5.4 Funktionen höherer Ordnung

Beispiel 5.16 (Insertion Sort)**.** Wir wollen eine Zahlenliste mittels „Insertion Sort“ sortieren, d. h. wir sortieren durch fortgesetztes sortiertes Einfügen. Hierzu definieren wir zunächst das sortierte Einfügen eines Elementes in eine schon sortierte Zahlenliste:

```
insert x []     = [x]
insert x (y:ys) = if x < y then x : y : ys
                           else y : insert x ys
```

Damit können wir nun das Sortieren durch Einfügen einfach definieren:

```
isort []     = []
isort (x:xs) = insert x (isort xs)

> isort [3,1,2]  ⇝ [1,2,3]
```

Der Nachteil dieser Lösung ist, dass sie zu speziell ist, denn diese implementiert z.B. nur

- das Sortieren von *Zahlen*listen,
- *aufsteigendes* Sortieren.

Um absteigendes Sortieren zu implementieren, müssten wir wie folgt vorgehen: Kopiere den Programmcode und ersetze „`x < y`“ durch „`x > y`“

- gut bei Bezahlung nach “lines of code”
- schlecht aus SW-technischer Sicht

Als Lösung dieses Problems parametrisieren wir `isort` über ein Vergleichsprädikat.

Definition 5.9 (Funktion höherer Ordnung)**.** *Eine **Funktion höherer Ordnung** ist eine Funktion, die eine andere Funktion als Parameter oder Ergebnis hat.*

In unserem Beispiel könnte `isort` auch eine Funktion höherer Ordnung sein, die zwei Parameter hat: ein Vergleichsprädikat `p` und eine zu sortierende Liste `xs`:

```
isort p []     = []
isort p (x:xs) = insert x (isort p xs)
  where insert x []     = [x]
        insert x (y:ys) = if p x y then x : y : ys
                                   else y : insert x ys
```

Nun können wir `isort` flexibel einsetzen:

```
isort (<) [3,1,2]  ⇝ [1,2,3]
isort (>) [3,1,2]  ⇝ [3,2,1]
```

Weitere nützliche Funktionen höherer Ordnung:

- Transformation einer Liste durch Anwendung einer Funktion auf jedes Listenelement:

  ```
  map _ []     = []
  map f (x:xs) = f x : map f xs
  ```

  Anwendung:

  ```
  map square [1,2,3]  ⇝ [1,4,9]
  map inc    [1,2,3]  ⇝ [2,3,4]
  ```

  (mit `inc x = x+1`)

  Falls `inc` nur einmal benutzt wird, ist es eigentlich überflüssig, diese Funktion explizit zu definieren. Daher kann man alternativ auch **anonyme Funktionen** bzw. **λ-Abstraktionen** benutzen:

  ```
  map (\x → x+1) [1,2,3]}  ⇝ [2,3,4]
  ```

  Hierbei bezeichnet „`\x -> x + 1`“ eine Funktion mit `x` als Argument und Ergebnis `x + 1`. Daher ist

  ```
  inc x = x + 1
  ```

  äquivalent zu

  ```
  inc = \x → x + 1
  ```

  Eine weitere Alternative zur Notation von `inc`:

  ```
  inc = (+) 1
  ```

  - „`(+)`“ ist eine Funktion, die zwei Argumente erwartet (die Klammern sind notwendig, da „`+`“ ein Infixoperator ist).
  - „`(+) 1`“ ist eine Funktion, die noch ein Argument erwartet (**partielle Applikation**).

  Voraussetzung hierfür ist, dass „`(+)`“ nicht ein Paar von Zahlen erwartet, sondern erst eine Zahl und danach die andere Zahl. Beachte hierzu den Unterschied in den Typen:

  ```
  plus' (x, y) = x + y
  ```

Hier hat `plus'` den Typ `plus' :: (Int, Int) -> Int`

Wenn wir dagegen die „Curry-Schreibweise“ verwenden:

```
plus x y = x + y
```

Hier hat `plus` den Typ `plus :: Int -> (Int -> Int)`

Beachte: „`f x`“ steht für die Anwendung von `f` auf `x`. Die Applikation ist dabei immer linksassoziativ:

```
f x y  ≈ (f x) y
```

Somit:

- `plus` nimmt ein Argument (eine Zahl) und liefert eine Funktion, die ein weiteres Argument nimmt und dann die Summe liefert.
- „`(+) 1`“ ist eine Funktion, die ein Argument nimmt und dazu 1 addiert.

