Als neuen Typ benötigen wir:

$$t \;::=\; \ldots \;\; \mathbf{ref}\; t \;\; \ldots$$

Neue Regeln:

$$\text{Ref:}\quad \frac{\Gamma \;\vdash\; e \;:\; t}{\Gamma \;\vdash\; (\mathbf{ref}\; e) \;:\; \mathbf{ref}\; t}$$

$$\text{Deref:}\quad \frac{\Gamma \;\vdash\; e \;:\; \mathbf{ref}\; t}{\Gamma \;\vdash\; (!\, e) \;:\; t}$$

$$\text{Assign:}\quad \frac{\Gamma \;\vdash\; e_1 \;:\; \mathbf{ref}\; t \qquad \Gamma \;\vdash\; e_2 \;:\; t}{\Gamma \;\vdash\; (e_1 := e_2) \;:\; ()}$$

Achtung:

Diese Regeln vertragen sich nicht mit Polymorphie !!!

Beispiel:

$$
\begin{array}{lcll}
\textbf{let}\ y & = & \textbf{ref}\ [\,]; & \\
\quad \_ & = & y := 1 : (!\, y); & \\
\quad \_ & = & y := \textbf{true} : (!\, y) & \\
\textbf{in}\ 1 & & &
\end{array}
$$

Achtung:

Diese Regeln vertragen sich nicht mit Polymorphie !!!

Beispiel:

$$
\begin{array}{lll}
\textbf{let } y & = & \textbf{ref } [\,]; \\
\quad \_ & = & y := 1 : (!\, y); \\
\quad \_ & = & y := \textbf{true} : (!\, y) \\
\textbf{in } 1 & &
\end{array}
$$

Für y erhalten wir den Typ: $\forall \alpha.\, \textbf{ref}\ (\mathsf{list}\ \alpha)$

$\Longrightarrow$ Die Typ-Inferenz liefert keinen Fehler

$\Longrightarrow$ Zur Laufzeit entsteht eine Liste mit **int** und **bool** :-(

Ausweg: Die Value-Restriction

- Generalisiere nur solche Typen, die Werte repräsentieren, d.h. keine Verweise auf Speicherstellen enthalten :-)
- Die Menge der Value-Typen lässt sich einfach beschreiben:

$$v \;::=\; \alpha \;\mid\; \mathbf{bool} \;\mid\; \mathbf{int} \;\mid\; \mathsf{list}\; v \;\mid\; (v_1, \ldots, v_m) \;\mid\; t \to t$$

Ausweg: Die Value-Restriction

- Generalisiere nur solche Typen, die Werte repräsentieren, d.h. keine Verweise auf Speicherstellen enthalten :-)
- Die Menge der Value-Typen lässt sich einfach beschreiben:

$$v \ ::= \ \alpha \ | \ \textbf{bool} \ | \ \textbf{int} \ | \ \text{list}\ v \ | \ (v_1, \ldots, v_m) \ | \ t \rightarrow t$$

... im Beispiel:

Der Typ: **ref** (list α) ist kein Value-Typ.

Darum darf er nicht generalisiert werden $\Longrightarrow$ Problem gelöst :-)

Matthias Felleisen, Northeastern University

Bemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Bemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Beispiel:

```
fun member x list = case list
                    of [ ]   → false
                     | h::t  → if x = h then true
                               else member x t
```

Bemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Beispiel:

$$
\begin{array}{lll}
\textbf{fun}\ \text{member}\ x\ \text{list} = & \textbf{case}\ \text{list} & \\
\quad \textbf{of} & [\,] & \rightarrow\ \textbf{false} \\
\quad \mid & h::t & \rightarrow\ \textbf{if}\ x = h\ \textbf{then}\ \textbf{true} \\
 & & \quad \textbf{else}\ \text{member}\ x\ t
\end{array}
$$

member hat den Typ: $\alpha' \rightarrow \text{list}\ \alpha' \rightarrow \textbf{bool}$ für jedes α' mit Gleichheit !!

Überladung

- Eine Funktion, eine Datenstruktur ist nicht generell polymorph, sondern verlangt Daten, die eine bestimmte Funktion unterstützen.
- Die Funktion sort ist z.B. nur auf Listen anwendbar, deren Elemente eine Operation $\leq$ zulassen.

