20.10.05

Theorie der Programmierung I

(Mitschrift von Benjamin Klein)

Weiter so vorgegeben:

- für jeden arithmetischen Operator op eine Funktion op^I: Int x Int → Int ∪ Exn
  z.B.: $/^I(n_1, n_2) = \begin{cases} \text{Division\_by\_zero, falls } n_2=0 \\ \text{ganzzahliger Quotient von } n_1 \text{ und } n_2, \text{ sonst} \end{cases}$
- für jeden Vergleichsoperator op eine Funktion op^I: Int x Int → Bool
- die Menge Val ⊆ Exp (Menge aller Ausdrücke) aller Werte (Values) v sei definiert durch: v ::= c | id | op n (z.B. + 1) | λid.e (Slogan: „λ-Abstraktion schützt vor Auswertung“)
  Bsp.: 0, true, + 1, λx. + x 1, λx. x x

Ein small step ist eine “Formel” e→e’, e→exn. Ein small step heißt gültig, wenn er sich mit den folgenden Regeln herleiten lässt: (siehe Extrablatt)

- (BETA-V) (λid. e)(v→e[v/id]) → der Ausdruck, der aus e entsteht, indem man v für id einsetzt
- Beobachtung: (LET-EXEC) und (BETA-V) ähneln sich:
  Deshalb: Man kann let-Ausdrücke als Abkürzungen erklären, nämlich:
  let id = e_1 in e_2 steht für (λid. e_2) e_1
  Dann ergeben sich die Regeln (LET-EXEC) und (LET-EVAL) aus den anderen:
  1) (λid.e) v → e[v/id] wegen (BETA-V)
     Abkürzung für |
     let id = v in e (LET-EXEC)
  2) Wenn sich e_1 zu e_2 umformen lässt. e_1→e_1’ dann gilt nach (APP-RIGHT):
     (λid.e_2) e_1 → (λid.e_2) e_1’
     Abkürzung für | | Abkürzung für
     let id = e_1 in e_2 … let id = e_1’ in e_2 (LET-EVAL)

Fazit: Mann kann let-Ausdrücke aus der Kernsyntax entfernen und als „syntaktischen Zucker“ betrachten.

Vorteil: Da Sätze oft durch Fallunterscheidung über die Form von Ausdrücken bewiesen werden, ist es günstig eine kleine Kernsyntax zu haben.

Bsp.: Sei c = let square = λx. * x x in square (square 5)

1. small step:
mit (LET-EXEC): (weil λx. * x x bereits ein Wert ist)
e → (square (square 5)) für square gilt: [λx. * x x / square] } (λx. * x x) ((λx. * x x) 5)

2. small step:
wegen (BETA-V): (λx. * x x) 5 → (* x x) [5/x] } * 5 5
mit (APP-RIGHT): (λx. * x x)((λx. * x x) 5) → (λx. * x x)(* 5 5)

3. small step:
mit (OP) und (APP-RIGHT): (λx. * x x)(* 5 5) → (λx. * x x) 25

4. small step:
wegen (BETA-V): (λx. * x x) 25 → * 25 25

5. small step:
wegen (OP): * 25 25 → 625

zusammen:

e → (λx. * x x) ((λx. * x x) 5) (LET-EXEC)
→ (λx. * x x)(* 5 5) (BETA-V), (APP-RIGHT)
→ (λx. * x x) 25 (OP), (APP-RIGHT)
→ * 25 25 (BETA-V)
→ 625 (OP)

„Merke-Regel" zum Weiterleiten von Exceptions:

Für jede small step-Regel der Form

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

führen wir eine zugehörige exception-Regel ein, nämlich wenn $e_1 \to exn$, dann auch $e_2 \to exn$, d.h. $\frac{e_1 \to exn}{e_2 \to exn}$

Bsp.: Aus (APP-RIGHT) $\frac{e \to e'}{v\ e \to v\ e'}$ entsteht (APP-RIGHT-EXN) $\frac{e \to exn}{v\ e \to exn}$

Schreibweise: Wenn kein small step e→… existiert, schreibt man c $\not\to$.

Lemma 1: Für alle Werte v gilt v$\not\to$.

Beweis: Klar, für Werte der Form c, id, λid.e, denn die treten nie links von „→" auf (in der Konklussion der Regel [siehe Regelblatt])
Es bleiben Werte der Form op n. Für sie kann höchstens (APP-LEFT) oder (APP-RIGHT) in Frage kommen. Dann müsste die Prämisse von der Form op→… oder n→… sein, was wir schon ausgeschlossen haben, denn op ist eine Konstante und n eine einzelne Zahl.

Satz 2: Die small step Semantik ist deterministisch, d.h. für jeden Ausdruck e existiert höchstens ein r∈Exp ∪ Exn mit e→r.

Beweis: Idee: In jeder Situation (für jeden Ausdruck e) ist höchstens eine Regl anwendbar, und diese Regel liefert ein eindeutiges Ergebnis.