Praktische Informatik 3 – Funktionale Programmierung
WiSe 2010/11, Universität Bremen
Dennis Walter, DFKI Bremen
Revision 1267 vom 2010-12-14

Vorlesung 6: Zum Typsystem von Haskell

1 Motivation

Typen erlauben das statische Zurückweisen unsinniger Ausdrücke

- beim Kompilieren
- **nicht** erst zur Laufzeit

Beispiel

```
"a" ++ 7

let f x = -x in f length

foldr [1, 2] (+) 0
```

2 Was ist ein Typsystem?

> *Ein Typsystem ist eine handhabbare syntaktische Methode, um die Abwesenheit bestimmter Programmverhalten zu beweisen, indem Ausdrücke nach der Art der Werte, die sie berechnen, klassifiziert werden.*
>
> *(Benjamin C. Pierce, [Pie02])*

Slogan:

> Well-typed programs can't go wrong
>
> (Robin Milner)

3 Vorteile von Typsystemen

- Frühzeitiges Aufdecken "offensichtlicher" Fehler
- "Once it type checks, it usually works"

- Hilfestellung bei Änderungen von Programmen
- Strukturierung großer Systeme auf Modul- bzw. Klassenebene
- Quellcodedokumentation
- Effizienz

4 Typen in Haskell: Was bisher geschah

- Primitive Basisdatentypen:
  `Bool`, `Double`
- Funktionstypen
  `Int` →`Int` →`Int`, `[Double]` →`Double`
- Allgemeiner: Typkonstruktoren zur Erzeugung von Typen:
  `[]`, `Maybe`, `MyList`
  - Listenkonstruktor angewandt auf Typen `Bool` ergibt `[Bool]`; `Maybe` angewandt auf `String` ergibt `Maybe String`, ebenso: `MyList Int`
- Typvariablen für polymorphe Funktionen und Datentypen
  `id :: a` →`a`
  `length :: [a]` →`Int`
  `map :: (a` →`b)` →`[a]` →`[b]`
- Typklassen (heute nicht Thema) schränken polymorphe Typen auf solche ein, für die bestimmte Funktionen existieren:
  `elem :: Eq a` ⇒`a` →`[a]` →`Bool`
  (`Eq a` fordert insbesondere Existenz des Gleichheitsoperators `==` auf dem Typen `a`)
  `max :: Ord a` ⇒`a` →`a` →`a`
  (`Ord a` fordert eine Ordnung auf dem Typen `a`, d.h. das Vorhandensein der Operatoren `<`, `<=`, etc.)

5 Typen für Ausdrücke

Wir wollen für einen Haskell-Ausdruck e einen zu diesem "passenden" Typen t ableiten. e ist hierbei als **Platzhalter** für einen beliebigen Ausdruck zu verstehen (genauso steht t für einen beliebigen Typen). Dabei stellen sich zunächst folgende Fragen:

1. Wie werden Ausdrücke gebildet?
2. Wie werden Typen gebildet?

5.1 Ausdrücke

Wir beschreiben die Menge der für unsere Zwecke relevanten Haskell-Ausdrücke über ihre **abstrakte Syntax**. Vereinfachend gesagt ignorieren wir bei dieser im Gegensatz zu einer konkreten Syntaxdefinition – etwa mithilfe einer kontextfreien Grammatik – "technische Details" wie Präzedenzen von Operatoren und die Verwendung konkreter Symbole wie etwa den Gruppierungssymbolen `(` und `)`. Uns interessiert lediglich die abstrakte **Struktur** der Ausdrücke. Weiteres hierzu kann in [Rey98, Kap. 1] nachgelesen werden.

Die abstrakte Syntax für Ausdrücke geben wir wie folgt vor:

$$
\begin{array}{rcl}
e & ::= & var \\
& | & \lambda\, var.\, e_1 \\
& | & e_1\ e_2 \\
& | & \textbf{let}\ var = e_1\ \textbf{in}\ e_2 \\
& | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
& | & \textbf{case}\ e_1\ \textbf{of} \\
& & \quad C_1\ var_1 \cdots var_n \rightarrow e_1 \\
& & \quad \cdots
\end{array}
\tag{1}
$$

Der Weg von einer abstrakten Syntax zur Definition eines diese repräsentierenden Datentypen ist nicht weit:

```
type Var = Int
type Pattern = ... -- hier unwichtig
data Expr = Var Var
          | Abs Var Expr
          | App Expr Expr
          | Let Var Expr Expr
          | If Expr Expr Expr
          | Case Expr [(Pattern, Expr)]
```

Ein Ausdruck ist also entweder ein beliebiger **Bezeichner** (Kategorie *var*; in einer abstrakten Grammatik geben wir die lexikalischen Eigenschaften von Bezeichnern nicht vor: sie sind ein technisches Detail), eine **λ-Abstraktion** (d.h. eine anonyme Funktion)[1], eine (Funktions-) **Applikation**, ein **let-Ausdruck**, ein **bedingter Ausdruck** oder ein **Pattern Matching**.