Die Curry-Schreibweise ist zunächst ungewohnt, da in vielen Programmiersprachen die Tupel-Schreibweise üblich ist. Sie ermöglicht aber durch partielle Applikation universeller einsetzbare Programme.

- Filtern bestimmter Elemente einer Liste:

```
filter _ []     = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise =     filter p xs
```

Mit dieser Definition hat `filter` den Typ

```
filter :: (a → Bool) → [a] → [a]
```

hierbei ist `a` eine Typvariable, die für einen beliebigen Typ (vgl. Kapitel 5.5) steht.

```
filter (\x → x < 3) [1,3,4,2]  ⇝ [1,2]
```

Es gibt eine spezielle Syntax für partielle Applikationen bei Infixoperation:

```
(3 <)   ≈ (\x → 3 < x)
(< 3)   ≈ (\x → x < 3)
```

Obiges Beispiel:

```
filter (< 3) [1,3,4,2]  ⇝ [1,2]
```

Anwendung: Quicksort:

```
qsort []     = []
```

```
qsort (x:xs) = qsort (filter (< x) xs)
            ++ x : qsort (filter (>= x) xs)
```

- Akkumulation von Listenelementen:

  `foldr` $\oplus$ z `[`x_1`,...,`x_n`]` $\rightsquigarrow$ $x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus z) \ldots))$

  Hierbei ist $\oplus$ eine binäre Verknüpfung und z kann als „neutrales“ Element (oder auch Startwert) der Verknüpfung aufgefasst werden.

  ```
  foldr :: (a → b → b) → b → [a] → b
  foldr _ z []     = z
  foldr f z (x:xs) = f x (foldr f z xs)
  ```

  Damit kann man die folgenden Funktionen einfach definieren:

  `sum = foldr (+) 0` (Summe aller Listenelemente)

  `prod = foldr (*) 1` (Produkt aller Listenelemente)

  `concat = foldr (++) []` (Konkatenation einer Liste von Listen, `++` ist die Listenkonkatenation)

  Produkt der ersten 10 positiven Zahlen: „`prod (take 10 (from 1))`“

Kombinator-Stil: Programmierung durch Kombination von Basisfunktionen (Programmiersprachen basierend hierauf: APL, FP)

Kontrollstrukturen als Funktionen höherer Ordnung

Eine while-Schleife zur Veränderung eines Wertes besteht aus den folgenden Komponenten:

- Bedingung (Prädikat für den Wert)
- Transformation bisheriger Wert $\mapsto$ neuer Wert
- Anfangswert

Beispiel: kleinste Zweierpotenz größer als 10000:

```
x := 1;           -- Anfangswert
while x <= 10000  -- Bedingung
do x := 2 * x     -- Transformation
```

Die Kontrollstruktur der while-Schleife können wir auch als Funktion höherer Ordnung definieren:

```
while :: (a → Bool) → (a → a) → a → a
while p f x | p x       = while p f (f x)
            | otherwise = x
```

Obiges Beispiel: „`while (<= 10000) (2 *) 1`“ $\rightsquigarrow$ `16384`

Anmerkungen:

- neue Kontrollstrukturen können ohne Spracherweiterung eingeführt werden (z.B. `until` ist analog in Haskell vordefiniert)
- einfache Basissprache
- flexible anwendungsorientierte Erweiterungen

5.5 Typsysteme

Wir betrachten zunächst noch einmal einige Begriffe aus Kapitel 2.3.

- **Statisch getypte Programmiersprache** (manchmal auch streng getypte Programmiersprache): Jeder Bezeichner hat einen Typ, und dieser steht zur Compilezeit fest.
- Die **Semantik eines Typs** ist dabei die Menge möglicher Werte.

