2.5 Typsystem und Typinferenz

Haskell (auch Standard ML oder Java > 5) ist eine *streng getypte Sprache mit einem polymorphen Typsystem*. Die Bedeutung dieser Begriffe wollen wir zunächst informell erläutern, bevor wir dann später diese genau definieren.

Streng getypte Sprache:

- Jedes Objekt (Wert, Funktion) hat einen *Typ*
- Typ $\approx$ Menge möglicher Werte:

  `Int` $\approx$ Menge der ganzen Zahlen bzw. endliche Teilmenge,

  `Bool` $\approx \{$`True`, `False`$\}$

  `Int` $\to$ `Int` $\approx$ Menge aller Funktionen, die ganze Zahlen auf ganze Zahlen abbilden
- "Objekt hat Typ τ" $\approx$ Objekt gehört zur Wertemenge des Typs τ
- *Typfehler:* Anwendung einer Funktion auf Werte, die nicht ihrem Argumenttyp entsprechen.

  Beispiel: `3 + 'a'`

  Falls ein Aufruf von "`+`" ohne Typprüfung der Argumente erfolgen würde, dann kann dies ein unkontrolliertes Verhalten zur Folge haben (Systemabsturz, Sicherheitslücken), was unbedingt vermieden werden sollte.
- Wichtige Eigenschaft streng getypter Sprachen:

  Bei wohlgetypten Programmen können Typfehler zur Laufzeit nicht auftreten ("well-typed programs do not go wrong", [Milner 78]).

  Aus diesem Grund wird der Ausdruck

  ```
  3 + 'a'
  ```

  durch den Compiler zurückgewiesen, ebenso wie der Ausdruck

  ```
  foldr (+) 0 [1, 3, 'a', 5]
  ```

  Beachte: der Typfehler im ersten Ausdruck ist „unmittelbar" klar, wohingegen der Typfehler im zweiten Ausdruck etwas mehr Überlegung benötigt.

Schwach getypte Sprachen (Scheme, Smalltalk, JavaScript, PHP, Ruby,...):

- Objekte haben Typ (werden zur Laufzeit mitgeführt)
- Funktionen prüfen zur Laufzeit, ob ihre Argumente typkorrekt sind

Vorteile stark getypter Sprachen:

- Programmiersicherheit: Typfehler treten nicht auf
- Programmiereffizienz: Compiler meldet Typfehler, bevor das Programm ausgeführt wird (Compiler prüft „Spezifikation“)
- Laufzeiteffizienz: Typprüfung zur Laufzeit unnötig
- Programmdokumentation: Typangaben können als (automatisch überprüfbare!) Teilspezifikation angesehen werden

Nachteile:

- strenge Typsysteme schränken die Flexibilität ein, um eine automatische Prüfung zu ermöglichen
- nicht jedes Programm ohne Laufzeittypfehler ist zulässig, z.B. ist

  ```
  takeWhile (<3) [1, 2, 3, 'a']
  ```

  unzulässig, obwohl kein Laufzeittypfehler auftreten würde.

Wichtige Ziele bei der Entwicklung von Typsystemen:

- Sicherheit
- Flexibilität (möglichst wenig einschränken)
- Komfort (möglichst wenig explizit spezifizieren)

In funktionalen Sprachen wird dies erreicht durch

Polymorphismus: Objekte (Funktionen) können mehrere Typen haben

Typinferenz: nicht alle Typen müssen deklariert werden, sondern Typen von Variablen und Funktionen werden inferiert

Nachfolgend werden wir beide Konzepte genauer erläutern.

2.5.1 Typpolymorphismus

Wie oben erläutert, bedeutet **Polymorphismus** in Programmiersprachen, dass Objekte, typischerweise Funktionen, mehrere Typen haben bzw. auf Objekte unterschiedlicher Typen angewendet werden können.

Generell unterscheiden wir zwei Arten von Polymorphismus.

Ad-hoc Polymorphismus: Funktionen, die mehrere Typen haben,können sich auf unterschiedlichen Typen verschieden verhalten.