Idee: Phil Wadler

- Erlaube Bedingungen an Typparameter.
- Eine Bedingung gibt an, welche Operationen dieser Typ implementieren muss.
- Eine Typklasse C versammelt alle Typen, die eine Operation unterstützen.
- Eine Instanzdeklaration für einen Typ τ und eine Klasse C stellt (möglicherweise unter Angabe weiterer Bedingungen) eine Implementierung des Operators der Klasse bereit.

Phil Wadler, Univerität Edinburgh

Beispiele für Typklassen

- Gleichheitstypen; Operation: $= \quad : \quad \alpha \to \alpha \to \mathbf{bool}$
- Vergleichstypen; Operation: $\leq \quad : \quad \alpha \to \alpha \to \mathbf{bool}$
- Druckbare Typen; Operation: $\mathsf{to\_string} \quad : \quad \alpha \to \mathbf{string}$
- Hashbare Typen; Operation: $\mathsf{hash} \quad : \quad \alpha \to \mathbf{int}$

Neue Typschemata:

$$\sigma \ ::= \ \tau \ \mid \ \forall \alpha \in C_1 \wedge \ldots \wedge C_k \,.\, \sigma$$

Klassendeklaration:

$$\textbf{class}\ C\ \textbf{where}\ \mathsf{op} : \forall\, \alpha \in C \,.\, \tau$$

Dabei enthält τ nur die Typvariable α.

Instanzdeklaration:

$$\textbf{inst}\ \alpha_1 \in C_1, \ldots \alpha_k \in C_k \Rightarrow \tau \in C$$
$$\textbf{where}\ \mathsf{op} = e$$

sofern op die Operation der Klasse C ist.

Beispiel

$$\textbf{class}\ \mathsf{Eq}\ \textbf{where}$$
$$(=) : \quad \forall\, \alpha \in C.\ \alpha \to \alpha \to \textbf{bool}$$

$$\textbf{inst}\ \beta \in \mathsf{Eq} \Rightarrow \textbf{list}\ \beta \in \mathsf{Eq}$$
$$\begin{aligned}
\textbf{where}\ (=) = \ & \textbf{letrec}\ f = \textbf{fn}\, l_1 \Rightarrow \textbf{fn}\, l_2 \Rightarrow \textbf{case}\ l_1\ \textbf{of} \\
& \quad [] \to \ \textbf{case}\ l_2\ \textbf{of}\ [] \to \textbf{true} \mid _ \to \textbf{false} \\
& \mid\ x : xs \to \ \textbf{case}\ l_2\ \textbf{of}\ [] \to \textbf{false} \\
& \qquad\qquad \mid\ y : ys \ \to \ \textbf{if}\ (=)\, x\, y\ \textbf{then}\ f\, xs\, ys\ \textbf{else false} \\
& \textbf{in}\ \ f
\end{aligned}$$

Bemerkungen

- I.a. ist es praktischer, mehrere Operationen zu einer Klasse zusammenzufassen – z.B. um eine Klasse Number zu definieren, mit den üblichen vier Grundrechenarten.

  In dieser Hinsicht verhält sich eine Klasse ganz ähnlich wie ein Interface ;-)
- Eine Klassendeklaration kann auch direkt diverse abgeleitete Operationen implementieren, wie z.B. eine Gleichheit, falls es nur ein $\leq$ gibt.

  Insofern könnte man damit generisch eine Klasse zu einer Unterklasse einer andern machen :-)
- Praktisch wird man zusätzlich zu den vom System bereit gestellten Typen auch Systemklassen bereitstellen, in die die eingebauten Typen eingeordnet sind.

Wie inferiert man Klassen?

Idee 1:

1. Ignoriere die Klassenbedingungen;

   Inferiere für jeden Ausdruck den polymorphen Typ;

2. Überprüfe für jedes Vorkommen von überladenen Operatoren, dass die entsprechenden Typen den Operator auch implementieren!

3. Wie übersetzt man getypte Programme?

Idee 2:

- Modifiziere polymorphe Typinferenz so, dass sie bei der Einführung eines Typschemas jeweils die notwendigen Bedingungen mit vermerkt;
- Verwalte dazu neben Γ eine Sortenumgebung S, die für jede Typvariable die Menge der für sie benötigten Klassen sammelt;
- neben dem Typ für jeden Teilausdruck eine Übersetzung liefert ...