Beispiele: Ausdrücke, die über diese Grammatik gebildet werden können sind etwa

1. **if** *odd x* **then** *True* **else** $f\ x$.

   Strukturell ist dies ein bedingter Ausdruck, dessen Bedingung die Applikation *odd x* ist, und dessen Zweige der Bezeichner *True* sowie die Applikation $f\ x$ sind. *True*, f, x und *odd* sind gültige Ausdrücke gemäß der abstrakten Grammatik und gehören der Kategorie *var* der Bezeicher an.

[1] Wir schreiben wie in der Literatur üblich λ statt wie in Haskell den `\` zu verwenden und ersetzen Haskell's `->` durch einen simplen Punkt. `\x-> e` wird zu $\lambda x.\, e$

2. $\lambda x.\ \textbf{let}\ y = x + (x - 1)\ \textbf{in}\ y$

   Dies ist eine λ-Abstraktion; Interessant hier ist die Diskrepanz zwischen konkreter Syntax und abstrakter Syntax im Teilausdruck $x + (x - 1)$: Der Lesbarkeit halber schreiben wir den Ausdruck in **Infix**-Notation (mit den Symbolen $+$ und $-$ zwischen ihren Operanden), doch strukturell liegt hier eine Applikation vor, was klar wird, wenn wir den Ausdruck äquivalent umformen und vollständig klammern: $((+)\ x)\ (((-)\ x)\ 1)$. $+$ und $-$ werden hier genauso als Bezeichner (*var*) angesehen wie x.

Andere aus Haskell bekannte Ausdrücke können in diesem **Kern** codiert werden:

- Die Mehrfachapplikation $f\ x_1\ x_2\ \ldots\ x_n$ ist äquivalent zu der geschachtelten unären Applikation $((((f\ x_1)\ x2\ )\ \ldots\ )\ x_n)$. Dies bezeichnet man auch als **Currying**.
- Guards über **if then else** -Ketten:

  ```
  f x | x < 0 = a
      | otherwise = b
  ```

  kann kodiert werden als $\textbf{let}\ f\ = \lambda x.\ \textbf{if}\ x < 0\ \textbf{then}\ a\ \textbf{else}\ b\ \textbf{in}\ f$.
- **where** kann über **let** ausgedrückt werden.
- Mehrere Definitionsgleichungen über Pattern Matching mit **case**. Vgl.

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

  und

  $$\begin{array}{lll} \textbf{let}\ f & = \lambda xs. & \textbf{case}\ xs\ \textbf{of} \\ & & \quad [] \rightarrow 0 \\ & & \quad (x : xs) \rightarrow 1 + f\ xs \\ \textbf{in}\ f & & \end{array}$$

- Tupel (x, y) können als Anwendung des Typelkonstruktors $(,)$ (hier als Funktion zu interpretieren) auf die Argumente x und y verstanden werden: $(((,)\ x)\ y$ (dies ist übrigens gültiges Haskell!).
- Sogar: **if –then–else** über **case**:

  ```
  case b of
    True → e1
    False → e2
  ```

  (Wegen der Häufigkeit der Booleschen Unterscheidung haben wir **if-then-else** dennoch in den Kern mit aufgenommen.)

5.2 Typen

Typen (auch als Typausdrücke bezeichnet) haben eine simple abstrakte Syntax:

$$\begin{array}{rcl} t & ::= & tvar \\ & | & T_0 \\ & | & T_1\ t_1 \\ & | & T_2\ t_1\ t_2 \\ & | & T_3\ t_1\ t_2\ t_3 \\ & | & \cdots \end{array} \tag{2}$$

Wir haben also:

- Die Kategorie der Typvariablen *tvar*, deren Elemente wir gewöhnlich mit a, b, c, oder auch α, β, γ, etc. bezeichnen.
- Für alles andere: Typkonstruktoren
  - Nullstellige Konstruktoren T_0 stehen für Basistypen: *Bool*, *Int*, etc.
  - Zu den einstelligen Konstruktoren gehört der Konstruktor des Listentyps: [] bildet den Elementtyp a auf den Typ von Listen über a ab: $([])\ a$ bzw. lesbarer $[a]$. Allgemein: Gegeben eine Typdefinition **`data T a = C1 a | C2`**, bildet T den Typen a auf $T\ a$ ab.
  - Der wichtigste zweistellige Typkonstruktor ist der Funktionstypkonstruktor: $(\rightarrow)$, dessen Anwendung auf die Typvariablen a und b wir gewöhnlich als $a \rightarrow b$ schreiben. Aber auch Paartypen werden über einen zweistelligen Typkonstruktor gebildet: $(,)\ a\ b$ bzw. (a, b) (syntaktisch identisch, aber nicht zu verwechseln mit der Anwendung des Wertkonstruktors $(,)$ von oben!)

