2.5.2 Typinferenz

Die Aufgabe der Typinferenz kann wie folgt formuliert werden:

> Suche (bzgl. vordefinierter Funktionen) die allgemeinsten Typen neu definierter Funktionen.

Nachfolgend zeigen wir die Grundidee der Typinferenz, wie sie z.B. in [Damas/Milner 82] zu finden ist.

Die Aufgabe der Typinferenz ist ja das Raten einer allgemeinsten Typannahme, sodass die definierten Funktionen unter der Typannahme typkorrekt sind. Statt nun den Typ zu raten, gehen wir wie folgt vor:

- Setze für einen unbekannten Typ zunächst eine neue Typvariable ein
- Formuliere Gleichungen (Bedingungen) zwischen Typen
- Berechne allgemeinste Lösung für das (Typ-)Gleichungssystem

Zunächst zeigen wir die Typinferenz für *eine* neue Funktion und erweitern dies später auf die Typinferenz für ganze Programme.

Vorgehen bei der Typinferenz:

1. **Variablenumbenennung:** Benenne Variablen in verschiedenen Gleichungen so um, dass verschiedene Gleichungen keine identischen Varablen enthalten:

   ```
   app []     x = x
   app (x:xs) y = x : app xs y
   ```

   wird umbenannt in

   ```
   app []     z = z
   app (x:xs) y = x : app xs y
   ```

2. **Erzeuge Ausdruck/Typ-Paare:** Für jede Regel

   l `|` c `=` r

   (wobei die Bedingung c auch fehlen kann) erzeuge die Ausdruck/Typ-Paare

   $l :: a$
   $r :: a$
   $c ::$ `Bool`

   wobei a eine *neue* Typvariable ist

3. **Vereinfache Ausdruck/Typ-Paare:** Ersetze

- $(e_1\ e_2) :: \tau$ durch $e_1 :: a \to \tau,\ e_2 :: a$ (hierbei ist a eine neue Typvariable)
- `if` e_1 `then` e_2 `else` $e_3 :: \tau$ durch $e_1 :: \texttt{Bool}, e_2 :: \tau, e_3 :: \tau$
- $e_1 \circ e_2 :: \tau$ durch $e_1 :: a, e_2 :: b, \circ :: a \to b \to \tau$ (a, b neue Typvariablen) (hierbei ist $\circ$ ein vordefinierter Infixoperator)
- $\backslash x \to e :: \tau$ durch $x :: a, e :: b$ und eine neue **Typgleichung** $\tau \doteq a \to b$, wobei a, b neue Typvariablen sind
- $f :: \tau$ durch die Typgleichung $\tau \doteq \sigma(\tau')$, falls f mit dem Typschema $\forall a_1 \ldots a_n : \tau'$ vordefiniert ist, $b_1, \ldots b_n$ neue Typvariablen sind und

  $\sigma = \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$

  eine Typsubstitution ist.

Die hierbei erzeugten Typgleichungen werden zusammen mit den im nächsten Schritt erzeugten Typgleichungen aufgesammelt.

4. **Erzeuge (Typ-)Gleichungen:**
   Nach dem vorigen Schritt können wir nun voraussetzen, dass alle Ausdruck/Typ-Paare von der Form $x :: \tau$ (mit x Bezeichner) sind. Es kann aber für einen Bezeichner x mehrere dieser Ausdruck/Typ-Paare geben. Da allerdings ein Bezeichner einen eindeutigen Typ haben muss, werden alle Typen für einen Bezeichner gleich gesetzt, d.h. es wird die folgende letzte Transformation durchgeführt:

   Erzeuge die Gleichung $\tau_1 \doteq \tau_2$ für alle Ausdruck/Typ-Paare $x :: \tau_1, x :: \tau_2$

   Hierbei müssen auf Grund der Transitivität der Gleichheit nicht unbedingt alle diese Paare erzeugt werden. Z.B. reicht es für die Ausdruck/Typ-Paare $x :: \tau_1, x :: \tau_2, x :: \tau_3$ aus, die Typgleichungen $\tau_1 \doteq \tau_2$ und $\tau_1 \doteq \tau_3$ zu erzeugen, da die Gleichung $\tau_2 \doteq \tau_3$ aus der Transitivität und Kommutativität von $\doteq$ folgt. Dies werden wir in Beispielen ausnutzen.

