fnord.hs

Sitzung 1: Grundlegende Konzepte, Syntax

13. August 2017

λ-Kalkül

- $f(x) = x + 1$
- $g(x, y) = x \cdot y$
- `int h(int a) { return a; }`

λ-Kalkül

- $f(x) = x + 1$
- $\Rightarrow \quad \lambda x.(x + 1)$
- $g(x, y) = x \cdot y$
- $\Rightarrow \quad \lambda x.\lambda y.(x \cdot y)$
- `int h(int a) { return a; }`
- $\Rightarrow \quad \lambda a.a$

α-Konversion

- Namen sind Schall und Rauch
- $\lambda x.\lambda y.(x + y)$

α-Konversion

- Namen sind Schall und Rauch
- $\lambda x.\lambda y.(x + y)$
- $\stackrel{\alpha}{\Rightarrow} \quad \lambda a.\lambda b.(a + b)$

α-Konversion

- Namen sind Schall und Rauch
- $\lambda x.\lambda y.(x + y)$
- $\overset{\alpha}{\Rightarrow}$ $\lambda a.\lambda b.(a + b)$
- $\overset{\alpha}{\Rightarrow}$ $\lambda horst.\lambda fritz.(horst + fritz)$

β-Reduktion

- „Funktionsanwendung“
- $(\lambda f.\lambda x.(f\ x))\ (\lambda x.\lambda y.(x + y))\ 3\ 4$

β-Reduktion

- „Funktionsanwendung“
- $(\lambda f.\lambda x.(f\ x))\ (\lambda x.\lambda y.(x + y))\ 3\ 4$
- $\overset{\alpha}{\Rightarrow}\quad (\lambda f.\lambda x.(f\ x))\ (\lambda a.\lambda y.(a + y))\ 3\ 4$

β-Reduktion

- „Funktionsanwendung“
- $(\lambda f.\lambda x.(f\ x))\ (\lambda x.\lambda y.(x + y))\ 3\ 4$
- $\overset{\alpha}{\Rightarrow}\quad (\lambda f.\lambda x.(f\ x))\ (\lambda a.\lambda y.(a + y))\ 3\ 4$
- $\overset{\beta}{\Rightarrow}\quad (\lambda x.((\lambda a.\lambda y.(a + y))\ x))\ 3\ 4$

β-Reduktion

- „Funktionsanwendung“
- $(\lambda f.\lambda x.(f\ x))\ (\lambda x.\lambda y.(x + y))\ 3\ 4$
- $\overset{\alpha}{\Rightarrow}\quad (\lambda f.\lambda x.(f\ x))\ (\lambda a.\lambda y.(a + y))\ 3\ 4$
- $\overset{\beta}{\Rightarrow}\quad (\lambda x.((\lambda a.\lambda y.(a + y))\ x))\ 3\ 4$
- $\overset{\beta}{\Rightarrow}\quad ((\lambda a.\lambda y.(a + y))\ 3)\ 4$

β-Reduktion

- „Funktionsanwendung“
- $(\lambda f.\lambda x.(f\ x))\ (\lambda x.\lambda y.(x + y))\ 3\ 4$
- $\overset{\alpha}{\Rightarrow}\quad (\lambda f.\lambda x.(f\ x))\ (\lambda a.\lambda y.(a + y))\ 3\ 4$
- $\overset{\beta}{\Rightarrow}\quad (\lambda x.((\lambda a.\lambda y.(a + y))\ x))\ 3\ 4$
- $\overset{\beta}{\Rightarrow}\quad ((\lambda a.\lambda y.(a + y))\ 3)\ 4$
- $\overset{\beta}{\Rightarrow}\quad (\lambda y.(3 + y))\ 4$

