

Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

18. Dezember 2007

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
g x y = Node True (g x y) (g y x)
```

g ist iterativ typisierbar und Milner-typisierbar,
aber Milner-Typisierung ergibt **spezielleren** Typ.

Milner-Typ:

```
g :: forall a t. t -> t -> Tree Bool
```

iterativer Typ:

```
g :: forall a b c. b -> c -> Tree Bool
```

Denn das iterative Verfahren erzwingt
keinen Zusammenhang zwischen den Argumenten x,y von g

Milner-Verfahren: Vereinfachung bei rekursiven Superkombinatoren:

In den Typannahmen:

- bereits berechnete Typen von Superkombinatoren mit $\forall(\delta)$
- aktuell zu typisierende Superkombinatoren mit **einer** Typvariablen: $sc_1 : \tau_1; sc_2 : \tau_2, \dots$
(analog zu lokalen Variablen)

```
[]      ++ ys = ys  
(x:xs) ++ ys = x : (xs++ys)
```

Verwendung der Typregeln unter der Annahme,
dass (++) als case übersetzt ist

Annahmen: $(++) : a; (x:xs) : b_1; ys : b_2.$

	$a = b_1 \rightarrow b_2 \rightarrow b_3$
<code>[] ++ ys = ys</code>	$b_1 \doteq [a_1], b_3 \doteq b_2$
<code>(x:xs)</code>	$x :: a_2, xs :: a_3, \text{ ergibt: } a_3 \doteq [a_2] \doteq b_1$
<code>(x:xs) ++ ys</code>	$b_1 \doteq [a_1] \doteq [a_2], \text{ also } a_1 \doteq a_2.$
<code>(xs++ys)</code>	$:: b_2$ keine neue Gleichung;
<code>x: (xs++ys)</code>	$b_2 \doteq [a_1]$

Erst hier wird der Resultattyp eingeschränkt!

Ergebnis: $(++) :: [a_1] \rightarrow [a_1] \rightarrow [a_1]$

```
foldr f z as = case_List as [] -> z;  
              x:xs -> f x (foldr f z xs)
```

Typannahmen: $\text{foldr} :: \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4, f : \alpha_1, z : \alpha_2, as : \alpha_3$

```
case_List as [] -> z; ...:  $\alpha_3 = [a_5], \alpha_2 = \alpha_4$   
x:xs                        $x:\alpha_6, xs: [a_6] = [a_5], \text{d.h. } \alpha_6 = \alpha_5$   
f x (foldr f z xs):        $\alpha_1 = \alpha_6 \rightarrow \alpha_2 \rightarrow \alpha_2:$   
                           Da der Resultattyp durch  $z$  festgelegt wird.
```

$\implies :$ $\text{foldr} :: (\alpha_5 \rightarrow \alpha_2 \rightarrow \alpha_2) \rightarrow \alpha_2 \rightarrow [a_5] \rightarrow \alpha_2.$

Typ in Haskell: $\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

Was passiert mit im Programm vorgegebenen Typen (Typzusicherungen)?

- Typchecker berechnet den Typ trotzdem
- Test: es muss gelten: berechneter Typ \geq vorgegebener Typ
- Der (speziellere) vorgegebene Typ eines SK wird beim Typcheck anderer SK' verwendet!

Bemerkungen zu unendlichen Typen

- Man kann z.B. (eingeschränkte) Bäume mittels geschachtelter Listen implementieren.
- Die Unifikation im Typ-Checker kann ohne Occurs-Check implementiert werden
- **Nachteil:** eine bestimmte Klasse von Typfehlern bleiben unentdeckt: z.B. **falsche** Schachtelung von Datenstrukturen

Unendliche Typen: Beispiel

$A = [B]$

$B = [A]$

Typ von A, B : “unendlich tief geschachtelte Liste”

Darstellungsmöglichkeit:

$A :: a \text{ where } a == [a]$

$B :: a \text{ where } a == [a]$

$f\ x = f$ (Argument-ignorierende Funktion)

Annahmen: $\{f : \tau_1 \rightarrow \tau_2; x : \tau_3\}$

f : $\tau_1 \doteq \tau_3$
Superkomb-Regel: $\tau_1 \rightarrow \tau_2 \doteq \tau_2$.

Milner-Typisierung mit unendlichen Typen liefert :

$\tau_1 \rightarrow \tau_2 \doteq \tau_2$.

iterative Typisierung mit unendlichen Typen liefert:

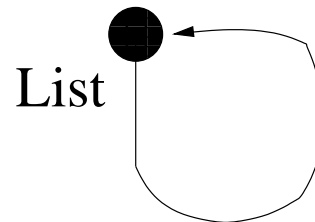
$\tau_1 \rightarrow (\tau_1 \rightarrow (\tau_1 \rightarrow (\tau_1 \dots)))$.

Ein “unendlicher Typ” :

bei Milner-Typisierung:

- “regulärer Baum” (bei Milner-Typis.)
unendlicher Baum mit endlich vielen Knoten (bei Milner)
gerichteter Graph mit endlich vielen Knoten.

$a = [a]$:



Bei iterativer Typisierung:

- unendlicher Baum mit unendlich vielen Knoten
- gerichteter Graph mit unendlich vielen Knoten.

Typsystem für Typklassen mit Grundtypen

Darstellung einer (evtl. unendlichen) Menge von Typen durch polymorphen Typ, der durch Typklassen “eingeschränkt” ist.

Typklassen = einstellige Prädikate auf Typen.

Syntax eines polymorphen TK-Typs:

$$TKL_1 \alpha_1, \dots, TKL_n \alpha_n \Rightarrow \tau$$

Typklasse TKL_i ; Typvariablen α_i ; polymorpher Typ τ

\implies polymorphe Typen mit Constraints

Semantik:

$\text{TKL}_1 \alpha_1, \dots, \text{TKL}_n \alpha_n \Rightarrow \tau$

entspricht

$\{\sigma(\tau) \mid \sigma \text{ ist Typsubst. und } \sigma(\alpha_i) \in S(\text{TKL}_i) \text{ für alle } i\}$

(Menge aller Grundinstanzen)

Beispiel

bestimme den Typ von `[] == []`

