

Kapitel 4

Typen und Typcheck

4.1 Typsysteme, Typüberprüfung

Wir haben bisher algebraische Datentypen in KFPT, Erweiterungen von KFPT und Haskell kennengelernt, und gesehen, dass es in KFPT dynamische Typfehler geben kann, und dass Haskell ein strenges Typsystem hat, so dass es keine Typfehler zur Laufzeit gibt. Wir werden ein polymorphes Typsystem vorstellen, und zwei Varianten des dazugehörigen Typüberprüfungsalgorithmus (Hindley, Milner, Mycroft, Abramsky, Damas).

Wir legen die Sprache KFPT mit der Erweiterung (im 2. Kapitel mit KFPTSP bezeichnet) bezüglich der Typangaben der Konstruktoren zugrunde, so dass für jeden Konstruktor im Deklarationsteil ein Typ mit angegeben werden muss. Ebenso nehmen wir an, dass Superkombinatoren erlaubt sind.

Die Typregeln, die wir angeben werden, können analog auch auf andere funktionale Programmiersprachen angewendet werden, wobei auch für kompliziertere Konstrukte Typregeln entworfen werden können.

Erwünschte Eigenschaften eines Typsystems:

- Wenn ein Ausdruck positiv überprüft worden ist, d.h. einen Typ hat, dann sollte nach der Kompilierung zur Laufzeit kein dynamischer Typfehler auftreten.
- Außerdem sollten keine zu starken Programmierbeschränkungen durch das Typsystem erzwungen werden.
- Typisierung sollte effizient entscheidbar sein.

Das Milnersche polymorphe Typsystem ist ein strenges und statisches Typsystem, d.h. jeder erlaubte Ausdruck (und Unterausdruck) muss einen Typ haben. Die Typen existieren nur zur Compilezeit. Das Laufzeitsystem hat (fast) keine Typinformation mehr. Das ist im Gegensatz zu dynamischen Typsystemen, bei denen Typen auch zur Laufzeit der Programme geprüft werden.

Zur Wiederholung nochmals die Syntax von KFPT:

$\text{Exp} ::= V$ Hierbei steht V für Variablen
 $| (\backslash V \rightarrow \text{Exp})$ wobei V eine Variable ist.
 $| (\text{Exp}_1 \text{Exp}_2)$
 $| (c_{\text{Typ}} \text{Exp}_1 \dots \text{Exp}_n)$ wobei c Konstruktor mit $n = \text{ar}(c)$
 $| (\text{case}_{\text{Typname}} \text{Exp} \text{ of } \{ \text{Pat}_1 \rightarrow \text{Exp}_1; \dots; \text{Pat}_n \rightarrow \text{Exp}_n \})$
 Hierbei ist Pat_i Pattern zum Konstruktor i ,
 Es kommen genau die Konstruktoren zum **Typname** vor
 $\text{Pat}_i \rightarrow \text{Exp}_i$ heißt auch **case-Alternative**.

$\text{Pat} ::= (c V_1 \dots V_{\text{ar}(c)})$
 Die Variablen V_i müssen alle verschieden sein.
 Das sind *Pattern* bzw. *Muster*)

Zur Typisierung nehmen wir an, dass:

- **case** getypt ist, d.h. einen Typindex hat.
- Die Konstruktoren getypt sind; d.h. einen polymorphen Typ haben.

Man sieht schnell ein, dass es kein Typsystem geben kann, das alle KFPT-Programme erkennt, die keine Typfehler machen werden. Denn:

Satz 4.1.1 *Es gibt keinen Algorithmus, der für jedes eingegebene KFPT-Programm entscheidet, ob es einen dynamischen Typfehler (zur Laufzeit) geben wird oder nicht.*

Beweis. Die Zurückführung auf das Halteproblem ist einfach: Betrachte den Ausdruck

```

if t then case_Bool Nil {p1-> (Nil Nil) ; ... }
      else case_Bool Nil {p1-> (Nil Nil) ; ... }
  
```

Dieser Ausdruck ergibt bei beliebigem t (das keine Typfehler erzeugt) genau dann einen Typfehler, wenn t terminiert. Da man in KFPT eine (getypte) Turingmaschine codieren kann (Einsetzen für t), hat man das Halteproblem zu entscheiden. \square

Das Typsystem kann Ausdrücke als ungetypt zurückweisen, d.h. es werden bestimmte Programme (die ungetypten) verworfen und nicht akzeptiert. Die Konsequenz aus der Unentscheidbarkeit ist jedoch, dass es in jedem Typsystem ungetypte Programme gibt, die sinnvoll sind, bzw. die zumindest keine dynamischen Typfehler provozieren. Typüberprüfung kann zur Verbesserung der Effizienz beitragen, da i.a. nach der Typüberprüfung keine dynamische Typprüfungen zur Laufzeit mehr notwendig sind.

4.1.1 Polymorphes Typsystem nach Milner (& Mycroft)

Syntax für Typ-Ausdrücke:

$\text{Typ} ::= \text{Variable} \mid (\text{Typkonstruktor } \text{Typ} \dots \text{Typ}) \mid \text{Typ} \rightarrow \text{Typ}$

Die Anzahl der Argumente eines Typkonstruktors muss der Stelligkeit des Typkonstruktors entsprechen.

Wir erlauben (zunächst) nicht: Variablen als Typkonstruktoren und partiell angewendete Typkonstruktoren.

(Typ-)Variable: bedeutet beliebiger Typ, Bezeichnung: $\alpha, \beta, \gamma, \delta, \dots$. Gleiche Variablen bedeuten gleichen Typ.

Typkonstruktor: zu jedem algebraischen Datentyp gibt es einen. Die Stelligkeit wird bei der Definition der Daten-Konstruktoren festgelegt: Das ist die Anzahl der verschiedenen Typvariablen in den Konstruktoren. z.B. bei **List** (bzw. $[\cdot]$) ist es eine Typvariable (der Typ der Elemente). Bei **Paar** (bzw. (\cdot, \cdot)) sind es zwei Typvariablen.

\rightarrow ist der zweistellige Typ-Konstruktor für Funktionen.

Typkonstruktoren können benutzerdefiniert sein (z.B. **Baum** \cdot). Eingebaute Typkonstruktoren sind:

Liste	$[\cdot]$	ein Typ-Parameter erlaubt
Tupel	(\cdot, \dots, \cdot)	Stelligkeiten ≥ 2
Funktion	\rightarrow	Infix und Stelligkeit 2

Typen mit \rightarrow sind sogenannte *Funktionstypen*. Die Konvention ist, dass man normalerweise $a \rightarrow b \rightarrow c \rightarrow d$ schreiben darf, und der gemeinte Typ dann durch Rechtsklammerung entsteht, d.h. $a \rightarrow (b \rightarrow (c \rightarrow d))$. Es gibt spezielle syntaktische Konventionen für Tupel und Listen. Zum Beispiel ist $(\text{Int}, \text{Char})$ der Typ eines Paares von Zahlen und Zeichen; $[\text{Int}]$ ist der Typ einer Liste von Zahlen des Typs **Int**.

Man verwendet den doppelten Doppelpunkt zur Typangabe. $t :: \tau$ soll bedeuten, dass der Ausdruck t den Grund-Typ τ hat. Dies ist syntaktisch auch in Haskell-Programmen erlaubt.

- Basistypen nennt man auch *elementare Typen*.
- Einen Typ nennt man auch *Grundtyp*, wenn er keine Typvariablen enthält. Einen solchen Typ nennt man manchmal auch *monomorph*.
- Einen Typ nennt man *polymorph*, wenn er Typvariablen enthält.

Beispiel 4.1.2

- *Typ von $*$* : $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$.
- *Typ von Id*: $\alpha \rightarrow \alpha$.
- *Typ des Paar-Konstruktors*: $\text{Paar}: \alpha \rightarrow \beta \rightarrow (\alpha, \beta)$.

Hier bedeuten Vorkommen der gleichen Typvariable auch, dass dort immer Objekte des gleichen Typs gemeint sind: **Id** nimmt Objekte vom Typ α und gibt als Resultat wieder solche vom Typ α zurück.

Beispiel 4.1.3

- $\mathbf{length} :: [\alpha] \rightarrow \mathbf{Int}$
*D.h. die Funktion \mathbf{length} hat als Argument ein Objekt vom Typ $[\alpha]$, wobei α noch frei ist. Das Ergebnis muss vom Typ \mathbf{Int} sein.
 Will man den Typ als prädikatenlogische Formel ausdrücken, so könnte man schreiben:*

$$\forall \alpha. \mathbf{length} :: [\alpha] \rightarrow \mathbf{Int}$$

Das kann man interpretieren als: Für alle Typen α ist \mathbf{length} eine Funktion, die vom Typ $[\alpha] \rightarrow \mathbf{Int}$ ist.

Da das eine Aussage über Argumente und Resultate ist, sollte folgendes gelten: $\forall x. (x :: [\alpha]) \Rightarrow ((\mathbf{length} \ x) :: \mathbf{Int})$.

- $(: :: \alpha \rightarrow [\alpha] \rightarrow [\alpha])$.
Der Listenkonstruktor hat zwei Argumente. Das erste hat irgendeinen Typ α . Das zweite Argument ist eine Liste, deren Elemente vom Typ α , d.h. von gleichem Typ wie das erste Argument sein müssen. Das Ergebnis ist eine Liste vom Typ $[\alpha]$.
- $\mathbf{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$ *beschreibt den Typ der Funktion \mathbf{map} .*

Die Bedeutung des Typs $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$ kann man auf zwei Arten interpretieren:

1. α ist allquantifiziert. D.h. der Typ ist: $\forall \alpha. \alpha \rightarrow [\alpha] \rightarrow [\alpha]$.
2. Als eine Beschreibung einer Menge von Grundtypen, die ein Ausdruck haben kann. In diesem Falle ist das die Menge $\{\tau \rightarrow [\tau] \rightarrow [\tau] \mid \tau \text{ ist ein Grundtyp}\}$.

Annahme Damit wir eine Basis für konkrete Typen von Ausdrücken haben, wird davon ausgegangen, dass die **Datenkonstruktoren** einen vorgegebenen Typ haben, der bei der Deklaration der algebraischen Datentypen in einem Programm mitangegeben werden muss.

