Tutorium 05: λ-Kalkül

Paul Brinkmeier

8. Dezember 2020

Tutorium Programmierparadigmen am KIT

Heutiges Programm

Programm

- Übungsblatt 3
- λ-Kalkül: Wiederholung der Basics
- λ-Kalkül: Auswertungsstrategien, Church-Zahlen, SKI

Übungsblatt 3

1 — Typen und Typklassen in Haskell

```
fun7 x = if show x /= [] then x else error
```

Ansatz?

```
fun7 x = if show x /= [] then x else error
```

Ansatz?

- `error :: String -> a`
- Wegen if: Auch `x :: String -> a`
- Wegen `show x`: Constraint `Show (String -> a)`
- Also: `fun7 :: Show (String -> a) => (String -> a) -> String -> a`

2.1, 2.3 — AST: Datenstruktur

```haskell
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
```

- Zum Auswerten der Summe müssen erstmal die Summanden bekannt sein $\rightsquigarrow$ rekursiv `eval` aufrufen

2.3 — AST: Boolsche Ausdrücke

```
module AstEval2 where


eval :: Env a -> Exp a -> Integer
eval env (Less e1 e2)
  | eval env e1 < eval env e2 = 1
  | otherwise                 = 0
eval env (And e1 e2)
  | eval env e1 /= 0 && eval env e2 /= 0 = 1
  | otherwise                            = 0
eval env (Not e)
  | not $ eval env e /= 0 = 1
  | otherwise             = 0
```

- Aufgabe sorgfältig lesen, nur 0 ist „falsey“ in C
- Alternative Lösung: `bool2int`/`int2boot` implementieren

```
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.
```

- `Show t => Show (Exp t)`: „Wenn man ein `t` anzeigen kann, kann man auch eine `Exp t` 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

Cheatsheet: Lambda-Kalkül/Basics

- Terme t: Variable (x), Funktion ($\lambda x.t$), Anwendung ($t\ t$)
- α-Äquivalenz: Gleiche Struktur
- η-Äquivalenz: Unterversorgung
- Freie Variablen, Substitution, RedEx
- β-Reduktion:
  $(\lambda p.b)\ t \Rightarrow b\,[p \rightarrow t]$

Church-Zahlen

Peano-Axiome

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

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

Church-Zahlen

- „Zahlen" im λ-Kalkül werden durch Funktionen in Normalform dargestellt
- $c_n \; f \; x = f$ n-mal angewendet auf x
- Bspw. $(c_3 \; g \; y) = g \; (g \; (g \; y)) = g^3 \; y$
  Mit $c_3 = \lambda f.\lambda x.f \; (f \; (f \; x))$
- Schreibt eine λ-Funktion *succ*, die eine Church-Zahl nimmt und zu deren Nachfolger auswertet

Church-Zahlen

- „Zahlen“ im λ-Kalkül werden durch Funktionen in Normalform dargestellt
- $c_n\ f\ x = f\ n$-mal angewendet auf x
- Bspw. $(c_3\ g\ y) = g\ (g\ (g\ y)) = g^3\ y$
  Mit $c_3 = \lambda f.\lambda x.f\ (f\ (f\ x))$
- Schreibt eine λ-Funktion *succ*, die eine Church-Zahl nimmt und zu deren Nachfolger auswertet

$$\text{succ} = \lambda n.\, \lambda s.\, \lambda z.\, s\ (n\ s\ z)$$

Auswertungsstrategien

Auswertungsstrategien

$$\overbrace{\underline{\mathsf{succ}}\ \overbrace{(\underline{\mathsf{succ}}\ c_0)}^{\text{Redex 2}}}^{\text{Redex 1}}$$

mit

$$\begin{aligned} c_0 &= \qquad\quad \lambda s.\, \lambda z.\, z \\ \mathsf{succ} &= \lambda n.\, \lambda s.\, \lambda z.\, s\ (n\ s\ z) \end{aligned}$$

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

Normalreihenfolge

$$
\begin{aligned}
& \underline{\mathsf{succ}}\ (\underline{\mathsf{succ}}\ c_0) \\
\Rightarrow_\beta\quad & \lambda s.\, \lambda z.\, s\ ((\underline{\mathsf{succ}}\ c_0)\ s\ z) \\
\Rightarrow_\beta\quad & \lambda s.\, \lambda z.\, s\ (\underline{(\lambda s.\, \lambda z.\, s\ (\underline{c_0}\ s\ z))}\ s\ z) \\
\Rightarrow_\beta^2\quad & \lambda s.\, \lambda z.\, s\ (s\ (\underline{c_0}\ s\ z)) \\
\Rightarrow_\beta^2\quad & \lambda s.\, \lambda z.\, s\ (s\ z)\ \not\Rightarrow
\end{aligned}
$$

Normalreihenfolge: Linkester Redex zuerst.

Call-by-Name

$$\underline{\text{succ}}\ (\underline{\text{succ}}\ c_0)$$
$$\Rightarrow_\beta \quad \lambda s.\, \lambda z.\, s\ ((\underline{\text{succ}}\ c_0)\ s\ z) \not\Rightarrow_{\text{CbN}}$$

Call-by-Name: Linkester Redex zuerst, aber:

- Funktionsinhalte werden nicht weiter reduziert
- $\rightsquigarrow$ Betrachte nur Redexe, die nicht von einem λ umgeben sind
- So funktioniert auch Laziness in Haskell (mit Auflagen)

Call-by-Value

$$\begin{aligned} & \underline{\mathrm{succ}}\ (\underline{\mathrm{succ}}\ c_0) \\ \Rightarrow_\beta \quad & \underline{\mathrm{succ}}\ (\lambda s.\, \lambda z.\, s\ (\underline{c_0}\ s\ z)) \\ \Rightarrow_\beta \quad & \lambda s.\, \lambda z.\, \underline{(\lambda s.\, \lambda z.\, s\ (\underline{c_0}\ s\ z))}\ s\ z \ \not\Rightarrow_{\mathrm{CbV}} \end{aligned}$$

