Funktionale Programmierung Mitschrieb

Finn Ickler

October 22, 2015

„Avoid sucess at all cost “

Simon Peyton Jones

List of Listings

Vorlesung 1

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

Codebeispiel 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
sementics	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 $\Leftrightarrow \{m \mid m \in m \in \{2, \ldots, n-1\}, n mod m = 0\} = \{\}$

```c
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;
}
```

Codebeispiel 2: isPrime in C

```haskell
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)
```

Codebeispiel 3: isPrime in Haskell

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

Codebeispiel 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 precedere (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 precedure, 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"
```

Codebeispiel 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 : `!#%&*+/<=>?@\^ |`

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

Codebeispiel 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 assignement is either given explicitly or inferred by the computer

Types

Type	Description	Value
Int	fixed precision integers ($-2^{63} \ldots 2^{63}-1$)	`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)	`()`

```
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
```

Defining Values (and thus: Functions)

- = binds naems to values, names must not start with A-Z (Haskell style: camelCase)