In Haskell könnten wir Typen über den folgenden Datentypen repräsentieren:

```
data Typ = TVar Var
         | TCtor String [Typ]
```

Beispiele:

Int ist dann **`TCtor "Int"[]`**.

Funktionen $a \rightarrow b$ sind dann **`TCtor "→"[TVar a, TVar b]`**.

Listen über a (d.h. der Typ der Form $[a]$) sind **`TCtor "[]"[TVar a]`**.

5.3 Ziel

Gegeben einen Ausdruck e und einen Typen t wollen wir feststellen, ob e den Typen t hat. Wir schreiben hierfür dann $e :: t$. Diese Aufgabe nennen wir **Typprüfung** bzw. **type checking**

Auch lösbar ist die scheinbar schwierigere Aufgabe der **Typinferenz**: gegeben e, finde t, so dass $e :: t$. Insbesondere aufgrund der Polymorphie haben Ausdrücke ggf. mehrere mögliche Typen.

Beispiel:

$$(\lambda x.\ x) :: Int \to Int$$
$$(\lambda x.\ x) :: Bool \to Bool$$
$$(\lambda x.\ x) :: a \to a$$

Üblicherweise sind wir also am **allgemeinsten Typen** (**principal type**) für einen gegebenen Ausdruck interessiert.

Im Beispiel ist $a \to a$ ein allgemeinster Typ aus dem alle anderen gültigen Typen durch eine **Instanziierung** von a erhalten werden können. So ergibt sich der Typ $Int \to Int$ durch die Instanziierung von a zu Int. Bei einer Instanziierung ersetzen wir eine oder mehrere Typvariablen konsistent durch beliebige Typausdrücke. Wird a also in einem Typen zu Int instanziiert, so muss jedes Vorkommen von a durch Int ersetzt werden. Nur Typvariablen lassen sich instanziieren, Typkonstruktoren hingegen nicht. Der Typ $Int \to Int$, der keine Typvariablen enthält, ist also nicht polymorph.

6 Ableitungsregeln für Typurteile

Üblicherweise sind wir am Typen eines Ausdrucks in einem gegebenen Kontext interessiert. Beispielsweise wollen wir den Typen des Ausdrucks *not True* ermitteln. Hierfür benötigen wir die Information, welchen Typ die Bezeichner *not* und *True* haben. Dies legt folgende Definition nahe:

Ein Typurteil (**typing judgment**) ist ein Tripel

$$\Gamma \vdash e :: t \tag{3}$$

das ausdrückt, dass der Ausdruck e im Kontext Γ den Typen t hat.

Genauer ist Γ ein **Variablenkontext**, d.h. eine Abbildung von Variablen auf Typen. Die Intuition ist, dass Γ Variablentypen "verwaltet", wir also über Γ die Typen von Variablen erfragen können.

Wir schreiben

- $\emptyset$ für den leeren Kontext
- $\Gamma, x :: t$ für die Erweiterung von Γ um die **Bindung** der Variable x an den Typen t. $\Gamma, x :: t$ bildet also fast dieselben Variablen auf dieselben Typen ab wie Γ, allerdings wird zusätzlich x auf t abgebildet. Wenn Γ bereits x auf t' abbilden sollte, so wird diese Variablenbindung in $\Gamma, x :: t$ durch die neue Bindung $x :: t$ **verschattet**.
- Wir schreiben $\Gamma(x)$, um den Typen von x in Γ zu erfragen.

Beispiel: für $\Gamma = x :: t, y :: t'$ ist $\Gamma(y) = t'$. Für den Kontext $\Gamma, x :: t'$ gilt hingegen $(\Gamma, x :: t')(x) = t'$. Für $\Gamma = \mathit{null} :: \mathit{Bool} \to \mathit{Bool}, \mathit{True} :: \mathit{Bool}$ würden wir erwarten, dass $\Gamma \vdash \mathit{null}\ \mathit{True} :: \mathit{Bool}$ ein gültiges Judgment ist.

