

Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

18. Dezember 2007

Berechnung von allgemeinsten und polymorphen Typen von Ausdrücken

Einige Annahmen:

- Nur polymorphe Typen werden berechnet (keine allgemeinen Typmengen)
- pro Superkombinator und Konstruktor ein polymorphes Typschema in A .
- Notation: τ, μ beliebige Typen (mit Typvariablen)
 α : Typvariablen

- Gleichheit von polymorphen Typen ist effizient testbar:
Denn:
 $\tau_1 = \tau_2$ (als Mengen)
gdw.:
 $\tau_1 = \tau_2$ (syntaktisch)
(bis auf Umbenennung von Typvariablen.)
- Subsumption von polymorphen Typen ist effizient testbar:
 $\tau_1 \subseteq \tau_2$ (als Mengen)
gdw.:
 $\exists \sigma : \sigma(\tau_2) = \tau_1$ (syntaktisch)

Wiederholung der Regeln (ohne Steuerung)

Sei G eine Multimenge von Gleichungen.

- $$\frac{\{(c\ t_1 \dots t_n) \doteq (c\ s_1 \dots s_n)\} \cup G}{\{s_1 \doteq t_1, \dots, s_n \doteq t_n\} \cup G}$$

- $$\frac{\{\alpha \doteq \tau\} \cup G}{\{\alpha \doteq \tau\} \cup G[\tau/\alpha]}$$

wenn $\tau \neq \alpha$ und α kommt in τ nicht vor
und α kommt in G vor.

- $$\frac{\{\alpha \doteq \alpha\} \cup G}{G}.$$

- $$\frac{\{(c\dots) \doteq (d\dots)\} \cup G}{FAIL}$$

wenn c und d verschiedene Typkonstruktoren sind.

- $$\frac{\{\alpha \doteq \tau\} \cup G}{FAIL}$$

wenn $\tau \neq \alpha$ und α kommt in τ vor

Der Occurs-Check erscheint als “infinite type“ in Fehlermeldungen

Occurs-Check-Fehler tritt auf zum Beispiel
beim Haskell-Ausdruck:

```
(\xs -> case xs of y:ys -> y++ys).
```

Eigenschaften der Unifikation:

1. das Ergebnis ist bis auf Umbenennung der Variablen eindeutig
2. Der Algorithmus ist total korrekt.
Er terminiert und findet eine allgemeinste Lösung,
wenn irgendeine Lösung existiert.
Ist effizient implementierbar

Gleichungssystem G ist **gelöst**, wenn es von der Form

$\{\alpha_1 \doteq \tau_1, \dots, \alpha_n \doteq \tau_n\}$ ist,

und alle α_i sind verschieden und kommen nicht in den τ_j vor.

Die Lösung ist als Substitution ablesbar:

$\{\alpha_1 \doteq \tau_1, \dots, \alpha_n \doteq \tau_n\}$ ergibt $\{\alpha_1 \rightarrow \tau_1, \dots, \alpha_n \rightarrow \tau_n\}$ als Lösung

mittels Fixpunktiteration „von oben“

Start mit allgemeinsten Annahmen $sc: \forall \alpha$

- Verkleinerung der Mengen
(bzw. Spezialisierung der polymorphen Typen)
bis Typannahmen konsistent sind
- das ergibt **deterministisches** Berechnungsverfahren für den
größten Fixpunkt.
- Behandlung der Typgleichungen mit Unifikation
- **Bei Nichtterminierung: kein** polymorpher Typ für den Super-
kombinator

1. Abhängigkeitsanalyse der Superkombinatoren
2. Typisierung startet “von unten” in der Abhängigkeitshierarchie
3. Bereits berechnete Typen werden später mitverwendet
Vorgegebene Typ-Annahmen über die Konstruktoren,
und bereits berechneten Typen einiger Superkombinatoren
5. lokale Terminierung, für eine abhängige Menge von Superkombinatoren
falls alle zugehörigen Annahmen konsistent sind
6. Typisierung der Rümpfe der Superkombinatoren

Zwei geschachtelte Iterationen:

1. Abhängigkeitshierarchie der Superkombinatoren.
2. Typannahmen-Verfeinerung

Schreibweisen: $t : \tau, E$ ist das Paar aus hergeleitetem Typ
und den Gleichungen E zwischen Typen.

