```

- Infinite List `[1..]`

Vorlesung 4

Pattern Matching

The idiomatic way to define functions by cases: `f ::`a_1 `->...->`a_k `-> b`
`f` $p_{11} \dots p_{1k} = e_1$
$\vdots \;\; \vdots \;\; \vdots \;\; \vdots$
`f` $p_{m1} \dots p_{nk} = e_n$
For all e_i `:: b` on a_i call `f` $x_1 x_2 \dots x_k$ each x_i is matched against patterns $p_{i1} \dots p_{in}$ in order. Result is e_r if the rth branch is the first in which all patterns

match.

Pattern	Matches if...	Bindings in e_r
constant c	x_1 == c	
variable v	always	v = x_i
wildcard _	always	
tuple $(p_1, \ldots, p_n)$	components of x_i match type component patterns	Those bound by the component patterns
[]	x_i == []	
$p_1 : p_2$	`head` x_1 matches p_1, `tail` x_i matches p_2	
$v@p$	p matches	those bound by p and $v = x_i$

Note: In a pattern, a variable may only occur once (linear patterns only)

```haskell
--(1) if then else
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 = do
  print $ sum' [1,2,3]
  print $ sum'' [1,2,3]
  print $ sum''' [1,2,3]
```

Code example 9: sum in Haskell

Pattern matching in expressions (case)

`case e of` p_1 `|` q_{11} `->` e_{11}

$\vdots$

p_n `|` q_{n1} `->` e_{n1}

```haskell
type Dictionary a b = [(a,b)]
type Person = String
type Age = Integer

people :: Dictionary Person Age
people = [("Darth", 46), ("Chewie",200), ("Yoda", 902)]

ageOf :: Dictionary Person Age -> Person -> Maybe Age
-- The old way
--ageOf pas p | fst (head pas) == p = snd (head pas)
--            | otherwise           = ageOf (tail pas) p
ageOf []          p'              = Nothing
ageOf ((p,a):pas) p' | p == p'    = Just a
                     | otherwise = ageOf pas p'

main :: IO ()
main = do
  print $ ageOf people "Luke"
```

Code example 10: ageOf in Haskell

```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..]
```

Code example 11: take in Haskell

Vorlesung 5

Algebraic Data Types (Sum of Product Types)

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

$$\texttt{data T } a_1\ a_2\ \dots\ a_n = \begin{array}{l} K_1\ b11\ \dots\ b_{1n_1} \\ |K_2\ b_{21}\ \dots\ b_{2n_2} \\ \vdots \quad \vdots \\ |K_r\ b_{r1}\ \dots\ b_{rnr} \end{array}$$

```haskell
-- Merge two sorted lists respecting their orderings
--
-- merge (<) [0,3,5] [1,2,4] = [0,1,2,3,4,5]

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 $ merge (<) [1,3..19] [2,4..20]
```

Code example 12: merge in Haskell

- Defines *Type constructor* T and r *value constructor* with types
- `K`$_i$`::`$b_{i1} \ldots b_{ini}$`->` `T`$a_1\ a_2 \ldots a_n$
- K_i identifier with uppercase first letter or symbol starting with `:`
- Example: [weekday.hs]
    - Sum (or enumeration, choice)

```
Wed
No instance for (Show Weekday) arising from a use of print
Thu == Sun
No instance for (Eq Weekday) arising from a use of '=='
Mon > Sat
No instance for (Ord Weekday) arising form a use of '>'
```

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

c (class)	operations
`Eq`	equality (`==`,`/=`)
`Show`	printing (`show`)
`Ord`	ordering (`<`,`<=`,`max`)
`Enum`	enumeration (`[x..y]`)
`Bounded`	bounds (`minBound`,`maxBound`)

- Product, r = 1, $n_1 = 2$ ()
- Sum of Products:

    `Maybe a = Nothing | Just a`

    `data Either a b = Left a | Right a`

```haskell
--Sortes a list

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..100]
```

Code example 13: mergeSort in Haskell

```haskell
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq,Show,Ord,Enum,Bounded)
weekend :: Weekday -> Bool
weekend Sat = True
weekend Sun = True
weekend _   = False

main :: IO ()
main = do
  print $ weekend Mon
  print $ [Mon..Fri]
```

Code example 14: weekday.hs

```haskell
List a = Nil
     |Cons a (List a)
```

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

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

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 $ outcome Paper Scissor
```

Code example 15: RockPaperScissors.hs

```haskell
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'..'z'])
```

Code example 16: sequence.hs

```
data List a = Nil
            | Cons a (List a)
  deriving(Show)

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

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

mapList :: (a -> b) -> List a -> List b
mapList f  Nil         = Nil
mapList f (Cons x xs) = Cons (f x) (mapList f xs)

liftList f = toList . f . fromList

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

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

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

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

Code example 17: cons.hs

```haskell
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 = do
  print $ ex1
  print $ evaluate ex1
```

Code example 18: eval-compile-run.hs