Die Frage ist nun, wie wir bestimmen können, ob ein Judgment gültig ist. Dies geschieht über ein sogenanntes **Regelsystem** oder **Ableitungssystem**. Abbildung 1 zeigt das Ableitungssystem für die hier betrachtete Teilmenge von Haskell-Ausdrücken unter Auslassung von Pattern Matchings über **case**.[2]

$$\frac{\Gamma(x) = t}{\Gamma \vdash x :: t} \quad \text{(Var)}$$

$$\frac{\Gamma, x :: t_1 \vdash e :: t_2}{\Gamma \vdash \lambda x.\, e :: t_1 \to t_2} \quad \text{(Abs)}$$

$$\frac{\Gamma \vdash e_1 :: t_2 \to t \qquad \Gamma \vdash e_2 :: t_2}{\Gamma \vdash e_1\ e_2 :: t} \quad \text{(App)}$$

$$\frac{\Gamma \vdash e_1 :: t_1 \qquad \Gamma, x :: t_1 \vdash e_2 :: t_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :: t_2} \quad \text{(Let)}$$

$$\frac{\Gamma, x :: t_1 \vdash e_1 :: t_1 \qquad \Gamma, x :: t_1 \vdash e_2 :: t_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :: t_2} \quad \text{(LetRec)}$$

$$\frac{\Gamma \vdash e :: \mathit{Bool} \qquad \Gamma \vdash e_1 :: t \qquad \Gamma \vdash e_2 :: t}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 :: t} \quad \text{(If)}$$

Abbildung 1: Das Ableitungssystem für die betrachtete Teilsprache von Haskell.

Die **Ableitungsregeln** (Var, Abs, App, etc.) sind zu lesen wie Implikationen: Wenn wir die Gültigkeit der Judgments oder Bedingungen oberhalb des ——— herleiten können, erlauben uns die Regeln zu schließen, dass auch das unter dem Strich stehende Judgment gültig ist. Die Regel (App) kann also gelesen werden als "*Wenn wir für einen beliebigen Ausdruck e_1 und einen Kontext Γ nachweisen können, dass e_1 den Typen $t_2 \to t$ (also einen Funktionstypen) hat, und wir zudem zeigen können, dass ein Ausdruck e_2 den Typen t_2 hat, dann können wir schließen, dass die Applikation $e_1\ e_2$ bzgl. Γ den Typen t hat.*"

Die Regeln können allerdings genauso gut "rückwärts", d.h. von unten nach oben gelesen werden: Regel (Abs) würde dann verbalisiert als "*Um zu zeigen, dass eine λ-Abstraktion $\lambda x.\, e$ den Typen $t_1 \to t_2$ bzgl. des Kontexts Γ hat, muss gezeigt werden, dass im erweiterten Kontext $\Gamma, x :: t_1$ (d.h. unter der zusätzlichen Annahme, dass x den Typen t_1 hat) der Körper der Abstraktion, e, den Typen t_2 hat.* ".

Wir sagen, dass ein Ausdruck e bzgl. eines Kontexts Γ **wohlgetypt** ist, wenn wir

[2] Die Formulierung einer entsprechenden Regel bleibt dem Leser als Übung vorbehalten.

ein Judgment $\Gamma \vdash e :: t$ für irgend einen Typen t ableiten können. Die Menge der **gültigen Judgments** ist genau die Menge aller Judgments, für die wir eine Ableitung mittels der gegebenen Regeln finden können.

6.1 Praktischer Nutzen

Die Formalisierung des Typsystems einer Programmiersprache hat den Vorteil, dass bestimmte Eigenschaften des Typsystems mathematisch stringent nachgewiesen und analysiert werden können. Eine entscheidende Eigenschaft des Typsystems von funktionalen Sprachen wie Haskell oder Standard ML [Mil78] wird häufig unter dem Slogan "*well-typed programs don't go wrong*" zusammengefasst. Damit wird ausgedrückt, dass typbezogene Laufzeitfehler wie insbesondere die Anwendung einer Funktion auf Argumente eines unzulässigen Typs unter allen Umständen statisch, d.h. zur Zeit der Typprüfung durch den Compiler, ausgeschlossen werden können (wenn der Compiler nicht fehlerhaft programmiert ist!).