Ein Beispiel für diese Art ist **Overloading**, **überladene Bezeichner**. Hierbei verwendet man einen Bezeichner für unterschiedliche Funktionen.

Java: "+" steht für

- Addition auf ganzen Zahlen
- Addition auf Gleitkommazahlen
- Stringkonkatenation

Parametrischer Polymorphismus: Funktionen haben gleiches Verhalten auf allen ihren zulässigen Typen.

Beispiel: Berechnung der Länge einer Liste:

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

Die Berechnung ist unabhängig vom Typ der Elemente. Somit kann der Typ von `length` wie folgt ausgedrückt werden:

```
length :: [a]  → Int
```

Hierbei ist `a` eine **Typvariable**, die durch jeden anderen Typ ersetzbar ist. Dadurch ist `length` anwendbar auf den Argumente der Typen

```
[Int]  [Float]  [[Int]]  [(Int, Float)]  ...
```

Z.B. ist der Ausdruck

```
length [0,1] + length ['a', 'b', 'c']
```

zulässig!

Im Gegensatz zum ad-hoc Polymorphismus arbeitet `length` auf allen zulässigen Typen gleich!

Beachte: Bei streng getypten Programmiersprachen ohne parametrischen Polymorphismus (wie Pascal oder C) muss für jeden Listentyp eine eigene `length`-Funktion (mit identischer Struktur) definiert werden!

Beispiel: Identität:

```
id x = x
```

Allgemeinster Typ: `id:: a → a`

Also: `id` auf jedes Argument anwendbar, aber es gilt immer:
Ergebnistyp = Argumenttyp

$\Rightarrow$ `id` $\underbrace{\texttt{[1]}}_{\texttt{[Int]}}$ `==` $\underbrace{\texttt{['a']}}_{\texttt{[Char]}}$ Typfehler

$\underbrace{\underbrace{\texttt{[Int]}}_{\texttt{[Int]}}}$

Unter **Typinferenz** verstehen wir die Herleitung allgemeinster Typen, sodass das Programm noch *typkorrekt* ist.

Vorteile:

1. Viele Programmierfehler werden als Typfehler entdeckt
   Beispiel: Vertauschung von Argumenten:

   ```
   foldr 0 (+) [1, 3, 5]   → Type error ...
   ```

2. Auch, falls kein Typfehler auftritt, kann der allgemeinste Typ auf Fehler hinweisen:

   ```
   revf []     = []
   revf (x:xs) = revf xs ++ x   -- statt [x]
   ```

   Allgemeinster Typ:

   ```
   revf :: [[a]]  → [a]
   ```

   Allerdings ist der Argumenttyp `[[a]]` nicht beabsichtigt!

Was bedeutet aber genau **typkorrekt**?

Hier gibt es keine allgemeine Definition, sondern dies ist durch die jeweilige Programmiersprache festgelegt. Es gibt aber die allgemeine Forderung:

„Typkorrekte“ Programme haben keine Laufzeittypfehler.

Im folgenden betrachten wir die **Typkorrektheit á la Hindley/Milner** [Milner 78, Damas/Milner 82].[1] Wir benötigen zur präzisen Definition zunächst einige grundlegende Begriffe.

Typausdrücke τ werden gebildet aus:

1. *Typvariablen* (a, b, c, ...)
2. *Basistypen* (Bool, Int, Float, Char, ...)
3. *Typkonstruktoren* (dies sind Operationen auf Typen, die aus gegebenen Typen neue bilden), wie z.B. $\cdot \to \cdot$, `[·]`, `Tree` $\cdot$

Beispiel: $\tau =$ `Int` $\to$ `[Tree Int]`

Wir unterscheiden:

- **monomorpher Typ**: enthält keine Typvariablen
- **polymorpher Typ**: enthält Typvariablen

Eine (Typ-)**Substitution** ist eine Ersetzung von Typvariablen durch Typausdrücke. Notation:

$$\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$$

bezeichnet eine Abbildung Typvariablen $\to$ Typausdrücke mit der Eigenschaft

$$\sigma(a) = \begin{cases} \tau_i & \text{falls } a = a_i \\ a & \text{sonst} \end{cases}$$