E_σ das Gleichungssystem ist gelöst mit Lösung σ

τ bezeichnet ab jetzt Typen die Typvariablen enthalten können.

Axiom für Variablen

$A \cup \{x : \tau\} \vdash x : \tau, \emptyset$ τ ist in den Annahmen eine Typvariable.

Axiom für Konstanten und Superkombinatoren

$A \cup \{c : \forall \delta\} \vdash \{c : \tau', \emptyset\}$

wobei τ' eine (frisch) umbenannte Version von δ

Anwendungsregel

$$\frac{A \vdash a : \tau_1, E_1; A \vdash b : \tau_2, E_2}{A \vdash (a \ b) : \tau_3, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \tau_3\}}$$

wobei τ_3 neue Typvariable.

Unifikation:

$$\frac{A \vdash a : \tau, E}{A \vdash a : \tau, E'}$$

wobei E' aus E durch Anwendung von Unifikationsregeln entsteht, wobei kein FAIL auftritt darf.

$$\frac{A \vdash a : \tau, E}{FAIL}$$

wenn bei Unifikation von E ein FAIL auftritt

case-Regel:

$$A \vdash e : \mu_0, F_0$$

$$A \cup \{x_{1,1} : \tau_{1,1} \dots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash (c_1 x_{1,1} \dots x_{1,n(1)}) : \mu_1, F_1$$

...

$$A \cup \{x_{m,1} : \tau_{m,1} \dots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash (c_m x_{m,1} \dots x_{m,n(m)}) : \mu_m, F_m$$

$$A \cup \{x_{1,1} : \tau_{1,1} \dots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash e_1 : \tau_1, E_1$$

...

$$A \cup \{x_{m,1} : \tau_{m,1} \dots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash e_m : \tau_m, E_m$$

$$A \vdash (\text{case}_T e \{(c_1 x_{1,1} \dots x_{1,n(1)}) \rightarrow e_1; \dots; (c_m x_{m,1} \dots x_{m,n(m)}) \rightarrow e_m\}) : \tau,$$
$$F_0 \cup F_1 \cup \dots \cup F_m \cup \{\mu_0 \doteq \mu_1, \dots, \mu_0 \doteq \mu_m\} \cup$$
$$E_1 \cup \dots \cup E_m \cup \{\tau \doteq \tau_1, \dots, \tau \doteq \tau_m\}$$

τ ist eine neue Typvariable

Abstraktion:

$$\frac{A \cup \{x : \tau_1\} \vdash t : \tau_2, E}{A \vdash (\lambda x.t) : \tau_1 \rightarrow \tau_2, E}$$

wobei τ_1 neue Typvariable.

Typberechnung:

$$\frac{A \vdash t : \tau, E}{A \vdash t : \sigma(\tau), E_\sigma}$$

σ ist die mit Unifikation berechnete Lösung (kein FAIL erlaubt)

Superkombinator-Regel zur Berechnung neuer Annahmen

$$\frac{A \cup \{x_1 : \tau_1, \dots, x_n : \tau_n\} \vdash e : \tau, E}{A \vdash sc : \sigma(\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau)}$$

Wenn σ Lösung von E und
wenn $sc \ x_1 \ \dots \ x_n = e$ Definition für sc ist.

$\text{length as} = \text{case_list as } \{\text{Nil} \rightarrow 0; (\text{Cons } x \text{ xs}) \rightarrow (1 + \text{length xs})\}$

In A lassen wir die Typen der Konstruktoren und Konstanten weg:

$0 : \text{Int}, 1 : \text{Int}, \dots$

$A = \{\text{length} : \forall \alpha, \text{as} : \tau_1, x : \tau_2, \text{xs} : \tau_3, + : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$

$A \vdash \text{length} : \alpha'$

$A \vdash (\text{length xs}) : \tau_4; \tau_3 \rightarrow \tau_4 \doteq \alpha'$

Anwendungs-Regel; case-Regel auf Resultate:

$A \vdash 1 + (\text{length xs}) : \text{Int}, \tau_4 \doteq \text{Int}$

und $0 : \text{Int}$

case-Regel auf Pattern

$\text{Nil} : [\beta_1]$

$\text{Cons} : \beta_2 \rightarrow [\beta_2] \rightarrow [\beta_2]$

$A \vdash (\text{Cons } x \text{ xs}) : [\beta_2], \{\tau_2 = \beta_2, \tau_3 = [\beta_2]\}$

