Funktionale Programmierung
- Grundlagen -

λ

λ-Ausdrücke

1) Variablen: x, y, z, ...V,V_1,...
2) Applikationen: (E_1 E_2)
   Bsp. (sum($\underline{m}$, $\underline{n}$)) bezeichnet $\underline{m+n}$
3) Abstraktionen: (λV.E)
   Bsp. λx.sum(x,$\underline{1}$) Inkrementierfunktion

in BNF:

< λ-Ausdruck> ::= <Variable>
| (< λ-Ausdruck> <(λ-Ausdruck)>
| λ<Variable>.< λ-Ausdruck>)>

Vereinbarungen zur Notation

- Unterstrichene Bezeichnungen stehen für λ-Teilausdrücke
  Bsp. $\underline{1}$ steht für den λ-Ausdruck, der den Wert 1 repräsentiert.
- Funktionen werden linksassoziativ ausgewertet:
  $E_1\ E_2 \ldots E_n$ steht für $(\ldots((E_1\ E_2)\ E_3)\ldots E_n)$
  $\lambda V.E_1 \ldots E_n$ steht für $(\lambda V.(E_1 \ldots E_n))$
  $\lambda V_1 \ldots V_n.E$ steht für $(\lambda V_1.(\ldots(\lambda V_n.(E)\ldots))$
  Bsp. $\lambda x.y.\mathrm{add}\ y\ x$ bezeichnet $(\lambda x.\ \lambda y.((\mathrm{add}\ y)x)))$
- E[E‘/V] steht für das Ergebnis einer textuellen Ersetzung (Substitution) jeder freien Variablen V in E durch E‘.

Frei und gebundene Variablen

gebunden

$$(\lambda x.y\ x)(\lambda y.x\ y)$$

frei

Eine Substitution E[E‘/V] wird genau dann als gültig bezeichnet, wenn keine freie Variable in E‘ zu einer gebundenen Variablen in E[E‘/V] wird.

Konvertierungsregeln

α-Regel
Jede Abstraktion der Form λV.E kann zu λV E[V‘/V] konvertiert werden, vorausgesetzt, die Substitution von V in E durch V‘ ist gültig.

β-Regel
Jede Applikation der Form $(\lambda V.E_1)E_2$ kann zu $E_1[E_2/V]$ konvertiert werden, vorausgesetzt, die Substitution von V in E_1 durch E_2 ist gültig.

η-Regel
Jede Abstraktion der Form λV.(E V), in der V in E nur gebunden auftritt, kann zu E reduziert werden

Gleichheit und Identität

Zwei λ-Ausdrücke nennen wir **gleich**, wenn sie mit einer Sequenz von Anwendungen der Konvertierungsregeln ineinander überführt werden können (vorwärts wie rückwärts).

$$E_1 = E_2$$

Zwei λ-Ausdrücke nennen wir **identisch**, wenn sie aus genau der gleichen Folge von Zeichen bestehen.

$$E_1 \equiv E_2$$

Bsp.

(λx.x)((λy.y)1) = 1

(λx. λy.add x y)3 4 = add 3 4

Repräsentation von Objekten

LET <u>true</u> = $\lambda x.\ \lambda y.x$
LET <u>false</u> = $\lambda x.\ \lambda y.y$
LET <u>not</u> = $\lambda t.t$ <u>false</u> <u>true</u>

Exemplarischer Nachweis „üblicher" Eigenschaften:

<u>not</u> <u>true</u>	= ($\lambda t.t$ <u>false</u> <u>true</u>) <u>true</u>	(* Def. von not *)
	= <u>true</u> <u>false</u> <u>true</u>	(* β-Regel *)
	= ($\lambda x.\ \lambda y.x$) <u>false</u> <u>true</u>	(* Def. von true *)
	= ($\lambda y.$<u>false</u>) <u>true</u>	(* β-Regel *)
	= <u>false</u>	(* β-Regel *)

Bedingte Ausdrücke:
LET $(E \rightarrow E_1|E_2) = (E\ E_1\ E_2)$

es gilt:

(<u>true</u> $\rightarrow E_1|E_2$) = <u>true</u> $E_1\ E_2$
= $(\lambda x\ y.x)\ E_1\ E_2$
= E_1