Fortsetzung dieser Abbildung auf Typausdrücken:

$\sigma(b) = b$ (für alle Basistypen b)

$\sigma(k(\tau_1, \ldots, \tau_n)) = k(\sigma(\tau_n), \ldots, \sigma(\tau_n))$
(für alle n-stelligen Typkonstruktoren k und Typen $\tau_1, \ldots, \tau_n$)

Beispiel: Falls $\sigma = \{$`a` $\mapsto$ `Int`, `b` $\mapsto$ `Bool`$\}$, dann ist

$\sigma($`a` $\to$ `b` $\to$ `a`$)$ = `Int` $\to$ `Bool` $\to$ `Int`

Polymorphismus von Funktionen können wir nun wie folgt charakterisieren:

> Falls eine Funktion den (polymorphen) Typ $\tau_1 \to \tau_2$ hat und σ eine beliebige Substitution ist, dann hat sie auch den speziellen Typ $\sigma(\tau_1 \to \tau_2)$. Den spezielleren Typ bezeichnet man auch als **Typinstanz**.

[1] Tatsächlich basiert Haskell auf einer Verallgemeinerung des Hindley/Milner-Typsystems mit „Typklassen“, die auch die Beschreibung von Überladung erlauben. Eine Präzisierung von Typklassen ist allerdings wesentlich aufwändiger. Daher beschränken wir uns hier auf ein polymorphes Typsystem ohne Typklassen.

Beispiel:

```
length :: [a]  → Int
```

hat auch die Typen

```
length :: [Int]   → Int   (σ = {a ↦ Int})
length :: [Char]  → Int   (σ = {a ↦ Char})
```

Es gilt in dem von uns betrachteten Typsystem die folgende wichtige Einschränkung:

> Eine Typinstanzbildung ist bei Parametern innerhalb der Definition derselben Funktion nicht erlaubt.

Beispiel:

```
f :: (a  → a)  → (a  → a)
f g =      g       g    ist unzulässig!
       (b→b)→(b→b)  b→b
```

Ursache für diese Einschränkung: die Typinferenz wäre sonst unentscheidbar

Daher müssen wir Funktionen/Parameter *mit* und *ohne* Instanzbildung unterscheiden. Dazu definieren wir:

Ein **Typschema** hat die Form

$$\forall a_1 \dots a_n : \tau$$

wobei $a_1, \dots, a_n$ Typvariablen sind und τ ein Typausdruck ist.
Beachte: für $n = 0$ ist jedes Typschema auch ein Typausdruck

Der Typausdruck $\sigma(\tau)$ ist eine **generische Instanz** des Typschemas $\forall a_1 \dots a_n : \tau$ falls $\sigma = \{a_1 \mapsto \tau_1, \dots, a_n \mapsto \tau_n\}$ eine Typsubstitution ist.

Vordefinierte Funktionen (d.h. alle Funktionen, deren Definition festliegt) haben Typschemata, während die zu definierenden Funktionen Typausdrücke haben.

Vorgehen bei Typprüfung:

1. Rate für die zu definierenden Funktionen und deren Parameter Typausdrücke und prüfe alle Regeln (s.u.) für diese.
2. Bei Erfolg: Fasse diese Funktionen von nun an als vordefiniert auf, d.h. abstrahiere die Typvariablen zu einem Typschema.

Voraussetzung: Parameter in verschiedenen Regeln haben verschiedene Namen (dies können wir immer durch Umbenennung erreichen).