β-Reduktion

- „Funktionsanwendung“
- $(\lambda f.\lambda x.(f\ x))\ (\lambda x.\lambda y.(x + y))\ 3\ 4$
- $\overset{\alpha}{\Rightarrow}\quad (\lambda f.\lambda x.(f\ x))\ (\lambda a.\lambda y.(a + y))\ 3\ 4$
- $\overset{\beta}{\Rightarrow}\quad (\lambda x.((\lambda a.\lambda y.(a + y))\ x))\ 3\ 4$
- $\overset{\beta}{\Rightarrow}\quad ((\lambda a.\lambda y.(a + y))\ 3)\ 4$
- $\overset{\beta}{\Rightarrow}\quad (\lambda y.(3 + y))\ 4$
- $\overset{\beta}{\Rightarrow}\quad (3 + 4)$

η-Reduktion

- $(\lambda f.\lambda x.(f\ x))\ g\ 3$
- $(\lambda f.f)\ g\ 3$

η-Reduktion

- $(\lambda f.\lambda x.(f\ x))\ g\ 3$
- $\overset{\beta}{\Rightarrow} \quad (\lambda x.(g\ x))\ 3$

- $(\lambda f.f)\ g\ 3$

η-Reduktion

- $(\lambda f.\lambda x.(f\ x))\ g\ 3$
- $\overset{\beta}{\Rightarrow}\quad (\lambda x.(g\ x))\ 3$
- $\overset{\beta}{\Rightarrow}\quad g\ 3$
- $(\lambda f.f)\ g\ 3$

η-Reduktion

- $(\lambda f.\lambda x.(f\ x))\ g\ 3$
- $\overset{\beta}{\Rightarrow}\quad (\lambda x.(g\ x))\ 3$
- $\overset{\beta}{\Rightarrow}\quad g\ 3$
- $(\lambda f.f)\ g\ 3$
- $\overset{\beta}{\Rightarrow}\quad g\ 3$

η-Reduktion

- $(\lambda f.\lambda x.(f\ x))\ g\ 3$
- $\overset{\beta}{\Rightarrow}\quad (\lambda x.(g\ x))\ 3$
- $\overset{\beta}{\Rightarrow}\quad g\ 3$
- $(\lambda f.f)\ g\ 3$
- $\overset{\beta}{\Rightarrow}\quad g\ 3$
- Daher: $\lambda f.\lambda x.(f\ x)\ \overset{\eta}{\Rightarrow}\ \lambda f.f$

Haskell

- Synonyme: λ-Abstraktion, Funktion, Abbildung, Morphismus, Pfeil, Reader

Haskell

- Synonyme: λ-Abstraktion, Funktion, Abbildung, Morphismus, Pfeil, Reader
- $\lambda x.(x + 1)$
  In Haskell: \x $\rightarrow$ x+1

Haskell

- Synonyme: λ-Abstraktion, Funktion, Abbildung, Morphismus, Pfeil, Reader
- $\lambda x.(x + 1)$
  In Haskell: \x $\rightarrow$ x+1
- Funktionsdefinitionen:
  addone = \x $\rightarrow$x+1

Haskell

- Synonyme: λ-Abstraktion, Funktion, Abbildung, Morphismus, Pfeil, Reader
- $\lambda x.(x + 1)$
  In Haskell: \x $\rightarrow$ x+1
- Funktionsdefinitionen:
  addone = \x $\rightarrow$x+1
- addone' x = x+1

Haskell

- Synonyme: λ-Abstraktion, Funktion, Abbildung, Morphismus, Pfeil, Reader
- $\lambda x.(x + 1)$
  In Haskell: \x $\rightarrow$ x+1
- Funktionsdefinitionen:
  addone = \x $\rightarrow$x+1
- addone' x = x+1
- addone'' = (+1)

ghci-Sitzung

```
Prelude> let addone x = x+1
Prelude> addone 1
2
Prelude> addone (addone 2)
4
```

Typen

- Bool, Int, Integer, Float, Double, Char, ...

Typen

- Bool, Int, Integer, Float, Double, Char, ...
- Abbildungen: **Int** $\rightarrow$ **Int**, **Char** $\rightarrow$ **Bool**, ...

