

Abschnitt 3.3

Unterabschnitt 3.3.1

Polymorphie und Funktionen höherer Ordnung

Typisierung

Abstraktion mittels Polymorphie und Funktionen höherer Ordnung

Überblick:

- Grundbegriffe der Typisierung
- Polymorphie als Abstraktionsmittel
- Typsysteme und Typinferenz
- Einführung in Funktionen höherer Ordnung
- Wichtige Funktionen höherer Ordnung

Typisierung

Inhalte:

- Was ist ein Typ?
- Ziele der Typisierung
- Polymorphie und parametrische Typen
- Typsystem von Haskell und Typinferenz

Fast alle modernen Spezifikations- und Programmiersprachen besitzen ein Typsystem.

Was ist ein Typ?

Ein Typ beschreibt Eigenschaften von Elementen der Modellierung oder Programmierung:

- Elementare Typen stehen häufig für die Eigenschaft, zu einer bestimmten Wertemenge zu gehören (Bool, Int, Char, String, ...).
- Zusammengesetzte Typen beschreiben die genaue Struktur ihrer Elemente (Tupel-, Listen- und Funktionstypen).

©2010

TU Kaiserslautern

445

Was ist ein Typ? (2)

- Parametrische Typen beschreiben bestimmte Eigenschaften und lassen andere offen; z.B.:
 - Elemente vom Typ `[a]` sind homogene Listen; man kann also `null`, `head`, `tail` anwenden. Offen bleibt z.B. der Ergebnistyp von `head`.
 - Elemente vom Typ `(a, [a])` sind Paare, so dass die Funktionen für Paare angewendet werden können. Außerdem besitzen sie die Eigenschaft, dass die zweite Komponente immer eine Liste ist, deren Elemente vom selben Typ sind wie die erste Komponente:

```
somefun :: ( a, [a] ) -> [a]
somefun p = let (fstc, sndc) = p
              in fstc : sndc
```

©2010

TU Kaiserslautern

446

Ziele der Typisierung

Die Typisierung verfolgt drei Ziele:

- Automatische Erkennung von Programmierfehlern (durch Übersetzer, Interpreter);
- Verbessern der Lesbarkeit von Programmen;
- Ermöglichen effizienterer Implementierungen.

Wir konzentrieren uns hier auf das erste Ziel.

©2010

TU Kaiserslautern

447

Zentrale Idee:

- Für jeden Ausdruck und jede Funktion wird ein Typ festgelegt.
- Prüfe, ob die Typen der aktuellen Parameterausdrücke mit der Signatur der angewendeten Funktion übereinstimmen.

Beispiele: (Typprüfung von Ausdrücken)

```
f :: Int -> Int
```

dann sind:

```
f 7, f (head [1,2,3]), f (f 78)+ 9 typkorrekt;
f True, [f,5.6], f head nicht typkorrekt.
```

©2010

TU Kaiserslautern

448

Bemerkung:

Typisierung war lange Zeit nicht unumstritten.
Hauptgegenargumente sind:

- zusätzlicher Schreib- und Entwurfsaufwand
- Einschränkung der Freiheit:
 - inhomogene Listen
 - Nutzen der Repräsentation von Daten im Rechner

©2010

TU Kaiserslautern

449

Beispiel:

Es gibt viele Programme, die nicht typkorrekt sind, sich aber trotzdem zur Laufzeit gutartig verhalten; z.B.:

Aufgabe 1:

Schreibe eine Funktion frp:

- Eingabe: Liste von Paaren entweder vom Typ `(Bool, Int)` oder `(Bool, Float)`
- Zulässige Listen: Wenn 1. Komponente `True`, dann 2. Komponente vom Typ `Int`, sonst vom Typ `Float`
- Summiere die Listenelemente und liefere ein Paar mit beschriebener Eigenschaft.

©2010

TU Kaiserslautern

450

Beispiel: (2)

Realisierung in Haskell-Notation (kein Haskell-Programm, da nicht typkorrekt!):

```
frp :: ( Bool, ? ) -> ?
frp [ ]      = ( True, 0 )
frp ((True,n):xs) =
  case frp xs of
    ( True, k) -> ( True, k+n )
    ( False, q) -> ( False, q+(fromInteger n) )
frp ((False,r):xs) =
  case frp xs of
    ( True, k) -> ( False, (fromInteger k)+r )
    ( False, q) -> ( False, q+r )
```

©2010

TU Kaiserslautern

451

Beispiel: (3)

Aufgabe 2:

Schreibe eine Funktion,

- die ein n-Tupel ($n \geq 2$) nimmt und
- die erste Komponente des Tupels liefert.

Kann in Haskell nicht definiert werden:

```
arbitraryfst :: (a,b,...) -> a
arbitraryfst (n,...) = n
```

©2010

TU Kaiserslautern

452

Polymorphie und parametrische Typen

Programmierer möchten vom Typsystem nicht weiter eingeeengt werden als nötig. Ziel:

- mächtige und flexible Typsysteme
- insbesondere Polymorphie und Parametrisierung

Im Allg. bedeutet Polymorphie Vielgestaltigkeit.

