Beispiel:

$$
\begin{array}{l}
\textbf{let}\ \mathsf{apply} \quad = \quad \textbf{fn}\ f \ \Rightarrow\ \textbf{fn}\ x \ \Rightarrow\ f\ x; \\
\qquad \mathsf{inc} \quad = \quad \textbf{fn}\ y \ \Rightarrow\ y + 1; \\
\qquad \mathsf{single} \quad = \quad \textbf{fn}\ y \ \Rightarrow\ y : [\,] \\
\textbf{in}\ \ \mathsf{apply}\ \mathsf{single}\ (\mathsf{apply}\ \mathsf{inc}\ 1) \\
\textbf{end}
\end{array}
$$

Beispiel:

$$
\begin{array}{lll}
\textbf{let}\ \mathsf{apply} & = & \textbf{fn}\ f\ \Rightarrow\ \textbf{fn}\ x\ \Rightarrow\ f\ x; \\
\quad \mathsf{inc} & = & \textbf{fn}\ y\ \Rightarrow\ y+1; \\
\quad \mathsf{single} & = & \textbf{fn}\ y\ \Rightarrow\ y : [\,] \\
\textbf{in}\ \mathsf{apply\ single}\ (\mathsf{apply\ inc}\ 1) & & \\
\textbf{end} & &
\end{array}
$$

Wir finden:

$$
\begin{array}{lll}
\alpha[\mathsf{apply}] & = & (\alpha \to \beta) \to \alpha \to \beta \\
\alpha[\mathsf{inc}] & = & \textbf{int} \to \textbf{int} \\
\alpha[\mathsf{single}] & = & \gamma \to \mathsf{list}\ \gamma
\end{array}
$$

- Durch die Anwendung: apply single erhalten wir:

$$\begin{aligned} \alpha &= \gamma \\ \beta &= \mathsf{list}\ \gamma \\ \alpha[\mathsf{apply}] &= (\gamma \to \mathsf{list}\ \gamma) \to \gamma \to \mathsf{list}\ \gamma \end{aligned}$$

- Durch die Anwendung: apply inc erhalten wir:

$$\begin{aligned} \alpha &= \mathbf{int} \\ \beta &= \mathbf{int} \\ \alpha[\mathsf{apply}] &= (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \end{aligned}$$

$\Longrightarrow$ Typ-Fehler ???

Idee 1: Kopiere jede Definition für jede Benutzung ...

... im Beispiel:

```
let apply  =  fn f ⇒ fn x ⇒ f x;
    inc    =  fn y ⇒ y + 1;
    single =  fn y ⇒ y : [ ]
in  (fn f ⇒ fn x ⇒ f x) single
   ((fn f ⇒ fn x ⇒ f x) inc 1)
end
```

Idee 1: Kopiere jede Definition für jede Benutzung ...

... im Beispiel:

```
let apply  =  fn f ⇒ fn x ⇒ f x;
    inc    =  fn y ⇒ y + 1;
    single =  fn y ⇒ y : [ ]
in  (fn f ⇒ fn x ⇒ f x) single
   ((fn f ⇒ fn x ⇒ f x) inc 1)
end
```

\+ Die beiden Teilausdrücke $(\mathbf{fn}\ f \Rightarrow \mathbf{fn}\ x \Rightarrow f\ x)$ erhalten jeweils einen eigenen Typ mit unabhängigen Typ-Variablen :-)

\+ Das expandierte Programm ist typbar :-))

– Das expandierte Programm kann seeehr groß werden :-(

– Typ-Checking ist nicht mehr modular :-((

Idee 2: Kopiere die Typen für jede Benutzung ...

- Wir erweitern Typen zu Typ-Schemata:

$$
\begin{array}{lcl}
t & ::= & \alpha \mid \textbf{bool} \mid \textbf{int} \mid (t_1, \ldots, t_m) \mid \mathsf{list}\ t \mid t_1 \to t_2 \\
\sigma & ::= & t \mid \forall\, \alpha_1, \ldots, \alpha_k.t
\end{array}
$$

- Achtung: Der Operator $\forall$ erscheint nur auf dem Top-Level !!!
- Typ-Schemata werden für **let**-definierte Variablen eingeführt.
- Bei deren Benutzung wird der Typ im Schema mit frischen Typ-Variablen instantiiert ...