1. Dieser Typ gibt die Stelligkeit an, und die möglichen Typen und Typkombinationen der Argumente.
2. Der Zieltyp muss von der Form $T \alpha_1 \dots \alpha_n$ sein, wobei T jeweils der Typkonstruktor des zugehörigen algebraischen Datentyps ist und α_i Typvariablen sind.
3. Alle Typvariablen, die im Typ der Argumente vorkommen, sollten auch im Zieltyp des Datenkonstruktors vorkommen.

Beispiel 4.1.4

Paare: $(.,.) : \alpha \rightarrow \beta \rightarrow (\alpha, \beta)$

Listen: `List` (bzw. `[]`) wird als einstelliger Typkonstruktor definiert.

`Nil` und `:` sind die (Daten-) Konstruktoren.

`Nil`: $[\alpha]$ und `'::'`: $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$

Dies definiert Listen, bei denen die Elemente den gleichen Typ haben.

`(1 : (True : Nil))` ist dann nicht mehr möglich, da das zweite Argument von `:` einen Listentyp haben muss.

Die Konstruktoren von `Bool` haben folgende Typangaben: `True`: `Bool` und `False`: `Bool`.

Beispiel 4.1.5 markierte binäre Bäume: Definition in Haskell:

```
Data Tree a = Node (Tree a) (Tree a) | Leaf a
```

- `Tree` ist der einstellige Typkonstruktor.
- `Node`, `Leaf` sind die Konstruktoren.
- `Node`: $(\text{Tree } \alpha) \rightarrow (\text{Tree } \alpha) \rightarrow (\text{Tree } \alpha)$
- `Leaf`: $\alpha \rightarrow (\text{Tree } \alpha)$

4.1.2 Instanz eines polymorphen Typs

Dazu benötigt man (Typ-)Substitutionen σ : das sind Abbildung, die Variablen Typterme zuordnen. Diese kann man dann auch auf komplexere Typterme anwenden durch die Regel $\sigma(\mathbf{c} \ t_1 \dots t_n) = (\mathbf{c} \ (\sigma t_1) \dots (\sigma t_n))$

Dann ist $\sigma(\delta)$ Instanz eines Typs δ . Wenn nach der Instanziierung der Typ keine Variablen mehr enthält, dann sprechen wir von *Grundinstanz* (Grundtyp, monomorpher Typ).

z.B. Instanzen von $[\alpha]$ sind `[Int]` und `[[α]]`, wobei `[Int]` eine Grundinstanz ist.

4.2 Typregeln (auf Grundtypen) für KFPTSP

Wir geben Typregeln an, die es erlauben, aus den bereits berechneten Grundtypen von Unterausdrücken (bzw. Mengen von Grundtypen) die Typen der direkten Oberausdrücke herzuleiten. Da man mit diesen Regeln zunächst mal alle Typen eines Ausdrucks herleiten kann, wirkt das Verfahren etwas umständlich. Man kann aber auf diesem im Prinzip die anderen Verfahren logisch aufsetzen. Die Regeln sind folgendermaßen aufgebaut:

$$\frac{\text{bisherige Folgerungen}}{\text{neue Folgerung}}$$

Oberhalb der Linie stehen bisherige Folgerungen, die von der Form

$$\text{Typannahme} \vdash t_i : \tau_i$$

sind, und unterhalb steht die neu zu schließende Folgerung

$$\text{Typannahme} \vdash t : \tau$$

Hierbei stehen τ, τ_i jeweils für einen Grundtyp. Regeln ohne Voraussetzungen (mit leerer oberer Zeile) nennt man auch **Axiome**. Bei diesen wird die Annahme und der Strich weggelassen.

Die Typannahmen A sind eine Menge von Angaben über die möglichen Typen der freien Variablen und bekanntes Wissen über die Typen der Konstanten, wobei die Konstanten sowohl Konstruktoren als auch andere Funktionssymbole sein können.

Von den Variablen nehmen wir an, dass jede genau eine Typannahme $x : \tau$ in einer Typannahme A hat. Die Konstanten können mehrere Typannahmen haben: $c : \tau_1, \dots, c : \tau_n$. Im allgemeinen ist dies eine Menge T von Grundinstanzen eines einzigen (polymorphen) Typs mit Variablen. In diesem Fall schreiben wir diese Menge als eine Typangabe $c : \forall(\delta)$. Diese repräsentiert $T = \{\tau' \mid \tau' \text{ ist Grundinstanz von } \delta\}$. Die Menge der zugehörigen Grundtypen ist i.a. unendlich groß.

Die Regeln definieren eine Herleitbarkeitsrelation \vdash zwischen Annahmen über Typen von Variablen und Konstanten. $A \vdash s : \tau$ bedeutet, dass aus den Annahmen A über die Typen der Variablen und Konstanten in s das Typsystem den Typ τ von s herleiten kann. Da man für Konstanten mehrere Typannahmen machen kann, ergibt sich somit auch, dass man möglicherweise mehrere Typinferenzen ziehen kann. Damit kann man i.a. **eine Menge von Typen** für einen Term herleiten. Hierbei kann man in vielen Fällen die Menge der hergeleiteten Typen wieder als Instanzmenge eines polymorphen Typen berechnen. Interessanterweise kann man damit auch die Beschränkungen des Typklassensystem modellieren. Z.B. die Menge der Typen des Zeichens $+$ ist endlich: $\{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}, \dots\}$.

Das Semikolon trennt verschiedene mit “und“ verbundene Annahmen über die Ableitbarkeit von Typen.

4.2.1 Typisierungsregeln

In folgenden stehen τ, τ_i für Grundtypen, T für eine Menge von Grundtypen.

Axiom für Konstanten

$$A \cup \{c : T\} \vdash c : \tau \text{ wenn } \tau \in T$$

Axiom für Konstanten (polymorphe Typen)

$$A \cup \{c : \forall(\delta)\} \vdash c : \tau' \text{ wenn } \tau' \text{ Grundinstanz von } \delta$$

Axiome für freie Variablen

$$A \cup \{x : \tau\} \vdash x : \tau$$

Anwendungsregel:

$$\frac{A \vdash a : (\tau_1 \rightarrow \tau_2); A \vdash b : \tau_1}{A \vdash (a \ b) : \tau_2}$$

Superkombinator-Regel

$$\frac{A \cup \{x_1 : \tau_1, \dots, x_n : \tau_n\} \vdash e : \tau; \text{sc } x_1 \dots x_n = e \text{ ist die Definition für } sc}{A \vdash sc : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau}$$

Nebenbedingung ist, dass A keine Annahmen über x_i hat.

Abstraktion:

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

case-Regel:

$$\frac{\begin{array}{l} \mu \text{ ist ein Typ zum Typkonstruktor } D \\ A \vdash e : \mu \\ A \cup \{x_{1,1} : \tau_{1,1} \dots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash (c_1 \ x_{1,1} \dots x_{1,n(1)}) : \mu \\ \dots \\ A \cup \{x_{m,1} : \tau_{m,1} \dots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash (c_m \ x_{m,1} \dots x_{m,n(m)}) : \mu \\ A \cup \{x_{1,1} : \tau_{1,1} \dots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash e_1 : \tau \\ \dots \\ A \cup \{x_{m,1} : \tau_{m,1} \dots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash e_m : \tau \end{array}}{A \vdash (\text{case}_D \ e \ \{(c_1 \ x_{1,1} \dots x_{1,n(1)}) \rightarrow e_1; \dots; (c_m \ x_{m,1} \dots x_{m,n(m)}) \rightarrow e_m\}) : \tau}$$

Mit diesen Regeln kann man bereits Typen für nicht rekursiv definierte Superkombinatoren erschließen. Man kann oft für Superkombinatoren polymorphe Typen angeben, die alle herleitbaren Grundtypen repräsentieren. Wir geben dafür jedoch hier keine Regeln an, da das im Typsystem für Typen mit Typvariablen besser und verständlicher gemacht werden kann.

Die Typregel für Abstraktionen und die **case**-Regel verändern die Menge der Typannahmen, indem die Annahmen für gebundene Variablen gelöscht werden.

Definition 4.2.1 *Ein Term t ist (bzgl des Grundtypensystem) wohlgetypt, wenn man für t mindestens einen Typ herleiten kann, D.h. $A \vdash t$ soll herleitbar sein. Hierbei darf A nur Annahmen zu Variablen enthalten, die frei in t vorkommen.*

Das ist offenbar nur dann möglich, wenn auch alle Unterausdrücke wohlgetypt sind, d.h. einen Typ haben.

Beispiel 4.2.2 *Wir berechnen den Typ der Abstraktion $Id := \lambda x. x$:*

$$\frac{\{x : \tau\} \vdash x : \tau}{\emptyset \vdash \lambda x. x : \tau \rightarrow \tau}$$

wobei τ ein (beliebiger) Grundtyp ist.

D.h. man kann für `Id` alle Typen der Form $\{\tau \rightarrow \tau\}$ herleiten. Das ist gerade die Menge der Instanzen von $\alpha \rightarrow \alpha$.

Beispiel 4.2.3 `unit-list x = Cons x Nil`

Sei $A = \{x : \tau, \forall(\text{Nil} : [\beta]), \text{Cons} : \forall(\alpha \rightarrow [\alpha] \rightarrow [\alpha])\}$ die Menge der Annahmen. Beachte, dass $[\beta]$ hier für die Menge aller Grundinstanzen steht, entsprechend bei `Cons`. Hierbei ist τ ein fester Grundtyp.

Aus dem Typ von `Cons` ergibt sich, dass

$$A \vdash (\text{Cons } x) : [\tau] \rightarrow [\tau]$$

d.h.: `(Cons x)` hat den Typ $[\tau] \rightarrow [\tau]$. Weitere Anwendung auf `Nil` erfordert, dass der Typ des Arguments und der erforderliche Typ des Funktionsausdrucks übereinstimmen. Da $[\tau]$ eine Instanz von $[\beta]$ ist, erhalten wir

$$A \vdash (\text{Cons } x \text{ Nil}) : [\tau]$$

D.h. für alle Grundtypen τ gilt: `unit-list`: $\tau \rightarrow [\tau]$. Da das für alle Grundtypen τ gilt, ist der Typ von `unit-list` somit eine Menge von Grundtypen, die man mit dem polymorphen Typ $\alpha \rightarrow [\alpha]$ beschreiben kann.