  Aus $\Gamma\,,\ S \vdash e :\quad \forall\alpha \in \mathcal{C}.\sigma$ wird:

$$\mathbf{fn}\,\alpha \Rightarrow e'$$

- Insbesondere benötigen wir eine modifizierte Unifikation ...

Modifizierte Unifikation

Um den Algorithmus $\mathcal{W}$ zu modifizieren, benötigen wir eine Unifikationsfunktion, die die Klasseninformation mit verwaltet:

$$\begin{array}{lcl} \textbf{fun}\ \mathsf{class-unify}\ (\tau_1, \tau_2)\ S & = & \textbf{case}\ \mathsf{unify}\ (\tau_1, \tau_2)\ \emptyset \\ \quad \textbf{of}\ \ \mathsf{Fail} & \rightarrow & \mathsf{Fail} \\ \quad \mid\ \ \theta & \rightarrow & (\theta, \theta\, S) \end{array}$$

Dabei liefert $\theta\, S$ die Klassenannahmen, die sich aus den Klassenannahmen in S für die Typvariablen im Bild von θ ergeben, wenn man die Instanz-Deklarationen berücksichtigt ...

Beispiel

Instanz-Deklarationen:

$$\begin{array}{lcl} \mathsf{list} & : & \alpha \in \mathsf{Eq} \Rightarrow \mathsf{list}\,\alpha \in \mathsf{Eq} \\ \mathsf{set} & : & \alpha \in \mathsf{Comp} \Rightarrow \mathsf{set}\,\alpha \in \mathsf{Eq} \end{array}$$

Dann haben wir für:

$$S = \{\alpha \mapsto \mathsf{Eq}\} \qquad \theta = \{\alpha \mapsto \mathsf{list}\,(\mathsf{set}\,\beta)\}$$

die neue Menge:

$$\theta\,S = \{\beta \mapsto \mathsf{Comp}\}$$

Insbesondere ist die substituierte Variable aus S verschwunden.

Modifizierter Abschluss

Der Aufruf close $(t, e)\ \Gamma\ S$ macht alle Typ-Variablen in t beschränkt generisch gemäß S, die nicht auch in Γ vorkommen ...

$$
\begin{aligned}
&\textbf{fun}\ \mathsf{close}\ (t, e)\ \Gamma\ S\ =\ \textbf{let}\\
&\qquad \textbf{val}\ \alpha_1, \ldots, \alpha_k = \mathsf{free}\,(t) \setminus \mathsf{free}\,(\Gamma)\\
&\qquad \textbf{val}\ \sigma = \forall\, \alpha_1 \in S(\alpha_1), \ldots, \alpha_k \in S(\alpha_k).\, t\\
&\qquad \textbf{val}\ S = S \setminus \{\alpha_1, \ldots, \alpha_k\}\\
&\qquad \textbf{in}\ (\sigma, \textbf{fn}\ \alpha_1 \Rightarrow \ldots \textbf{fn}\ \alpha_k \Rightarrow e, S)\\
&\qquad \textbf{end}
\end{aligned}
$$

Modifizierte Instantiierung

Die Instantiierung mit frischen Typ-Variablen leistet die Funktion:

$$
\begin{array}{l}
\textbf{fun}\ \mathsf{inst}\ (\sigma, x)\ =\ \textbf{let} \\
\qquad \textbf{val}\ \forall\, \alpha_1 \in S_1, \ldots, \alpha_k \in S_k.\, t = \sigma \\
\qquad \textbf{val}\ \beta_1 = \mathsf{new}()\quad \ldots\quad \textbf{val}\ \beta_k = \mathsf{new}() \\
\qquad \textbf{val}\ t = t[\beta_1/\alpha_1, \ldots, \beta_k/\alpha_k] \\
\qquad \textbf{in}\ (t, x\,\beta_1 \ldots \beta_k, \{\beta_1 \mapsto S_1, \ldots, \beta_k \mapsto S_k\}) \\
\qquad \textbf{end}
\end{array}
$$

Bemerkung

- Bei der Transformation sollten nur diejenigen Typparameter zu Funktionsparametern werden, die durch Typklassen beschränkt sind :-)
- Die Transformation fügt nicht-generische Typvariablen in die Ausgabeausdrücke ein :-)
- Während der Unifikation werden diese Variablen gebunden. Entsprechend werden sie nicht nur in den Typen, sondern auch in den Ausdrücken substituiert.
- Ein Aufruf $\mathsf{op}\ \tau$ für einen Operator op der Klasse C kann dann zur Laufzeit aufgelöst werden, indem die Implementierung des Operators in der Instanzdeklaration von τ nachgeschlagen wird.

