

Tutorium 05: λ -Kalkül

Paul Brinkmeier

19. November 2019

Tutorium Programmierparadigmen am KIT

Heutiges Programm

- Übungsblatt 4
- λ -Kalkül: Basics + Church-Zahlen

- Übungsblatt 4
- λ -Kalkül: Basics + Church-Zahlen
- λ -Kalkül in Haskell

Übungsblatt 4

2.1, 2.3 — AST: Datenstruktur

```
module AstType where

data Exp t
  = Var t
  | Const Integer
  | Add (Exp t) (Exp t)
  | Less (Exp t) (Exp t)
  | And (Exp t) (Exp t)
  | Not (Exp t)
  | If (Exp t) (Exp t) (Exp t)
```

- `t` ist Typvariable, um bspw. Ints als Namen zuzulassen
- Das kommt bspw. bei Compiler-Optimierungen zum Einsatz

2.2 — AST: Auswertung

```
module AstEval where
import AstType

type Env a = a -> Integer

eval :: Env a -> Exp a -> Integer
eval env (Var v) = env v
eval env (Const c) = c
eval env (Add e1 e2) = eval env e1 + eval env e2
```

2.3 — AST: Boolsche Ausdrücke

```
module AstEval2 where

eval :: Env a -> Exp a -> Integer
eval env (Less e1 e2) = b2i $
    (eval env e1) < (eval env e2)
eval env (And e1 e2) = b2i $
    (i2b $ eval env e1) && (i2b $ eval env e2)
eval env (Not e) = b2i $ not $ i2b $ eval env e

b2i b = if b then 0 else 1
i2b i = if i == 0 then False else True
```

- Aufgabe sorgfältig lesen, nur 0 ist „falsey“ in C
- \rightsquigarrow kann einem in der Klausur in den Arsch beißen

2.4 — AST: Show

```
module AstShow where
import AstType

instance Show t => Show (Exp t) where
  show (Const c) = show c
  show (Var    v) = show v -- Darf man wegen Show t
  show (Add a b) =
    "(" ++ show a ++ " + " ++ show b ++ ")"
  -- etc.
```

- $\text{Show } t \Rightarrow \text{Show (Exp } t) \Leftrightarrow$ „Wenn man t s anzeigen kann, kann man auch $\text{Exp } t$ s anzeigen“

3.1 — ropeLength

```
module RopeLength where
import RopeType

ropeLength :: Rope -> Int
ropeLength (Leaf s)      = length s
ropeLength (Inner l w r) = w + ropeLength r
```

3.2 — ropeConcat

```
module RopeConcat where
import RopeType
import RopeLength

ropeConcat :: Rope -> Rope -> Rope
ropeConcat r1 r2 = Inner r1 (ropeLength r1) r2

(+++) = ropeConcat
```

3.3 — ropeSplitAt

```
module RopeSplitAt where
import RopeType
import RopeConcat

ropeSplitAt :: Int -> Rope -> (Rope, Rope)
ropeSplitAt i (Leaf s)      = (Leaf sl, Leaf sr)
  where (sl, sr) = (take i s, drop i s)
ropeSplitAt i (Inner l w r)
  | i < w                    = (ll, lr ++ r)
  | i > w                    = (l ++ rl, rr)
  | otherwise                = (l, r)
  where (ll, lr) = ropeSplitAt i      l
        (rl, rr) = ropeSplitAt (i - w) r
```

3.4 — ropeInsert

```
module RopeInsert where
import RopeType
import RopeConcat
import RopeSplitAt

ropeInsert :: Int -> Rope -> Rope -> Rope
ropeInsert i toInsert rope =
    ropeL ++ toInsert ++ ropeR
    where (ropeL, ropeR) = ropeSplitAt i rope
```

3.5 — ropeDelete

```
module RopeInsert where
import RopeType
import RopeConcat
import RopeSplitAt

ropeDelete :: Int -> Int -> Rope -> Rope
ropeDelete from to rope =
    left ++ right
    where (left, _ ) = ropeSplitAt from rope
          (_, right) = ropeSplitAt to   rope
```

Wiederholung

```
module DataExamples where

data Bool = True | False

data Category = Jackets | Pants | Shoes
  deriving Show

data Filter
  = InSale
  | IsCategory Category
  | PriceRange Float Float
```