Beispiel 4.2.4 Die Funktionen `twice` sei definiert als: `twice f x = f (f x)`

Die Typannahme sei $A = \{x : \tau_1, f : \tau_2\}$.

Der Typ von `(f x)` muss existieren: deshalb muss $f : \tau_2 = \tau_1 \rightarrow \tau_3$ sein, wobei τ_3 noch unbestimmt ist, aber der Typ von `(f x)` muss existieren: $(f x) : \tau_3$. Dies bedeutet eigentlich, dass nur Annahmen folgender Form sinnvoll sind: $A' = \{x : \tau_1, f : \tau_1 \rightarrow \tau_3\}$.

Der Typ von `f (f x)` muss existieren: Wir haben angenommen, dass Argumentvariablen nur einen einzigen Typ haben, der in die Typberechnung eingeht: Dann ist $\tau_1 = \tau_3$ und $f (f x) : \tau_1$. Da wir die möglichen Annahmen weiter eingeschränkt haben, gilt jetzt: $A'' = \{x : \tau_1, f : \tau_1 \rightarrow \tau_1\}$ und daraus lässt sich jetzt `f (f x)`: τ_1 herleiten. Damit erhalten wir als Typ von `twice` mit der Superkombinatorregel: $(\tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$. Da dies ohne Annahmen über `twice` hergeleitet werden konnte, und auch für alle Grundtypen τ hergeleitet werden kann, kann man als Typ von `twice` $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ angeben.

Betrachtet man die Annahmen genauer, wird klar, dass man statt einer unendlichen Menge von verschiedenen Annahmen für die freien Variablen, wobei für jede Kombination dieser Annahmen jeweils eine Typherleitung gemacht werden muss, diese auch mit Typvariablen versehen kann.

Die Annahme, dass man für die Eingabeargumente nur einen Grundtyp zulässt (statt einer Menge), ist wesentlich für die Einfachheit des Typsystems. Das ist die sogenannte *Monomorphie-Annahme*, die im allgemeineren Typsystem ebenfalls verwendet wird (allerdings in etwas anderer Form). Man kann sich auch Typsysteme vorstellen, die für Funktionen als Argumente verschiedene

Typinstanzen dieser Funktionen im Rumpf verwenden. Als Beispiel wollen wir die Funktion `selfapply` betrachten, deren Rumpf als Unterausdruck in der Definition des Fixpunktkombinators Y vorkommt:

Beispiel 4.2.5 *Berechne den Typ von `selfapply x = x x` nach zwei verschiedenen Methoden.*

1. *Argumente dürfen nur mit einem Typ angewendet werden (sogenannte Monomorphie-Einschränkung):*

Annahme: $\{\text{selfapply} : \alpha, x : \tau\}$

$(x\ x)$ hat einen Typ, deshalb gilt $\tau = \tau_1 \rightarrow \tau_2$ und $\tau_1 = \tau$. Dies ergibt einen Fehler, denn es gibt keine Grundtypen, die diese Annahme erfüllen. Aber: es gibt Argumente, so dass $(\text{selfapply } x)$ vernünftig ist, Z.B: $\text{selfapply } I \rightarrow (I\ I) \rightarrow I$.

2. *Argumente dürfen eine Menge von Typen haben; das geht über die bisherigen Regeln hinaus. Wir rechnen mal, als ob die Regeln entsprechend angepasst wären:*

$\{\text{selfapply} : S; x : T\}$

$(x\ x)$ muss einen Typ haben $\Rightarrow T$ muss folgende Eigenschaft haben:

(\ddagger) Es gibt Typen $\tau_1 \in T$ und $\tau_1 \rightarrow \tau_2 \in T$.

Der von T abgeleitete Typ von $(x\ x)$ ist dann: $T_1 = \{\tau_2 \mid \text{Es gibt } \tau_1 \rightarrow \tau_2 \in T \text{ und } \tau_1 \in T\}$

Die Typisierung von `selfapply` kann dann nicht mehr als Menge von Typen beschrieben werden, die Instanz eines polymorphen Typausdrucks ist, sondern nur noch als komplexere Bedingung an Mengen von Grundtypen.

Bemerkung Die Typisierung beschreiben wir für die Sprache KFPTS, damit wir möglichst viele Ausdrücke typen können, und da es in KFP Ausdrücke gibt, die wir benötigen, z.B. Y , die aber keinen polymorphen Typ haben. In KFPTS kann man durch die rekursive Definitionsmöglichkeit von Superkombinatoren dieses Problem verdecken, so dass man auch für rekursive Superkombinatoren einen polymorphen Typ erhält.

4.3 Typen für rekursive Superkombinatoren

Unser bisheriges Typsystem kann noch keinen Typ für rekursiv definierte Superkombinatoren erschließen, da man den Typ eines rekursiven Superkombinators bereits in die Typannahmen A aufnehmen muss.

Definition 4.3.1 (*Fixpunkt, konsistente Annahme*) *Sei A eine Menge von (nichtleeren) Annahmen über alle Konstruktoren und definierte Superkombinatoren sc_i , wobei $A = sc_1 : S_1, \dots, sc_n : S_n$. Wenn für alle Superkombinatoren $sc_j, j = 1, \dots, n$, für alle $\tau_i \in S_j : A \vdash sc_j : \tau_i$, wobei die letzte angewendete Regel der Herleitung jeweils die Superkombinatorregel ist, dann ist A eine konsistente Typannahme.*

Diese Bedingung kann man auch schreiben als:

$$\forall i \forall (\tau_1 \rightarrow \dots \rightarrow \tau_{n+1}) \in S_i : A \cup x_1 : \tau_1, \dots, x_n : \tau_n \vdash e_i : \tau_{n+1}$$

wenn $sc_i x_1 \dots x_{n_i} = e_i$ die jeweilige Definition von sc_i ist.

Die Konsistenz bedeutet, dass man mindestens alle Typen von Superkombinatoren, die in den Annahmen drin stecken, auch aus diesen Annahmen herleiten kann.

Definition 4.3.2 *Ein Programm ist wohlgetypt, wenn es eine konsistente Typannahme gibt.*

Wenn es eine allgemeinste, konsistente Typannahme gibt, dann nennt man die entsprechenden Angaben $sc_i : S_i$ auch die allgemeinsten Typen der Superkombinatoren.

Wünschenswert ist, dass dieser allgemeinste Typ sich endlich ausdrücken lässt. Diese sind i.a. polymorphe Typen mit Typvariablen.

Beispiel 4.3.3 *Typisierung eines rekursiven Kombinator.*

`repeat x = Cons x (repeat x)`

Wir nehmen als Typannahme:

$$A = \{\text{repeat} : \forall \alpha, \text{Cons} : \forall (\beta \rightarrow [\beta] \rightarrow [\beta]), x : \tau\}$$

Dann ergibt sich: (repeat x) ist getypt, deshalb hat repeat den Typ $\tau \rightarrow \alpha'$

(Cons x (repeat x)) ist getypt, deshalb $[\tau] = \alpha'$

und damit (Cons x (repeat x)): $[\tau]$.

Das ergibt zunächst $\tau \rightarrow [\tau]$.

Das Ergebnis in der Typschreibweise mit neuen Variablen: `repeat: $\alpha' \rightarrow [\alpha']$`

Wir testen die neuen Annahmen auf Konsistenz:

$$A_1 = \{\text{repeat} : \forall (\alpha \rightarrow [\alpha]), \text{Cons} : \beta \rightarrow [\beta] \rightarrow [\beta]\}$$

(repeat x): $[\tau]$

(Cons x (repeat x)): $[\alpha]$ und $\beta = \alpha = \tau$

Der Typ ist damit wieder $\tau \rightarrow [\tau]$ für alle τ , wobei dies mit der Superkombinatorregel herleitbar war. Damit ist A_1 eine konsistente Typannahme.

Bevor wir zu einem Typalgorithmus kommen, der allgemeinste Typen berechnet, argumentieren wir, dass das Grundtypsystem garantiert, dass von der Auswertung keine dynamischen Typfehler gemacht werden können, wenn die Programme wohlgetypt sind.

Lemma 4.3.4 *Für direkt dynamisch ungetypte Ausdrücke lässt sich kein Typ herleiten.*

Begründung. Es gibt folgende Fälle, in denen ein Ausdruck dynamisch ungetypt ist: Wir können der Einfachheit halber annehmen, dass alle Unterausdrücke einen Typ haben.

1. (`case_T {c ...; ... }`) und c ist kein Konstruktor zum Typ. In dem Fall kann für den `case`-Unterausdruck keine Typregel verwendet werden, da der Typ des zu untersuchenden Ausdrucks und der Pattern verschieden ist.
2. (`case_T e ...`) und e ist in WHNF, aber nicht von der richtigen Form. Wenn e eine Kombinator-WHNF mit zuwenig Argumenten ist, dann ist der Typ ebenfalls von der Form $\alpha \rightarrow \beta$.
Das gleich gilt für eine WHNF, die ein Abstraktion ist.
3. e ist eine Anwendung einer Konstruktor-Anwendung auf ein Argument: $((c\ t_1 \dots t_n)\ t)$. Da eine Konstruktor-Anwendung einen Typ der Form $(K\ r_1 \dots r_m)$ hat, kann dies kein Funktionstyp sein, und somit ist $(c\ t_1 \dots t_n)\ t_{n+1}$ nicht typisierbar. \square

Lemma 4.3.5 *Angenommen, wir haben eine konsistente Typannahme für alle Superkombinatoren. Dann gilt: Wenn $t \rightarrow t'$ eine Reduktion ist und man für t den Grundtyp τ herleiten kann, dann kann man auch für t' den Typ τ herleiten.*

Beweis. Wir betrachten die Fälle der verschiedenen Reduktionen: Es genügt, einen Redex zu betrachten. Bei Betrachtung der Oberterme genügt es, sich davon zu überzeugen, dass nur die Typen, nicht aber die Struktur der Terme eine Rolle spielt.