Typannahme A: Zuordnung Namen → Typschemata

Typprüfung: Beweisen von Aussagen der Form $A \vdash e :: \tau$ („unter der Typannahme A hat der Ausdruck e den Typ τ“)

Beweis durch das folgende *Inferenzsystem*:

(lese $\frac{P_1 \ldots P_n}{Q}$ so: falls $P_1 \ldots P_n$ richtig ist, dann auch Q)

Axiom $\quad \dfrac{}{A \vdash x :: \tau}$ falls τ generische Instanz von $A(x)$

Applikation $\quad \dfrac{A \vdash e_1 :: \tau_1 \to \tau_2, \quad A \vdash e_2 :: \tau_1}{A \vdash e_1\ e_2 :: \tau_2}$

Abstraktion $\quad \dfrac{A[x \mapsto \tau] \vdash e :: \tau'}{A \vdash \backslash x \to e :: \tau \to \tau'}$ wobei τ Typausdruck

$$A[x \mapsto \tau](y) = \begin{cases} \tau & \text{, falls } y = x \\ A(y) & \text{, sonst} \end{cases}$$

Bedingung $\quad \dfrac{A \vdash e_1 :: \texttt{Bool}, \quad A \vdash e_2 :: \tau, \quad A \vdash e_3 :: \tau}{A \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: \tau}$

Eine Gleichung $f\ t_1 \ldots t_n = e$ ist **typkorrekt** bzgl. A genau dann wenn $A(f)$ und $A(x)$ (für alle Variablen x in $t_1, \ldots, t_n$) Typausdrücke sind und $A \vdash f t_1 \ldots t_n :: \tau$ und $A \vdash e :: \tau$ für einen Typausdruck τ ableitbar ist.

Falls alle Gleichungen für alle zu definierenden Funktionen aus A typkorrekt sind, abstrahiere deren Typen zu Typschemata (durch Quantifizierung der Typvariablen).

Beispiel: Es seien die folgenden vordefinierten Funktionen gegeben:

$$\begin{array}{lcl} A(\texttt{+}) & = & \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\ A(\texttt{[]}) & = & \forall a.[a] \\ A(\texttt{:}) & = & \forall a.a \to [a] \to [a] \\ A(\texttt{not}) & = & \texttt{Bool} \to \texttt{Bool} \end{array}$$

Die Funktion `twice` sei wie folgt definiert:

```
twice f x = f (f x)
```

Wir erweitern unsere Typannahme um folgende Typausdrücke für `twice`, `f` und `x`:

$$\begin{array}{lcl} A(\texttt{twice}) & = & (a \to a) \to a \to a \\ A(\texttt{f}) & = & a \to a \\ A(\texttt{x}) & = & a \end{array}$$

Damit können wir die Typkorrektheit wie folgt nachweisen:

Typ der linken Seite:

$$\dfrac{\dfrac{\dfrac{}{A \vdash \texttt{twice} :: (a \to a) \to a \to a} \quad \dfrac{}{A \vdash \texttt{f} :: a \to a}}{A \vdash \texttt{twice f} :: a \to a} \quad \dfrac{}{A \vdash \texttt{x} : a}}{A \vdash \texttt{twice f x} :: a}$$

Typ der rechten Seite:

$$\dfrac{\dfrac{}{A \vdash \texttt{f} : a \to a} \quad \dfrac{\dfrac{}{A \vdash \texttt{f} :: a \to a} \quad \dfrac{}{A \vdash \texttt{x} :: a}}{A \vdash (\texttt{f x}) :: a}}{A \vdash \texttt{f (f x)} :: a}$$

Hieraus folgt, dass die Gleichung typkorrekt ist.

Damit können wir den geratenen Typ zu einem Typschema abstrahieren:

```
twice :: ∀a : (a → a) → a → a
```

Nun ist die polymorphe Anwendung von `twice` möglich, sodass die folgenden Ausdrücke typkorrekt sind:

```
twice (+2) 3
twice not True
```

Beachte: wenn in Haskell ein Funktionstyp definiert wird, dann wird dies implizit immer als Typschema interpretiert, bei dem die darin vorkommenden Typvariablen allquantifiziert sind. Somit entspricht das obige Typschema für `twice` der Typdeklaration

```
twice :: (a → a) → a → a
```

in Haskell.

τ heißt **allgemeinster Typ** eines Objektes (Funktion), falls

- τ ein korrekter Typ ist, und
- ist τ' ein korrekter Typ für dasselbe Objekt, dann gilt $\tau' = \sigma(\tau)$ für eine Typsubstitution σ.

