25.10.05

Theorie der Programmierung I

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

Satz 2: Small Stepp $e \to r$ eindeutig

Beweistechnik: Fallunterscheidung nach der Form e und Induktion über die Größe von e (oder „Struktur“ von e, d.h. die Induktionsannahme bezieht sich immer auf Teilausdrücke von e)

1. Fall: e Konstante: Name oder λ-Abstraktion: Dann gibt es (nach Lemma 1) keinen small Stepp: $e \not\to$.

2. Fall: e Applikation e_1e_2: Es kommen nur (APP-LEFT), (APP-RIGHT), (BETA-V) und (OP) in Frage

Fall 2a: $e_1 \to e_1'$: Hier kommt <u>nur</u> (APP-LEFT) in Frage, denn in den anderen 3 Regeln ist die linke Seite e_1 der Applikation ein Wert, also $e_1 \not\to$ nach Lemma 1. (APP-LEFT) führt zu $e_1e_2 \to e_1'e_2$. Nach Induktionsannahme für e_1 ist e_1' eindeutig durch e_1 bestimmt, also ist $e_1'e_2$ eindeutig durch e_1e_2 bestimmt.

Fall 2b: $e_1 \to exn$: Es kommt nur (APP-LEFT-EXN) in Frage (gleiche Argumentation wie in 2a). Dies führt zu $e_1e_2 \to exn$. Da exn durch e_1 eindeutig bestimmt ist, ist sie auch durch e_1e_2 eindeutig bestimmt.

Fall 2c: $e_1 \not\to$ und $e_2 \to e_2'$: Hier kommt <u>höchstens</u> (APP-RIGHT) in Frage (nämlich dann, wenn $e_1 \in Val$), denn (BETA-V) und (OP) benötigen einen <u>Wert</u> e_2 auf der rechten Seite, d.h. $e2 \not\to$. Wenn (APP-RIGHT) möglich ist, dann ergibt sich $e_1e_2 \to e_1e_2'$. Da e_2' eindeutig durch e_2 bestimmt ist, ist e_1e_2' eindeutig durch e_1e_2 bestimmt.

Fall 2d: $e_1 \not\to$ und $e_2 \to exn$: analog

Fall 2e: $e_1 \not\to$ und $e_2 \not\to$: Es kommt höchstens (BETA-V) oder (OP) in Frage. Sie schließen sich gegenseitig aus, da (BETA-V) links eine λ-Abstraktion benötigt und (OP) einen Ausdruck der Form op n. In beiden Fällen ergibt sich die rechte Seite des small stepps (nämlich e[v/id] bzw. $op^i[n_1, n_2]$ eindeutig aus der ersten Seite.

Weitere Fälle: siehe Übung

Bezeichnung: $\xrightarrow{*}$ reflexiver transitiver Abschluss von $\to$ (d.h. $\xrightarrow{*}$ steht für eine endlich Folge von small stepps)
$\xrightarrow{+}$ transitiver Abschluss von $\to$ (d.h. $\xrightarrow{+}$ steht für eine nichtleere endliche Folge)

Begriffe: Eine **<u>Berechnungsfolge</u>** ist eine endliche oder unendliche Folge von small stepps $e_0 \to e_1 \to \ldots \to e_n \to \ldots$
Eine **<u>Berechnung</u>** (für den Ausdruck e) ist eine **<u>maximale</u>** Berechnungsfolge (die mit e beginnt). „Maximal“ bedeutet, sie ist entweder unendlich oder sie lässt sich nicht mehr verlängern.
Eine Berechnung **<u>divergiert</u>**, falls sie unendlich ist. Sie **<u>terminiert</u>** (mit dem Wert v bzw. der exception exn), wenn sie mit einem Wert v oder einer exception exn endet. Sie **<u>bleibt stecken</u>**, wenn sie mit einem Ausdruck $e \notin Val \cup Exn$ endet.

Bemerkung: Alle 4 Fälle können in $\mathcal{L}_1$ auftreten (siehe Übung 1). In $\mathcal{L}_0$ fehlt das Terminieren mit exception.

Ausdrücke, mit denen eine Berechnung stecken bleiben kann sind z.B.:

- \+ 1 true — denn (OP) verlangt zwei integers n_1, n_2 als Argumente
- = true falls — denn (OP) verlangt zwei integers n_1, n_2 als Argumente
- \+ true — denn nur op n mit integern zählt zu den Werten
- \+ + +, + + , if 1 than 2 else 3, …

In Lisp / Scheme kann man solche Ausdrücke eingeben, sie führen zu „Laufzeittypfehlern“.

Nachteile:

1. Es ist eine gewisse Typüberprüfung zur Laufzeit notwendig (Verschwendung von Laufzeit)
2. Fehlermeldungen können unverständlich sein, da sie „tief im Programm“ entstehen können → ursprünglicher Programmierfehler nicht erkennbar

In **statisch getypten Sprachen** (Pascal, C, Java, ML) versucht man solche Typfehler (d.h. das „Steckenbleiben“) von vornherein zu vermeiden, durch eine Typüberprüfung zur **Compilezeit**.
In den meisten dieser Sprachen (Pascal, C, Java) ist es dann nötig, dass der Programmierer die Typen der Parameter und des Resultats einer Funktion angibt.
Manche Sprachen (ML, Haskell, Miranda) erlauben es dem Programmierer, diese Typinformation ganz oder teilweise wegzulassen. Der Compiler findet diese Typen wieder mit Hilfe eines **Typinferenzalgorithmus**.

Wichtiges Thema der Vorlesung:

- Definition unterschiedlicher Typsysteme
- Jeweils Beweis der Typsicherheit, d.h. es wird gezeigt, dass wphlgetypte Ausdrücke niemals stecken bleiben.

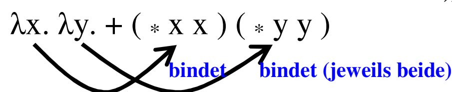
Zunächst: Weiter mit den wohlgetypten Sprachen $\mathcal{L}_0, \mathcal{L}_1$.

Noch zu definieren: Schreibweise e[v/id]
Allgemeiner e[e'/id]

Frei vorkommende Namen und Substitutionen

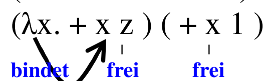
Intuition: Durch λ-Abstraktionen entstehen „Bindungen“, z.B.:

$\lambda x. \lambda y. + (* \; x \; x) (* \; y \; y)$



(Vorkommen von) Namen, die nicht „gebunden“ sind, nennt man „frei“.

$(\lambda x. + x \; z) (+ x \; 1)$



Ähnlich für „let“: Da wir „let“ als syntaktischen Zucker auffassen, können wir die Bedingungen in einem let-Ausdruck erkennen, indem wir ihn in die Kernsyntax übersetzen, z.B.:

let x = + x 1 in $*$ x 2

steht für $(\lambda x. * \; x \; 2) (+ x \; 1)$