Der Algorithmus $\mathcal{W}$ (erweitert):

$$
\begin{array}{lll}
 & \ldots & \\
| & x & \rightarrow \quad \textbf{let}\ (t, e, S') = \mathsf{inst}\ (\Gamma(x), x) \\
 & & \quad \textbf{in}\ (t, e, S \cup S', \theta) \\
 & & \quad \textbf{end} \\
| & (\textbf{let}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0) & \\
 & & \rightarrow \quad \textbf{let}\ \textbf{val}\ (t_1, e_1, S, \theta) = \mathcal{W}\ e_1\ (\Gamma, S, \theta) \\
 & & \qquad \textbf{val}\ (\sigma_1, e_1, S) = \mathsf{close}\ (\theta\ t_1, \theta\ e_1)\ (\theta\ \Gamma)\ S \\
 & & \qquad \textbf{val}\ \Gamma = \Gamma \oplus \{x_1 \mapsto \sigma_1\} \\
 & & \qquad\quad \ldots \\
 & & \qquad \textbf{val}\ (t_m, e_m, S, \theta) = \mathcal{W}\ e_m\ (\Gamma, S, \theta) \\
 & & \qquad \textbf{val}\ (\sigma_m, e_m, S) = \mathsf{close}\ (\theta\ t_m, \theta\ e_m)\ (\theta\ \Gamma)\ S \\
 & & \qquad \textbf{val}\ \Gamma = \Gamma \oplus \{x_m \mapsto \sigma_m\} \\
 & & \qquad \textbf{val}\ (t_0, e_0, S, \theta) = \mathcal{W}\ e_0\ (\Gamma, S, \theta) \\
 & & \qquad \textbf{val}\ e = \textbf{let}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0 \\
 & & \quad \textbf{in}\ (t_0, e, S, \theta) \\
 & & \quad \textbf{end}
\end{array}
$$

Bemerkungen

- Die Typinferenz/Transformation startet mit $S_0 = \emptyset$ und

$$\Gamma_0 = \{\mathsf{op} \mapsto \sigma_{\mathsf{op}} \mid \mathsf{op} \text{ Operator}\}$$

- Bei jeder Instanz-Deklaration

$$\beta_1 \in \mathcal{C}_1, \ldots, \beta_k \in \mathcal{C}_k \Rightarrow \mathsf{c}(\beta_1, \ldots, \beta_k) \in C$$

muss überprüft werden, ob für die Definition des Operators op : $\forall\, \alpha \in C.\, \tau$ gilt:

$$\mathcal{W}\ e\ (\Gamma_0, \emptyset, \emptyset) \quad = \quad (\tau', e, S, \theta) \qquad \text{mit} \qquad \theta\tau' = \tau[\mathsf{c}(\beta_1, \ldots, \beta_k)/\alpha]$$

wobei:

$$S \subseteq \{\beta_1 \mapsto \mathcal{C}_1, \ldots, \beta_k \mapsto \mathcal{C}_k\}$$

Bemerkungen (Forts.)

...

- Am Ende wird die Substitution θ auf alle (freien Vorkommen von) Typvariablen im transformierten Ausdruck angewendet.
- Durch Pattern Matching auf den Typausdrücken wird die richtige Implementierung der Operatoren ausgewählt ...

$$\begin{aligned} \mathsf{op} = \mathbf{fn}\ \beta \Rightarrow\ & \mathbf{case}\ \beta\ \mathbf{of} \\ & \ldots \\ & \mathsf{c}(\beta_1, \ldots, \beta_k) \rightarrow \mathsf{op}_\mathsf{c}\ \beta_1\ \ldots\ \beta_k \\ & \ldots \end{aligned}$$