Call-by-Name: Linkester Redex zuerst, aber:

- Funktionsinhalte werden nicht weiter reduziert
- $\rightsquigarrow$ Betrachte nur Redexe, die nicht von einem λ umgeben sind
- Berechne Argumente vor dem Einsetzen
- $\rightsquigarrow$ Betrachte nur Redexe, deren Argument <u>unter CbV</u> nicht weiter reduziert werden muss

Cheatsheet: Lambda-Kalkül/Fortgeschritten

- Auswertungsstrategien (von lässig nach streng):
  - Volle β-Reduktion
  - Normalreihenfolge
  - Call-by-Name
  - Call-by-Value
- Datenstrukturen:
  - Church-Booleans
  - Church-Zahlen
  - Church-Listen
- Rekursion durch Y-Kombinator

Klausuraufgaben zum λ-Kalkül

19SS A4 — SKI-Kalkül (13P.)

$$\begin{aligned} S = &\quad \lambda x.\, \lambda y.\, \lambda z.\, x\; z\; (y\; z) \\ K = &\quad \lambda x.\, \lambda y.\, x \\ I = &\quad \lambda x.\, x \end{aligned}$$

- SKI-Kalkül kann alles, was λ-Kalkül auch kann, allein mit den Kombinatoren S, K und I
- Definiere $U = \lambda x.\, x\; S\; K$
- Aufgabe: Beweise, dass man S, K und I durch U darstellen kann:

19SS A4 — SKI-Kalkül (13P.)

$$\begin{aligned} S &= \lambda x.\, \lambda y.\, \lambda z.\, x\ z\ (y\ z) \\ K &= \lambda x.\, \lambda y.\, x \\ I &= \lambda x.\, x \end{aligned}$$

- SKI-Kalkül kann alles, was λ-Kalkül auch kann, allein mit den Kombinatoren S, K und I
- Definiere $U = \lambda x.\, x\ S\ K$
- Aufgabe: Beweise, dass man S, K und I durch U darstellen kann:
  - $U\ U\ x \overset{?}{\Longrightarrow} x$
  - $U\ (U\ (U\ U)) = U\ (U\ I) \overset{?}{\Longrightarrow} K$
  - $U\ (U\ (U\ (U\ U))) = U\ K \overset{?}{\Longrightarrow} S$

18SS A4 — Currying im λ-Kalkül (13P.)

$$\begin{aligned}
\text{pair} =& \quad \lambda a.\, \lambda b.\, \lambda f.\, f \; a \; b \\
\text{fst} =& \quad \lambda p.\, p \, (\lambda x.\, \lambda y.\, x) \\
\text{snd} =& \quad \lambda p.\, p \, (\lambda x.\, \lambda y.\, y) \\
\text{fst } (\text{pair } a \; b) =& \quad a \\
\text{snd } (\text{pair } a \; b) =& \quad b
\end{aligned}$$

- Schreibe curry und uncurry, sodass:
  - $(\text{curry } f) \; a \; b = f \; (\text{pair } a \; b)$
  - $(\text{uncurry } g) \; (\text{pair } a \; b) = g \; a \; b$

18WS A5 — Listen im λ-Kalkül (10P.)

$$\begin{aligned} \text{nil} &= \qquad \lambda n.\, \lambda c.\, n \\ \text{cons} &= \lambda x.\, \lambda xs.\, \lambda n.\, \lambda c.\, (c\ x\ xs) \end{aligned}$$

- Schreibe *head* und *tail*, sodass:
  - head (cons A B) $\overset{*}{\Longrightarrow} A$
  - tail (cons A B) $\overset{*}{\Longrightarrow} B$

18WS A5 — Listen im λ-Kalkül (10P.)

$$\begin{aligned} \text{nil} &= \qquad\qquad \lambda n.\, \lambda c.\, n \\ \text{cons} &= \lambda x.\, \lambda xs.\, \lambda n.\, \lambda c.\, (c\ x\ xs) \end{aligned}$$

- Schreibe *head* und *tail*, sodass:
  - head (cons A B) $\stackrel{*}{\Longrightarrow}$ A
  - tail (cons A B) $\stackrel{*}{\Longrightarrow}$ B
- Schreibe replicate, sodass:
  - replicate c_n $A = \underbrace{\text{cons } A \text{ (cons } A \text{ ...(cons } A \text{ nil))}}_{n \text{ mal}}$
- Erinnerung: $c_n\ f\ x = \underbrace{f\ (f\ ...(f\ x))}_{n \text{ mal}}$

18WS A5 — Listen im λ-Kalkül (10P.)

$$\begin{aligned}\text{nil} &= \qquad \lambda n.\, \lambda c.\, n \\ \text{cons} &= \lambda x.\, \lambda xs.\, \lambda n.\, \lambda c.\, (c\ x\ xs)\end{aligned}$$

- Schreibe *head* und *tail*, sodass:
  - head (cons A B) $\overset{*}{\Longrightarrow}$ A
  - tail (cons A B) $\overset{*}{\Longrightarrow}$ B
- Schreibe replicate, sodass:
  - replicate c_n $A = \underbrace{\text{cons } A \text{ (cons } A \text{ ...(cons } A \text{ nil))}}_{n \text{ mal}}$
- Erinnerung: $c_n\ f\ x = \underbrace{f\ (f\ ...(f\ x))}_{n \text{ mal}}$
- Werte aus: replicate c_3 A $\overset{*}{\Longrightarrow}$?