(<u>false</u> $\rightarrow E_1|E_2$) = <u>false</u> $E_1\ E_2$
= $(\lambda x\ y.y)\ E_1\ E_2$
= E_2

Tupel:

LET <u>fst</u> = $\lambda p.p$ <u>true</u>
LET <u>snd</u> = $\lambda p.p$ <u>false</u>
LET (E_1,E_2) = $\lambda f.f\ E_1\ E_2$

es gilt:

fst (E_1, E_2) = ($\lambda p.p$ <u>true</u>) (E_1, E_2)
= (E_1, E_2) <u>true</u>
= ($\lambda f.f\ E_1\ E_2$) <u>true</u>
= <u>true</u> $E_1\ E_2$
= $(\lambda x\ y.x)\ E_1\ E_2$
= E_1

LET $(E_1, E_2, \ldots, E_n) = (E_1,(E_2,(\ldots (E_{n-1},E_n)\ldots)))$

Zahlen und arithmetische Operationen

n-te Applikation:

LET E^0 E' = E'

LET E^n E' = $\underbrace{E(E(\ldots(E}_{\text{n Es}}$ E')...))

Zahlen:

LET $\underline{0}$ = λf x.x

LET $\underline{1}$ = λf x.f x

LET $\underline{2}$ = λf x.f(f x)

...

LET $\underline{n}$ = λf x f^n x

arithmetische Operationen:

LET $\underline{\text{suc}}$ λn f x.n f(f x)

LET $\underline{\text{add}}$ λm n f x.m f (n f x)

LET $\underline{\text{iszero}}$ = λn.n (λx. $\underline{\text{false}}$) $\underline{\text{true}}$

Vorgängerfunktion

Idee: der Vorgänger von n wird definiert, indem in der Repräsentation von $\underline{n}$ = (λf x . f^n x) die erste Anwendung von f in f^n entfernt wird.

Hilfsfunktion prefn:

„n Anwendungen von prefn auf (true, x) sollen n-1 Anwendungen von f auf x ergeben, also ($\underline{\text{prefn}}$ f)^n ($\underline{\text{true}}$, x) = ($\underline{\text{false}}$, f^{n-1}x) für n>0"

LET $\underline{\text{prefn}}$ = λf p.($\underline{\text{false}}$, ($\underline{\text{fst}}$ p → $\underline{\text{snd}}$ p | (f ($\underline{\text{snd}}$ p))))

Vorgängerfunktion pre:

LET $\underline{\text{pre}}$ = λn f x.$\underline{\text{snd}}$ (n ($\underline{\text{prefn}}$ f) ($\underline{\text{true}}$, x))

$\underline{\text{pre}}$ $\underline{n}$ f x	= $\underline{\text{snd}}$ ($\underline{n}$ ($\underline{\text{prefn}}$ f) ($\underline{\text{true}}$, x))	(*definition von pre*)
	= $\underline{\text{snd}}$ (($\underline{\text{prefn}}$ f)^n ($\underline{\text{true}}$, x))	(*definition von $\underline{n}$*)
	= $\underline{\text{snd}}$ ($\underline{\text{false}}$, f^{n-1} x)	(* n -anwendungen von prefn *)
	= f^{n-1} x	

Hilfsfolie zur Vorgängerfunktion

LET prefn = λf p.(false, (fst p → snd p | (f (snd p))))

$(\text{prefn } f)^n\ (\text{true}, x)$

$= \underbrace{(\text{prefn } f)(\ (\text{prefn } f)(\ (\text{prefn } f)(\ \ldots\ (\text{prefn}}_{n\text{-mal}}\ f\ (\text{true}, x))\ \ldots\ )))$

= .. “.. (λf p.(false, *snd p*)(true, x)))...

= .. “.. (prefn f (false, x)) ...

= .. “.. (false, f x)

= (prefn f (false, f(f(f...(f x)...))))