Beispiel 5.17 (Typen)**.**

Einige Beispiele für Typbezeichnungen und ihre Bedeutung:

```
Int
```

Dies bezeichnet eine (endliche Teil-)Menge der ganzen Zahlen.

```
Bool
```

Dieser Typ bezeichnet die Menge {`True`, `False`}.

```
Int → Bool
```

Damit bezeichen wir die Menge aller Funktionen, die ganze Zahlen auf Wahrheitswerte abbilden.

- **Typkorrektheit**: Ein Ausdruck e ist **typkorrekt**/**wohlgetypt**, wenn
  - e den Typ τ hat und
  - falls die Auswertung von e den Wert v ergibt, dann gehört v zum Typ τ

Beispiel 5.18 (Typkorrektheit)**.** Der Ausdruck

```
3 + 6 * 7
```

hat den Typ `Int`. Dagegen ist der Ausdruck

```
3 + False
```

nicht typkorrekt, denn die Auswertung liefert keinen Wert oder `error`, was zu keinem Typ gehört. Ebenso ist der Ausdruck

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

nicht typkorrekt, aber der Ausdruck

```
map (+1) [1,3,4,5]
```

hat den Typ `[Int]`

Ein Programm heißt **typkorrekt**, wenn alle Ausdrücke, Funktionen, etc. typkorrekt sind.

Bei einer statisch getypten Programmiersprache gilt (beziehungsweise sollte gelten):

Zulässige Programme sind typkorrekt, d. h. es gibt keine Typfehler zur Laufzeit ("well-typed programs do not go wrong" (Milner, 1978)).

Die Vorteile statisch getypter Programmiersprachen sind:

- Programmiersicherheit
- Produktivitätsgewinn: Der Compiler meldet Typfehler (potentielle Laufzeitfehler)
- Laufzeiteffizienz: keine Typprüfung zur Laufzeit erforderlich
- Dokumentation: Typen $\approx$ partielle Spezifikation

Die Nachteile statisch getypter Programmiersprachen sind:

- Die Typkorrektheit ist im Allgemeinen unentscheidbar, daher ist eine Einschränkung der zulässigen Programme notwendig.
- Beispiel: Der Haskell-Ausdruck

  ```
  map (+1) (take 2 [1,2,True])
  ```

  ist typkorrekt, aber im Allg. unzulässig in einer statisch getypten Programmiersprache.

Da aber die Vorteile die Nachteile überwiegen, sind viele moderne Programmiersprachen statisch getypt.

Kriterien für gute Typsysteme sind:

- Sicherheit (wohlgetypt $\Rightarrow$ keine Laufzeitfehler)
- Flexibilität (möglichst wenig Einschränkungen)
- Benutzung (Typangaben weglassen, falls diese nicht notwendig sind)

Am besten wird dies erreicht in funktionalen Programmiersprachen durch

- Polymorphismus ($\rightsquigarrow$ Flexibilität)
- Typinferenz ($\rightsquigarrow$ Benutzung)

5.5.1 Polymorphismus

Polymorphismus bedeutet, dass Objekte (Funktionen) mehrere Typen haben können. Hierbei unterscheiden wir zwei Arten von Polymorphismus:

Ad-hoc-Polymorphismus: Hier können Objekte ein unterschiedliches Verhalten auf unterschiedlichen Typen haben (üblich: Overloading).

Ein Beispiel für den Ad-hoc-Polymorphismus ist die Operation „+“ in Java:

`1+2`: Addition auf ganzen Zahlen

`"ab"+"cd"`: Konkatenation von Strings

Parametrischer Polymorphismus: Hier haben polymorphe Objekte das gleiche Verhalten auf allen Typen. Typischerweise enthalten Typen hierbei Typparameter (vgl. Kapitel 2.2.6: parametrisierte ADTs).

Ein Beispiel für den parametrischen Polymorphismus ist die Definition der Länge einer Liste in Haskell:

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

Der Typ von `length` ist

```
length :: [a] → Int
```

Hierbei ist `a` ein **Typparameter**, der bei der Anwendung von `length` durch einen beliebigen anderen Typ ersetzt werden kann.

⇒ `length` ist anwendbar auf `[Int]`, `[(Float,Bool)]`,...

