15.11.05

Theorie der Programmierung I

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

Definition: Sei (A, <) eine irreflexive partielle Ordnung, $a \in A$ heißt minimal (in A), wenn es kein $b \in A$ mit $b < 0$ gibt.

Satz 5: (Noethersche Ordnung)
Sei (A, <) eine Noethersche Ordnung. Um zu beweisen, dass eine Eigenschaft P für alle $a \in A$ gilt, genügt es zu zeigen, dass für jedes $a \in A$:
(1) Induktionsanfang: Wenn a minimal, dann gilt P(a)
(2) Induktionsschritt: Wenn P(b) für alle b<a gilt, so gilt auch P(a).
(Bemerkung: (1) kann man weglassen, denn es ist Spezialfall von (2). Für minimale a existiert kein b mit b<a, also ist in (2) P(a) ohne Voraussetzung zu zeigen.)

Beweis: Sei (2) für die Eigenschaft P erfüllt. Angenommen, es existiert ein $a_0 \in A$ mit $\neg P(a_0)$. Wegen (2) muss dann ein $a_1 < a_0$ existieren mit $\neg P(a_1)$. Analog erhält man $a_2 < a1$ in A. Widerspruch zur Definition Noethersche Ordnungen.

Wozu big step Semantik?

(a) Beweis von Eigenschaften der Programmiersprache
(b) Beweis von Eigenschaften einzelner Programme („Korrektheitsbeweis“)
(c) Vorlage zur Implementation eines Interpreters

Zu (b) Bsp.: Programm zum schnellen Potenzieren
let rec exp x y = if y = 0 then 1 else if y mod 2 = 0 then exp (x * x) (y / 2) else x * exp (x * x) (y / 2) in exp m n

Behauptung: Für alle m, n ∈ Int mit $n \geq 0$ gilt: Das Programm terminiert mit Resultat m^n (wenn man Integer overflow ignoriert).

Beweis: (Zuerst) „informell“, d.h. mit informativem Verständnis der Semantik.
Induktion über n.

Induktionsannahme n = 0: exp m 0 bewirkt: if 0=0 then 1 else … liefert als Resultat $1 = m^0$.

Induktionsschritt n>0: exp m n bewirkt: if n=0 then 1 else … Das führt zum else-Teil:
if n mod 2 = 0 …

n gerade: Führt zu (m*m) (n/2)
liefert m^2 | | liefert die Zahl n/2
Wegen n>0 ist n/2>n, also Induktionsannahme anwendbar, d.h. der Aufruf liefert $(m^2)^{n/2} = m^{2*n/2} = m^n$
| n gerade

n ungerade: Führt zu m * exp (m*m) (n/2) liefert nach Induktionsannahme wieder $m^{2*n/2} = m^{n-1}$
| n ungerade
also Ergebnis $m*m^{n-1}=m^n$

jetzt formal: d.h. mit der mathematischen Definition der big step Semantik:

Dazu: abgeleitete Regel

(APP*) Für alle $n \geq 2$ gilt:

$$\frac{e_1 \Downarrow v_1 \ \ldots \ e_n \Downarrow v_n \qquad v_1 \ldots v_n \Downarrow v}{e_1 \ldots e_n \Downarrow v}$$

Beweis: Induktion über n

Induktionsannahme n = 2: (APP*) = (APP)

Induktionsschritt n>2: Wenn $v_1 \ldots v_n \Downarrow v$ gilt, dann kann es nur mit (APP) aus Prämissen der Form:
(1) $v_1 \ldots v_{n-1} \Downarrow v$

(2) $v_n \Downarrow v_n$
(3) $v_{n-1} v_n \Downarrow v$ entstanden sein.
Aus (1) und den Prämissen $e_1 \Downarrow v_1 \ldots e_{n-1} \Downarrow v_{n-1}$ folgt nach Induktionsannahme mit (APP*) (4) $e_1 \ldots e_{n-1} \Downarrow v'$
Aus (3), (4) und der Prämisse $e_n \Downarrow v_n$ folgt mit (APP) $e_1 \ldots e_n \Downarrow v$.

zu zeigen ist: für alle m, n ∈ Int, n≥0:
let exp = rec exp λy. If y=0 … in exp m n ⇓ m^n
wegen (UNFOLD), (LET-EXEC) genügt es zu zeigen: (λx. λy. if y=0 then …) m n ⇓ m^n

Induktion über n:

n=0: (1) (λx. λy. …) m 0 ⇓ 1
↳ (2) (BETA-V) if 0=0 then 1 else … ⇓ 1
↳ (3) 0=0⇓ true (COND-TRUE)
↳ (4) (OP) 1⇓1 (VAL)

n>0: (1) (λx. λy. …) m n ⇓ m^n
↳ (2) 2x (BETA-V) if n=0 then 1 else … ⇓ m^n
↳ (3) (COND-FALSE) n=0 ⇓ false, da n>0
↳ (4) (OP) if n mod 2 = 0 then ⇓ m^n

n gerade:
↳ (5) (COND-TRUE) n mod 2 = 0 ⇓ true (APP)+2x(OP)
↳ (6) (rec …) (m∗m) (n/2) ⇓ m^n
↳ (7) (APP*) rec … ⇓ λx. λy. …
↳ (8) (UNFOLD) λx. λy. … ⇓ λx. λy. … (VAL)
↳ (9) m∗m ⇓ m^2 (genauer: $*^I(m, m)$)
↳ (10) n/2 ⇓ n/2 (genauer: $/^I(n, 2)$)
↳ (11) (λx. λy. …) (m^2) (n/2) ⇓ m^n

wegen n/2<n liefert die Induktionsannahme: (λx. λy. …) (m^2) (n/2) ⇓ $(m^2)^{n/2} = m^n$ ← wie im informellen Beweis

n ungerade: analog

zu (c): von der big step Semantik zum Interpreter

Wir wissen: für jeden Ausdruck e existiert ein r ∈ Val ∪ Exn mit e ⇓ r. Also kann man eine partielle Funktion definieren:
eval Exp <→ Val ∪ Exn

$$\text{eval}(e) = \begin{cases} r, \text{ falls } e \Downarrow r \\ \text{undefiniert, sonst} \end{cases}$$

Eval lässt sich rekursiv definieren wie folgt:
eval(c) = c, eval(id) = id, eval(op n) = on n, eval(λid.e) = λid. e, eval(op n_1, n_2) = $op^I(n_1, n_2)$, eval((λid. e)v) = e[v/id]

$$\text{eval}(e_1, e_2) = \begin{cases} \text{eval}(v_1, v_2), \text{ falls eval}(e_1)=v_1 \text{ und eval}(e_2)=v_2 \\ \text{exn, falls eval}(e_1)=\text{exn oder eval}(e_1) \in \text{Val und eval}(e_2)=\text{exn} \\ \text{undefiniert, sonst} \end{cases}$$

…

Bemerkung: Bei big step Herleitungen hat man **lokalen Nichtdeterminismus**, denn auf Ausdrücke der Form $v_1 v_2$ ist Regel (APP) anwendbar und eventuell (OP) oder (BETA-V). Bei der Implementierung vermeidet man das, indem man den Regeln (BETA-V) und (OP) den Vorrang vor (APP) gibt (→ Vermeidung einer Endlos-Rekursion durch (APP)).

Nachteile: dieser Interpreter ist (für realistische Programme) ineffizient, da die Substitution e[v/id] lineare Laufzeit in der Größe von e hat.