7 Beispielableitungen

Ableitungen kann man auf natürliche Weise als Bäume darstellen, die man dann passenderweise **Ableitungsbäume** nennt. Zum Beispiel können wir den Typen der Identitätsfunktion $\lambda x.\ x$ angewandt auf 1 im Kontext $\Gamma = 1 :: Int$ herleiten:

$$\dfrac{\dfrac{\dfrac{(\Gamma, x :: Int)(x) = Int}{\Gamma, x :: Int \vdash x :: Int^{1}}(\text{Var})}{\Gamma \vdash \lambda x.\ x :: Int \to Int^{3}}(\text{Abs}) \qquad \dfrac{\Gamma(1) = Int}{\Gamma \vdash 1 :: Int^{2}}(\text{Var})}{\Gamma \vdash (\lambda x.\ x)\ 1 :: Int^{4}}(\text{App}) \tag{4}$$

Hierbei werden also die jeweils angewandten Regeln übereinander geschrieben, indem ihre Schlussfolgerungen direkt als Vorbedingungen weiterer Regelanwendungen dienen. Diese Art der Darstellung ist allerdings für größere Ableitungen schwer handhabbar. Man behilft sich, indem die einzelnen Zwischenfakten (oben mit ^{1} bis ^{4} gekennzeichnet) linear untereinander geschrieben und mit einer Angabe der verwendeten Regel zur Ableitung des Judgments sowie der verwendeten Vorbedingungen versehen werden.

7.1 Identitätsfunktion

Behauptung:

$$\emptyset \vdash \lambda x.\ x :: a \to a \tag{5}$$

Herleitung:

$$\begin{array}{lll} 0. & (x :: a)(x) = a & \text{Definition des Kontextauslesens} \\ 1. & x :: a \vdash \quad x :: a & \text{Var}[0] \\ 2. & \emptyset \vdash \lambda x.\, x :: a \to a & \text{Abs}[1] \end{array} \tag{6}$$

Die Erfülltheit der Vorbedingung der Regel (Var), also dass eine Variable auf einen bestimmten Typen abgebildet wird, ist bei konkret vorliegendem Kontext direkt ersichtlich. So kann oben Fakt 0. direkt aus 1. entnommen werden. Wir verwenden daher eine abkürzende Schreibweise, bei der die Anwendung der Regel (Var) nicht explizit begründet wird:

$$\begin{array}{lll} 1. & x :: a \vdash \quad x :: a & \text{Var} \\ 2. & \emptyset \vdash \lambda x.\, x :: a \to a & \text{Abs}[1] \end{array} \tag{7}$$

7.2 Identitätsfunktion angewandt auf 1

Hier noch einmal die obige Ableitung im linearen Stil notiert. Behauptung:

$$1 :: \mathit{Int} \vdash (\lambda x.\, x)\ 1 :: \mathit{Int} \tag{8}$$

Herleitung:

$$\begin{array}{llll} 1. & 1 :: \mathit{Int}, x :: \mathit{Int} \vdash & x :: \mathit{Int} & \text{Var} \\ 2. & 1 :: \mathit{Int} \vdash & 1 :: \mathit{Int} & \text{Var} \\ 3. & 1 :: \mathit{Int} \vdash & \lambda x.\, x :: \mathit{Int} \to \mathit{Int} & \text{Abs}[1] \\ 4. & 1 :: \mathit{Int} \vdash & (\lambda x.\, x)\ 1 :: \mathit{Int} & \text{App}[3,2] \end{array} \tag{9}$$

7.3 Nicht typkorrekter Ausdruck

Behauptung:

$$\Gamma \vdash 1 + \text{`}a\text{'} :: \mathit{Int} \tag{10}$$

mit $\Gamma = 1 :: \mathit{Int}, (+) :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}, \text{`}a\text{'} :: \mathit{Char}$

Versuch einer Herleitung:

$$\begin{array}{lll} 1. & \Gamma \vdash \quad 1 :: \mathit{Int} & \text{Var} \\ 2. & \Gamma \vdash \quad (+) :: \mathit{Int} \to (\mathit{Int} \to \mathit{Int}) & \text{Var} \\ 3. & \Gamma \vdash (+)\ 1 :: \mathit{Int} \to \mathit{Int} & \text{App}[2,1] \\ 4. & \Gamma \vdash \quad \text{`}a\text{'} :: \mathit{Char} & \text{Var} \\ 5. & \quad \lightning \quad (\mathit{Int} \neq \mathit{Char}) & \end{array} \tag{11}$$

Aufgrund dieser Beispielrechnung dürfen wir zwar noch nicht formal schließen, dass es keine Ableitung für das behauptete Judgment geben kann. Wir sehen aber anschaulich, dass die einzig anwendbaren Ableitungsregeln hier nicht ans Ziel führen. (Es gibt in der Tat keine Ableitung für das behauptete Judgment).

8 Typinferenz

Die Regeln aus Abbildung 1 können auch als eine Vorschrift für einen Algorithmus zur Typinferenz verstanden werden.