⇒ „`length [0,1] + length [True,False]`“ ist typkorrekt.

Der parametrische Polymorphismus ist

- nicht vorhanden in vielen, insbesondere ältern imperativen Programmiersprachen (wie Pascal, Modula, Java (erst ab Version 5), ...),
- besonders wichtig bei Funktionen höherer Ordnung (vgl. die Typen von `filter`, `foldr`), mit denen man allgemeine Programmiermuster realisieren will.

5.5.2 Typinferenz und Typkorrektheit

Der Begriff **Typinferenz** bezeichnet die Herleitung von (möglichst allgemeinen) Typen für Objekte, so dass das Programm typkorrekt ist. Dies beinhaltet zwei Aspekte:

Herleitung der Parametertypen: Falls

```
f :: Int → Bool → Int
f x y = ...
```

dann hat `x` den Typ `Int` und `y` hat den Typ `Bool`.

Herleitung von Funktionstypen: In diesem Fall muss man nur die Regeln für `f` hinschreiben und daraus wird automatisch der Typ von `f` hergeleitet. Dies ist besonders angenehm und sinnvoll bei lokalen Deklarationen.

Ein **Typsystem**

- definiert die Sprache der Typen
- definiert die Eigenschaft „Wohlgetyptheit“ (die im Allgemeinen eine konstruktive Einschränkung von „typkorrekt“ ist),
- legt evtl. auch die Typinferenz fest (dann gibt es oft weitere Einschränkungen).

Im Folgenden betrachten wir ein Typsystem wie es von Hindley/Milner vorgeschlagen wurde (Damas and Milner, 1982) (man beachte, dass Haskell ein komplexeres Typsystem mit Typklassen hat). Wir definieren dabei nur die Wohlgetyptheit, nicht aber die Typinferenz. Dies wird z. B. in der Vorlesung „Deklarative Programmierung“ behandelt.

Zunächst definieren wir dazu die **Sprache der parametrisierten Typen**, also den Aufbau der **Typausdrücke**.

Definition 5.10 (Typausdruck)**.** *Ein* Typausdruck τ *ist*

- *eine* Typvariable a*, oder*
- *ein* Basistyp *wie* `Bool`*,* `Int`*,* `Float`*,* `Char`*, . . . , oder*
- *die (vollständige)* Anwendung eines Typkonstruktors TC auf Typausdrücke

$$TC\ \tau_1\ \ldots\ \tau_n \quad (n \geq 0)$$

wobei TC ein ***Typkonstruktor*** *und $\tau_1, \ldots, \tau_n$ Typausdrücke sind.*

In der Programmiersprache Haskell gilt zudem Folgendes:

- Typvariablen beginnen mit einem Kleinbuchstaben.
- Typkonstruktoren beginnen mit einem Großbuchstaben.
- Typkonstruktoren werden durch `data`- bzw. `newtype`-Deklarationen eingeführt.
- Für häufige Typkonstruktoren gibt es eine eingebaute Syntax:
  - Für Listen `[`τ`]` statt `List` τ
  - Für Tupel `(`τ_1, τ_2`)` statt `Tuple` τ_1 τ_2

– Für Funktionen τ_1 `->` τ_2 statt `Func` τ_1 τ_2

In diesem Zusammenhang bedeutet parametrischer Polymorphismus, dass Substitutionen auf Typvariablen zugelassen werden (**Typsubstitution**), analog zu Substitutionen auf Ausdrücken (vgl. Kapitel 5.3). Somit hat die Funktion

```
length :: [a] → Int
```

auch die Typen:

`[Int] → Int`	Typsubstitution: $\{$`a` $\mapsto$ `Int`$\}$
`[(Char, Float)] → Int`	Typsubstitution: $\{$`a` $\mapsto$ `(Char, Float)`$\}$

Damit die Eigenschaft „wohlgetypt“ entscheidbar ist, sind jedoch einige Einschränkungen notwendig. In dem hier betrachteten Typsystem ist die wesentliche Einschränkung, dass die Bildung von Typinstanzen (d. h. die Anwendung von Typsubstitutionen) innerhalb von Funktionsregeln bei Typen von Funktionsparametern *nicht* erlaubt ist.