$A \vdash (\text{case_list as } \{\text{Nil} \rightarrow 0; (\text{Cons } x \text{ xs}) \rightarrow (1 + \text{length xs})\}) : \tau,$

$\{\tau \doteq \text{Int}, \tau_2 \doteq \beta_2, \tau_3 \doteq [\beta_2], \tau_4 \doteq \text{Int}, \tau_1 \doteq [\beta_2] \doteq [\beta_1]\}$

alle Gleichungen:

$$\{\tau_3 \rightarrow \tau_4 \doteq \alpha', \tau_4 \doteq \text{Int}, \tau_3 = [\beta_2], \\ \tau_2 = \beta_2, \tau \doteq \text{Int}, \tau_3 \doteq [\beta_2], \tau_2 \doteq \beta_2, \tau_4 \doteq \text{Int}, \tau_1 \doteq [\beta_2] \doteq [\beta_1]\}$$

gültige Gleichungen

Nach Unifikation und Umformung; (nur relevante Variablen):

$$\{\tau \doteq \text{Int}, \tau_1 \doteq \tau_3 \doteq [\tau_2], \tau_4 = \text{Int}\}$$

Superkombinatorregel ergibt:

length: $[\tau_2] \rightarrow \text{Int}$.

Danach Konsistenz-Test der neuen Annahme (Neuer Typ-Check)

$g\ x = (\text{Cons } 1\ (g\ (g\ 'c')))$

$A := \{x : \tau_1, g : \forall a\}$

Anwendungsregel

$(g\ 'c')$ mit $g : \alpha_1$ ergibt:

$\{\alpha_1 = \alpha_2 \rightarrow \alpha_3; \alpha_2 = \text{Char}\}$.

$g\ (g\ 'c')$ ergibt $g : \beta$ und $\{\beta \doteq \beta_1 \rightarrow \beta_2, \beta_1 \doteq \alpha_3\}$,

$(\text{Cons } 1\ (g\ (g\ 'c')))$: ergibt $\beta_2 = [\text{Int}]$

Unifikation:

$\{\alpha_1 = \alpha_2 \rightarrow \alpha_3; \alpha_2 = \text{Char}, \beta \doteq \beta_1 \rightarrow \beta_2, \beta_1 \doteq \alpha_3, \beta_2 = [\text{Int}]\}$

Superkombinatorregel:

$g: \tau_1 \rightarrow [\text{Int}]$.

$g\ x = (\text{Cons } 1\ (g\ (g\ 'c')))$

Nächste Typannahme für g : $\forall \alpha. \alpha \rightarrow [\text{Int}]$.

Danach ergibt sich der gleiche Typ;

D.h. das iterative Verfahren terminiert für das Beispiel

Beachte: In Haskell ist diese Funktion nicht typisierbar.

```
g x = (Cons 1 (g (g 'c')))
```

```
:t (let g = \x -> 1:g(g 'c') in g)
```

```
Couldn't match '[a]' against 'Char'
```

```
Expected type: [a]
```

```
Inferred type: Char
```

```
In the first argument of 'g', namely 'c'
```

```
In the first argument of 'g', namely '(g 'c')'
```

Haskell's weitergehende Monomorphie-Beschränkung:
rekursiv definierte Funktionen dürfen im rekursiven Skopus
nur mit einem Typ verwendet werden

Beispiel mit 3 Iterationen!

$g\ x = \text{Cons } x\ (g\ (g\ 'c'))$

1. Iteration

$A = \{x : \tau_1, g : \forall \alpha\}$

$(g\ 'c') : \alpha_2; \alpha_1 \doteq \alpha_2 \rightarrow \alpha_3$ und $\alpha_2 \doteq \text{Char}$

$g(g\ 'c')$

$g : \beta$ ergibt $\beta \doteq \beta_1 \rightarrow \beta_2, \beta_1 \doteq \alpha_3,$