   Beispiel: betrachte die Definition von `twice`:

   ```
   twice f x = f (f x)
   ```

   Die Anwendung von Schritt 2 ergibt:

   `twice f x` $:: a$
   `f (f x)` $:: a$

   Anwendung von Schritt 3:

   `twice f x` $:: a \quad \Rightarrow \quad$ `twice f` $:: b \to a$, `x` $:: b$
   `twice f` $:: b \to a \quad \Rightarrow \quad$ `twice` $:: c \to b \to a$, `f` $:: c$

   `f (f x)` $:: a \quad \Rightarrow \quad$ `f` $:: d \to a$, `(f x)` $:: d$
   `(f x)` $:: d \quad \Rightarrow \quad$ `f` $:: e \to d$, `x` $:: e$

Insgesamt haben wir am Ende also die Ausdruck/Typ-Paare:

$\texttt{x} :: b$
$\texttt{twice} :: c \to b \to a$
$\texttt{f} :: c$
$\texttt{f} :: d \to a$
$\texttt{f} :: e \to d$
$\texttt{x} :: e$

Daraus erzeugen wir die folgenden Typgleichungen:

$c \doteq d \to a$
$c \doteq e \to d$
$b \doteq e$

5. Löse nun das sich ergebende Gleichungssystem, d.h. finde einen „allgemeinsten Unifikator“ σ hierfür (s.u.). Dann ist, für alle Ausdruck/Typ-Paare $x :: \tau$, $\sigma(\tau)$ der allgemeinste Typ für x. Falls kein allgemeinster Unifikator existiert, dann sind die Regeln nicht typkorrekt.

   Somit gilt:

   Für typkorrekte Regeln existiert immer ein allgemeinster Typ!

   Diese Aussage ist nicht trivial, z.B. ist sie falsch, wenn man ad-hoc-Polymorphismus mit Overloading betrachtet.

Zur Lösung des letzten Punktes, d.h. die Berechnung eines allgemeinsten Unifikators, definieren wir zunächst folgende Begriffe:

- Eine Substitution σ heißt **Unifikator** für ein Gleichungssystem E, falls für alle Gleichungen $l \doteq r \in E$ gilt: $\sigma(l) = \sigma(r)$
- Ein Unifikator σ für ein Gleichungssystem E heißt **allgemeinster Unifikator** (**mgu**, **most general unifier**) für E, falls für alle Unifikatoren σ' für E eine Substitution φ existiert mit $\sigma' = \varphi \circ \sigma$ (wobei $\varphi \circ \sigma(\tau) = \varphi(\sigma(\tau))$), d.h. alle anderen Unifikatoren sind Spezialfälle von σ.

Satz 2.2 ([Robinson 65]) *Falls ein Unifikator existiert, dann existiert auch ein mgu, der effektiv berechenbar ist.*

Hier: mgu-Berechnung nach [Martelli/Montanari 82] durch Transformation von E:

Wende folgende Transformationsregeln auf E an, d.h. die Regel $\frac{\text{A}}{\text{B}}$ steht für das Transformationsschema, dass A durch B ersetzt:

Decomposition:	$\dfrac{\{k\ s_1 \ldots s_n \doteq k\ t_1 \ldots t_n\} \cup E}{\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\} \cup E}$	k Typkonstruktor
Clash:	$\dfrac{\{k\ s_1 \ldots s_n \doteq k'\ t_1 \ldots t_m\} \cup E}{\texttt{fail}}$	k, k' Typkonstruktoren, $k \neq k'$ oder $m \neq n$
Elimination:	$\dfrac{\{x \doteq x\} \cup E}{E}$	x Typvariable
Swap:	$\dfrac{\{k\ t_1 \ldots t_n \doteq x\} \cup E}{\{x \doteq k\ t_1 \ldots t_n\} \cup E}$	x Typvariable
Replace:	$\dfrac{\{x \doteq \tau\} \cup E}{\{x \doteq \tau\} \cup \sigma(E)}$	x Typvariable, kommt in E aber nicht in τ vor, $\sigma = \{x \mapsto \tau\}$
Occur check:	$\dfrac{\{x \doteq \tau\} \cup E}{\texttt{fail}}$	x Typvariable, $x \neq \tau$, x kommt in τ vor