1. $sc\ t_1 \dots t_n \rightarrow e[t_1/x_1 \dots t_n/x_n]$ und $sc\ x_1 \dots x_n = e$ sei die Definition für sc . Für den Term $sc\ t_1 \dots t_n$ haben wir unter einer Annahme A die Typen $t_1 : \tau_1, \dots, t_n : \tau_n$, hergeleitet und mit der Annahme $sc : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \in A$ dann auch $(sc\ t_1 \dots t_n) : \tau$.

Da die Typannahmen konsistent sind, können wir auch diesen Typ für sc mittels der Superkombinatorregel herleiten und somit auch für den Rumpf e . Da die Typisierungsregeln nur den Typ der Terme, nicht aber deren innere Struktur beachten, erhalten wir, dass für $e[t_1/x_1 \dots t_n/x_n]$ auch der Typ τ herleitbar ist.

2. $(\lambda x.st) \rightarrow s[t/x]$. Geht analog zur Superkombinatorreduktion.
- 3.

$$\text{case}_A (c_i\ t_1 \dots t_{n(i)}) \quad \left\{ \begin{array}{l} (c_1\ x_{1,1}, \dots, x_{1,n(1)}) \rightarrow \text{exp}_1; \dots \\ (c_m\ x_{m,1}, \dots, x_{m,n(m)}) \rightarrow \text{exp}_m \end{array} \right\}$$

reduziert zu:

$$\text{exp}_i[t_1/x_{i,1}, \dots, t_{n(i)}/x_{i,n(i)}]$$

Die Typregel für `case` erzwingt, dass die Typen für alle Ausdrücke $(c_i\ t_1 \dots t_{n(i)})$ und alle Pattern $(c_j\ x_{j,1} \dots x_{j,n(j)})$ gleich sind und als obersten Typkonstruktor K_A haben. Wegen der Konvention über die Typvariablen in der Definition von Datenkonstruktoren zu einem Typ gilt,

dass die Typen der Argumente von $(c_i t_1 \dots t_n(i))$ und $(c_i x_{i,1}, \dots x_{i,n(i)})$ sich aus dem gemeinsamen Typ wieder eindeutig bestimmen lassen, so dass für alle j die Typen von t_j und $x_{i,j}$ die gleichen sind. Somit stimmt der herleitbare Typ des reduzierten Ausdrucks mit dem des `case` überein.

Damit ist jeder Typ des Ausdruck vorher auch noch Typ des Ausdrucks nach Reduktion, insbesondere, wenn der Ausdruck in WHNF (oder Normalform) ist. Außerdem sind alle Unterausdrücke jedes Terms innerhalb der Reduktion getypt. \square

Lemma 4.3.6 *Für dynamisch ungetypte Ausdrücke lässt sich kein Typ herleiten.*

Beweis. Folgt aus Lemma 4.3.4 und aus Lemma 4.3.5. Erinnerung: dynamisch ungetypte Ausdrücke sind solche, die mit evtl. mehreren Normalordnungsreduktion auf einen direkt ungetypten Ausdruck reduzieren. \square

Damit gilt der Satz:

Satz 4.3.7 : *(Milner-) Wohlgetypte Programme machen keine dynamischen Typfehler.*

Bemerkung: Dieser Satz ist etwas allgemeiner als für das Milnersche Typsystem, da in Milners System eine etwas eingeschränkere Typberechnungsmethode verwendet wird. D.h. Milners Satz sagt aus, dass Milner-wohlgetypte Programme keine Typfehler machen.

4.4 Berechnung von allgemeinsten und polymorphen Typen von Ausdrücken. Das iterative Verfahren

Wir erlauben in diesem Abschnitt nur eine Annahme für jeden Superkombinator und Konstruktor, wobei diese Annahme sich jeweils als polymorpher Typ ausdrücken lassen muss.

Als Notation verwenden wir jetzt:

τ, μ stehen für Typen, die Typvariablen enthalten dürfen.

Typvariablen sind mit α bezeichnet.

Die Gleichheit von polymorphen Typen lässt sich leicht testen, denn für polymorphe Typen mit Variablen gilt:

τ_1 repräsentiert die gleiche Menge von Grundinstanzen wie τ_2 , wenn τ_1 und τ_2 bis auf Umbenennung von Variablen gleich sind.

Analog repräsentiert τ_2 mehr Typen als τ_1 , wenn $\sigma(\tau_2) = \tau_1$ gilt für eine Typsubstitution σ (mit Variablen). Dieser Matching-Test ist in linearer Zeit durchführbar.

Die Definitionen sind anzupassen, Z.B.:

Definition 4.4.1 *Ein Programm ist wohlgetypt, wenn es eine konsistente Typannahme für alle Superkombinatoren gibt und der auszuwertende Ausdruck einen polymorphen Typ hat bezüglich dieser Typannahmen.*

Wenn es eine allgemeinste, konsistente Typannahme gibt, dann nennt man die entsprechenden Angaben $sc_i : \tau_i$ auch die allgemeinsten Typen der Superkombinatoren.

Superkombinatoren, die frei von Rekursion sind, kann man mit dem Typsystem jetzt schon behandeln. Was noch fehlt, ist eine systematische Vorgehensweise bei der Berechnung eines einzigen, polymorphen Typs.

Bei rekursiven Superkombinatoren ergibt sich das Problem, dass man deren Typ schon in der Voraussetzung benötigt. Man möchte natürlich möglichst den allgemeinsten Typ der rekursiven Superkombinatoren berechnen, da dieser am informativsten ist und die besten Verwendungsmöglichkeiten erlaubt. Dazu benötigt man eine Fixpunktiteration und der Einfachheit halber eine systematische Behandlung der Darstellung mittels polymorpher Typen; bei Typklassen muss man die Darstellung um die Typklassen-Bedingungen erweitern.

Macht man zunächst die allgemeinsten Annahmen, und verkleinert in einer Fixpunktiteration diese Annahmen solange, bis die Eigenschaft der Konsistenz erfüllt ist, dann hat man ein Berechnungsverfahren für den größten Fixpunkt. Diese Fixpunktiteration terminiert möglicherweise nicht, was bedeutet, dass es keinen allgemeinsten polymorphen Typ für den betrachteten Superkombinator gibt, wie wir noch sehen werden.

Im folgenden beschreiben wir einen Typcheck-Algorithmus.

Wir schreiben die Annahmen, die Typvariablen enthalten, wie bereits oben, mit einem Präfix \forall , um anzudeuten, dass die Typvariablen unter einem Quantor stehen und somit deren Name keine Rolle spielt. Im Algorithmus hat das die Bedeutung, dass diese Typen kopiert werden dürfen.

Wir fügen eine bessere Behandlung der Gleichungen zwischen Typen hinzu, d.h. der Beschränkungen der Annahmen, durch. Dazu benötigt man Unifikation (Gleichungslösen) von polymorphen Typen, um mit diesen Beschränkungen umzugehen.

4.4.1 Unifikation

Unifikation operiert auf einer Menge von Gleichungen $s \doteq t$ zwischen Termen, wobei wir \doteq als symmetrisch betrachten. Die Regeln verändern die Menge der Gleichungen. Sie sind nichtdeterministisch formuliert, und können in irgendeiner Reihenfolge angewendet werden (don't care non-determinism).

Sei G eine Multimenge von Gleichungen.

1.
$$\frac{\{(c\ t_1 \dots t_n) \doteq (c\ s_1 \dots s_n)\} \cup G}{\{s_1 \doteq t_1, \dots, s_n \doteq t_n\} \cup G}$$
2.
$$\frac{\{\alpha \doteq \tau\} \cup G}{\{\alpha \doteq \tau\} \cup G[\tau/\alpha]}$$

wenn $\tau \neq \alpha$ und α kommt in τ nicht vor und α kommt in G vor.
3.
$$\frac{\{\alpha \doteq \alpha\} \cup G}{G}.$$
4.
$$\frac{\{(c \dots) \doteq (d \dots)\} \cup G}{FAIL}$$

wenn c und d verschiedene Typkonstruktoren sind.
5.
$$\frac{\{\alpha \doteq \tau\} \cup G}{Fail}$$

wenn α Typvariable, $\tau \neq \alpha$ und α kommt in τ vor (Occurs-Check, erscheint als "infinite type" in Fehlermeldungen).

Der 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.

Bei naiver (ungünstiger) Implementierung ist der Algorithmus zeit-exponentiell. Er kann aber quadratisch, in dieser Variante als $O(n * \log(n))$ und in einer anderen Variante sogar linear implementiert werden.

Wir sagen, dass ein Gleichungssystem *gelöst* ist, wenn es von der Form $\{\alpha_1 \doteq \tau_1, \dots, \alpha_n \doteq \tau_n\}$ ist, und die Variablen α_i alle verschieden sind und nicht in den τ_j vorkommen. Wenn ein Gleichungssystem gelöst ist, kann man eine Substitution ablesen: Aus $\{\alpha_1 \doteq \tau_1, \dots, \alpha_n \doteq \tau_n\}$ kann man $\{\alpha_1 \rightarrow \tau_1, \dots, \alpha_n \rightarrow \tau_n\}$ ablesen.

4.5 Der iterative Typisierungsalgorithmus

Er soll die bestmöglichen (d.h. allgemeinsten) polymorphen Typen für die Superkombinatoren berechnen, und ebenso den Typ aller Unterausdrücke und natürlich auch des auszuwertenden Ausdrucks.

Normalerweise macht man zuerst eine Abhängigkeitsanalyse zwischen den Superkombinatoren und startet die Typisierung von unten, d.h. mit den Superkombinatoren, die von den wenigsten anderen abhängen. Danach kann man diese berechneten Typen in der Typisierung mitverwenden. Wir betrachten die Situation, in der schon einige Superkombinatoren Typen haben und von anderen verwendet werden, und wollen die Typen der nächsten Schicht berechnen. Die schon berechneten Typen der SK bezeichnen wir mit $\forall(\delta)$, um anzudeuten, dass die Typvariablen in δ umbenannt werden können.

Er startet mit

- vorgegebenen Typ-Annahmen über die Konstruktoren,
- bereits berechneten Typen einiger Superkombinatoren
- den vorgegebenen Typen der Variablen und
- mit $\forall\alpha$ als Typen der Superkombinatoren (deren Typ zu bestimmen ist)

Danach werden die Rümpfe der Superkombinatoren getypt, wobei folgende Regeln verwendet werden. Diese Regeln sind die Regeln für Grundtypen, erweitert um Unifikation. Wir fügen zu den hergeleiteten Typen noch die notwendigen Gleichungen zwischen den Typen hinzu.