Beispiel 5.19 (Typinstanzen von Funktionsparametern)**.** Wir betrachten die Funktion

```
f :: (a → a) → (a → a)
f g = g g
```

Diese Funktion ist prinzipiell typkorrekt, falls im Rumpf „`g g`“

- das erste `g` den Typ „`(a → a) -> (a → a)`“ hat (d. h. die Typsubstitution $\{$`a` $\mapsto$ `(a → a)`$\}$ angewendet wird), und
- das zweite `g` den Typ „`(a → a)`“ hat.

Hierzu müsste also der Typ des Parameters `g` innerhalb der Regel auf verschiedene Arten substituiert werden. Dies ist allerdings aus Entscheidbarkeitsgründen nicht zugelassen, d. h. in Haskell ist diese Funktion *nicht wohlgetypt*.

Um dies genau zu beschreiben, erweitern wir die Typsprache um **Typschemata**.

Definition 5.11. *Ein* Typschema *hat die folgende Form*

$$TypeScheme := \forall a_1, \ldots, a_n.\tau \quad (n \geq 0)$$

wobei τ ein Typausdruck und $a_1, \ldots, a_n$ Typvariablen aus τ sind. Damit entspricht das triviale Typschema $\forall.\tau$ dem Typ τ.

Beispiel 5.20 (Typschema)**.**

$\forall a$. `[`a`]` →`Int`

ist ein Typschema für die Funktion `length`.

Hierbei sollte beachtet werden, dass in Haskell ein Funktionstyp wie

```
f :: τ
```

immer für das Typschema

```
f :: ∀a_1, ..., a_n.τ
```

steht, wobei $a_1, \ldots, a_n$ alle Typvariablen aus τ sind.

Wichtig ist nun, dass von Typschemata Instanzen gebildet werden können:

Definition 5.12 (Generische Instanz)**.** *τ ist eine **generische Instanz** des Typschemas $\forall a_1, \ldots, a_n.\tau'$ $(n \geq 0)$, falls eine Typsubstitution $\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$ existiert mit $\tau = \sigma(\tau')$.*

5.5.3 Wohlgetyptheit von Ausdrücken

Nun ist die Wohlgetyptheit mittels eines Inferenzsystems einfach definierbar. Hierzu betrachten wir eine **Typannahme** als Zuordnung A von Namen zu Typschemata. Das Raten einer korrekten Typannahme ist gerade die Aufgabe der Typinferenz, die wir hier nicht weiter betrachten.

Dann hat die Basissprache des Inferenzsystems zur Typkorrektheit die struktur $A \vdash e :: \tau$, wobei A eine Typannahme, e ein Ausdruck und τ ein Typausdruck ist. Das Inferenzsystem zur Typkorrektheit besteht aus folgenden Regeln:

Variable

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

falls τ eine generische Instanz von $A(x)$ ist. Dies drückt genau den parametrischen Polymorphismus aus!.

Applikation

$$\frac{A \vdash e_1 :: \tau_1 \to \tau_2 \quad A \vdash e_2 :: \tau_1}{A \vdash e_1\ e_2 :: \tau_2}$$

Abstraktion

$$\frac{A[x/\tau] \vdash e :: \tau'}{A \vdash \backslash x \to e :: \tau \to \tau'}$$

wobei τ ein *Typausdruck* ist.

Bedingung

$$\frac{A \vdash e_1 :: \mathit{Bool} \quad A \vdash e_2 :: \tau \quad A \vdash e_3 :: \tau}{A \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: \tau}$$

`let`-Deklaration

$$\frac{A[x/\tau] \vdash e_1 :: \tau \quad A[x/\sigma] \vdash e_2 :: \tau'}{A \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 :: \tau'}$$

Hierbei ist τ ein *Typausdruck* und $\sigma = \forall a_1, \ldots, a_n.\tau$, wobei jedes a_i in τ vorkommt, aber nicht in A.
Dies spezifiziert die polymorphe Verwendung lokaler Funktionen.