```
==          hat Typ  Eq a => a -> a -> Bool.  
[]         hat Typ  [b]  
((==) [])  hat Typ  Eq [b] => [b] -> Bool
```

Es gilt: `Eq [b]` gdw. `Eq b`. Deshalb:

```
((==) [])  hat Typ  Eq b => [b] -> Bool
```

das zweite `[]` hat Typ `[c]`

Ergebnis: `[] == [] :: Eq b => Bool`

```
[] == [] :: Eq b => Bool
```

`b` ist irrelevant: man kann kein Objekt vom Typ `b` benutzen:

deshalb ist der vernünftige Typ: `[] == [] :: Bool`

`b` könnte man in Haskell als existenziell quantifizierten Typ sehen

Frühere Versionen von Haskell: `[] == []` war typisierbar, aber nicht auswertbar da sich die Überladung nicht auflösen ließ.

Beispiel

$(\lambda x \rightarrow x) == (\lambda x \rightarrow x)$

ist nicht typisierbar:

$((==) (\lambda x \rightarrow x)) :: \text{Eq } (a \rightarrow a) \Rightarrow (a \rightarrow a) \rightarrow \text{Bool}.$

Aber $\text{Eq } (a \rightarrow a)$ gilt nicht \Rightarrow Typfehler.

Beispiel

```
summe xs = case_List xs of {[] -> 0; y:ys -> y+summe ys}
```

Annahme: $\{xs : \tau_1, \text{summe} : \forall \alpha\}$.

$\tau_1 = [\tau_2], y : \tau_2, ys : [\tau_2],$

$0 :: \text{Num } a \Rightarrow a,$

$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a,$

$(y +) : \text{Num } \tau_2 \Rightarrow \tau_2 \rightarrow \tau_2.$

Superkombinatorregel ergibt: $\text{Num } \tau_2 \Rightarrow [\tau_2] \rightarrow \tau_2.$

Das ist auch der Fixpunkt.

Wird auch vom Haskell Typchecker berechnet.

Beispiel Lösen von linearen Gleichungen nach dem Gauss-Verfahren (gauss.hs).

Die Funktion `gauss` hat als Argument die Parameter des Gleichungssystems, die letzte Spalte sind die Werte der rechten Seiten.

```
gauss :: Fractional a => [[a]] -> [a]
```

```
matrix :: Num a => [[a]]
```

So soll es sein:

```
:t (gauss matrix) :: Num a => [a]
```

Constraint-Verfahren liefert:

```
gauss matrix :: (Fractional a, Num a) => [a]
```

Da $\text{Fractional} \subseteq \text{Num}$, reicht `Fractional`

Versuch: hypothetische Erweiterung um **Subtypen**
Analog zu Objektorientierung (ohne Haskell-Typklassen)

Beispiel einer Subtypenhierarchie

Nat < Int < Float < Complex

Gegeben: Unterausdruck $s :: \tau$ im Programm;

Prinzip: s darf typmäßig durch
jeden Ausdruck $t : \tau'$ ersetzt werden, wenn $\tau' \leq \tau$.

Fortsetzung der Untertypen-Beziehung auf Typausdrücke:

Listentypen: $[Int] < [Complex]$
Paare: $(Int, Float) < (Float, Complex)$

Allgemein:

Für benutzerdefinierte Typkonstruktoren TC , wenn Argumente der Konstruktoren keinen \rightarrow -Typ haben:

aus $\forall i : \tau_i \leq \tau'_i$ folgt $(TC \ \tau_1 \ \dots \ \tau_n) \leq (TC \ \tau'_1 \ \dots \ \tau'_n)$.

Das nennt man *kovariant*

Kovarianter Typkonstruktor

Fortsetzung der Untertypen-Beziehung auf Typausdrücke:

Was ist mit Funktionstypen ?

Gilt

$$\tau_1 \leq \tau'_1 \text{ und } \tau_2 \leq \tau'_2 \implies \tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2 ?$$

Gilt nicht:

Man kann aus $\tau_1 \leq \tau'_1$ und $\tau_2 \leq \tau'_1$ **nicht schließen**, dass
 $\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2$:

Gegenbeispiel:

$f \quad :: \quad Int \rightarrow Complex$

$g \quad :: \quad Nat \rightarrow Float$

In

$f((-1) :: Int) \quad + \quad Complex \quad i$

kann f **nicht** durch g ersetzt werden,
denn g kann keine Int -Argumente.

$Nat \rightarrow Float \not\leq Int \rightarrow Complex$

Aber:

$$\begin{aligned} f' &:: \text{Nat} \rightarrow \text{Complex} \\ g' &:: \text{Int} \rightarrow \text{Float} \end{aligned}$$

In $f'(1 :: \text{Nat}) +_{\text{Complex}} i$
ist f' durch g' ersetzbar

Verallgemeinerung:

$$\tau_1 \leq \tau'_1 \text{ und } \tau_2 \leq \tau'_2 \text{ impliziert } \tau'_1 \rightarrow \tau_2 \leq \tau_1 \rightarrow \tau'_2$$

Diese Fortsetzung der Ordnung
ist *kontravariant* im ersten Argument von \rightarrow