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

E_σ bedeutet, dass das Gleichungssystem in gelöster Form ist und der Substitution σ entspricht.

Eine implizite Annahme ist, dass verschachtelte **cases** keine Variablen mehrfach benutzen, sondern stets neue Variablen einführen.

Axiom für Variablen

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

Axiom für Konstanten

$A \cup \{c : \forall\delta\} \vdash \{c : \tau', \emptyset\}$ c ist ein Konstruktor oder ein bereits getypter Superkombinator. wobei τ' eine umbenannte Version des Typs δ ist mit neuen Typvariablen.

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, wenn kein FAIL auftritt.

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

wenn bei Unifikation von E ein FAIL auftritt

case-Regel:

$$\begin{array}{l}
 A \vdash e : \mu_0, F_0 \\
 A \cup \{x_{1,1} : \tau_{1,1} \dots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash (c_1 x_{1,1} \dots x_{1,n(1)}) : \mu_1, F_1 \\
 \dots \\
 A \cup \{x_{m,1} : \tau_{m,1} \dots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash (c_m x_{m,1} \dots x_{m,n(m)}) : \mu_m, F_m \\
 A \cup \{x_{1,1} : \tau_{1,1} \dots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash e_1 : \tau_1, E_1 \\
 \dots \\
 A \cup \{x_{m,1} : \tau_{m,1} \dots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash e_m : \tau_m, E_m \\
 \hline
 A \vdash (\text{case}_T e \{(c_1 x_{1,1} \dots x_{1,n(1)}) \rightarrow e_1; \dots; (c_m x_{m,1} \dots x_{m,n(m)}) \rightarrow e_m\}) : \tau, \\
 F_0 \cup F_1 \cup \dots \cup F_m \cup \{\mu_0 \doteq \mu_1, \dots, \mu_0 \doteq \mu_m\} \cup \\
 E_1 \cup \dots \cup E_m \cup \{\tau \doteq \tau_1, \dots, \tau \doteq \tau_m\} \\
 \tau \text{ ist eine neue Typvariable}
 \end{array}$$

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:

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

wobei σ die Substitution ist, die durch Unifikationsregeln aus E berechnet wird; hierbei darf kein FAIL auftreten.

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.

4.5.1 Typberechnung mittels Iteration:

Man startet mit Annahmen $A_0 = \{sc_i : \alpha_i \mid i = 1, \dots, n\}$ und berechnet mit der Superkombinatorregel neue Typen $sc_i : \tau'_i$. Danach iteriert man diesen Zyklus, indem man mit den berechneten Typen als neue Annahmen $A' = \{sc_i : \tau'_i \mid i = 1, \dots, n\}$ weitermacht.

Die Iteration terminiert, wenn die Typannahmen konsistent sind.

Es gelten folgende Aussagen zum Typsystem, wobei wir allerdings auf einen Beweis in diesem Skript verzichten:

- Die berechneten Typen für Superkombinatoren nach einem Iterationsschritt sind eindeutig bis auf Umbenennung von Variablen. Deshalb sind auch die berechneten Typen der Superkombinatoren im Falle des Terminierens eindeutig.
- Die erreichten Typannahmen werden in jedem Schritt spezieller, so dass bei dieser Iteration die Abbruchbedingung nur erreicht werden kann, wenn die Typannahmen pro Superkombinator bis auf Umbenennung gleich sind.
- Wenn das Verfahren nicht terminiert, dann gibt es keinen polymorphen Typ für den rekursiven Superkombinator, bzw. die verschränkt rekursiven Superkombinatoren.

Zum Beweis der zweiten Aussage benötigt man eine Monotonie-Eigenschaft: Wenn die Annahmen für die Superkombinatoren spezieller sind, dann sind auch die hergeleiteten Typen spezieller.

Die dritte Aussage gilt, da Nichtterminierung bedeutet, dass die beteiligten polymorphen Typen immer größer werden. Da das Verfahren aber den größten Fixpunkt berechnet (im Sinne der Mengen der Instanzen), kann es keinen Grundtyp τ für den Superkombinator geben, denn τ kann nicht Instanz eines Typs mit Variablen sein, der syntaktisch größer ist (mehr Zeichen hat).

Die Überprüfung auf Konsistenz macht man mit einem Subsumtionscheck für polymorphe Typen. τ_1 ist allgemeiner als τ_2 , wenn man eine Substitution σ findet mit $\sigma(\tau_1) = \tau_2$. Mit Unifikation ist das leicht zu berechnen, indem man die Variablen in τ_1 umbenennt, dann die Variablen in τ_2 zu Konstanten erklärt und danach unifiziert.

Beispiel 4.5.1 *Typ von length:*

`length as = case_list as {Nil -> 0; (Cons x xs) -> (1 + length xs)}`

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

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

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

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

Verwendung der case-Regel:

Es gilt $0 : \text{Int}$ *und* $\text{Nil} : [\beta_1]$

`cons`: $\beta_2 \rightarrow [\beta_2] \rightarrow [\beta_2]$

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

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

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

Da die meisten Variablen irrelevant sind, genügt es, bis auf die Variablen τ und τ_1 , nur die Lösbarkeit der Gleichungen festzustellen. Das Gleichungssystem nach Unifikation sieht so aus: $\{\tau \doteq \text{Int}, \tau_1 \doteq \tau_3 \doteq [\tau_2], \tau_4 = \text{Int}\}$

Die berechnete Annahme für den Typ von length ist dann: $\text{length} : [\tau_2] \rightarrow \text{Int}$.

Danach ist die Annahme für length: $\forall. [\tau_2] \rightarrow \text{Int}$.

Eine weitere Iteration zeigt die Konsistenz dieser Annahme.

Beispiel 4.5.2 $g\ x = (\text{Cons } 1\ (g\ (g\ 'c')))$

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

$(g\ 'c')$ mit $g : \alpha_1$ ergibt zunächst $\alpha_1 = \alpha_2 \rightarrow \alpha_3$ und $\alpha_2 = \text{Char}$.

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

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

Dies ergibt als nächste Typannahme für $g : \forall \alpha. \alpha \rightarrow [\text{Int}]$. Verwendet man diese, so erhält man, dass dies der richtige Typ ist.

Beachte: In Haskell ist diese Funktion nicht typisierbar.

Haskell hat eine weitergehende Monomorphie-Beschränkung: rekursiv definierte Funktionen dürfen im rekursiven Skopus **nur mit einem Typ** verwendet werden. Dies ist bei obigem Beispiel verletzt.

Beispiel 4.5.3 Betrachte $g\ x = g\ 1$:

$g : \alpha \rightarrow \beta, x : \tau$ ergibt: $g\ 1 : \beta$. D.h. $g : \alpha \rightarrow \beta$.

Dieser Typ bedeutet, dass das Resultat alle Typen hat. Das kann immer so interpretiert werden, dass die Auswertung von $(g\ x)$ zu einer WHNF nicht terminiert. Ein Objekt beliebigen Typs (bzw. eines das alle Typen hat), kann nicht in WHNF sein, denn ein Konstruktor als oberstes Funktionssymbol impliziert, dass der Typ des Objektes mit dem entsprechenden Typkonstruktor anfängt. Ein Funktionsausdruck als WHNF hätte ein \rightarrow als obersten Typkonstruktor. Es bleibt nur ein Objekt ohne WHNF, d.h. ein nicht-terminierender Ausdruck. D.h. In gewissen Fällen liefert der Type-Checker Nichtterminierungsinformation. Aber nicht jeder nichtterminierende Ausdruck hat einen solchen allgemeinen Typ.

4.5.2 Frage: Benötigt das Verfahren tatsächlich mehrere Iterationen?

JA:

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

- 1. Iteration
 - Annahme $\{x : \tau_1, g : \forall \alpha\}$
 - $(g\ 'c') : \alpha_1 \doteq \alpha_2 \rightarrow \alpha_3$ und $\alpha_2 \doteq \text{Char}$
 - $g(g\ 'c')$
 - Annahme: $g : \beta : \beta \doteq \beta_1 \rightarrow \beta_2, \beta_1 \doteq \alpha_3$,
 - $g(g\ 'c') : \beta_2$
 - $(\text{Cons } x\ (g(g\ 'c')))$: ergibt $\beta_2 = [\tau_1]$
 - Gesamttyp von $g : \tau_1 \rightarrow [\tau_1]$
- 2. Iteration
 - Neue Annahme: $\{x : \tau_1, g : \forall \alpha \rightarrow [\alpha]\}$
 - $(g\ 'c') : \alpha_1 \rightarrow [\alpha_1]$ und $\alpha_1 = \text{Char}$
 - $g(g\ 'c')$ ergibt $g : \beta_1 \rightarrow [\beta_1], \beta_1 = [\text{Char}]$
 - $(\text{Cons } x\ (g(g\ 'c')))$: ergibt $\tau_1 = [\text{Char}]$ und als Typ
 - $(\text{Cons } x\ (g(g\ 'c')))$: $[[\text{Char}]]$

- 3. Iteration
 Neue Annahme: $\{x : \tau_1, g : [\text{Char}] \rightarrow [[\text{Char}]]\}$
 $(g \text{ 'c'}) : [\text{Char}] = \text{Char}$ liefert FAIL. D.h. der Ausdruck ist nicht typisierbar.

g hat also keinen Typ im iterativen Typsystem.

4.5.3 Frage: Terminiert das Verfahren?

Die Antwort ist NEIN:

Betrachte $g \ x = \text{Cons } x \ (g \ (g \ \text{bot}))$, wobei bot nichtterminierender Superkombinator mit Typ $\forall \gamma$.

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 Gesamttyp von g mittels der Superkombinatorregel ist: $\tau_1 \rightarrow [\tau_1]$

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

$(g \ \text{bot}) :: [\alpha_1]$ 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 $g : [\alpha_1] \rightarrow [[\alpha_1]]$

Wir berechnen dies mit einer allgemeineren Annahme, wobei wir die n -fache Anwendung des Listen-Typ-Konstruktors mit $[\cdot]^n$ notieren.