Die Herleitung des Typen eines Ausdrucks ist leicht im Falle einer **expliziten Typisierung** der Ausdrücke, bei dem alle Variablen mit ihrem Typen annotiert sind. Schreiben wir überall $\lambda x :: t.\ e$ statt $\lambda x.\ e$ sowie **let** $x :: t = e_1$ **in** e_2 anstelle von **let** $x = e_1$ **in** $e2$, dann ergeben sich die Variablenkontexte der Ableitungen aus Abschnitt 7 sofort, und der Algorithmus kann schlicht die Regeln (in Vorwärtsrichtung) anwenden und so stückweise den Typen des Ausdrucks aufbauen.

Haskell ist aber **implizit getypt**, d.h. die Angabe des Typs jeder eingeführten Variablen ist nicht erforderlich. Ein Algorithmus zur Typinferenz muss also die Typen von Variablen herleiten, d.h. anhand ihrer Verwendung **inferieren**.

Wie ermittelt ein Compiler die Typen? Die Grundidee ist, zunächst einen allgemeinstmöglichen Typen für jede Variable und jeden Teilausdruck anzunehmen und diesen Typen im Verlauf der Typinferenz zu konkretisieren. Konkretisiert wird der Typ, indem aufgrund der Verwendung der Teilausdrücke im Gesamtausdruck sog. **Constraints** (Bedingungen) akkumuliert werden, die mittels **Unifikation** (siehe Abschnitt 9) gelöst werden. Die Constraints werden dabei aus dem Regelsystem abgeleitet.

Weisen wir bspw. zunächst einem Teilausdruck e' die Typvariable a zu und entdecken anschließend einen Teilausdruck $e'\ e''$, also das Vorkommen von e' in Funktionsposition, so wissen wir, dass a ein Funktionstyp, also gleich $b \to c$ für entsprechende Typen b und c sein muss.

Wir beschränken uns bzgl. der Typinferenz auf ein **Tafelbeispiel** (unter Verwendung von $succ = (+)\ 1$); Interessierte können Details z.B. in [Pie02] nachlesen.

Wir wagen für den nachstehenden Ausdruck eine Inferenz im als gegeben angenommenen Kontext Γ:

$$\textbf{let}\ len = \lambda xs.\ \textbf{if}\ null\ xs\ \textbf{then}\ 0\ \textbf{else}\ succ\ (len\ (tail\ xs))\ \textbf{in}\ len \tag{12}$$

$$\Gamma = null :: [a] \to Bool, tail :: [b] \to [b], 0 :: Int, succ :: Int \to Int \tag{13}$$

Die hierbei aufgrund der Typregeln resultierenden Constraints sind in Abbildung 2 dargestellt. [1-4] ergeben sich aus dem vorgegebenen Kontext Γ; [5-8] ergeben sich über die Regel (If): die Bedingung *null xs* muss vom Typ *Bool* sein;

[9-11], [12-14], [15-17], [18-20] entstehen aus der Regel (App): alle Ausdrücke in Funktionsposition müssen von Funktionstyp sein. [21-23] entstammt der Regel (Abs), während [24,25] schließlich aus (LetRec) hervorgeht und sicherstellt, dass der Typ der per **let** gebundenen Variablen gleich seiner Definition ist.

1	$null$	$: [a] \to Bool$
2	$tail$	$: [b] \to [b]$
3	0	$: Int$
4	$succ$	$: Int \to Int$
5	$null\ xs$	$: Bool$
6	0	$: c$
7	$succ\ (len\ (tail\ xs))$	$: c$
8	$\textbf{if}\ null\ xs\ \textbf{then}\ 0\ \textbf{else}\ succ\ (len\ (tail\ xs))$	$: c$
9	$null$	$: d \to e$
10	xs	$: d$
11	$null\ xs$	$: e$
12	$tail$	$: f \to g$
13	xs	$: f$
14	$tail\ xs$	$: g$
15	len	$: h \to i$
16	$tail\ xs$	$: h$
17	$len\ (tail\ xs)$	$: i$
18	$succ$	$: k \to l$
19	$len\ (tail\ xs)$	$: k$
20	$succ(len\ (tail\ xs))$	$: l$
21	xs	$: m$
22	$\textbf{if}\ null\ xs\ \textbf{then}\ 0\ \textbf{else}\ succ\ (len\ (tail\ xs))$	$: n$
23	$\lambda xs.\ \textbf{if}\ null\ xs\ \textbf{then}\ 0\ \textbf{else}\ succ\ (len\ (tail\ xs))$	$: m \to n$
24	len	$: p$
25	$\lambda xs.\ \textbf{if}\ null\ xs\ \textbf{then}\ 0\ \textbf{else}\ succ\ (len\ (tail\ xs))$	$: p$

Abbildung 2: Constraints bei der Typinferenz von (12) (auch in [Car87])

