Funktionales Programmieren

Teil 13

Carl Philipp Reh

Universität Siegen

24. November 2023

Unifikation

Wir wollen im Folgenden Typgleichungssysteme lösen. Zur Erinnerung: Wir lösen diese, indem wir Typvariablen ersetzen. Eine *Substitution* ist eine partielle Funktion TVar $\to$ Type. Analog zu Umgebungen schreiben wir $[\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n]$ für die Substitution s mit $s(\alpha_i) = \tau_i$ für $1 \leq i \leq n$.
Das Anwenden einer Substitution s auf einen Typ $\tau \in$ Type schreiben wir als τs, wobei

$$\tau s = \begin{cases} \tau_1 s \to \tau_2 s & \text{falls } \tau = \tau_1 \to \tau_2, \\ s(\alpha) & \text{falls } \tau = \alpha \text{ und } s(\alpha) \text{ definiert,} \\ \tau & \text{sonst.} \end{cases}$$

Das heißt, für jede Typvariable α, für die s einen Eintrag hat, ersetzen wir jedes Vorkommen von α durch $s(\alpha)$.
Beispiel: $\alpha \to \alpha_1[\alpha/\mathrm{Int}] = \mathrm{Int} \to \alpha_1$.

Unifikation

Analog können wir eine Substitution s auf Typgleichungen und auch auf Typgleichungssysteme anwenden, was wir als $(\tau = \tau')s$ bzw. Es für ein Typgleichungssystem E schreiben.

Die *Erweiterung* einer Substitution s_1 durch eine Substitution s_2 ist die Substitution $s_1 s_2$, wobei $(s_1 s_2)(\alpha) = (s_1(\alpha))s_2$. Das heißt, die Einträge von s_2 „wirken hinterher“ auf die von s_1. Beispiel: Sei $s_1 = [\alpha/\alpha_1]$ und $s_2 = [\alpha_1/\mathrm{Int}]$. Dann ist $s_1 s_2 = [\alpha/\mathrm{Int}, \alpha_1/\mathrm{Int}]$ und $s_2 s_1 = [\alpha_1/\mathrm{Int}, \alpha/\alpha_1]$.

Eine Substitution s heißt *Lösung* einer Typgleichung $\tau = \tau'$ genau dann, wenn τs und $\tau s'$ gleich sind. Entsprechend heißt s Lösung eines Typgleichungssystems, wenn es Lösung aller Typgleichungen in ihm ist.

Unifikation

Beim Lösen eines Typgleichungssystem sind wir an der „besten“ Lösung interessiert. Eine Substitution s_1 ist *allgemeiner als* eine Substitution s_2, wenn es eine Substitution s_3 gibt mit $s_2 = s_1 s_3$. Das heißt, man erhält aus dem „allgemeineren“ s_1 ein „spezielleres“ s_2, indem man weitere Ersetzungen, nämlich die aus s_3, durchführt. Beispiel: $s_1 = [\alpha/\alpha_1]$ ist allgemeiner als $s_2 = [\alpha/\mathrm{Int}, \alpha_1/\mathrm{Int}]$, weil $s_2 = s_1[\alpha_1/\mathrm{Int}]$. Eine Lösung heißt *allgemeinste Lösung*, wenn sie allgemeiner als alle anderen Lösungen ist.

Beispiel: Sei $E = \{\alpha \to \mathrm{Int} = \alpha_1 \to \alpha_2\}$. Eine allgemeinste Lösung für E wäre $s_1 = [\alpha/\alpha_1, \alpha_2/\mathrm{Int}]$, also $Es_1 = \{\alpha_1 \to \mathrm{Int} = \alpha_1 \to \mathrm{Int}\}$. Eine andere Lösung wäre $[\alpha/\mathrm{Int}, \alpha_1/\mathrm{Int}, \alpha_2/\mathrm{Int}]$, also $Es_2 = \{\mathrm{Int} \to \mathrm{Int} = \mathrm{Int} \to \mathrm{Int}\}$.

Die *freien Variablen* eines Typs sind definiert als $\mathrm{free}(\alpha) = \{\alpha\}$ für $\alpha \in \mathrm{TVar}$, $\mathrm{free}(\mathrm{Int}) = \emptyset$ und $\mathrm{free}(\tau \to \tau') = \mathrm{free}(\tau) \cup \mathrm{free}(\tau')$.

Unifikationsalgorithmus

Wir geben nun einen Algorithmus an, der ein Typgleichungssystem E bekommt und

- „nicht lösbar“ liefert, wenn es keine Lösung für E gibt,
- und andernfalls eine allgemeinste Lösung für E liefert.