Anmerkungen

Eine Gleichung $f\ t_1 \ldots t_n = e$ ist wohlgetypt bzgl. A genau dann wenn:

1. $A(f) = \forall a_1 \ldots a_m.\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau$
2. Es existieren *Typausdrücke* $\tau'_1, \ldots, \tau'_k$ für die Variablen $x_1, \ldots, x_k$ in $t_1, \ldots, t_n$ mit folgender Eigenschaft: Sei

   $$A' = A[x_1/\tau'_1, \ldots, x_k/\tau'_k]$$

   dann sind $A' \vdash t_i :: \tau_i$ $(i = 1, \ldots, n)$ und $A' \vdash e :: \tau$ ableitbar.

Gleichungen mit Bedingungen, d. h. Gleichungen der Form $l \mid b \ = \ r$ können wir hier auffassen als

l `=` `if` b `then` r `else` $\bot$

mit $A(\bot) = \forall a.a$.

Ein Programm ist wohlgetypt, falls alle Gleichungen bzgl. einer festen Typannahme A wohlgetypt sind.

Die Konsequenz von Punkt 2 ist, dass, wie schon oben erwähnt, bestimmte typkorrekte Ausdrücke als nicht typisierbar betrachtet werden. Betrachten wir dazu die folgende Funktion:

```
funsum f l1 l2 = f l1 + f l2
```

Dann ist der Ausdruck

```
(funsum length [1,2] "abc")
```

nicht wohlgetypt, obwohl dessen Auswertung, wenn man Typangaben ignoriert, den Wert 5 liefert. Tatsächlich könnte dieser Ausdruck als typkorrekt aufgefasst werden, falls die Funktion `funsum` den Typ

`funsum ::` $\forall a, b.(\forall c.[c] \to$ `Int`$) \to [a] \to [b] \to$ `Int`

hätte, aber im Hindley/Milner-Typsystem sind Typschemata als Typen von Parametern nicht zulässig. Typen wie diese werden auch als Rank-2-Typen bezeichnet, weil hier

innerhalb von Typausdrücken eines Typschemas wieder ein Typschema steht, d.h. Typquantifikation geschachtelt auftritt.

Haskell bietet aber eine Reihe von Erweiterungen des Hindley/Milner-Typsystems, mit dem man z.B. auch Rank-2-Typen angeben (aber nicht inferieren) kann. Unser Beispiel kann mit der Spracherweiterung `Rank2Types` wie folgt korrekt umgesetzt werden:

```
{-# LANGUAGE Rank2Types #-}

funsum :: (forall a . [a] → Int) → [b] → [c] → Int
funsum f l1 l2 = f l1 + f l2

main = funsum length [1,2] "Hello"
```

Beispiel 5.21 (Wohlgetyptheit)**.** Die Funktion

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

ist wohlgetypt mit der Typannahme $A(\texttt{twice}) = \forall a.(a \to a) \to a \to a$ und geeigneten Annahmen für `f` und `x` ($\to$ Übung).

Die Aufgabe der **Typinferenz** ist es, eine geeignete Typannahme zu raten bzw. konstruktiv zu finden. Dies ist

- unproblematisch für Parameter
- schwieriger für Typschemata (Typen der Funktionen), daher fordert man dann weitere Einschränkungen, die allerdings nur bei komplexen Beispielen relevant werden.

Beispiel 5.22 (Notwendigkeit der Typannahme)**.** Wir betrachten das folgende Programm:

```
f :: [a] → [a]
f x = if length x == 0 then fst (g x x) else x

g :: [a] → [b] → ([a], [b])
g x y = (f x, f y)

h = g [3,4] [True, False]

fst (x,y) = x   -- first
```

Bezüglich der Wohlgetyptheit stellen wir Folgendes fest:

- Mit Typangaben für `f` und `g` ist das Beispiel wohlgetypt.
- Ohne Typangaben für `f` und `g` ist das Beispiel nicht wohlgetypt, da der Typ

```
g :: [a] → [a] → ([a], [a])
```

hergeleitet wird.

Das **Prinzip bei der Typinferenz** ist das Folgende: Innerhalb von rekursiven Aufrufen/ Definitionen haben Funktionen kein Typschema, sondern nur einen Typ, anderenfalls wäre die Typinferenz unentscheidbar.