Tutorium 04: Entwurf in Haskell

Paul Brinkmeier

1. Dezember 2020

Tutorium Programmierparadigmen am KIT

Heutiges Programm

Programm

- Übungsblatt 3
- Lambda-Kalkül
- ... und Implementierung in Haskell

Übungsblatt 3

1 — Listenkombinatoren

```
module Polynom where

type Polynom = [Double]

cmult :: Polynom -> Double -> Polynom
cmult poly c = map (* c) poly

eval :: Polynom -> Double -> Double
eval poly x = foldr evalFold 0 poly
  where evalFold aN acc = aN + x * acc

deriv :: Polynom -> Polynom
deriv [] = []
deriv poly = zipWith (*) [1..] $ tail poly
```

2 — Collatz-Vermutung

```
module Collatz where

collatz = iterate next
  where next aN | even aN   = aN `div` 2
                | otherwise = 3 * aN + 1

num = length . takeWhile (/= 1) . collatz

maxNum a b = bestNum [(m, num m) | m <- [a..b]]
  where bestNum = foldl maxSecond (0, 0)
        maxSecond a b
          | snd a >= snd b = a
          | otherwise      = b
```

2 — Collatz-Vermutung

```
module CollatzAlt where

import Collatz (num)
import Data.Function (on)
import Data.List (maximumBy)

maxNum a b =
  maximumBy
    (compare `on` snd)
    [(m, num m) | m <- [a..b]]
```

- „eleganter“
- In der Klausur aber eher nur Funktionen aus der Prelude verwenden

3 — Stream-Kombinatoren

```
module Merge where

import Primes (primes)

merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
merge xs ys = xs ++ ys

primepowers n = mergeAll $ map primesExp [1..n]
  where mergeAll = foldl merge []
        primesExp i = map (^i) primes
```

- Für `i` in `1..n` unendliche Liste der Primzahlen hoch `i` erstellen
- Wegen Laziness: wird nur so weit ausgewertet wie nötig
- Dann: Alle miteinander vereinigen

:sprint

```
*Merge> pp3 = primepowers 3
*Merge> take 10 pp3
[2,3,4,5,7,8,9,11,13,17]
*Merge> :sprint pp3
pp3 = 2 : 3 : 4 : 5 : 7 : 8 : 9 : 11 : 13 : 17 : _
```

- `:sprint a` gibt aktuelle Speicherrepräsentation für `a` aus
- `_` steht dabei für „noch nicht ausgewertet“
- $\rightsquigarrow$ praktisch für Debugging unendlicher Listen

Wiederholung: Algebraische Datentypen

Algebraische Datentypen

TODO

λ-Kalkül

λ-Kalkül

- „Funktionales Gegenstück zur Turingmaschine“
- 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

λ-Terme

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

λ-Terme

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

λ-Terme in Haskell

```
module Lambda where


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

- github.com/pbrinkmeier/pp-tut
- Modul `x` liegt in `slides/demos/x.hs`

Begriffe im λ-Kalkül

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

Freie Variablen

- $fv(t)$ 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:
  - $fv(\lambda x.x) = \emptyset$
  - $fv(\lambda x.y) = \{y\}$

Freie Variablen

- $fv(t)$ 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
- $t\,[a \to b]$ — Ersetze a durch b in t
- Beispiele:
  - $a\,[a \to b] = b$
  - $a\,[b \to c] = a$
  - $(f\ x)\,[f \to g]\,[x \to y] = g\ y$

Substitution

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

Substitution

- Substitution ersetzt alle freien Variablen in einem Term
- $t\,[a \to b]$ — Ersetze a durch b in t
- Beispiele:
  - $a\,[a \to b] = b$
  - $a\,[b \to c] = a$
  - $(f\ x)\,[f \to g]\,[x \to y] = g\ y$
  - $(\lambda x.f\ x)\,[x \to y] = \lambda x.f\ x$ (x ist nicht frei)
  - $(\lambda x.f\ x)\,[f \to g] = \lambda x.g\ x$ (f ist frei)
- Implementiert
  `substitute :: (String, Term) -> Term -> Term`
  - `type Term = LambdaTerm`
  - `fv` braucht ihr dafür nicht

α-Äquivalenz

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

α-Äquivalenz

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

η-Äquivalenz

- $\lambda x.f\ x \overset{\eta}{=} f$, 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

β-Reduktion

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

β-Reduktion

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

β-Reduktion

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

β-Reduktion

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

β-Reduktion

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

Auswertungsstrategien

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

Auswertungsstrategien

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

- **Volle β-Reduktion** — Beliebiger Redex
- **Normalreihenfolge** — „Linkester“ Redex

Auswertungsstrategien

- 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 (linkester Redex) aus
- Wenn kein Redex vorkommt, wird derselbe Term zurückgegeben
- Bindet `LambdaShow` ein für `instance Show LambdaTerm`

Church-Zahlen im λ-Kalkül

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
- $n\ f\ x = f$ 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

Church-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
- Übertragt 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