Funktionale Programmierung
- Grundlagen -

λ

λ-Ausdrücke

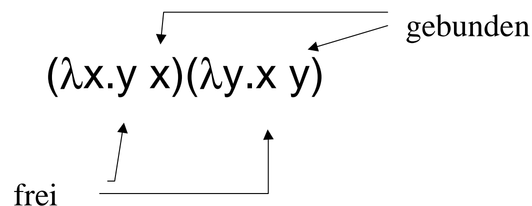
1) Variablen: x, y, z, ...V,V_1,...
2) Applikationen: (E_1 E_2)
   Bsp. (sum(m, n)) bezeichnet m+n
3) Abstraktionen: (λV.E)
   Bsp. λx.sum(x,1) Inkrementierfunktion

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:
  $E_1\ E_2\ ...\ E_n$ steht für $(...((E_1\ E_2)\ E_3)...E_n)$
  $\lambda V.E_1...\ E_n$ steht für $(\lambda V.(E_1\ ...\ E_n))$
  $\lambda V_1...V_n.E$ steht für $(\lambda V_1.(...(\lambda V_n.(E)...))$
  Bsp. λx.y.add y x bezeichnet (λx. λy.((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



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 true = λx. λy.x
LET false = λx. λy.y
LET not = λt.t false true

Exemplarischer Nachweis „üblicher“ Eigenschaften:

not true	= (λt.t false true) true	(* Def. von not *)
	= true false true	(* β-Regel *)
	= (λx. λy.x) false true	(* Def. von true *)
	= (λy.false) true	(* β-Regel *)
	= false	(* β-Regel *)

Bedingte Ausdrücke:

LET $(E \rightarrow E_1|E_2) = (E\ E_1\ E_2)$

es gilt:

$(\text{true} \rightarrow E_1|E_2)$ = true $E_1\ E_2$
= (λx y.x) $E_1\ E_2$
= E_1

$(\text{false} \rightarrow E_1|E_2)$ = false $E_1\ E_2$
= (λx y.y) $E_1\ E_2$
= E_2

Tupel:

LET fst = λp.p true
LET snd = λp.p false
LET (E_1,E_2) = λf.f $E_1\ E_2$

es gilt:

fst (E_1, E_2) = (λp.p true) (E_1, E_2)
= (E_1, E_2) true
= (λf.f $E_1\ E_2$) true
= true $E_1\ E_2$
= (λx 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‘ = $\underbrace{E(E(...(E}_{\text{n Es}}$ E‘)...))

Zahlen:

LET <u>0</u> = λf x.x

LET <u>1</u> = λf x.f x

LET <u>2</u> = λf x.f(f x)

...

LET <u>n</u> = λf x f^n x

arithmetische Operationen:

LET <u>suc</u> λn f x.n f(f x)

LET <u>add</u> λm n f x.m f (n f x)

LET <u>iszero</u> = λn.n (λx. <u>false</u>) <u>true</u>

Vorgängerfunktion

Idee: der Vorgänger von n wird definiert, indem in der Repräsentation von <u>n</u> = (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 (<u>prefn</u> f)^n (<u>true</u>, x) = (<u>false</u>, f^{n-1}x) für n>0“

LET <u>prefn</u> = λf p.(<u>false</u>, (<u>fst</u> p → <u>snd</u> p | (f (<u>snd</u> p))))

Vorgängerfunktion pre:

LET <u>pre</u> = λn f x.<u>snd</u> (n (<u>prefn</u> f) (<u>true</u>, x))

<u>pre</u> <u>n</u> f x	= <u>snd</u> (<u>n</u> (<u>prefn</u> f) (<u>true</u>, x))	(*definition von pre*)
	= <u>snd</u> ((<u>prefn</u> f)^n (<u>true</u>, x))	(*definition von <u>n</u>*)
	= <u>snd</u> (<u>false,</u> f^{n-1} x)	(* n -anwendungen von prefn *)
	= f^{n-1} x	

Hilfsfolie zur Vorgängerfunktion

LET <u>prefn</u> = λf p.(<u>false</u>, (<u>fst</u> p → <u>snd</u> p | (f (<u>snd</u> p))))

(<u>prefn</u> f)^n (<u>true</u>, x)
= (<u>prefn</u> f)((<u>prefn</u> f)((<u>prefn</u> f)(... (<u>prefn</u> f (<u>true</u>, x)) ...)))
(n-mal)
= .. ".. (λf p.(<u>false</u>, *snd p*)(<u>true</u>, x)))...
= .. ".. (<u>prefn</u> f (<u>false</u>, x)) ...
= .. ".. (false, f x)
= (<u>prefn</u> f (false, f(f(f...(f x)...))))
= (<u>false</u>, f^{n-1} x)

<u>pre</u> <u>n</u> f x = f^{n-1} x
λf x. <u>pre</u> <u>n</u> f x = λf x .f^{n-1} x (*Leibnitz bzw. Extensionalität *)
(* η-Regel bzw. Def. von <u>n-1</u> *)
<u>pre</u> <u>n</u> = <u>n-1</u>

Rekursion

Idee: <u>mult</u> m n = <u>add</u> n (<u>add</u> n (...(<u>add</u> n <u>0</u>)...))

<u>mult</u> m n = (<u>iszero</u> m→ <u>0</u> | <u>add</u> n (<u>mult</u> (<u>pre</u> m) n)) bzw.

<u>mult</u> = λm n .(<u>iszero</u> m→ <u>0</u> | <u>add</u> n (<u>mult</u> (<u>pre</u> m) n))

!!! ungebunden !!!

Lösung: Definiere einen zunächst ein <u>Y</u> mit der folgenden Eigenschaft:
<u>Y</u> E = E (<u>Y</u> E)

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

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

LET <u>Y</u> = λf. (λx. f(x x)) (λx. f(x x))

LET <u>Y</u> = λf. (λx. f(x x)) (λx. f(x x))

Die rekursiv definierte Multiplikationsfunktion:

LET <u>mult</u> = <u>Y</u> <u>multfn</u>

wobei

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

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

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

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

b) Definiere die Funktion durch: LET <u>fktn</u> = <u>Y</u> (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 <u>m</u> <u>n</u> = <u>m+n</u>

b) f (x_1,...,x_n) sum (<u>m</u>, <u>n</u>) = <u>m+n</u>

LET <u>curry</u> = λf x_1 x_2. f (x_1, x_2)
LET <u>uncurry</u> = λf p. f (<u>fst</u> p) (<u>snd</u> p)

nun gilt: sum = uncurry add

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

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 $m \neq n$ 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).