Wir starten mit einer Substitution s, die am Anfang leer ist und verwalten ein Typgleichungssystem E', das zu Anfang auf E gesetzt wird. In jedem Schritt wählen wir aus E' eine Gleichung $\tau = \tau'$ aus, die wir auch aus E' entfernen. Wir sind fertig, wenn E' leer ist. Dann ist s die allgemeinste Lösung.

Im Fall, dass die aktuelle Typgleichung schon gelöst ist, also die Form $\tau = \tau$ hat, ist nichts weiter zu tun.

Unifikationsalgorithmus

Im Fall, dass die aktuelle Typgleichung $\tau = \alpha$ ist, wobei $\alpha \in \text{TVar}$, schauen wir, ob α in τ vorkommt, also $\alpha \in \text{free}(\tau)$. Wenn ja, ist das Ergebnis „nicht lösbar“. Beispiel: $\alpha = \alpha \to \text{Int}$ hat keine Lösung. Ansonsten setzen wir s auf $s[\alpha/\tau]$. Außerdem müssen wir $[\alpha/\tau]$ auf E' anwenden. Der Fall, dass die Typgleichung die Form $\alpha = \tau$ hat, ist analog.

Eine Typgleichung der Form $\tau_1 \to \tau_1' = \tau_2 \to \tau_2'$ lösen wir, indem wir die Typgleichungen $\tau_1 = \tau_2$ und $\tau_1' = \tau_2'$ zu E' hinzufügen. Wir ersetzen also eine Typgleichung durch zwei. Beispiel: $\text{Int} \to \alpha = \alpha_1 \to \text{Bool}$ ersetzen wir durch $\text{Int} = \alpha_1$ und $\alpha = \text{Bool}$.

Falls wir auf eine Gleichung $\tau = \tau'$ stoßen, die nicht auf die vorherigen Fälle passt, ist das Ergebnis „nicht lösbar“. Beispiel: $\text{Int} = \alpha \to \text{Bool}$.

Typinferenz

Ein Typ τ heißt *allgemeiner* als ein Typ τ', wenn es eine Substitution s gibt mit $\tau s = \tau'$. Beispiel: $\alpha \to \mathrm{Int}[\alpha/\mathrm{Int}] = \mathrm{Int} \to \mathrm{Int}$. Also ist $\alpha \to \mathrm{Int}$ allgemeiner als $\mathrm{Int} \to \mathrm{Int}$. Zu einem Ausdruck e heißt τ *allgemeinster Typ* von e, wenn $\Gamma \vdash e : \tau$ und für alle anderen Typen τ' mit $\Gamma \vdash e : \tau'$ gilt, dass τ allgemeiner ist als τ'.

Insgesamt liefert uns Typinferenz den allgemeinsten Typ zu einem Ausdruck e, wenn dieser wohlgetypt ist. Wir starten mit $\Gamma \vdash e : \alpha$, wobei $\alpha \in \mathrm{TVar}$ und erhalten daraus ein Gleichungssystem E. Darauf wenden wir den Unifikationsalgorithmus an. Dieser liefert „nicht lösbar“, wenn E keine Lösung hat. In dem Fall ist e nicht wohlgetypt. Wenn E lösbar ist, dann liefert der Unifikationsalgorithmus eine allgemeinste Lösung s von E. In dem Fall ist $s(\alpha)$ der allgemeinste Typ von e.

Beispiel für Unifikation

In der letzten Vorlesung haben wir angefangen, Typinferenz für $\Gamma \vdash ((+)\ 1)\ 2 : \alpha$ durchzuführen. Dabei haben wir folgendes Typgleichungssystem erhalten:

$$E := \{\alpha_2 \to (\alpha_1 \to \alpha) = \mathrm{Int} \to (\mathrm{Int} \to \mathrm{Int}),\ \alpha_2 = \mathrm{Int},\ \alpha_1 = \mathrm{Int}\}.$$

Wir zeigen nun, wie der Unifikationsalgorithmus dafür durchgeführt werden kann. Wir starten mit $E' := E$ und $s = []$.