Aus diesem System von Typconstraints können wir nun Schlussfolgerungen ziehen, etwa dass $d = f = m$ (aus [10, 13, 21]), da alle diese Variablen als Typ von xs angenommen wurden. Weiterhin muss gelten: $p = m \to n$ ([23, 24]), $n = c = Int$ ([22, 6, 3]), $d = [a]$ ([9, 1]) und damit $p = [a] \to Int$. Wir haben somit den erwarteten Typ für *len* als $[a] \to Int$ ermittelt. Dass die weiteren Constraints keinen Widerspruch beinhalten, und a nicht weiter instanziiert werden muss, nehmen wir an dieser Stelle als gegeben hin (es stimmt auch!).

9 Unifikation

Schließlich betrachten wir eine systematische Weise, Constraints zu lösen, die bei der Typinferenz entstehen. Wir sprechen in diesem Abschnitt allgemein von Variablen und Ausdrücken; in der praktischen Anwendung wie in Abschnitt 8 geht es hierbei um Typvariablen und Typausdrücke.

9.1 Begrifflichkeiten

Gegeben seien Variablen $v_1, \ldots, v_n$ und Ausdrücke $t_1, \ldots, t_n$ (etwa v_i, $v_i \to v_j$ oder (Int, v_i))

Eine Menge von Abbildungen von Variablen zu Ausdrücken

$$\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\} \tag{14}$$

heißt **Substitution**.

Für eine Substitution η und einen Typausdruck t schreiben wir t_η für die Anwendung von η auf t. Hierbei werden alle Vorkommen der v_i durch t_i ersetzt.

Beispiel: Für $t = a \to b \to c$ und $\eta = \{a \mapsto (Int, Bool), b \mapsto c\}$ ist $t_\eta = (Int, Bool) \to c \to c$.

Wir definieren weiter die **Komposition** von Substitutionen. Für $\eta = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ und $\theta = \{x_1 \mapsto u_1, \ldots, x_m \mapsto u_m\}$ bezeichnet $\eta \circ \theta$ ihre Komposition. Diese erhalten wir aus

$$\{x_1 \mapsto (u_1)_\eta, \ldots, x_m \mapsto (u_m)_\eta, v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$$

indem wir alle $x_i \mapsto (u_i)_\eta$ entfernen, bei denen $(u_i)_\eta = x_i$ (Identitätssubst.) sowie alle $v_i \mapsto t_i$ bei denen $v_i = x_j$ für irgend ein $j \leq m$.

Beispiel:

$$\{a \mapsto (c, c), b \mapsto c, d \mapsto c\} \circ \{b \mapsto a, c \mapsto d\} = \{a \mapsto (c, c), b \mapsto (c, c), d \mapsto c\}$$

Eine Substitution η ist ein **Unifikator** für eine Menge von Ausdrücken $E_1, \ldots, E_k$, wenn gilt

$$(E_1)_\eta = (E_2)_\eta = \cdots = (E_k)_\eta,$$

d.h. wenn ihre Anwendung auf die Ausdrücke diese syntaktisch identisch macht.

Beispiel: Die Substitution $\eta = \{a \mapsto Int, b \mapsto c\}$ identifiziert die Ausdrücke $(Int, b \to c)$ und $(a, c \to b)$ und ist somit ein Unifikator für diese:

$$(Int, b \to c)_\eta = (a, c \to b)_\eta = (Int, c \to c)$$

Ein Unifikator heißt **allgemeinster Unifikator** (most general unifier) für $E_1, \ldots, E_k$, wenn für alle anderen Unifikatoren θ immer eine Substitution ι existiert, so dass $\theta = \iota \circ \eta$. (Intuition: θ ist dann spezieller und instanziiert mehr Variablen). Da ι auch eine Substitution sein kann, die lediglich Variablen umbenennt (etwa $\iota = \{a \mapsto a', b \mapsto b'\}$, wobei a' und b' nicht in den Termen vorkommen, auf die ι angewandt wird), ist ein allgemeinster Unifikator nur bis auf Umbenennungen von Variablen eindeutig bestimmt.

9.2 Unifikationsalgorithmus

Ein **Unifikationsalgorithmus** findet für eine gegebene Menge von Ausdrücken einen allgemeinsten Unifikator oder schlägt fehl, weil kein Unifikator exisitert. Er terminiert also immer.

Aus Sicht der Typinferenz sind wir allgemeiner interessiert an einem Unifikator, der nicht nur eine Menge von Ausdrücken unifiziert, sondern für eine Sequenz von Ausdrucksmengen jede enthaltene Menge unifiziert. Für die Constraints aus Abbildung 2 (vgl. letzten Absatz in Abschnitt 8) enthielte die Sequenz insbesondere die Mengen $\{d, f, m, [a]\}$, $\{p, m \to n\}$ und $\{n, c, Int\}$.

