Theorie der Programmierung I

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

Satz 11: Sei e :: τ. Dann gilt: Entweder e∈Val oder es existiert ein small step e→e' oder e→exn.

Satz 12: (Typsicherheit, „Type Safety")
Die Berechnung für einen abgeschlossenen wohlgetypten Ausdruck bleibt nicht stecken, d.h. sie kann
(a) divergieren
(b) oder mit einer exception terminieren
(c) oder mit einem Wert terminieren

Beweis: „Safety: Preservation & Progress"
Sei e:: τ. Wenn die Berechnung für e unendlich ist, oder mit einer exception endet, so gilt (a) bzw. (b). Es bleibt der Fall, dass sie mit einem Ausdruck endet, d.h. sie ist von der Form: $e = e_0 \rightarrow \ldots e_n \not\rightarrow$
Mit Preservation folgt aus e :: τ, dass e_i :: τ für alle i<n, also insbesondere e_n :: τ.
Daraus folgt mit „Progress" $e_n \in$ Val.

Beachte: Das Steckenbleiben wird allein durch die statische Typüberprüfung (zur Compile-Zeit) verhindert. Zur Laufzeit stehen die Typen zwar noch in den Ausdrücken, aber sie spielen keine Rolle mehr bei den small steps. Das lässt sich deutlicher machen, indem man diese Typen nach der Typüberprüfung aus den Ausdrücken entfernt:
Sei $\text{erase}_{\mathcal{L}_2^t} \rightarrow L_2$ definiert durch:
erase(c) = c
erase(id) = id
$\text{erase}(e_1 e_2) = (\text{erase}(e_1))(\text{erase}(e_2))$
$\text{erase}(\lambda id{:}\tau.e) = \lambda id.\ \text{erase}(e)$
erase(rec id erase(e)) = rec id e

Dann gilt folgendes: $e \rightarrow e' \Leftrightarrow \text{erase}(e) \rightarrow \text{erase}(e')$ und $e \rightarrow \text{exn} \Leftrightarrow \text{erase}(e) \rightarrow \text{exn}$
$|\ \text{in}\ \mathcal{L}_2^t$ $|\ \text{in}\ \mathcal{L}_2$

Analog für $\xrightarrow{*}$, d.h. die Berechnung für e läuft „genauso" ab wie die Berechnung für erase(e).

Also neue Formulierung von Satz 12:
Wenn $e \in \mathcal{L}_2^t$ abgeschlossen und wohlgetypt ist, dann bleibt die Berechnung für erase(e) nicht stecken.

3. Typinferenz für die einfach getypten Sprachen

Bisher: Um Steckenbleiben zu verhindern, muss der Programmierer Typinformationen in die Programme schreiben → Dann kann Wohlgetyptheit überprüft werden.

Frage: Kann man dem Programmierer diese Arbeit ersparen, und es dem Compiler überlassen, die „richtige" Typinformation zu finden?

→ Definition von Sprachen $\mathcal{L}_0^{ti}, \mathcal{L}_1^{ti}, \mathcal{L}_2^{ti}$ (ti: „Typinferenz")

$\mathcal{L}_i^{ti}$ werden so definiert:

- die kontextfreie Syntax ist die von $\mathcal{L}_i$
- ein Ausdruck $e \in \mathcal{L}_i^{ti}$ heißt wohlgetypt bezüglich Γ, wenn es einen Ausdruck $e \in \mathcal{L}_i^t$ gibt mit e ist wohlgetypt bezüglich Γ und e = erase(e).

Frage: Gibt es eine eigenständige Definition der Sprachen $\mathcal{L}_i^{ti}$, die nicht auf $\mathcal{L}_i^t$ Bezug nimmt? – Ja, man muss nur die Typregeln etwas verändern:

Die Typregeln (ABSTR) und (REC) werden ersetzt durch:

(ABSTR') $\dfrac{\Gamma[\tau/id] \triangleright e :: \tau'}{\Gamma \triangleright \lambda id.\ e :: \tau \rightarrow \tau'}$ **(REC')** $\dfrac{\Gamma[\tau/id] \triangleright e :: \tau}{\Gamma \triangleright \text{rec id e} :: \tau}$

Lemma 13: Ein Ausdruck $e \in \mathcal{L}_i^{ti}$ ist genau dann wohlgetypt bezüglich Γ, wenn sich mit diesen neuen Regeln ein typurteil $\Gamma \triangleright e' :: \tau$ herleiten lässt.