Wir wählen zunächst $\alpha_2 \to (\alpha_1 \to \alpha) = \mathrm{Int} \to (\mathrm{Int} \to \mathrm{Int})$ aus. Dies müssen wir durch $\alpha_2 = \mathrm{Int}$ und $\alpha_1 \to \alpha = \mathrm{Int} \to \mathrm{Int}$ ersetzen und erhalten

$$E' := \{\alpha_1 \to \alpha = \mathrm{Int} \to \mathrm{Int},\ \alpha_2 = \mathrm{Int},\ \alpha_1 = \mathrm{Int}\}.$$

Dann wählen wir $\alpha_1 \to \alpha = \mathrm{Int} \to \mathrm{Int}$ und erhalten

$$E' := \{\alpha = \mathrm{Int},\ \alpha_2 = \mathrm{Int},\ \alpha_1 = \mathrm{Int}\}.$$

Beispiel für Unifikation

Wir wählen $\alpha = \mathrm{Int}$ und erhalten $s := s[\alpha/\mathrm{Int}] = [\alpha/\mathrm{Int}]$ mit

$$E' := \{\alpha_2 = \mathrm{Int},\ \alpha_1 = \mathrm{Int}\}.$$

Dann wählen wir $\alpha_2 = \mathrm{Int}$ und erhalten
$s := s[\alpha_2/\mathrm{Int}] = [\alpha/\mathrm{Int}, \alpha_2/\mathrm{Int}]$ mit

$$E' := \{\alpha_1 = \mathrm{Int}\}.$$

Zuletzt wählen wir $\alpha_1 = \mathrm{Int}$ und erhalten
$s := s[\alpha_1/\mathrm{Int}] = [\alpha/\mathrm{Int}, \alpha_2/\mathrm{Int}, \alpha_1/\mathrm{Int}]$ mit $E' := \emptyset$.

Damit ist der Algorithmus fertig und wir haben die allgemeinste Lösung gefunden. Wegen $s(\alpha) = \mathrm{Int}$ ist Int der allgemeinste Typ von $((+)\ 1)\ 2$.

Polymorphie

Eine Funktion wie `f x = x` nennt man polymorph, weil man sie mit allen Typen benutzen kann. In Haskell schreiben wir `f :: a -> a`, was *nicht* gleich $\Gamma \vdash f : \alpha \to \alpha$ ist!
Zum Beispiel bekommen wir bei der Typinferenz für

```
let f = \x -> x
in case f True of { True -> f 0 }
```

heraus, dass $\alpha = \mathrm{Int}$ und $\alpha = \mathrm{Bool}$, also $\mathrm{Int} = \mathrm{Bool}$ (Übung).

In Haskell ist `f :: a -> a` hingegen eine Abkürzung für `f :: forall a. a -> a`. Um das in unserem Typsystem abzubilden, definieren wir die neuen Typen

$$\mathsf{PType} = \{\forall\alpha_1 \cdots \forall\alpha_n.\tau \mid \tau \in \mathsf{Type}\,,\ \alpha_1, \ldots, \alpha_n \in \mathsf{TVar}, n \geq 0\}.$$

Man beachte, dass $\mathsf{Type} \subseteq \mathsf{PType}$.

Polymorphie

Wir werden bei **let** $f = e_1$ **in** e_2 erlauben, dass f einen polymorphen Typ in e_2 bekommt. In unserem vorherigen Beispiel würde also für f der Typ $\forall\alpha.\alpha \to \alpha$ in Γ eingetragen, sodass f sowohl auf $\underline{\mathrm{True}}$, als auch auf 0 angewandt werden kann.
In e_1 hingegen, also der Definition von f, darf f keinen polymorphen Typ bekommen.
Wichtig ist außerdem, dass nur die Typvariablen ein $\forall$ in e_2 bekommen dürfen, die nicht schon in Γ vorkommen. Man könnte zum Beispiel die Identität folgendermaßen definieren:

$$\textbf{let}\ \mathrm{id} = \backslash x \text{->} (\textbf{let}\ g = \backslash y \text{->} x\ \textbf{in}\ g\ 0)\ \textbf{in}\ e$$

Könnte man einfach $\forall$ vor alle Typvariablen schreiben, könnte man g den Typ $\forall\alpha_1\forall\alpha_2.\alpha_1 \to \alpha_2$ in $g\ 0$ geben, was wiederum dazu führt, dass auch id diesen Typ in e bekommt.

Polymorphie

Für Typkonstruktoren mit Kind

$$T : \underbrace{* \to \cdots \to *}_{n \text{ mal}} \to *$$

müssen wir unsere induktive Definition von Typen erweitern: Wenn $\tau_1, \ldots, \tau_n \in \text{Type}$, dann $T(\tau_1, \ldots, \tau_n) \in \text{Type}$. Zum Beispiel sind $\mathrm{List}(\alpha)$ und $\mathrm{List}(\mathrm{Int})$ Typen. Entsprechend müssen Substitutionen s auch in solchen Typen die Typvariablen ersetzen, also

$$T(\tau_1, \ldots, \tau_n)s = T(\tau_1 s, \ldots, \tau_n s).$$

Zum Beispiel ist $\mathrm{List}(\alpha)[\alpha/\mathrm{Int}] = \mathrm{List}(\mathrm{Int})$.