$g(g\ 'c') : \beta_2$

$(\text{Cons } x\ (g(g\ 'c')))$ ergibt $\beta_2 = [\tau_1]$

Gesamttyp nach erster Iteration: $g : \tau_1 \rightarrow [\tau_1]$

2. Iteration

$A_2 := \{x : \tau_1, g : \forall \alpha \rightarrow [\alpha]\}$

$(g \text{ 'c'})$ ergibt: $[\alpha_1]$ mit $g : \alpha_1 \rightarrow [\alpha_1]$ und $\alpha_1 = \text{Char}$

$g(g \text{ 'c'})$ ergibt $g : \beta_1 \rightarrow [\beta_1], \beta_1 = [\text{Char}]$

$(\text{Cons } x \ (g(g \text{ 'c'})))$ ergibt: $\tau_1 = [\text{Char}]$ und als Typ

$(\text{Cons } x \ (g(g \text{ 'c'})))$: $[[\text{Char}]]$

Gesamttyp nach 2. Iteration: $g : [\text{Char}] \rightarrow [[\text{Char}]]$.

3. Iteration

$A_3 := \{x : \tau_1, g : [\text{Char}] \rightarrow [[\text{Char}]]\}$

$(g \text{ 'c'}) : [\text{Char}] = \text{Char}$ liefert FAIL.

D.h. der Ausdruck ist **nicht typisierbar**

g hat **keinen Typ** im iterativen Typsystem mit polymorphen Typen

Beispiel dazu:

$g\ x = \text{Cons } x\ (g\ (g\ \text{bot}))$ $\text{bot}::\forall\gamma.$

Annahme: $\{x : \tau_1, g : \forall\alpha, \text{bot} : \forall\gamma\}$

- $(g\ \text{bot})$: $\alpha_1 \doteq \alpha_2 \rightarrow \alpha_3$ und $\alpha_2 \doteq \gamma_1$
- $g(g\ \text{bot})$: $g : \beta$ ergibt: $\beta \doteq \beta_1 \rightarrow \beta_2, \beta_1 \doteq \alpha_3,$
- $(\text{Cons } x\ (g(g\ \text{bot})))$: ergibt $\beta_2 = [\tau_1]$

Der Typ von g mit Superkombinatorregel: $\tau_1 \rightarrow [\tau_1]$

Neue Annahme: $\{x : \tau_1, g : \forall(\alpha \rightarrow [\alpha]), \text{bot} : \forall\gamma\}$

- $(g \text{ bot})$ ergibt $[\alpha_1]$ und $\alpha_1 \doteq \gamma_1$.
- $g(g \text{ bot})$: $g : \alpha_2 \rightarrow [\alpha_2], \alpha_2 \doteq [\alpha_1]$,
- $(\text{Cons } x (g(g \text{ bot})))$: ergibt $[[\alpha_1]] \doteq [\tau_1]$

Superkombinatorregel für $g: [\alpha_1] \rightarrow [[\alpha_1]]$

$g\ x = \text{Cons } x\ (g\ (g\ \text{bot}))$

Allgemeine Typbestimmung von g mit vollständiger Induktion;

$[\cdot]^n$ bedeutet n -fach $[\cdot]$.

$g\ x = \text{Cons } x\ (g\ (g\ \text{bot}))$

Annahme: $\{x : \tau_1, g : \forall([\alpha]^n \rightarrow [\alpha]^{n+1}), \text{bot} : \forall\gamma\}$

- $(g\ \text{bot})$ hat Typ $[\alpha_1]^{n+1}$
- $g(g\ \text{bot})$ $g : [\alpha_2]^n \rightarrow [\alpha_2]^{n+1},$
 $[\alpha_2]^n \doteq [\alpha_1]^{n+1} \Rightarrow \alpha_2 \doteq [\alpha_1].$
Es ergibt sich: $g(g\ \text{bot}) : [\alpha_1]^{n+2}$
- $\text{Cons } x\ (g(g\ \text{bot}))$ Wegen $\text{Cons} : \beta \rightarrow [\beta] \rightarrow [\beta]$ ergibt sich
 $[\tau_1] \doteq [\alpha_1]^{n+2}$ und somit $\tau_1 \doteq [\alpha_1]^{n+1}.$
- $\text{Cons } x\ (g\ (g\ \text{bot})) : [\alpha_1]^{n+2}$

Der hergeleitete Typ von g ist dann: $[\alpha_1]^{n+1} \rightarrow [\alpha_1]^{n+2}$

Der Typ wird immer tiefer geschachtelt: \implies Iteration terminiert nicht

$A = [B]$

$B = [A]$

Iteration schachtelt die Typen immer tiefer:

Die Annahme $A : \tau_1, B : \tau_2$ liefert

$A : [\tau_2], B : [\tau_1]$.