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

$(g \ \text{bot})$ 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 $\tau_1 \doteq [\alpha_1]^{n+1}$. Der Resultattyp ist $\text{Cons } x \ (g \ (g \ \text{bot})) : [\alpha_1]^{n+2}$

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

Da der Typ immer tiefer geschachtelt wird, terminiert dieses Verfahren nicht.

Beispiel 4.5.4 : Ein weiteres Beispiel zur Nichtterminierung, das leichter zu überschauen ist, aber zwei rekursive Superkombinatoren benötigt:

Man kann konstante Superkombinatoren A, B definieren:

$A = [B]$

$B = [A]$

Man wird mit dem hier angegebenen Verfahren iterativ immer tiefer geschachtelte Typen bekommen. Denn die Annahme $A : \tau_1, B : \tau_2$ mit beliebigen Typtermen

τ_1 und τ_2 liefert danach $A : [\tau_2], B : [\tau_1]$. Typkonsistenz kann nicht erfüllt sein, da wir dann eine zyklische Abhängigkeit hätten: $\tau_1 \subseteq [\tau_2]$, und $\tau_2 \subseteq [\tau_1]$ dies kann nur sein, wenn $\tau_1 \subseteq [[\tau_1]]$, was niemals stimmt. Damit kann das iterative Typinferenzverfahren nicht terminieren.

Nochmal zur Erinnerung:

- 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.
Begründung: Die Iteration muss immer tiefer geschachtelte Typen liefern, sonst wird ein Fixpunkt gefunden.
- 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 aber: (das war damals auch für die Fachleute überraschend:)

Satz 4.5.5 *Es gilt: (iterative) Typisierung ist unentscheidbar.*

Das Kern des Problems ist die sogenannte 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)

4.5.4 Terminierung:

Die Terminierung kann auf Kosten der Berechnung des allgemeinsten Typs erzwungen werden (bzw. auf die Gefahr hin, dass ein Ausdruck als ungetypt abgelehnt wird), wenn man nach einigen Iterationen einen "Milner"-Iterationsschritt anschließt:

Erlaube nur noch Instanzen des Typs in der Typ-Annahme ohne Umbenennung der Typvariablen. Anschließend unifiziere den berechneten Typ eines Superkombinators mit dem Typ in der Annahme.

Danach ist dann nämlich die Bedingung der konsistenten Typannahme automatisch erfüllt.

Beispiel 4.5.6 $g\ x = \text{Cons } x\ (g\ (g\ \text{bot}))$

Das iterative Typisierungsverfahren terminiert nicht für dieses Beispiel. Das Verbot von Kopien des Typs des aktuell zu typisierenden Superkombinators ergibt:

$\{x : \tau_1, g : \forall \alpha \rightarrow [\alpha], \text{bot} : \gamma\}$
 $(g \text{ bot}) : \alpha \rightarrow [\alpha]$ und $\alpha \doteq \gamma_1$
 $g(g \ c)$ muss getypt sein $\Rightarrow g : \alpha \rightarrow [\alpha], \alpha \doteq [\alpha]$:
 Die letzte Gleichung $\alpha \doteq [\alpha]$ ergibt einen Fehler bei der Unifikation.

Aussage 4.5.7 Werden für die zu typisierenden Superkombinatoren keine umbenannten Kopien von Typen erlaubt, dann terminiert das Typisierungsverfahren im ersten Iterationsschritt.

Begründung. Das Verfahren startet mit der Annahme $A = A_0 \cup \{sc_i : \alpha_i\}$, wobei A_0 die Konstruktoren und bereits typisierten Superkombinatoren enthält und sc_i die zu typisierenden Superkombinatoren sind. Da keine Kopien des Typs von Superkombinatoren sc_i gemacht werden, können alle Anforderungen an den Typ der Superkombinatoren sc_i berechnet und danach unifiziert werden. Am Ende ergibt sich entweder ein FAIL beim Unifizieren, oder die Unifikation ist erfolgreich und der für sc_i angenommene Typ ist gleich dem berechneten Typ.

4.5.5 Typisierungsverfahren von Milner

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. Dann sei \preceq^* der reflexiv-transitive Abschluß von \preceq . Sei \approx die Äquivalenzrelation zu \preceq^* , d.h.

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

Der Algorithmus

Das Typisierungsverfahren von Milner geht so vor wie das iterative Verfahren, mit folgenden Unterschieden,

- Superkombinatoren werden nach der Abhängigkeitsanalyse von unten nach oben typisiert: In jedem Typcheck-Schritt wird eine minimale \approx -Äquivalenzklasse getypt (deren Superkombinatoren rekursiv voneinander abhängig sind); Die Typen von bereits typisierten SK werden wie die von Konstanten verwendet.
- Von zu typisierenden Superkombinatoren wird nur eine Kopie des Typs erlaubt während dieser Typcheckphase erlaubt. Die Suoperkombinatorregel wird dazu abgewandelt:

Superkombinator-Regel ala Milner

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

Wenn σ Lösung von $E \cup \{\alpha \doteq \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau\}$ ist, wobei $\mathbf{sc} : \alpha$ die Typannahme $\mathbf{sc} \ x_1 \ \dots \ x_n = \mathbf{e}$ die Definition für \mathbf{sc} ist.

Werden mehrere Superkombinstoren gleichzeitig typisiert, muss man in die Unifikation alle Ergebnisse der Superkombinatorregeln eingehen lassen.

- Am Ende eines Typcheckschritts werden durch die Unifikation automatisch die Annahmen und der berechnete Typ pro zu typisierendem Superkombinator gleichgemacht.

Im nächsten Typisierungsschritt wird dann eine minimale Äquivalenzklasse von \approx typisiert.

Diese Vorgehensweise berechnet i.a. allgemeinere Typen als “alle auf einmal“, da für die bereits typisierten SK Kopien der Typen möglich sind.

Das Milner-Verfahren liefert eingeschränktere Typen als das oben beschriebene iterative Verfahren, oder lehnt womöglich einen (iterativ wohl-getypten) Ausdruck als ungetypt ab. Das Verfahren hat aber dafür den Vorzug, dass es terminiert und (gegenüber einem irgendwann abgebrochenen und mit Milner-Schritt beendeten iterativen Verfahren) eindeutige polymorphe Typen liefert. Es wird deshalb in mehreren Programmiersprachen verwendet. Da Programmiersprachen wie Haskell, ML, und Miranda u.a. letrec- Konstrukte haben, ist das Typisierungsverfahren etwas komplizierter als das hier beschriebene, denn es muss lokal rekursiv definierte Funktionen ebenfalls typen können.

Die worst-case Komplexität des Problems “ist eine Funktion Milner-typisierbar?“ ist EXPSPACE-hard. Dies wirkt sich so aus, dass es Funktionen gibt, deren Typ im Milner Typsystem exponentiell viele Typvariablen benötigt. Das Milner Typsystem macht Unterschiede zwischen let-gebundenen und lambda-gebundenen Variablen: let-gebundene dürfen polymorph sein, lambda-gebundene Variablen müssen monomorph sein. Außerdem dürfen rekursiv definierte Superkombinatoren im rekursiven Aufruf nur mit einem Typ benutzt werden.

Wir wollen die guten Eigenschaften des Milner-Typsystems begründen:

Aussage 4.5.8 *Das (Milner-)Typisierungsverfahren terminiert.*

Es genügt, sich klar zu machen, dass ein Typisierungsschritt für eine Äquivalenzklasse von \approx terminiert.

Beispiel 4.5.9 *Milner-Methode: Berechne den Typ der Funktion map.*

```
map f as = case_list as {Nil -> Nil;
                        (x:xs) -> (f x) : (map f xs)}
```

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

Typ-inferenz für `map` ergibt:

1. $\mathbf{map} \ f \ \mathbf{xs} \ \text{getypt} \Rightarrow \mathbf{map} : \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3, \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \doteq \alpha, \alpha_1 \doteq \tau_1, \alpha_2 \doteq \tau_4.$

2. $(f\ x)$ *getypt* $\Rightarrow \tau_1 \doteq \tau_3 \rightarrow \tau_5$
3. $(f\ x) : \text{map } f\ xs$ *getypt*: $\text{Cons}: \tau_6 \rightarrow [\tau_6] \rightarrow [\tau_6], \tau_5 \doteq \tau_6, [\tau_6] \doteq \alpha_3$
4. Nil und $(f\ x) : \text{map } f\ xs$ haben den gleichen Typ: $[\beta_1] \doteq [\tau_6]$
5. $x:xs$ ist *getypt*: $\tau_4 \doteq [\tau_3]$
6. as, Nil und $x:xs$ haben gleichen Typ: $\tau_2 \doteq [\beta_2] \doteq [\tau_3]$
7. Ergebnis ist vom Typ $[\tau_6] \doteq \alpha_3$
8. Unifikation ergibt: $\tau_5 \doteq \tau_6 \doteq \beta_1, \tau_3 \doteq \beta_2, \tau_4 \doteq \tau_2 \doteq [\tau_3] \alpha_1 \doteq \tau_1 \doteq \tau_3 \rightarrow \tau_5, \alpha_2 \doteq \tau_4, \alpha_3 \doteq \tau_2 \doteq [\tau_5]$

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

Beispiel 4.5.10 Der Milner Algorithmus kann Typfehler liefern, obwohl der iterative Algorithmus einen Typ liefert: z.B. für die Funktion $g\ x = 1: g(g\ 'c')$ liefert der Milner-Algorithmus, dass diese nicht wohlgetypt ist, obwohl wir iterativ dafür den Typ: $\alpha \rightarrow [\text{Int}]$ berechnet haben:

```
g:  $\alpha_1 \rightarrow \alpha_2, x: \alpha_1,$ 
g 'c':  $\alpha_2, \alpha_1 = \text{Char}$ 
g (g 'c'):  $\alpha_2; \alpha_2 = \text{Char}$ 
1:g(g 'c') erfordert mit cons:  $\beta \rightarrow [\beta] \rightarrow [\beta]$ 
 $\beta \doteq \text{Int}, [\text{Int}] = [\text{Char}]$ 
```

Das ist nicht unifizierbar.

Ein Test in Haskell liefert: nicht (Haskell-)typisierbar.