Ebenso definieren wir free als

$$\mathsf{free}(T(\tau_1, \ldots, \tau_n)) = \mathsf{free}(\tau_1) \cup \cdots \cup \mathsf{free}(\tau_n).$$

Polymorphie

Typumgebungen können nun polymorphe Typen enthalten, also sind diese jetzt partielle Funktionen TVar $\to$ PType. Typurteile sind immer noch von der Form $\Gamma \vdash e : \tau$, wobei $\tau \in$ Type!

Die *freien Variablen* in polymorphen Typen seien definiert als $\mathsf{free}(\forall\alpha_1 \ldots \forall\alpha_n.\tau) = \mathsf{free}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}$. Zum Beispiel ist $\mathsf{free}(\forall\alpha.\alpha \to \alpha_1) = \{\alpha_1\}$.

Für eine Typumgebung Γ sei

$$\mathsf{free}(\Gamma) = \bigcup\{\mathsf{free}(\Gamma(x)) \mid x \in \mathsf{Var}\,,\ \Gamma(x) \text{ definiert}\}.$$

Zum Beispiel ist $\mathsf{free}([x/\forall\alpha.\alpha, y/\mathrm{Int} \to \alpha_1]) = \{\alpha_1\}$.

Der *polymorphe Abschluss* $\mathsf{cl}_\Gamma(\tau)$ für $\Gamma \in$ TEnv und $\tau \in$ Type ist $\forall\alpha_1 \cdots \forall\alpha_n.\tau$, wobei $\{\alpha_1, \ldots, \alpha_n\} = \mathsf{free}(\tau) \setminus \mathsf{free}(\Gamma)$.
Zum Beispiel gilt $\mathsf{cl}_{[]}(\alpha \to \alpha_1 \to \mathrm{Int}) = \forall\alpha\forall\alpha_1.\alpha \to \alpha_1 \to \mathrm{Int}$, aber $\mathsf{cl}_{[x/\alpha]}(\alpha \to \alpha_1 \to \mathrm{Int}) = \forall\alpha_1.\alpha \to \alpha_1 \to \mathrm{Int}$.

Polymorphie

Wir passen nun unsere Regeln an: Für $x \in \text{Var}$ erlauben wir, einen polymorphen Typ zu instanziieren, also
$\Gamma \vdash x : \tau[\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n]$, falls $\Gamma(x) = \forall\alpha_1 \cdots \forall\alpha_n.\tau$. Beispiel:
$[f/\forall\alpha.\alpha \to \text{Int}] \vdash f : \text{Bool} \to \text{Int}$.

Ebenso erlauben wir dies für Wertkonstruktoren, also
$\Gamma \vdash \underline{\text{C}} : \tau[\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n]$, falls $\underline{\text{C}} : \forall\alpha_1 \cdots \forall\alpha_n.\tau$
Beispiel: Es gilt $\underline{\text{Nil}} : \forall\alpha.\text{List}(\alpha)$, also auch $\Gamma \vdash \underline{\text{Nil}} : \text{List}(\text{Int})$.

Die Typregel für Let ändern wir wie besprochen:

$$\frac{\Gamma \oslash [x/\tau'] \vdash e_1 : \tau' \quad \Gamma \oslash [x/\,\text{cl}_\Gamma(\tau')] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

Auch für das polymorphe Typsystem lässt sich Typinferenz implementieren, was wir allerdings nicht mehr behandeln können.

Beispiel für Polymorphie

$\Gamma \vdash \textbf{let } f = \backslash x \text{ -> } x \textbf{ in case } f \ \underline{\text{True}} \textbf{ of } \{\underline{\text{True}} \text{ -> } f \ 0\} : \text{Int}$

- $\Gamma \circledcirc [f/\alpha \to \alpha] \vdash \backslash x \text{ -> } x : \alpha \to \alpha$
  - $\Gamma \circledcirc [f/\alpha \to \alpha, x/\alpha] \vdash x : \alpha$
- $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash \textbf{case } f \ \underline{\text{True}} \textbf{ of } \{\underline{\text{True}} \text{ -> } f \ 0\} : \text{Int}$
  - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash f \ \underline{\text{True}} : \text{Bool}$
    - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash f : \text{Bool} \to \text{Bool}$
    - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash \underline{\text{True}} : \text{Bool}$
  - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash \underline{\text{True}} : \text{Bool}$
  - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash f \ 0 : \text{Int}$
    - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash f : \text{Int} \to \text{Int}$
    - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash 0 : \text{Int}$