13.12.05

Theorie der Programmierung I

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

neue small Step Regeln:

(HD) $hd(cons(v_1, v_2)) \rightarrow v_1$

(HD-EXN) $hd[] \rightarrow$ Empty_list ($\in$ Exn)

(TL) $tl(cons(v_1, v_2)) \rightarrow v_2$

(TL-EXN) $tl[] \rightarrow$ Empty_list ($\in$ Exn)

(IS_EMPTY_TRUE) is_empty[] $\rightarrow$ true

(IS_EMPTY_FALSE) is_empty$(cons(v_1, v_2)) \rightarrow$ false

(FST) $fst(v_1, v_2) \rightarrow v_1$

(SND) $snd(v_1, v_2) \rightarrow v_2$

(PAIR-LEFT) $\dfrac{e_1 \rightarrow e_1'}{(e_1, e_2) \rightarrow (e_1', e_2)}$

(PAIR-RIGHT) $\dfrac{e \rightarrow e'}{(v, e) \rightarrow (v, e')}$

Beispiel:
(a) let rec length l = if is_empty l then 0 else 1 + length (tl l) in length [1, 2, 3]
(b) let rec member x l = not(is_empty l) && (x=hd l || member x (lt l)) in member 5 [1, 2, 3]
(c) let rec append l_1 l_2 = if is_empty l_1 then l_2 else (hd l_1, append (tl l_1) l_2) in append [1, 2, 3] [4, 5]
(d) let rec map f l = if is_empty l then [] else cons (f (hd l), map f (tl l)) in map (λx. x $*$ x) [1, 2, 3]
(e) let rec exists p l = not(is_empty l) && (p (hd l) || exists (tl l)) in exists (λx. x mod 2 = 0) [1, 2, 3]
(f) let member x l = exists (λy. y = x) l; kürzer: let member x = exists (λy. y = x)

(a) allgemeiner Typ: α list $\rightarrow$ int
(b) allgemeiner Typ: int $\rightarrow$ int list $\rightarrow$ bool
(c) allgemeiner Typ: α list $\rightarrow$ α list $\rightarrow$ α list
(d) allgemeiner Typ: $(\alpha \rightarrow \alpha') \rightarrow \alpha$ list $\rightarrow \alpha'$ list
(e) allgemeiner Typ: $(\alpha \rightarrow$ bool$) \rightarrow \alpha$ list $\rightarrow$ bool
(f) allgemeiner Typ: int $\rightarrow$ int list $\rightarrow$ bool

Satz 17: Auch für $\mathcal{L}_3^{ti}$ gilt die **Typsicherheit**, d.h. die Berechnung für einen abgeschlossenen wohlgetypten Ausdruck bleibt nicht stecken.

Funktioniert die Typsicherheit noch?

Dazu: (1) Definition von tequs(Γ, e, α) erweitern!
(2) Unifikations-Algorithmus erweitern!

zu (1):

- tequs(Γ, (e_1, e_2), α) = tequs(Γ, e_2, α) $\cup$ tequs(Γ, e_2, α) $\cup$ $\{\alpha = \alpha_1 * \alpha_2\}$; α_1, α_2 neue Typvar.
- tequs(Γ, [], α) = { $\alpha = \alpha_1$ list }; α_1 neue Typvariable
- tequs(Γ, cons, α) = { $\alpha = \alpha_1 * \alpha_1$ list $\rightarrow \alpha_1$ list }; α_1 neue Typvariable

(Man beachte: Bei jedem Auftreten von [], cons, … wird eine neue Typvariable eingeführt $\rightarrow$ an verschiedenen Stellen darf [] verschiedene Typen haben.)

zu (2): Produkt- und Listentypen werden mit Funktionstypen behandelt:

- unify $(\{\tau_1 * \tau_2 = \tau_1' * \tau_2'\} \cup E) = $ unify $(\{\tau_1 = \tau_1', \tau_2 = \tau_2'\} \cup E)$
- unify $(\{\tau_1 \text{ list} = \tau_2 \text{ list}\} \cup E) = $ unify $(\{\tau_1 = \tau_2\} \cup E)$

Beachte: Der letzte Fall „nicht unifizierbar“ ändert sich implizit. Er wird z. B. benutzt bei unify $(\{\tau_1 \text{ list} = \tau_2 \rightarrow \tau_2\} \cup \ldots)$.

4. Polymorphie

Duden: polymorph = vielgestaltig
Eine polymorphe Funktion ist eine, die mehrere Typen annehmen kann, anders formuliert: Eine Funktion, die einen Typparameter hat.

Beispiel: Identität $\lambda x.\ x :: \tau \rightarrow \tau$
twice $\lambda f.\ \lambda x.\ f\ (f\ x) :: (\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$
Komposition $\lambda f.\ \lambda g.\ g\ (f\ x) :: (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_2 \rightarrow \tau_3) \rightarrow (\tau_1 \rightarrow \tau_3)$
cons, hd, tl, …
length τ list $\rightarrow$ int

Aber: Unser bisheriges Typsystem erlaubt es <u>nicht</u>, dass benutzerdefinierte Funktionen (d. h. mit let deklarierte) in <u>einem</u> Programm auf Argumente unterschiedlichen Typs angewandt werden.

Beispiel: let rec length l = … in length [1, 2, 3] + length [true, false] ist <u>nicht</u> wohlgetypt in $\mathcal{L}_3^{ti}$.

Begründung:
- wenn wir in der Typherleitung die (LET)-Regel anwenden, müssen wir uns beim Eintrag in die Typumgebung Γ auf einen Typ der Form τ list $\rightarrow$ int <u>festlegen</u>, d. h. wir müssen an dieser Stelle τ „erraten“.
- genauso mit dem Typinferenz-Algorithmus:
  in Γ trägt man length = τ list $\rightarrow$ int ein. ➔ das führt beim ersten Aufruf zur Gleichung α = int ➔ das führt beim zweiten Aufruf zur Gleichung α = bool ➔ nicht unifizierbar!

Problem: Die Typvariable α ist bisher nur ein Platzhalter für einen noch unbekannten („festen“) Typ τ. Wir brauchen: Typvariablen, die als „Typparameter“ dienen, d. h. Typvariablen, für die man in <u>einem</u> Programm unterschiedliche Typen einsetzen könnte.

Lösung: Einführung einer neuen Schreibweise, die unterscheidet zwischen:
- reinen Platzhaltern (wie bisher)
- Typparametern