Neue Regeln:

$$\text{Inst:}\quad \frac{\Gamma(x) = \forall\, \alpha_1, \ldots, \alpha_k.t}{\Gamma \;\vdash\; x \;:\; t[t_1/\alpha_1, \ldots, t_k/\alpha_k]} \qquad (t_1, \ldots, t_k \quad \text{beliebig})$$

$$\text{Let:}\quad \frac{\begin{array}{llll} \Gamma_0 & \vdash & e_1 \;:\; t_1 & \Gamma_1 = \Gamma_0 \oplus \{x_1 \mapsto \mathsf{close}\, t_1\, \Gamma_0\} \\ & \ldots & & \ldots \\ \Gamma_{m-1} & \vdash & e_m \;:\; t_m & \Gamma_m = \Gamma_{m-1} \oplus \{x_m \mapsto \mathsf{close}\, t_m\, \Gamma_{m-1}\} \\ \Gamma_m & \vdash & e_0 \;:\; t_0 & \end{array}}{\Gamma_0 \;\vdash\; (\mathbf{let}\; x_1 = e_1; \ldots; x_m = e_m\; \mathbf{in}\; e_0) \;:\; t_0}$$

Der Aufruf close $t\ \Gamma$ macht alle Typ-Variablen in t generisch (d.h. instantiierbar), die nicht auch in Γ vorkommen ...

$$\begin{array}{l}\textbf{fun}\ \mathsf{close}\ t\ \Gamma\ =\ \textbf{let}\\ \qquad\textbf{val}\ \alpha_1,\ldots,\alpha_k = \mathsf{free}\,(t) \setminus \mathsf{free}\,(\Gamma)\\ \qquad\textbf{in}\ \forall\,\alpha_1,\ldots,\alpha_k.\,t\\ \qquad\textbf{end}\end{array}$$

Eine Instantiierung mit frischen Typ-Variablen leistet die Funktion:

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

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

$$
\begin{array}{lll}
\ldots & & \\
\mid\ x & \rightarrow & (\mathsf{inst}\ (\Gamma(x)), \theta) \\
\mid\ (\textbf{let}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0) & & \\
 & \rightarrow & \textbf{let}\ \textbf{val}\ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & \quad \textbf{val}\ \sigma_1 = \mathsf{close}\ (\theta\ t_1)\ (\theta\ \Gamma) \\
 & & \quad \textbf{val}\ \Gamma = \Gamma \oplus \{x_1 \mapsto \sigma_1\} \\
 & & \quad \ldots \\
 & & \quad \textbf{val}\ (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
 & & \quad \textbf{val}\ \sigma_m = \mathsf{close}\ (\theta\ t_m)\ (\theta\ \Gamma) \\
 & & \quad \textbf{val}\ \Gamma = \Gamma \oplus \{x_m \mapsto \sigma_m\} \\
 & & \quad \textbf{val}\ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
 & & \textbf{in}\ (t_0, \theta) \\
 & & \textbf{end}
\end{array}
$$

Beispiel:

$$
\begin{array}{lll}
\textbf{let}\ \mathsf{apply} & = & \textbf{fn}\ f\ \Rightarrow\ \textbf{fn}\ x\ \Rightarrow\ f\ x; \\
\quad \mathsf{inc} & = & \textbf{fn}\ y\ \Rightarrow\ y + 1; \\
\quad \mathsf{single} & = & \textbf{fn}\ y\ \Rightarrow\ y : [\,] \\
\textbf{in}\ \mathsf{apply}\ \mathsf{single}\ (\mathsf{apply}\ \mathsf{inc}\ 1) & & \\
\textbf{end} & &
\end{array}
$$