Genauer: $\Gamma \triangleright e' :: \tau$ (mit den neuen Regeln) $\Leftrightarrow$ es existiert ein $e \in \mathcal{L}_i^{t}$ mit $\Gamma \triangleright e :: \tau$ (mit den alten Regeln) und e' = erase(e)

Beweis: „⇐“: Aus einer Herleitung für $\Gamma \triangleright e :: \tau$ ergibt sich eine für $\Gamma \triangleright e' :: \tau$ (e'=erase(e)), indem man die Typen aus e entfernt und jede Anwendung von (ABSTR) und (REC) durch eine neue Anwendung von (ABSTR') und (REC') ersetzt.

„⇒“: Aus einer Herleitung für $\Gamma \triangleright e' :: \tau$ ergibt sich ein passender Ausdruck e und eine Herleitung für $\Gamma \triangleright e :: \tau$, indem man jede Anwendung von (ABSTR) und (REC) durch „entsprechende“ Anwendungen von (ABSTR') und (REC') ersetzt. „Entsprechend“ bedeutet: Der Typ τ aus den Regeln (ABSTR') und (REC') schreibt man hinter den Namen id.
→ Damit haben wir eine eigenständige Definition der Sprache $\mathcal{L}_i^{ti}$ (durch KFG und Typregeln).

Es fehlt noch: ein Algorithmus, der zu Γ und $e' \in \mathcal{L}_i^{ti}$ eine Herleitung liefert

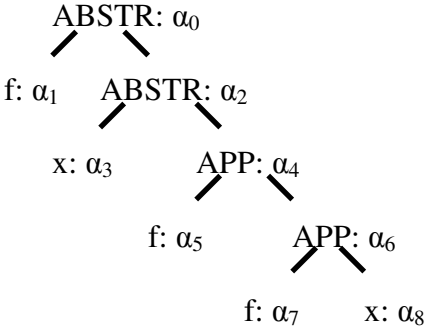
Problem:
- Eine einfache „Rückwärts-Anwendung“ der Typregeln funktioniert nicht mehr, da man für die neuen Regeln (ABSTR') und (REC') den Typ τ für die Prämisse erraten musste (denn aus der Konklusion ist ja nur Γ und e bekannt).
- Der Typ eines Ausdrucks e bezüglich Γ ist nicht mehr eindutig, z.B. $\lambda x.\ x :: \tau \to \tau$ für beliebiges τ.

Gesucht: Algorithmus, der Wohlgetyptheit überprüft und jeweils alle möglichen Typen eines Ausdrucks e bezüglich Γ bestimmt.

Idee: Man führt „Platzhalter“ für Typen ein, sogenannte **Typvariablen** und legt sich dann immer nur soweit fest, wie es nötig ist, z.B: wenn $\lambda x.\ x$ als Teilausdruck auftritt, erhält es zunächst den Typ $\alpha \to \alpha$. Später wird α eventuell genauer bestimmt.

Formal: Vorgegeben sei eine unendliche Menge TVar von **Typvariablen** α. Die Menge Type aller Typen wird erweitert durch die Produktion $\tau := \alpha$
Eine **Typgleichung** ist ein Paar $(\tau_1, \tau_2) \in \text{Type} \times \text{Type}$, das man in der Form $\tau_1 = \tau_2$ schreibt.
Der Typinferenzalgorithmus lässt sich in zwei Phasen aufteilen:
(1) Erzeugung einer Menge von Typgleichungen
(2) Lösen dieser Typgleichungen → man erhält Typen für alle Teilausdrücke mit Parameter, insbesondere einen Typ für den Gesamtausdruck.

Beispiel: Sei twice = $\lambda f.\ \lambda x.\ f(fx)$



Typgleichungen:
1) $\alpha_0 = \alpha_1 \to \alpha_2$ → ABSTR
2) $\alpha_2 = \alpha_3 \to \alpha_4$ → ABSTR
3) $\alpha_5 = \alpha_6 \to \alpha_4$ → APP
4) $\alpha_7 = \alpha_8 \to \alpha_6$ → APP
5) $\alpha_1 = \alpha_5$ → Bindungen
6) $\alpha_1 = \alpha_7$ → Bindungen
7) $\alpha_3 = \alpha_8$ → Bindungen

Lösung des Gleichungssystems:
4) $\alpha_1 = \alpha_3 \to \alpha_6$, 3) $\alpha_1 = \alpha_6 \to \alpha_4$

Folgerung:
$\alpha_3 = \alpha_4 = \alpha_6$, also $\alpha_1 = \alpha_3 \to \alpha_4$
2) $\alpha_2 = \alpha_3 \to \alpha_3$, 1) $(\alpha_3 \to \alpha_3) \to (\alpha_3 \to \alpha_3)$