- Keyword `data` definiert *neuen* Typ
- „enum auf Meth“


```
module TypeClassExamples where

-- Ersatz für null in C-likes
-- Auch bekannt als "Maybe"
data Optional a = Present a | NoValue

instance Show a => Show (Optional a) where
    show (Present x) = show x
    show NoValue     = "null"
```

- Typklassen stellen globale Operationen für Typen bereit
- Bspw. Eq und Ord für Vergleiche, Enum für Aufzählbarkeit

λ -Kalkül

- Funktionales Gegenstück zur Turingmaschine
- Entsprechend theoretisch
- Wurde u.a. genutzt um Unlösbarkeit des Halteproblems zu zeigen
- Gibt saftig Punkte in der Klausur
 - 13P. im 19SS
 - 10P. (+15P.) im 18WS
 - 20P. (+15P.) im 18SS
- Nicht kompliziert aber „schwierig“ (wie bspw. Go oder Schach)

Ein Term im λ -Kalkül hat eine der drei folgenden Formen:

Notation	Besteht aus	Bezeichnung
x	x : Variablenname	Variable
$\lambda p.b$	p : Variablenname b : λ -Term	Abstraktion
$f\ a$	f, a : λ -Terme	Funktionsanwendung

- „ λ -Term“: rekursive Datenstruktur
- Semantik definieren wir später

Ein Term im λ -Kalkül hat eine der drei folgenden Formen:

Notation	Besteht aus	Bezeichnung
x	x : Variablenname	Variable
$\lambda p.b$	p : Variablenname b : λ -Term	Abstraktion
$f a$	f, a : λ -Terme	Funktionsanwendung

- „ λ -Term“: rekursive Datenstruktur
- Semantik definieren wir später
- Jetzt: Ergänzt das Modul Lambda um die fehlenden Typen
 - +Fragen zur ÜB-Korrektur

```
module Lambda where

data LambdaTerm
  = Var String    -- Variable
  | App () ()     -- Funktionsanwendung: f a
  | Abs () ()     -- Abstraktion: \p.b
```

- [//github.com/pbrinkmeier/pp-tut](https://github.com/pbrinkmeier/pp-tut)
- Modul x liegt in slides/demos/x.hs

Begriffe im λ -Kalkül

Begriff	Formel	Bedeutung
α -Äquivalenz	$t_1 \stackrel{\alpha}{=} t_2$	t_1, t_2 sind gleicher Struktur
η -Äquivalenz	$\lambda x.f \stackrel{\eta}{=} f$	„Unterversorgung“
Freie Variablen	$fv(\lambda p.b) = b$	Menge der nicht durch λ s gebundenen Variablen
Substitution	$(\lambda p.b) [b \rightarrow c] = \lambda p.c$	Ersetzung nicht-freier Variablen
Redex	$(\lambda p.b) t$	„Reducible expression“
β -Reduktion	$(\lambda p.b) t \Rightarrow b [p \rightarrow t]$	„Funktionsanwendung“

- $fv(t)$ bezeichnet die frei vorkommenden Variablen im Term t
- Frei vorkommend \approx nicht durch ein λ gebunden
 - $fv(x) = \{x\}$, wenn x Variable
 - $fv(f\ x) = fv(f) \cup fv(x)$
 - $fv(\lambda p.b) = fv(b) \setminus \{p\}$
- Beispiele:
 - $fv(\lambda x.x) = \emptyset$
 - $fv(\lambda x.y) = \{y\}$

- $fv(t)$ bezeichnet die frei vorkommenden Variablen im Term t
- Frei vorkommend \approx nicht durch ein λ gebunden
 - $fv(x) = \{x\}$, wenn x Variable
 - $fv(f\ x) = fv(f) \cup fv(x)$
 - $fv(\lambda p.b) = fv(b) \setminus \{p\}$
- Beispiele:
 - $fv(\lambda x.x) = \emptyset$
 - $fv(\lambda x.y) = \{y\}$
- Implementiert `fv :: LambdaTerm -> Set String`
 - Benutzt `Set`, `union`, `delete` und `fromList` aus `Data.Set`

Substitution

- Substitution ersetzt alle *freien* Variablen in einem Term
- $t[a \rightarrow b]$ — Ersetze a durch b in t
- Beispiele:
 - $a[a \rightarrow b] = b$
 - $a[b \rightarrow c] = a$
 - $(f\ x)[f \rightarrow g][x \rightarrow y] = g\ y$