$= (\text{false}, f^{n-1}\ x)$

$\text{pre } n\ f\ x = f^{n-1}\ x$

$\lambda f\ x.\ \text{pre } n\ f\ x = \lambda f\ x\ .f^{n-1}\ x$ (*Leibnitz bzw. Extensionalität *)

(* η-Regel bzw. Def. von n-1 *)

pre n = n-1

Rekursion

Idee: mult m n = add n (add n (...(add n 0)...))

mult m n = (iszero m→ 0 | add n (mult (pre m) n)) bzw.

mult = λm n .(iszero m→ 0 | add n (mult (pre m) n))

!!! ungebunden !!!

Lösung: Definiere einen zunächst ein Y mit der folgenden Eigenschaft:

Y E = E (Y E)

Def. E‘ ist ein Fixpunkt von E, wenn E‘ = E E‘

Def. Fix heißt Fixpunktoperator, wenn
Fix E = E (Fix E) ∀ E

LET Y = λf. (λx. f(x x)) (λx. f(x x))

LET Y = λf. (λx. f(x x)) (λx. f(x x))

Die rekursiv definierte Multiplikationsfunktion:

LET mult = Y multfn

wobei

LET multfn = λ**f** m n .(iszero m→ 0 | add n (**f** (pre m) n))

mult m n = (Y multfn) m n (*Def. von mult *)
= multfn (Y multfn) m n (*Fix-Punkt-Eigenschaft von Y*)
= multfn mult m n (*Def. von mult *)
= (λ**f** m n .(iszero m→ 0 | add n (**f** (pre m) n))) mult m n
= (iszero m→ 0 | add n (mult (pre m) n)) (*β-Regel*)

Eine Gleichung der Form fktn x_1 ... x_n = E wird rekursiv genannt, wenn fktn in E als freie Variable vorkommt.

a) Schreibe die gewünschte Funktion als Gleichung der Form:
fktn x_1 ... x_n = ... fktn ....

b) Definiere die Funktion durch: LET fktn = Y (λf x_1 ... x_n . ..... f ....)

Repräsentation von Funktionen

Im λ-Kalkül gibt es zwei Varianten zur Repräsentation:

a) (f x ,... $x_{n)}$) (* gecurryt *) Bsp. add m n = m+n

b) f (x_1,...,x_n) sum (m, n) = m+n

LET curry = λf x_1 x_2. f (x_1, x_2)

LET uncurry = λf p. f (fst p) (snd p)

nun gilt: sum = uncurry add

sum (m, n) = uncurry add (m, n)
= (λf p. f (fst p)(snd p)) add (m, n)
= add (fst (m, n)) (snd (m, n))
= add m n
= m+n

Church-Rosser-Theorem

Wenn $E_1 = E_2$, dann existiert ein E, so dass $E_1 \rightarrow E$ und $E_2 \rightarrow E$.

1) Folgerung:
Zwei λ-Ausdrücke können in beliebiger Reihenfolge ausgewerten werden.
Beweis:
Gelte $E \rightarrow E_1$ und $E \rightarrow E_2$ und E_1, E_2 sind Normalformen, dann
folgt: $E=E_1$ und $E=E_2$ (* Def. der Gleichheit *)
und damit $E_1=E_2$.
Hieraus folgt: $\exists E'$ mit $E_1 \rightarrow E'$ und $E_2 \rightarrow E'$. (* Church-Rosser *)
Da E_1 und E_2 Normalformen sind, können höchstens α-Reduktionen durchgeführt werden, um E' zu erhalten. Daher müssen die Ausdrücke E_1 und E_2 bis auf die Namensgebung der gebundenen Variablen identisch sein.

2) Ähnlich kann gezeigt werden, dass mit m≠n auch $\underline{m} \neq \underline{n}$ folgen muss.

Korollar zum Church-Rosser-Theorem

(i) Wenn E eine Normalform hat,
dann ex. $E \rightarrow E'$ mit E' ist Normalform

(ii) Wenn E eine Normalform hat und $E=E'$,
dann hat E' eine Normalform

(iii) Wenn $E=E'$ und sowohl E als auch E' sind Normalformen,
dann sind E und E' identisch (außer α-Reduktionen).

Normalisierungs Theorem

Wenn E eine Normalform hat, dann erhält man aus E eine Normalform, indem wiederholend die am weitesten links arbeitende β- oder η-Reduktion angewendet wird (ggf. nach einer vorhergehenden α-Reduktion).