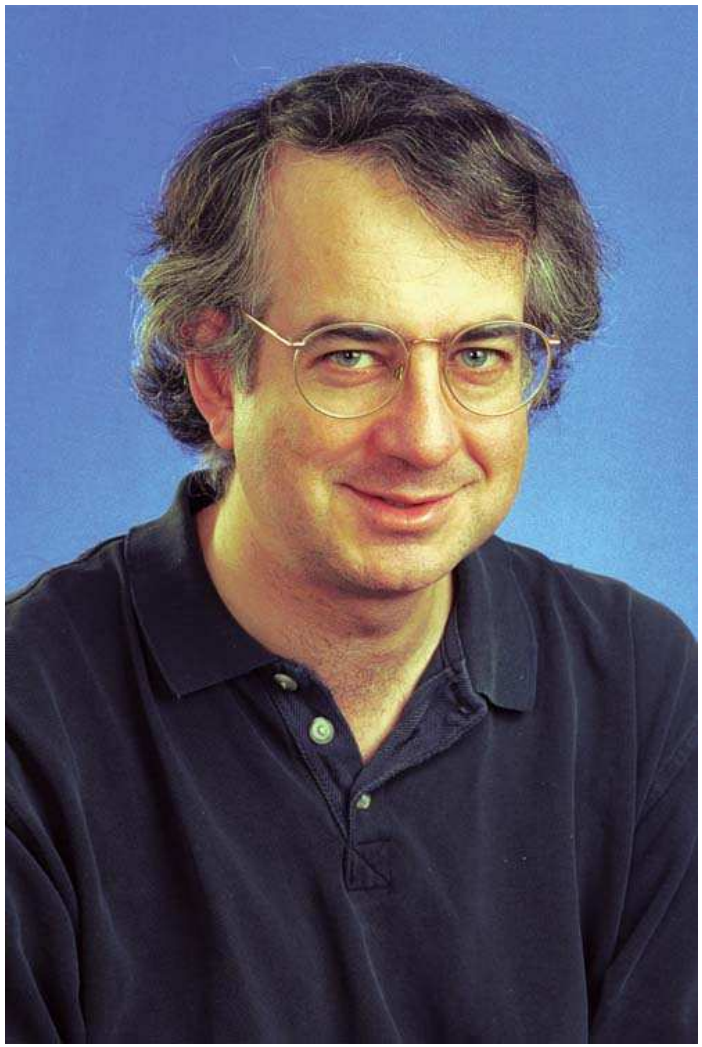
Wir finden:

$$
\begin{array}{lll}
\alpha[\mathsf{apply}] & = & \forall\, \alpha, \beta.\, (\alpha \to \beta) \to \alpha \to \beta \\
\alpha[\mathsf{inc}] & = & \textbf{int} \to \textbf{int} \\
\alpha[\mathsf{single}] & = & \forall\, \gamma.\, \gamma \to \mathsf{list}\ \gamma
\end{array}
$$

Bemerkungen:

- Der erweiterte Algorithmus berechnet nach wie vor allgemeinste Typen :-)
- Instantiierung von Typ-Schemata bei jeder Benutzung ermöglicht polymorphe Funktionen sowie modulare Typ-Inferenz :-))
- Die Möglichkeit der Instantiierung erlaubt die Codierung von DEXPTIME-schwierigen Problemen in die Typ-Inferenz ??

  ... ein in der Praxis eher marginales Problem :-)
- Die Einführung von Typ-Schemata ist nur für nicht-rekursive Definitionen möglich: die Ermittlung eines allgemeinsten Typ-Schemas für rekursive Definitionen ist nicht berechenbar !!!



Harry Mairson, Brandeis University

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz nützlich :-)
- Darum erweitern wir unsere kleine Programmiersprache um Referenzen:

$$e \;::=\; \ldots \;\mid\; \textbf{ref}\; e \;\mid\; !e \;\mid\; e_1 := e_2$$

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz nützlich :-)
- Darum erweitern wir unsere kleine Programmiersprache um Referenzen:

$$e \;::=\; \ldots \;\mid\; \textbf{ref}\; e \;\mid\; !e \;\mid\; e_1 := e_2$$

Beispiel:

```
let count = ref 0;
    new   = fn () ⇒ let
                      ret = !count;
                      _   = count := ret + 1
                    in ret
in new() + new()
```

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:

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

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:

$$v ::= \textbf{bool} \mid \textbf{int} \mid \text{list } v \mid (v_1, \ldots, v_m) \mid t \to t$$

... 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.