Substitution

- Substitution ersetzt alle *freien* Variablen in einem Term
- $t[a \rightarrow b]$ — Ersetze a durch b in t
- Beispiele:
 - $a[a \rightarrow b] = b$
 - $a[b \rightarrow c] = a$
 - $(f\ x)[f \rightarrow g][x \rightarrow y] = g\ y$
 - $(\lambda x.f\ x)[x \rightarrow y] = \lambda x.f\ x$ (x ist nicht frei)
 - $(\lambda x.f\ x)[f \rightarrow g] = \lambda x.g\ x$ (f ist frei)

Substitution

- Substitution ersetzt alle *freien* Variablen in einem Term
- $t[a \rightarrow b]$ — Ersetze a durch b in t
- Beispiele:
 - $a[a \rightarrow b] = b$
 - $a[b \rightarrow c] = a$
 - $(f\ x)[f \rightarrow g][x \rightarrow y] = g\ y$
 - $(\lambda x.f\ x)[x \rightarrow y] = \lambda x.f\ x$ (x ist nicht frei)
 - $(\lambda x.f\ x)[f \rightarrow g] = \lambda x.g\ x$ (f ist frei)
- Implementiert

```
substitute :: (String, Term) -> Term -> Term
```

 - `type Term = LambdaTerm`
 - `fv` braucht ihr dafür nicht

- $t_1 \stackrel{\alpha}{=} t_2$ — Strukturelle Äquivalenz der Terme t_1 und t_2
- Umformung von t_1 in t_2 allein durch Substitution der (gebundenen) Variablen möglich

- $t_1 \stackrel{\alpha}{=} t_2$ — Strukturelle Äquivalenz der Terme t_1 und t_2
- Umformung von t_1 in t_2 allein durch Substitution der (gebundenen) Variablen möglich
- Bspw.:
 - $x \stackrel{\alpha}{\neq} y$, da x und y frei sind
 - $\lambda x.x \stackrel{\alpha}{=} \lambda y.y$, durch Umbenennen von x zu y
 - $f (\lambda x.y) \stackrel{\alpha}{=} f (\lambda p.y)$
 - $\lambda x.y \stackrel{\alpha}{\neq} \lambda x.z$

- $t_1 \stackrel{\alpha}{=} t_2$ — Strukturelle Äquivalenz der Terme t_1 und t_2
- Umformung von t_1 in t_2 allein durch Substitution der (gebundenen) Variablen möglich
- Bspw.:
 - $x \stackrel{\alpha}{\neq} y$, da x und y frei sind
 - $\lambda x.x \stackrel{\alpha}{=} \lambda y.y$, durch Umbenennen von x zu y
 - $f (\lambda x.y) \stackrel{\alpha}{=} f (\lambda p.y)$
 - $\lambda x.y \stackrel{\alpha}{\neq} \lambda x.z$
- Aufgabe: Implementiert instance Eq Term als α -Äquivalenz
 - Benutzt substitute!

- $\lambda x.f \ x \stackrel{\eta}{=} f$, wenn $x \notin fv(f)$
- Wie bei Haskell:
 `all list = foldl (&&) True list \Leftrightarrow`
 `all = \list -> foldl (&&) True list \Leftrightarrow`
 `all = foldl (&&) True`
- Also:
 - η -Äquivalenz: eher Umformungsschritt als Gleichheitskriterium
 - Formelle Definition von Unterversorgung

- Bisher: λ -Terme als (seltsame) Datenstruktur
Jetzt: Ausführungssemantik

- Bisher: λ -Terme als (seltsame) Datenstruktur
Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“ \Leftrightarrow
Funktionsanwendung $(f\ a)$, mit $f = \lambda p.b$
- $(\lambda p.b)\ a$

- Bisher: λ -Terme als (seltsame) Datenstruktur
Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“ \Leftrightarrow
Funktionsanwendung $(f\ a)$, mit $f = \lambda p.b$
- $(\lambda p.b)\ a \Longrightarrow b[p \rightarrow a]$

- Bisher: λ -Terme als (seltsame) Datenstruktur
Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“ \Leftrightarrow
Funktionsanwendung $(f\ a)$, mit $f = \lambda p.b$
- $(\lambda p.b)\ a \implies b[p \rightarrow a]$
- „Ausführung“ (besser: Auswertung) von λ -Termen: Anwenden der β -Reduktion, bis Term „konvergiert“
- Term konvergiert \approx Normalform \approx enthält keinen Redex mehr
 - Notation: $t \not\Rightarrow$