Beispiel 4.5.11 Ein weiteres Beispiel zum Unterschied zwischen Milner und iterativer Typisierung: Das Milner-Verfahren kann einen eingeschränkteren Typ liefern als das iterative.

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
g x y = Node 1 (g x y) (g y x)
```

```
g :: forall a t. (Num a) => t -> t -> Tree a
```

Das iterative Verfahren erzwingt keinen Zusammenhang zwischen den beiden Argumenten von g , somit ist der iterativ bestimmte Typ, hier angegeben mit Typklassen:

```
g::Num a => t1 -> t2 -> Tree a|.
```

Eine vereinfachte Vorgehensweise bei der Milner-Typisierung:

Wir beschreiben eine Vereinfachung des Milner-Typalgorithmus, der in einfachen Fällen den polymorphen Typ rekursiv definierter Funktionen ausrechnet, und der auch mit Pattern umgehen kann. Das ist im obigen Verfahren durch eine Übersetzung in einen case-Ausdruck nachvollziehbar.

Bei der Berechnung wird unterschieden zwischen Konstanten und Funktionen, deren Typ schon bekannt ist bzw. berechnet wurde und solchen, deren Typ zu ermitteln ist. Der wichtige Unterschied ist:

Der Typ von Funktionen mit bekanntem Typ darf **umbenannt** verwendet werden, denn der Typ hat schon die Form $\forall a, b, \dots. \tau[a, b, \dots]$.

Das Vorgehen bei der Berechnung des Typs einer einzigen rekursiven Funktion die mittels $f \ x_1 \dots \ x_n = \dots$ definiert wird, kann man etwas vereinfacht so beschreiben:

- Es wird $f :: a$ angenommen, a eine Typvariable. Ebenso wird $x_1 :: b_1 \dots x_n :: b_n$ angenommen, wobei b_1, \dots, b_n Typvariablen. Die gleichen Annahmen machen wir bei mehreren Gleichungen, wobei das gleiche a genommen wird. Der Typ der linken Seite, d.h. $f \ x_1 \dots \ x_n$, wird als $b_{\{n+1\}}$ angenommen. Das ergibt bereits die erste Typgleichung $a = b_1 \rightarrow \dots \ b_n \rightarrow b_{\{n+1\}}$.
Bei der Verwendung von Pattern gehen ersatzweise neue Gleichungen $x_i = p_i$ ein, wobei die Variablen des Pattern ebenfalls als mit neuen Typvariablen getypt angenommen werden.
- Die Typregeln werden verwendet, um den Typ der in der Definition (bzw. Definitionsgleichungen) vorkommenden Ausdrücke zu berechnen. Hierbei wird für jedes Vorkommen von f die Typvariable a genommen, ohne Umbenennung, bzw. die berechnete Instanz. Die Typberechnung instanziiert die freien Typvariablen während der Typberechnung der Unterausdrücke. Bei der Typberechnung und bei der Lösung von Typgleichungen wird der Unifikationsalgorithmus verwendet.
- Man setzt die Typen der linken und rechten Seiten von Definitionsgleichungen gleich, und verwendet dazu den Unifikationsalgorithmus. Am Ende ergibt sich der Typ von f als :
 $\gamma(b_1 \rightarrow \dots \rightarrow b_n \rightarrow b_{\{n+1\}})$, wobei γ die insgesamt berechnete Typsubstitution ist.

Weiteres Beispiel, das auch einen Unterschied zwischen in Haskell vorgegebenen und selbst definierten Funktionen zeigt.

Beispiel 4.5.12

Wir berechnen den Typ der length-Funktion:

```
length []      = 0
length (x:xs) = (1 + length xs)
```

Start mit folgenden Annahmen:

$length :: a, x_1 :: b_1$, Typ der linken Seite; b_2 .

Hier eine Tabelle der Ausdrücke, die getypt werden und die Ergebnisse, wobei wir annehmen, dass Zahlen wie 0 als `Int` getypt werden. Die Typvariable a_i werden jeweils neu eingeführt, wenn nötig.

$x_1 = []$	$b_1 = [a_1]$
$length []$	$a = [a_1] \rightarrow b_2$
$= 0$	$b_2 = Int$
$(x:xs)$	$x :: a_2, xs :: a_3, a_3 = [a_2], (x:xs) :: [a_2]$
$x_1 == (x:xs)$	$[a_1] = [a_2], a_1 = a_2$
$length xs$	keine neuen Gleichungen
$(1 + length xs)$	keine neuen Gleichungen

Ergebnis: $length :: [a_1] \rightarrow Int$.

Versucht man, die etwas zu starke Beschränkung `Int` zu verallgemeinern durch `Num a => a`, dann erhält man: $length :: Num a => [b] \rightarrow a$, was allgemeiner als der Typ im Haskell-Prelude ist, aber von Haskell-Typchecker für obige Funktionsdefinition berechnet wird.

Die entsprechende Haskell-Funktion ist `genericLength` im Modul `List`.

Beispiel 4.5.13

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

Zum Berechnen gehen wir vor wie oben und tabellieren die Ergebnisse.

Annahmen: $x:xs :: b_1, ys :: b_2$.

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

Erst hier wird der Resultattyp eingeschränkt!

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

Bemerkung 4.5.14 : Zur Typisierung in Haskell: In Haskell und Gofer ist der Superkombinator $g\ x = \text{Cons } x\ (g\ (g\ 'c'))$ nicht typisierbar. Der Grund ist das Typsystem von Haskell / Gofer, das auch Typklassen unterstützt und Überladung auflöst. Die Überladungsauflösung soll zur Compilezeit auf einfache Weise möglich sein. Dies wird unterstützt durch die Berechnung von eingeschränkteren Typen (nach Milner).

Beispiel 4.5.15 Als zweites Beispiel die Berechnung des Typs von `foldr`:

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

1. $\text{foldr} :: \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4, f : \alpha_1, z : \alpha_2, as : \alpha_3$
2. Wegen $\text{case_List as } \{[] \rightarrow z; \dots : \alpha_3 = [\alpha_5], \alpha_2 = \alpha_4.$
3. Pattern $x:xs \quad x:\alpha_6, xs: [\alpha_6] = [\alpha_5], \text{ d.h. } \alpha_6 = \alpha_5.$
4. $f x (\text{foldr } f z xs) : \alpha_1 = \alpha_6 \rightarrow \alpha_2 \rightarrow \alpha_2$: Der Grund ist, dass der Resultattyp durch z festgelegt wird.
5. es ergibt sich: $\text{foldr} :: (\alpha_5 \rightarrow \alpha_2 \rightarrow \alpha_2) \rightarrow \alpha_2 \rightarrow [\alpha_5] \rightarrow \alpha_2.$
Das ist auch der Typ, den das Haskell-Typsystm berechnet.

Definition 4.5.16 Ein Typ τ_1 ist allgemeiner als ein Typ τ_2 , wenn es eine Substitution σ für die Typvariablen gibt, so dass $\sigma(\tau_1) = \tau_2$. Dieser Test ist in linearer Zeit durchführbar durch strukturelles Durchmustern der beiden Typen.

Erhält man mit dem Milner Typberechnungsverfahren einen allgemeineren Typ für einen Superkombinator, als den im Programm festgelegten – falls man einen festgelegt hat –, so ist das nicht weiter schlimm, aber die Information könnte nützlich für den Programmierer sein. Erhält man einen spezielleren oder einen anderen Typ als den vorgegebenen, so ist die vorgegebene Typisierung vom Typchecker abzulehnen. Spezieller darf der Typ nicht sein, da dann die Konsistenz der Annahmen verletzt wird.

4.5.6 Bemerkung zur Erweiterung von Typsystemen

Es ist möglich, ein Typsystem zu definieren, das es erlaubt, Bäume mit geschachtelten Listen zu implementieren. Dafür muss man aber bei der Unifikation den Occurs-check weglassen, und muss rekursiv definierte Typen akzeptieren, die sich nicht als endliche Terme darstellen lassen. Ein weiterer Nachteil ist, dass eine bestimmte Klasse von Typfehlern manchmal unentdeckt bleibt. Es sind gerade solche Typfehler, die eine “falsche“ Schachtelung von Datenstrukturen charakterisieren. Fehler in Haskell: “. . . infinite type . . . “.

Beispiel 4.5.17 Argument-ignorierende Funktion: $f \ x = f$
 $A : \{f : \tau_1; x : \tau_2\}$ Typ von $f : \tau_2 \rightarrow \tau_1$ Unifikation mit τ_1 ergibt $\tau_1 \doteq \tau_2 \rightarrow \tau_1$. Standardunifikation liefert einen Fehler. Weglassen des Occurs-check würde diesen Typ zulassen.

Beispiel 4.5.18 Funktionen auf geschachtelten Listen:

Die Funktion `nn` soll die Anzahl der Nil-Vorkommen in einer geschachtelten Liste zählen:

```
nn xs = case_list xs {Nil -> 1;   (Cons y ys) -> ((nn y) + (nn ys))}
```

Typisierung mit dem iterativer Typcheck:

$A = \{\mathbf{nn} : [\alpha] \rightarrow \mathbf{Int}, xs : \tau\}$
 Der *case*-Ausdruck hat Typ \mathbf{Int} .
 $xs : [\alpha_1] \doteq \tau$
 $y : \alpha_1, ys : [\alpha_1], (\mathbf{nn} \ y) : [\alpha_3] \doteq \alpha_1$
 ergibt den neuen Typ: $[[\alpha]] \rightarrow \mathbf{Int}$. Man sieht: Die Iteration terminiert nicht.
 In Haskell ist diese Funktion nicht typisierbar und wird zurückgewiesen:

