27.10.05

Theorie der Programmierung I

(Mitschrift von Lars Friedrich: email@lars-friedrich-home.de)

Intuition: Ein frei vorkommender Name ist einer, der nicht bekannt ist, d.h. der weder als Parameter noch durch eine Deklaration eingeführt wurde.

Definition: Die Menge aller in e frei vorkommenden Namen (Bezeichnung free(e)) ist induktiv definiert durch:

free(c) = ∅
free(id) = {id}
free($e_1 e_2$) = free(e_1) ∪free(e_2)
free(if…) = analog
free(λid.e) = free(e)\{id}
free(let…) = siehe Übung
Ein Ausdruck e heißt **abgeschlossen**, wenn free(e)= ∅.

In realen Programmiersprachen werden nur abgeschlossene Ausdrücke als „lauffähiges Programm“ zugelassen (bei nicht abgeschlossenen Ausdrücken: Fehlermeldung wegen unbekanntem Namen).

Hier: Wir lassen auch Ausdrücke mit freiem Namen zu, deren Berechnung bleibt stecken, sobald man auf einen freien Namen stößt.

Lemma 3: Wenn e→e’, dann gilt free(e’) ⊆ free(e) (d.h., beim small step entstehen keine neuen freien Namen).

Definition der Substitution

Intuition: Beispiele für Parameterübergabe, d.h. für (BETA-V)

1. (λx. * x x) 5 → * 5 5 Der aktuelle Parameter (5) wird für alle Vorkommen von x eingesetzt, die durch dieses λx gebunden sind.
2. (λx. (λx. * x x) (+ x 1)) 5 → (λx. * x x) (+ 5 1) Der aktuelle Parameter wird nicht durch Vorkommen von x eingesetzt, die durch ein anderes (weiter innen stehendes „lokales“) x gebunden sind.

Mit anderen Worten: Es wird für die Vorkommen von x eingesetzt, die „frei werden“, wenn man das äußere λx weglässt.

Weiter Komplikation entsteht, wenn der aktuelle Parameter freie Namen enthält.

3. (λx. λy. + y x)
   Die Funktion, die ihren Parameter y in x erhöht.
   Die Funktion, die jedem x die Funktion zuordnet, die ihren Parameter x um y erhöht.
   (λx. λy. + y x) 5 → λy. + y 5 Die Funktion, die ihren Parameter um 5 erhöht.
   (λx. λy. + y x) z → λy. + y z Die Funktion, die ihren Parameter um z erhöht.
   (λx. λy. + y x) y (naiv)→ λy. + y y Die Funktion, die den Parameter verdoppelt. Es ist eine neue „falsche“ Bindung entstanden (so genannte „Namenskollision“)
   Korrekte Variante: Das lokale y wird vor dem Einsetzen umbenannt („gebundene Umbenennung“).
   (λx. λy. + y x) y → λy’. + y’ y Die Funktion, die ihren Parameter um y erhöht.

Definition: e' [e/id] ist induktiv definiert durch:

$c\,[e/id] = c$

$id'\,[e/id] = \begin{cases} e & \text{falls } id' = id \text{ (syntaktisch)} \\ id' & \text{sonst} \end{cases}$

$(e_1 e_2)\,[e/id] = (e_1\,[e/id])(e_2\,[e/id])$

$(if\ldots)\,[e/id] = \text{analog}$

$(\lambda id'.\ e_1)\,[e/id] = \begin{cases} \lambda id'.\ e & \text{falls } id = id' \\ \lambda id'.\ e_1[e/id] & \text{falls } id \neq id' \text{ und keine Namenskollision, d.h. wenn } id' \notin free(e) \\ \lambda id''.\ e[id''/id'][e/id] & \text{falls } id' \in free(e), \text{ dabei ist } id'' \text{ ein} \\ & \underline{\text{neuer}} \text{ Name, d.h. } id'' \notin free(e) \cup free(\lambda id'.\ e_1) \cup \{id\} \end{cases}$

gebundener Name — Namenskollision

Bemerkung: 2. Zeile kann als Spezialfall der 3. Zeile aufgefasst werden.
Wenn wir es nur mit abgeschlossenen Ausdrücken zu tun haben (wi in echten Programmiersprachen), dann ist free(e)=∅, also tritt die 3. Zeile (Namenskollision) nicht auf.

Definition von free(e) und e[e'/id] übertragt sich auf let-Ausdrücke, indem man sie als syntaktischen Zucker auffasst (Übungsaufgabe).

Weiterer syntaktischer Zucker:

not für $\lambda x.$ if x then false else true
e_1 && e_2 für if e_1 then e_2 else false
e_1 || e_2 für if e_1 then true else e_2
Außerdem Infixnotation für die binären Operatoren, d.h. e_1 op e_2 für op e_1 e_2, z.B. x+y für +xy

Konkrete Syntax: Applikation hat höchste Priorität
dann $*$ / mod |
dann + - | alle linksassoziativ, z.B.
dann = < > ≤ ≥ | e_1-e_2-e_3 steht für (e_1-e_2)-e_3
dann && || |
dann if let λ

Abgeleitete small step Regeln für && und ||:

e_1 && e_2 → e_1' && e_2
steht für — steht für
if e_1 then e_2 else false → if e_1' then e_2 else false
true && e → e
steht für
if true then e else false → e (COND-TRUE)
false && e → false
steht für
if false then e else false → false (COND-FALSE)

Abgeleitete Regel (AND-EVAL):

$$\frac{e_1 \to e_1'}{e_1 \,\&\&\, e_2 \to e_1' \,\&\&\, e_2}$$

(AND-TRUE) true && e → e

(AND-FALSE) fals && e → false

Analog:

(OR-EVAL) $\dfrac{e_1 \to e_1'}{e_1 \,||\, e_2 \to e_1' \,||\, e_2}$
(OR-TRUE) true || e → true
(OR-FALSE) false || e → e