Für eine Implementierung ist es einfacher, anstelle von Mengen die Unifikation von jeweils nur zwei Ausdrücken zu betrachten. Die Suche nach einem Unifikator für eine Menge von Ausdrücken $E_1, E_2, \ldots, E_k$ kann auf die Suche nach einem Unifikator für eine Liste von Paaren von Ausdrücken reduziert werden: Nehmen wir eine Variable v, die in den E_i nicht vorkommt, so gelingt die Unifikation der E_i genau dann, wenn die paarweise Unifikation (mit demselben, lediglich um eine Substitution von v erweiterten Unifikator) von (v, E_1), (v, E_2) bis (v, E_k) gelingt.

Beispiel: Die Paare $(a, b \to c)$ und (c, Int). Erwartetes Ergebnis: $\eta = \{a \mapsto (b \to Int), c \mapsto Int\}$, so dass $(a, b \to c)_\eta = (b \to Int, b \to Int)$ und $(c, Int)_\eta = (Int, Int)$.

Gemäß diesen Überlegungen ist eine mögliche Haskell-Signatur des gesuchten Unifikationsalgorithmus:

```
unify :: [(Typ, Typ)] → Either String [(Var, Typ)]
```

Gegeben eine Liste von Paaren von (Typ-)Ausdrücken soll ein allgemeinster Unifikator gefunden (*Right* η) oder mit einer Fehlermeldung abgebrochen werden (*Left message*). Unifikatoren (bzw. Substitutionen) werden als Paarlisten dargestellt, die für jede (Typ-)Variable den Ausdruck angeben, durch den sie zu ersetzen sind.

9.2.1 Algorithmus $\mathcal{U}$ in Pseudo-Haskell

Der Algorithmus läuft schrittweise durch die Paarliste. Er bestimmt für das aktuelle Paar einen Unifikator η, wendet ihn auf die nachfolgenden Ausdrücke in der Paarliste an, berechnet dann einen Unifikator θ für diese und komponiert abschließend die Unifikatoren zum Rückgabewert $\theta \circ \eta$.

$\mathcal{U}([]) = \{\}$

Für $\mathcal{U}((t, t') : z)$: Fallunterscheidungen:

1. t ist Variable ($t = v$)
   (a) Wenn auch $t' = v$, direkt fortfahren mit $\mathcal{U}(z)$.

(*Vermeidung von Identitätssubstitutionen*)

(b) Sicherstellen, dass v nicht in t' vorkommt (**occurs check**); ansonsten Abbruch mit Fehler "Occurs Check".
(*Anschaulich: a kann niemals mit $a \to b$ unifiziert werden*)

(c) Sonst setze $\eta = \{v \mapsto t'\}$ und ermittle θ als $\mathcal{U}(z_\eta)$ (z_η steht hier für die Anwendung von η auf alle Elemente der Tupelliste z)
(*Die vom vorliegenden Paar erzwungene Gleichheit zwischen v und t' muss auf die Restliste propagiert werden*)

(d) Liefere den Unifikator $\theta \circ \eta$

2. t' ist Variable

(a) Berechne $\mathcal{U}((t', t) : z)$
(*Wegen Symmetrie wird dieser Fall auf den vorigen zurückgeführt*)

3. $t = C\ t_1 \cdots t_n$ und $t' = C'\ t'_1\ \cdots t'_n$ sind komplexe Ausdrücke mit $C = C'$. **Beispiel**: $t_1 \to t_2$, mit $\to$ als Konstruktor (hier Infix geschrieben, formal korrekt also $\to\ t_1\ t_2$)

(a) Berechne $\mathcal{U}((t_1, t'_1) : (t_2, t'_2) : \cdots : (t_n, t'_n) : z)$
(*Eine schöne strukturelle Eigenschaft des Algorithmus: die Argumente der Typkonstruktoren müssen paarweise unifiziert werden; wir können sie also schlicht in die vorhandene Liste der zu unifizierenden Paare einhängen. Beispiel: Um $t_1 \to t_2$ und $t'_1 \to t'_2$ zu unifizieren, muss die Paarliste $[(t_1, t'_1), (t_2, t'_2)]$ unifiziert werden.*)

(b) Wenn $n = 0$ (nullstelliger Konstruktor C) heißt dies: Berechne lediglich $\mathcal{U}(z)$.

4. Sonst: Abbruch mit Fehlermeldung "Kein Unifikator"

Literatur

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.