Typen

- Bool, Int, Integer, Float, Double, Char, ...
- Abbildungen: **Int** $\rightarrow$ **Int**, **Char** $\rightarrow$ **Bool**, ...
- Tupel: (**Int**, **Int**), (**Char** $\rightarrow$ **Bool**, **Double**), (**Bool**,**Bool**,**Bool**,(**Int**,**Int**),**Bool**), ...

Typen

- Bool, Int, Integer, Float, Double, Char, ...
- Abbildungen: **Int** $\rightarrow$ **Int**, **Char** $\rightarrow$ **Bool**, ...
- Tupel: (**Int**, **Int**), (**Char** $\rightarrow$ **Bool**, **Double**), (**Bool**,**Bool**,**Bool**,(**Int**,**Int**),**Bool**), ...
- Polymorphie: a $\rightarrow$ **Int**, (a, **Bool**), a $\rightarrow$ (b,c)

Typ von:

- ampersand = '&'
- addone x = x+1
- sincos a = (**sin** a, **cos** a)
- f $ a = f a

Typ von:

- ampersand :: **Char**
- ampersand = '&'

- addone x = x+1

- sincos a = (**sin** a, **cos** a)

- f $ a = f a

Typ von:

- ampersand :: **Char**
- ampersand = '&'
- addone :: **Int** → **Int**
- addone x = x+1

- sincos a = (**sin** a, **cos** a)

- f $ a = f a

Typ von:

- ampersand :: **Char**
- ampersand = '&'
- addone :: **Int** → **Int**
- addone x = x+1
- sincos :: **Double** →(**Double**,**Double**)
- sincos a = (**sin** a, **cos** a)

- f $ a = f a

Typ von:

- ampersand :: **Char**
- ampersand = '&'
- addone :: **Int** → **Int**
- addone x = x+1
- sincos :: **Double** →(**Double**,**Double**)
- sincos a = (**sin** a, **cos** a)
- ($) :: (a → b) → a → b
- f $ a = f a

Typ $\Rightarrow$ Implementierung?

- f :: a $\rightarrow$ a

Typ $\Rightarrow$ Implementierung?

- f :: a $\rightarrow$ a
- f x = x
- Andere totale Möglichkeiten für f existieren nicht!

Typ $\Rightarrow$ Implementierung?

- f :: a $\rightarrow$ a
- f x = x
- Andere totale Möglichkeiten für f existieren nicht!
- f' :: a $\rightarrow$ a
- f' x = **undefined**

Typ $\Rightarrow$ Implementierung?

- `f :: a -> a`
- `f x = x`
- Andere totale Möglichkeiten für `f` existieren nicht!
- `f' :: a -> a`
- `f' x = undefined`
- `f'' :: a -> a`
- `f'' = undefined`

Typ $\Rightarrow$ Implementierung?

- `f :: a -> a`
- `f x = x`
- Andere totale Möglichkeiten für f existieren nicht!
- `f' :: a -> a`
- `f' x = undefined`
- `f'' :: a -> a`
- `f'' = undefined`
- `f''' :: a -> a`
- `f''' = f'''`

Bottom ⊥

- ist Element *jedes* Typs!
- verkörpert:
  - Endlosrekursion
  - fehlgeschlagenes Pattern-Matching
  - **undefined**
  - **error** "bla"

Bottom $\bot$

- ist Element *jedes* Typs!
- verkörpert:
  - Endlosrekursion
  - fehlgeschlagenes Pattern-Matching
  - **undefined**
  - **error** "bla"
- $(\bot, 1) \neq (1, \bot) \neq (\bot, \bot) \neq \bot \neq (\lambda x \to \bot)$

Bottom $\bot$

- ist Element *jedes* Typs!
- verkörpert:
  - Endlosrekursion
  - fehlgeschlagenes Pattern-Matching
  - **undefined**
  - **error** "bla"
- $(\bot, 1) \neq (1, \bot) \neq (\bot, \bot) \neq \bot \neq (\lambda x \rightarrow \bot)$
- Halteproblem!