```

nn Nil = 1
nn (x:xs) = (nn x) + (nn xs)
  
```

Der Typ von \mathbf{nn} darf während des Typcheckens nicht kopiert werden. Das ergibt:
 $\{\mathbf{nn} : [\alpha] \rightarrow \mathbf{Int}, xs : \tau\}$
 Der *case*-Ausdruck hat Typ \mathbf{Int} .
 $xs : [\alpha_1] \doteq \tau$
 $y : \alpha_1, ys : [\alpha_1] (\mathbf{nn} \ x) : [\alpha] \doteq \alpha_1 (\mathbf{nn} \ xs) : [\alpha] \doteq [\alpha_1]$
 Unifikation (ohne Occurs-check) ergibt hier $\alpha \doteq \alpha_1 \doteq [\alpha]$.
 Erlaubt man Unifikation ohne Occurs-Check, so könnte man als Typ folgenden Ausdruck nehmen:

$$nn :: [\alpha] \rightarrow \mathbf{Int} \text{ where } \alpha = [\alpha]$$

Da $\mathbf{Nil} : [\alpha]$, wobei α beliebigen Typ haben kann, wird in diesem Typsystem eine verschachtelte Liste beschrieben, wobei nur \mathbf{Nil} oder $\text{bot}:\alpha$ als Blatt erlaubt ist. Man kann auf dieser Idee ein Typsystem aufbauen, das ebenfalls dynamische Typfehler verhindert. Allerdings erhält man andere Nachteile, wie z.B. nicht erkannte Fehler beim Programmieren.
 Empfehlung: Man sollte die Typcheckvariante mit Occurs-check benutzen.

4.5.7 Typisierung unter Beteiligung von Typklassen

Zur Konstruktion eines Typisierungsalgorithmus, wenn Typklassen beteiligt sind, setzen wir auf dem System für Grundtypen auf. Analog zur Darstellung einer Menge von Grundtypen nehmen wir in dem Fall die Darstellung einer Menge von Grundtypen durch einen polymorphen Typ, der durch Typklassen eingeschränkt ist.

Wir nehmen dafür an, dass die Sprache um Typklassen erweitert ist. Typklassen sind so definiert, dass sie genau Mengen von Grundtypen entsprechen, d.h. es sind einstellige Prädikate. Wir nehmen auch an, dass die Definitionen der Typklassen festlegen, welche Grundtypen zur Typklasse gehören und welche nicht. Die Syntax eines Typs ist:

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

wobei TKL_i eine Typklasse, α_i Typvariablen und τ ein polymorpher Typ ist. Damit kann man die Menge der Grundinstanzen eines Typs

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

hinschreiben:

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

wobei $S(\text{TKL})$ die Menge der Grundtypen sein soll, die zur Typklasse TKL gehören.

D.h. man kann einen erweiterten Typ als polymorphen Typ auffassen, der noch konjunktive Bedingungen für die Typvariablen erfüllen muss. D.h. man hat einen polymorphen Typ mit Constraints.

Beispiel 4.5.19 *Wir bestimmen den Typ des Ausdrucks*

`[] == []`

Der Typ von `==` ist `Eq a => a -> a -> Bool`. Der Typ von `[]` ist `[b]`. Damit ist der Typ von `((==) [])` der Ausdruck `Eq [b] => [b] -> Bool`, d.h. die Instanzen von `[b] -> Bool`, wobei `[b]` in der Typklasse `Eq` sein muss. Aus der Definition der Typklassen in Haskell ergibt sich, dass es eine Definition gibt, die impliziert, dass `Eq [b]` gilt gdw `Eq b`. Damit bleibt `Eq b => [b] -> Bool`. Der Typ des Vergleichs ergibt jetzt, dass man für das zweite `[]` den Typ `[c]` hat:

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

Vergleicht man mit Haskell, so sieht man, dass das der Typchecker von Haskell fast genauso macht. In früheren Versionen war es genau dieser Typ. In der neuesten Version werden offenbar die irrelevanten Bedingungen weggelassen. Das ergibt `[] == [] :: Bool`.

In früheren Versionen von Haskell war der Ausdruck typisierbar, aber nicht auswertbar, denn der benutzte Typ von `((==))`, der sich ergibt ist `Eq a => [a] -> [a] -> Bool`, aber `a` ist nicht spezifizierbar. D.h. die Fehlermeldung kam vom Versuch, die Überladung von `((==))` aufzulösen.

Dies konnte beheben durch explizite Typisierung:

`[] == ([] :: [Int])`

Dann ergibt sich `Eq Int => [Int] -> [Int] -> Bool` als benutzten Typ von `((==))`. Hierbei ist die Bedingung `Eq Int` aufgrund der Definition erfüllt, so dass man genau den Typ `[Int] -> [Int] -> Bool` hat.

In der aktuellen Version (2006) wird offenbar direkt ausgenutzt, dass die Typvariable `a` des Typs `[a]` von `[]` keine Rolle spielen kann bei der Auswertung, so dass es keinen Fehler gibt.

Der Ausdruck

`\x->x == \x->x`

ist nicht typisierbar: Man erhält analog zu oben als Typ von `((==) (\x->x))` `Eq (a->a) => (a->a) -> Bool`. Da aber `Eq (a->a)` nicht gilt, hat dieser Typ eine leere Instanzmenge, und man erhält einen Typfehler.

Beispiel 4.5.20 *Als zweites Beispiel die Funktion `summe`:*

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

```
summe :: Num a => [a] -> a
```

Als Annahme nehmen wir $\{x : \tau_1, \text{summe} : \forall \alpha\}$.

Es ergibt sich: $\tau_1 = [\tau_2], y : \tau_2, ys : [\tau_2]$,

$0 :: \text{Num } a \Rightarrow a, (+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$,

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

Aus der Superkombinatorregel folgt: $\text{Num } \tau_2 \Rightarrow [\tau_2] \rightarrow \tau_2$.

Das ist der Fixpunkt, was man durch einen weiteren Iterationsschritt verifizieren kann.

Dieser Typ wird auch vom Milner Typalgorithmus berechnet und auch vom Haskell Typsystem ausgegeben: `summe :: Num a => [a] -> a`.

Bei der Auflösung der Überladung von $(+)$ wird ebenfalls festgestellt, dass man den Grundtyp nicht zur Compilerzeit fixieren kann. Die Funktion `summe` muss hierbei so in die Kernsprache übersetzt werden, dass sie ein verstecktes Typargument (`[a]`) benötigt. Damit lässt sich die Überladung zur Laufzeit auflösen, da der genaue Typ für die Anwendung von $(==)$ daraus zu errechnen ist.

Beispiel 4.5.21 *Ein Beispiel zu einem Programm, das lineare Gleichungen nach dem Gauss-Verfahren löst (`gauss.hs`).*

Die Funktion `gauss` erwartet als Argument die Parameter des Gleichungssystems, die letzte Spalte sind die Werte der rechten Seiten. Das Gauss-Verfahren kann man so programmieren, dass es den Typ

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

hat. D.h. die Division muss erlaubt sein. Gibt man eine Matrix von Integer als konstanten Superkombinator ein, dann hat die Matrix den Typ

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

Vom Typchecker in Hugs98 erhält dieser leider den spezielleren Typ `[[Integer]]`, was beim Aufruf `gauss matrix` einen Fehler ergibt. Hier vergeblich der Typchecker offenbar fehlerhaft zu spezielle Typen. Gibt man den Typ von `matrix` explizit ins Programm ein,

```
ergebnis = gauss matrix
```

dann erhält man

```
:t ergebnis ----> [Double]
```

Aber:

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

Unser Constraint-Verfahren liefert:

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

Aber es gilt ja, dass $\text{Fractional} \subseteq \text{Num}$, so dass man `Num` weglassen kann. Fügt man die Typangabe ins Programm ein:

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

dann kann man im Interpreter durch Typangabe entweder die Gleitkommadarstellung oder die Bruchdarstellung abrufen.

```
Main> ergebnis :: [Double]
[46.4371,27.8623,48.759,6.19162]
Main> ergebnis :: [Ratio Integer]
[7755 % 167,4653 % 167,32571 % 668,1034 % 167]
```

Das analoge Verhalten zeigt auch *ghci* (2006).

4.6 Subtypen, Ko- und Kontravarianz

Man kann das Milnersche Typsystem mit Subtypen erweitern. Wir nehmen mal hypothetisch an, dass die Syntax dafür bereits existiert.

Um die typischen Effekte zu sehen, nehmen wir hierzu als Subtypenhierarchie $\text{Nat} < \text{Int} < \text{Float} < \text{Complex}$

Wenn ein Unterausdruck einen Typ τ hat, dann darf man typmäßig jeden Ausdruck dafür einsetzen, der einen Typ τ' mit $\tau' \leq \tau$ hat.

Aus den \leq -Beziehungen kann man weitere schließen, z.B. $[\text{Int}] \leq [\text{Complex}]$. Das gleiche für Tupel und die meisten benutzerdefinierten Typkonstruktoren. D.h. aus $\tau_i \leq \tau'_i$ folgt $(\tau_1, \dots, \tau_n) \leq (\tau'_1, \dots, \tau'_n)$. Dies nennt man *kovariante* Fortsetzung der Ordnung. Für benutzerdefinierte Typkonstruktoren gilt die Konvarianz, wenn man keine \rightarrow -Typen in den Datenkonstruktoren verwendet, genauer: wenn α eine Typvariable ist, die in einem Datenkonstruktor vorkommt, dann darf α nicht an einer "kovarianten Stelle vorkommen; das sind solche, bei denen man im Syntaxbaum des Typs von oben ungerade oft ins erste Argument eines Pfeil-Typs ($\tau_1 \rightarrow \tau_2$ geht, um zur Variablen α zu kommen).

Allerdings sieht es bei Funktionstypen anders aus:

Man kann aus $\tau_1 \leq \tau'_1$ und $\tau_2 \leq \tau'_2$ nicht schließen, dass $\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2$: Sei $f : \text{Int} \rightarrow \text{Complex}$ und $g : \text{Nat} \rightarrow \text{Float}$.

Dann kann man im Ausdruck $(f((-1) :: \text{Int}) +_{\text{Complex}} i)$ die Funktion f nicht durch g ersetzen, denn g kann nicht mit `Int`-Argumenten arbeiten.

Aber: Wenn man folgende Funktionen hat:

$f' : \text{Nat} \rightarrow \text{Complex}$ und $g' : \text{Int} \rightarrow \text{Float}$.

Dann kann man in $f'(1 :: \text{Nat}) +_{\text{Complex}} i$ die Funktion f' durch g' ersetzen.

Verallgemeinert man das, erhält man:

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

Diese Fortsetzung der Ordnung nennt man auch *kontravariant* im ersten Argument, während man das Verhalten der anderen Argumente und Typkonstruktoren *kovariant* nennt.