Satz 2.3 *Falls E mit den obigen Transformationen in $\{x_1 \doteq \tau_1, \ldots, x_n \doteq \tau_n\}$ überführbar und dann keine weitere Regel anwendbar ist, dann ist $\sigma = \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}$ ein mgu für E. Falls E in `fail` überführbar ist, dann existiert kein Unifikator für E.*

Beispiel: Wir betrachten die Berechnung eines mgus für unser obiges Gleichungssystem:

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{c \doteq d \to a},\ c \doteq e \to d,\ b \doteq e}{c \doteq d \to a,\ \boxed{d \to a \doteq e \to d},\ b \doteq e}\ \text{Replace}}{c \doteq d \to a,\ \boxed{d \doteq e},\ a \doteq d,\ b \doteq e}\ \text{Decomposition}}{c \doteq e \to a,\ d \doteq e,\ \boxed{a \doteq e},\ b \doteq e}\ \text{Replace}}{c \doteq e \to e,\ d \doteq e,\ a \doteq e,\ b \doteq e}\ \text{Replace}$$

Daher ist

$$\sigma = \{c \mapsto e \to e,\ d \mapsto e,\ a \mapsto e,\ b \mapsto e\}$$

ein allgemeinster Unifikator für das Gleichungssystem und

$$\texttt{twice} :: (e \to e) \to e \to e$$

ein allgemeinster Typ. Somit ist `twice` typkorrekt und hat das Typschema

$$\forall e.(e \to e) \to e \to e$$

Typinferenz für beliebige rekursive Funktionen

Bisher haben wir gezeigt, wie man den Typ einer Funktion interferieren kann. Wie wir oben bei der Typprüfung schon gesehen haben, ist es bei mehreren Funktionen wichtig, die Typen nacheinander zu prüfen, um möglichst allgemeine Typen zu unterstützen. Ein Problem ist allerdings, dass manche Funktionen gegenseitig rekursiv sind und damit wechselweise voneinander abhängig sind. Daher basiert die Typinferenz für beliebige rekursive Funktionen auf folgender Idee:

1. Sortiere Funktionen nach ihren Abhängigkeiten
2. Inferiere zunächst den Typ der Basisfunktionen. Falls diese typkorrekt sind, wandle deren Typen zu Typschemata um.
3. Inferiere die darauf aufbauenden Funktionen. Falls diese typkorrekt sind, wandle deren Typen zu Typschemata um.
4. ...

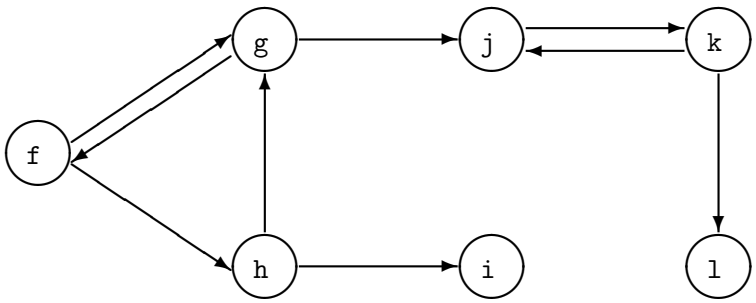
Zur Präzisierung dieser Idee benutzen wir einen **statischen Aufrufgraph**:

- Die Knoten sind mit Funktionen markiert.
- Eine Kante $f \to g$ existiert genau dann, wenn g in der Definition von f verwendet wird.

Beispiel:

```
f n  =  g n + h n
g n  =  j n + f n
h n  =  i n + g n
i n  =  n
j n  =  k n
k n  =  j n + l n
l n  =  5
```

Statischer Aufrufgraph für dieses Beispiel:



Die Relation „gegenseitig abhängig“ wird dann wie folgt definiert:

$$f \leftrightarrow g \;:\Leftrightarrow\; \text{es existiert ein Pfad von } f \text{ nach } g \text{ und ein Pfad von } g \text{ nach } f$$

Feststellung: $\leftrightarrow$ ist eine Äquivalenzrelation

Daher können wir die Äquivalenzklassen bzgl. $\leftrightarrow$ bilden (diese werden auch **starke Zusammenhangskomponenten** genannt). Im obigen Beispiel sind dies die Knotenmengen $\{\texttt{f},\texttt{g},\texttt{h}\}$, $\{\texttt{j},\texttt{k}\}$, $\{\texttt{i}\}$ und $\{\texttt{l}\}$.