Beispiel:

```
loop 0 = loop 0
```

Allgemeinster Typ:

```
loop:: Int → a
0 :: Int
```

Mit dieser Typananahme ist die Gleichung für `loop` typkorrekt:

$$\dfrac{\dfrac{}{A \vdash \texttt{loop} :: \texttt{Int} \to a} \quad \dfrac{}{A \vdash 0 :: \texttt{Int}}}{A \vdash \texttt{loop } 0 :: a}$$

⇒ Beide Gleichungsseiten haben Typ a

Der Typ `Int → a` deutet auf Programmierfehler hin, da die Funktion beliebige Ergebnisse erzeugen kann, was aber nicht möglich ist.

Die **Typprüfung für mehrere Funktionen** erfolgt analog, wobei wir nacheinander die Funktionen typisieren und dann deren Typausdrücke zu einem Typschemata abstrahieren. Beispiel:

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

f xs ys = length xs + length ys
```

Vordefiniert seien die folgenden Symbole:

```
[]   :: ∀ a. [a]
(:)  :: ∀ a. a→ [a] → [a]
0, 1 :: Int
(+)  :: Int → Int → Int
```

Typannahme:

```
length :: [a] → Int
x      :: a
xs     :: [a]
```

Wir können zeigen, dass unter dieser Typannahme die Regeln für `length` wohlgetypt sind. Damit können wir nun `length` mit folgendem Typschema als vordefiniert annehmen:

```
length:: ∀a. [a]  →Int
```

Nun wollen wir die Typkorrektheit von `f` prüfen. Dazu erweitern wir unsere Typannahme wie folgt:

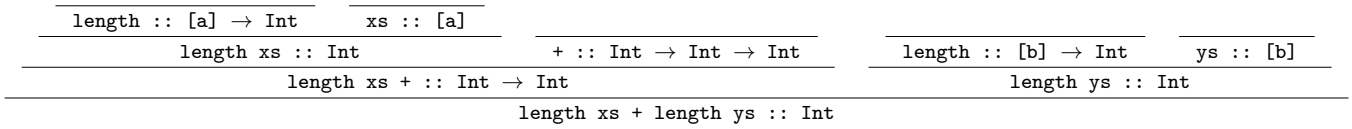
```
f  :: [a] → [b] → Int
xs :: [a]
ys :: [b]
```

Mit dieser Typannahme erhalten wir folgende Ableitung für den Typ der linken Seite der Definition von `f` (in den folgenden Ableitungen lassen wir „$A \vdash$“ weg):

$$\dfrac{\dfrac{\dfrac{}{\texttt{f :: [a] → [b] → Int}} \quad \dfrac{}{\texttt{xs :: [a]}}}{\texttt{f xs :: [b] → Int}} \quad \dfrac{}{\texttt{ys :: [b]}}}{\texttt{f xs ys :: Int}}$$

In ähnlicher Weise erhalten wir eine Ableitung für den Typ der rechten Seite der Definition von `f` (hier sollte man beachten, dass hier zwei verschiedene generische Instanzen

des Typschemas von `length` verwendet werden):



Somit erhalten wir das folgende Typschema für `f`:

```
f :: ∀a,b : [a] → [b] → Int
```

Daher ist

```
f [0,1] "Hallo"                                                    (*)
```

wohlgetypt (mit der generischen Instanz $\{\texttt{a} \mapsto \texttt{Int}, \texttt{b} \mapsto \texttt{Char}\}$).

Ohne Typschemabildung für `length` würden wir für `f` den Typ `[a] → [a] → Int` erhalten, der zu speziell ist, d.h. mit dem der Ausdruck `(*)` nicht typkorrekt wäre.

Aus diesem Verfahren ergeben sich unmittelbar die folgenden Fragen zur praktischen Anwendung:

- Wie findet man allgemeinste Typen?
- Wie kann man Typannahmen geeignet raten?

Diese Fragen sollen im nächsten Kapitel beantwortet werden.