- Bisher: λ -Terme als (seltsame) Datenstruktur
Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“ \Leftrightarrow
Funktionsanwendung $(f\ a)$, mit $f = \lambda p.b$
- $(\lambda p.b)\ a \Longrightarrow b[p \rightarrow a]$
- „Ausführung“ (besser: Auswertung) von λ -Termen: Anwenden der β -Reduktion, bis Term „konvergiert“
- Term konvergiert \approx Normalform \approx enthält keinen Redex mehr
 - Notation: $t \not\Rightarrow$
- $id\ a = (\lambda x.x)\ a \Longrightarrow x[x \rightarrow a] = a \not\Rightarrow$

- Welcher Redex soll zuerst ausgewertet werden?
- \rightsquigarrow verschiedene Auswertungsstrategien

- Welcher Redex soll zuerst ausgewertet werden?
- \rightsquigarrow verschiedene Auswertungsstrategien
- **Volle β -Reduktion** — Beliebiger Redex
- **Normalreihenfolge** — „Linkester“ Redex

- Welcher Redex soll zuerst ausgewertet werden?
- \rightsquigarrow verschiedene Auswertungsstrategien
- **Volle β -Reduktion** — Beliebiger Redex
- **Normalreihenfolge** — „Linkester“ Redex
- **Call-by-Name** — *Nur äußerster* „linkester Redex“
- **Call-by-Value** — „Linkester Redex“, der eine Normalform als Argument hat

Normalreihenfolge

```
module LambdaN where

data LambdaTerm
  = Var String
  | App LambdaTerm LambdaTerm
  | Abs String LambdaTerm
```

- Implementiert
normalBeta :: LambdaTerm -> LambdaTerm
- Führt einen β -Reduktionsschritt in Normalreihenfolge (linkster Redex) aus
- Wenn kein Redex vorkommt, wird derselbe Term zurückgegeben
- Bindet LambdaShow ein für instance Show LambdaTerm

Begriffe im λ -Kalkül

Begriff	Formel	Bedeutung
α -Äquivalenz	$t_1 \stackrel{\alpha}{=} t_2$	t_1, t_2 sind gleicher Struktur
η -Äquivalenz	$\lambda x.f \stackrel{\eta}{=} f$	„Unterversorgung“
Freie Variablen	$fv(\lambda p.b) = b$	Menge der nicht durch λ s gebundenen Variablen
Substitution	$(\lambda p.b) [b \rightarrow c] = \lambda p.c$	Ersetzung nicht-freier Variablen
Redex	$(\lambda p.b) t$	„Reducible expression“
β -Reduktion	$(\lambda p.b) t \Rightarrow b [p \rightarrow t]$	„Funktionsanwendung“

Church-Zahlen im λ -Kalkül

$$\begin{aligned}c_0 &= ? \\c_1 &= s(c_0) \\c_2 &= s(s(c_0)) \\c_3 &= s(s(s(c_0))) \\c_8 &= s(s(s(s(s(s(s(s(c_0))))))))\end{aligned}$$

1. Die 0 ist Teil der natürlichen Zahlen
2. Wenn n Teil der natürlichen Zahlen ist,
ist auch $s(n) = n + 1$ Teil der natürlichen Zahlen

- „Zahlen“ im λ -Kalkül werden durch Funktionen in Normalform dargestellt
- $n\ f\ x = f\ n$ -mal angewendet auf x
- Bspw. $(3\ g\ y) = g\ (g\ (g\ y)) = g^3\ y$
Mit $3 = \lambda f.\lambda x.f\ (f\ (f\ x))$
- Schreibt eine λ -Funktion *succ*, die eine Church-Zahl nimmt und zu deren Nachfolger auswertet

- „Zahlen“ im λ -Kalkül werden durch Funktionen in Normalform dargestellt
- $n\ f\ x = f\ n$ -mal angewendet auf x
- Bspw. $(3\ g\ y) = g\ (g\ (g\ y)) = g^3\ y$
Mit $3 = \lambda f.\lambda x.f\ (f\ (f\ x))$
- Schreibt eine λ -Funktion *succ*, die eine Church-Zahl nimmt und zu deren Nachfolger auswertet
- Überträgt die Funktion in euren Haskell-Code und wertet *succ* c_0 durch wiederholtes Anwenden von *normalBeta* aus
- Vergleicht euer Ergebnis mit dem von Wavelength.
 - [//pp.ipd.kit.edu/lehre/misc/lambda-ide/Wavelength.html](http://pp.ipd.kit.edu/lehre/misc/lambda-ide/Wavelength.html)