Konsistenz ist nicht erfüllt wg der Abhängigkeit: $\tau_1 \subseteq [\tau_2]$, und $\tau_2 \subseteq [\tau_1]$

Das iterative Typinferenzverfahren terminiert in diesem Fall nicht.

- Das iterative Typisierungsverfahren berechnet einen größten Fixpunkt der Typannahmen bzgl. eines Iterationsschritts.
- Typisierbar bedeutet: es gibt einen Fixpunkt (eine konsistente Typannahme).
- Das iterative Verfahren findet einen größten Fixpunkt, wenn es überhaupt einen Fixpunkt gibt, d.h. es ist vollständig und korrekt.
- Wenn es einen Typ gibt, dann wird das iterative Verfahren diesen finden, und kann höchstens einen allgemeineren polymorphen Typ liefern. Es wird also in dem Fall erfolgreich terminieren.

Es gilt:

Satz (iterative) Typisierung ist unentscheidbar.

Das Kern des Problems ist die sogenannten **Semi-Unifikation**, die unentscheidbar ist.

siehe: A. J. Kfoury, Jerzy Tiuryn, Pawel Urzyczyn: The Undecidability of the Semi-unification Problem. Information and Computation 102(1): 83-101 (1993)

(das war damals auch für die Fachleute überraschend)

Möglichkeiten, um Entscheidbarkeit zu erzwingen:

1. **Einschränkung der Sprache** (Typen, Programme)

Unklar:

2. **Abänderungen des Algorithmus**

z.B. Abbrechen, o.ä

Geht zu Lasten der Allgemeinheit der Ergebnisse

Nach n Iterationen mache einen “Milner”-Iterationsschritt

Beschränkung:

- Keine Umbenennungen der Typvariablen bei Instanzbildung aus den Typ-Annahmen
- Anschließend unifiziere die berechneten Typen der Superkombinatoren mit ihrem Typ in der Annahme

\implies Konsistenz der Typannahme ist danach automatisch erfüllt.

$g\ x = \text{Cons } x\ (g\ (g\ \text{bot}))$

iterative Typisierung terminiert nicht!

Anwenden des Milner-Schritts:

Annahme: $\{x : \tau_1, g : \forall \alpha \rightarrow [\alpha], \text{bot} : \gamma\}$

- $(g\ \text{bot})$: $\alpha \rightarrow [\alpha]$ und $\alpha \doteq \gamma_1$
- $g(g\ \text{bot})$ muss getypt sein $\Rightarrow g : \alpha \rightarrow [\alpha], \alpha \doteq [\alpha]$:
- Unifikation: $\alpha \doteq [\alpha]$ ergibt einen **Fehler**
- \implies **Nicht typisierbar**

Aussage

Werden für die zu typisierenden Superkombinatoren keine umbenannten Kopien von Typen erlaubt, dann terminiert das Typisierungsverfahren im ersten Iterationsschritt

Begründung:

Anfangs-Annahme $A = A_C \cup \{sc_i : \alpha_i\}$

Keine Kopien des Typs von Superkombinatoren sc_i :

\implies Typ der Superkombinatoren sc_i werden berechnet

Danach Unifikation des angenommenen Typs und des berechneten Typs

erfolgreich \implies Konsistenz der Typ-Annahmen

Oder: FAIL beim Unifizieren: dann so nicht typisierbar.

Abhängigkeitsanalyse

Seien $SK_i, i = 1, \dots, n$ die nicht typisierten Superkombinatoren.

Sei $SK_j \preceq SK_i$ gdw. SK_i benutzt SK_j im Rumpf.

\preceq^* sei reflexiv-transitive Hülle von \preceq .