Sortiere nun die Äquivalenzklassen in eine Liste

$$[A_1, A_2, \ldots, A_n]$$

wobei gilt: wenn ein Pfad von einem A_i-Knoten zu einem A_j-Knoten ($i \neq j$) existiert (d.h. A_i ist mit Hilfe von A_j definiert), dann ist $i \geq j$.

Sortierung für das obigen Beispiel:

$$[\{\texttt{i}\}, \{\texttt{l}\}, \{\texttt{j},\texttt{k}\}, \{\texttt{f},\texttt{g},\texttt{h}\}]$$

Nun tue das Folgende für alle Listenelemente A_i ($i = 1, \ldots, n$):

1. Berechne den allgemeinsten Typ für alle Funktionen in A_i.
2. Abstrahiere diese Typen durch Allquantifizierung aller vorkommenden Typvariablen.

Beispiel:

```
i     :: ∀a : a → a
l     :: ∀a : a → Int
j,k   :: ∀a : a → Int
f,g,h :: Int → Int
```

Ein Beispiel für eine nicht typisierbare Funktion ist

```
f x = x x
```

(der Ausdruck “`x x`” wird auch als **Selbstanwendung** bezeichnet). Für dieses Beispiel ergibt das Typinferenzverfahren nach der Vereinfachung der Ausdruck/Typ-Paare:

$$\texttt{f::}b \to a\texttt{, x::}b\texttt{, x::}c \to a\texttt{, x::}c$$

Nun stellen wir das Gleichungssystem auf und berechnen den mgu:

$$\dfrac{\dfrac{b \doteq c \to a,\ \boxed{b \doteq c}}{\boxed{c \doteq c \to a},\ b \doteq c}\ \text{Replace}}{\texttt{fail}}\ \text{Occur check}$$

Somit ist dieses Gleichungsssytem nicht lösbar und damit ist die Funktion `f` nicht typisierbar.

Nicht typisierbar sind aber auch sinnvolle Funktionsanwendungen. Betrachten wir dazu die Funktion

```
funsum f xs ys = f xs + f ys
```

und den Ausdruck

```
funsum length [1,2,3] "abc"
```

In einer ungetypten Sprache würde hierfür das Ergebnis `6` berechnet werden. Bei dem hier dargestellten Typsystem führt dieser Ausdruck allerdings zu einem **Typfehler**!

Die Ursache liegt im inferierten Typ von `funsum`:

$\forall a : (a \to \texttt{Int}) \to a \to a \to \texttt{Int}$

Bei Aufruf von `funsum` muss man eine generische Instanz festlegen, z.B. $a =$ `[Int]`, was aber zu einem Typfehler bei `"abc"` führt.

Ein geeigneter Typ wäre:

`funsum` $\forall b, c : (\forall a : a \to \texttt{Int}) \to b \to c \to \texttt{Int}$

d.h. der erste Parameter ist eine polymorph verwendbare Funktion.
Solche Typen sind prinzipiell denkbar ($\rightsquigarrow$ Polymorphismus 2. Ordnung), aber in dem hier vorgestellten Typsystem gilt:

Parameter sind nicht mehrfach typinstanziierbar

Praktisch ist dies aber kein Problem, da mehrfache Aufrufe wegtransformiert werden können:

```
funsum :: (a  → Int)  → (b  → Int)  → a  → b  → Int
funsum  f1 f2 l1 l2 = f1 l1 + f2 l2
```

Mit dieser Definition ist der Ausdruck

```
funsum length length [1,2,3] "abc"
```

typisierbar und ergibt `6`.
Tatsächlich unterstützt Haskell verschiedene Erweiterungen des hier vorgestellten Hindley/Milner-Typsystems. Insbesondere kann man mit der Spracherweiterung `Rank2Types` auch Rank-2-Typen, d.h. polymorphe Parameterfunktionen angeben. Das obige Beispiel kann in Haskell so umgesetzt werden:

```
{-# LANGUAGE Rank2Types #-}
funsum :: (forall a . [a]  → Int)  → [b]  → [c]  → Int
```

```
funsum f l1 l2 = f l1 + f l2

useFunSum = funsum length [1,2] "Hello"
```

Dieser Typ kann geprüft, aber nicht inferiert werden, d.h. dieses Programm wäre unzulässig, wenn man die Typangabe für `funsum` weglässt.