Strictness

- Eine Funktion f heißt *strict*, falls gilt: f $\perp = \perp$
- **id** x = x
- **snd** (a,b) = b
- wrap a = (1,a)

Strictness

- Eine Funktion f heißt *strict*, falls gilt: f $\perp = \perp$
- **id** x = x
- **id** $\perp = \perp \Rightarrow$ id ist strict!
- **snd** (a,b) = b

- wrap a = (1,a)

Strictness

- Eine Funktion f heißt *strict*, falls gilt: f $\perp = \perp$
- **id** x = x
- **id** $\perp = \perp \Rightarrow$ id ist strict!
- **snd** (a,b) = b
- **snd** $\perp = \perp \Rightarrow$ snd ist strict!
- wrap a = (1,a)

Strictness

- Eine Funktion f heißt *strict*, falls gilt: f $\perp = \perp$
- **id** x = x
- **id** $\perp = \perp \Rightarrow$ id ist strict!
- **snd** (a,b) = b
- **snd** $\perp = \perp \Rightarrow$ snd ist strict!
- wrap a = (1,a)
- wrap $\perp = (1,\perp) \Rightarrow$ wrap ist *nicht* strict!

Lazyness

- Eager Evaluation: Jeder Ausdruck wird ausgewertet, sobald er im Code auftritt.
- Lazy Evaluation: Jeder Ausdruck wird nur so weit ausgewertet, wie er gerade gebraucht wird.

```
Prelude> let wrap x = (1,x)
Prelude> wrap undefined
(1,*** Exception: Prelude.undefined
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/Err.hs:79:14
  undefined, called at <interactive>:3:6 in intera
Prelude> fst $ wrap undefined
1
```

⇒ Haskell ist faul!

Listen

```haskell
data [a] = a : [a] | []

head :: [a] → a
head (x:_) = x

tail :: [a] → [a]
tail (_:xs) = xs

null :: [a] → Bool
null [] = True
null (_:_) = False
```

lst head strict? lst null strict? lst (:[]) strict?

Unendliche Listen

```
Prelude> let repeat x = x : repeat x
Prelude> take 10 (repeat 1)
[1,1,1,1,1,1,1,1,1,1]
Prelude> take 20 (repeat 1)
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
Prelude> let countUp a = a : countUp (a+1)
Prelude> take 10 $ countUp 20
[20,21,22,23,24,25,26,27,28,29]
```

Lokale Bindings

```
splitParity :: [Int] → ([Int],[Int])
splitParity xs = (odds, evens)
  where odds = filter odd xs
        evens = filter even xs

splitParity' xs =
  let odds = filter odd xs
      evens = filter even xs
  in (odds, evens)
```

Lokale Bindings

```
splitParity :: [Int] → ([Int],[Int])
splitParity xs = (odds, evens)
  where odds = filter odd xs
        evens = filter even xs

splitParity' xs =
  let odds = filter odd xs
      evens = filter even xs
  in (odds, evens)
```

Kurzversion: splitParity '' = **filter odd &&& filter even**

Verzweigungen

```
foo :: Int → String
foo x = if x == 5 then "Bingo!" else "Nope."

foo' x | x == 5 = "Bingo!"
       | otherwise = "Nope."

foo'' 5 = "Bingo!"
foo'' _ = "Nope."

foo''' x = case x of
             5 → "Bingo!"
             _ → "Nope."

foo'''' x = case x of
              _ | x == 5 → "Bingo!"
                | otherwise → "Nope."
```

Verzweigungen

Wie ist **otherwise** definiert?

Verzweigungen

Wie ist **otherwise** definiert?

```
otherwise :: Bool
otherwise = True
```

Mehrere Parameter

- Tupelübergabe:

```
f :: (Int,Int) → Int
f (a,b) = a + b

... f (1,2) ...
```

- Currying:

```
f :: Int → (Int → Int)
f = \a → (\b → a + b)
-- oder
f :: Int → Int → Int
f a b = a + b

... f 1 2 ...
```