\approx die Äquivalenzrelation zu \preceq^*

$$SK_i \approx SK_j \text{ gdw. } SK_i \preceq^* SK_j \wedge SK_j \preceq^* SK_i$$

minimale \approx -Äquivalenzklasse: Superkombinatoren sind rekursiv voneinander abhängig

- Abhängigkeitsanalyse von unten nach oben
- In jedem Typcheck-Schritt wird eine minimale \approx -Äquivalenzklasse typisiert.
Bereits typisierte SK sind in Annahmen; Typ dürfen umbenannt werden.
- Aktuell zu typisierende Superkombinatoren: Milner- Schritt
Abschluss eines Milner-Schritts:
Unifikation: Typannahmen der aktuelle SK = deren berechneter Typ.

- Vorgehensweise mit Abhängigkeitsanalyse:
i.a. allgemeinere Typen als “alle auf einmal“
- Das Milner-Verfahren liefert i.a. speziellere Typen als das iterative Verfahren
bzw. Milner-ungetypt statt iterativ wohl-getypt
- Das Milner-Verfahren terminiert und
liefert eindeutige polymorphe Typen

- Verwendung in Haskell, Java: generische Typen, ML, und Miranda
- Die worst-case Komplexität ist **EXPSPACE-hard**
Selten, aber möglich: exponentiell viele Typvariablen nötig
- Milner-Typsystem und das iterative Typsystem unterscheidet let-gebundene und lambda-gebundene Variablen:
let-gebundene dürfen polymorph sein,
lambda-gebundene Variablen müssen monomorph sein.

Rekursiv definierte Superkombinatoren dürfen im rekursiven Aufruf nur mit einem Typ benutzt werden.

$\text{map } f \text{ as} = \text{case_list as } \{\text{Nil} \rightarrow \text{Nil}; (x:xs) \rightarrow (f \ x) : (\text{map } f \ xs)\}$

Typannahme: $A = \{\text{map} : \alpha, f : \tau_1, \text{as} : \tau_2, x : \tau_3, xs : \tau_4\}$.

- $\text{map } f \text{ xs:}$ $\text{map} : \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3,$
 $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \doteq \alpha, \alpha_1 \doteq \tau_1, \alpha_2 \doteq \tau_4.$
- $(f \ x)$ $\tau_1 \doteq \tau_3 \rightarrow \tau_5$
- $(f \ x) : \text{map } f \text{ xs:}$ $(:): \tau_6 \rightarrow [\tau_6] \rightarrow [\tau_6], \tau_5 \doteq \tau_6, [\tau_6] \doteq \alpha_3$
- $\text{Nil} =_{Typ} (f \ x) : \text{map } f \text{ xs}$ $[\beta_1] \doteq [\tau_6]$
- $x:xs$ $\tau_4 \doteq [\tau_3]$
- $\text{as} =_{Typ} \text{Nil} =_{Typ} x:xs:$ $\tau_2 \doteq [\beta_2] \doteq [\tau_3]$
- Unifikation ergibt: Ergebnis ist vom Typ: $[\tau_6] \doteq \alpha_3$
 $\tau_5 \doteq \tau_6 \doteq \beta_1, \tau_3 \doteq \beta_2, \tau_4 \doteq \tau_2 \doteq [\tau_3]$
 $\alpha_1 \doteq \tau_1 \doteq \tau_3 \rightarrow \tau_5, \alpha_2 \doteq \tau_4, \alpha_3 \doteq \tau_2 \doteq [\tau_5]$

Insgesamt: $\text{map}: (\tau_3 \rightarrow \tau_5) \rightarrow [\tau_3] \rightarrow [\tau_5]$ oder, standardisiert:

$\text{map} : (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$

iterativ typisierbar, aber nicht Milner-typisierbar:

$g\ x = 1: g(g\ 'c')$ iterativer Typ: $\alpha \rightarrow [\text{Int}]$

Milner-Typisierung:

Annahmen: $\{g : \alpha_1 \rightarrow \alpha_2, x : \alpha_1\}$

- $g\ 'c'$: α_2 , $\alpha_1 = \text{Char}$
- $g\ (g\ 'c')$: α_2 ; $\alpha_2 = \text{Char}$
- $1:g(g\ 'c')$: erfordert mit $(:)$: $\beta \rightarrow [\beta] \rightarrow [\beta]$:
 $\beta \doteq \text{Int}, [\text{Int}] = \text{Char}$
nicht unifizierbar

Test in Haskell: nicht (Haskell-)typisierbar.