In der Programmierung bezieht sich Polymorphie auf die Typisierung bzw. das Typsystem.

Begriffsklärung: (polymorphes Typsystem)

Das **Typsystem** einer Sprache *S* beschreibt,

- welche Typen es in *S* gibt bzw. wie neue Typen deklariert werden;
- wie den Ausdrücken von *S* ein Typ zugeordnet wird;
- welche Regeln typisierte Ausdrücke erfüllen müssen.

Ein Typsystem heißt **polymorph**, wenn Ausdrücke zu Werten bzw. Objekten unterschiedlichen Typs ausgewertet werden können.

Bemerkung:

- Man unterscheidet:
 - **Parametrische Polymorphie**
 - **Subtyp-Polymorphie** (vgl. Typisierung in Java)
- Oft spricht man im Zusammenhang mit der Überladung von Funktions- oder Operatorsymbolen von Ad-hoc-Polymorphie.

Beispiel:

Dem `+`-Operator könnte man in Haskell den Typ

`Int ->Int ->Int` oder `Float ->Float ->Float` geben.

Bemerkung: (2)

- In polymorphen Typsystemen gibt es meist eine Relation „ist_spezieller_als“ zwischen Typen *T1*, *T2*:
T1 heißt spezieller als *T2*, wenn die Eigenschaften, die *T1* garantiert, die Eigenschaften von *T2* implizieren (umgekehrt sagt man: *T2* ist allgemeiner als *T1*).

Beispiel:

Der Typ `[Int]` ist spezieller als der parametrische Typ `[a]`. Insbesondere gilt:

Jeder Wert vom Typ `[Int]` kann überall dort benutzt werden, wo ein Wert vom Typ `[a]` erwartet wird.

Typsystem von Haskell und Typinferenz

Typen werden in Haskell durch Typausdrücke beschrieben:

- Typkonstanten sind die Basisdatentypen: `Bool`, `Char`, `Int`, `Integer`, `Float`, `Double`
- Typvariablen: `a`, `meineTypvar`, `gTyp`
- Ausdrücke gebildet mit Typkonstruktoren:
 - Seien `TA`, `TA1`, `TA2`, `TA3`, ... Typausdrücke, dann sind die folgenden Ausdrücke auch Typausdrücke:
 - (`TA1`, `TA2`) Typ der Paare
 - (`TA1`, `TA2`, `TA3`) Typ der Triple
 - ...
 - [`TA`] Listentyp
 - `TA1` -> `TA2` Funktionstyp

Einem Typausdruck `TA` kann ein Typconstraint für Typvariablen `a` vorangestellt werden: (Eq `a`) => `TA`

©2010

TU Kaiserslautern

457

Beispiele: (Typausdrücke)

```
Int, Bool
a, b
(Int, Bool) , (Int, a) , (Int, a, b, a)
[Integer] , [a] , [(Float, b, a)]
Int -> Int , [a] -> [(Float, b, a)]
(Char -> Char) -> (Int -> Int)
(Eq a) => a -> [a] -> Bool
```

Hinweis:

-> ist rechtsassoziativ.

©2010

TU Kaiserslautern

458

Begriffserklärung: (Vergleich von Typen)

Seien `TA` und `TB` Typausdrücke und `var(TA)` die Menge der Typvariablen, die in `TA` vorkommen.

Eine *Variablensubstitution* ist eine Abbildung β von `var(TA)` auf die Menge der Typausdrücke.

Bezeichne TA_β den Typausdruck, den man aus `TA` erhält, wenn man alle Variablen `v` in `TA` konsistent durch $\beta(v)$ ersetzt.

TB ist spezieller als TA, wenn es eine *Variablensubstitution* gibt, so dass $TA_\beta = TB$.

`TA` und `TB` bezeichnen den gleichen Typ, wenn `TA` spezieller als `TB` ist und `TB` spezieller als `TA`.

©2010

TU Kaiserslautern

459

Beispiele:

```
[Int] ist spezieller als [a]
[a] ist spezieller als [b] und umgekehrt
a -> a ist spezieller als a -> b
([Int], b) und ([c], Bool) sind nicht vergleichbar, d.h. der erste
Ausdruck ist nicht spezieller als der zweite und der zweite nicht
spezieller als der erste.
(Eq a) => TA ist spezieller als TA
```

©2010

TU Kaiserslautern

460

Typen in Haskell

Bezeichne TE die Menge der Typausdrücke in Haskell.

- Die „ist_spezieller_als“ Relation auf $TE \times TE$ ist reflexiv und transitiv, aber nicht antisymmetrisch.
- Identifiziert man alle Typausdrücke, die den gleichen Typ bezeichnen, erhält man die Menge T der Typen. (T , $ist_spezieller_als$) ist eine partielle Ordnung.

Damit ist gesagt, was Typen in Haskell sind.

Bemerkung:

Das Typsystem von Haskell ist feiner als hier dargestellt (Typklassen und benutzerdefinierte Typconstraints).

©2010

TU Kaiserslautern

461

Beispiel:

```
foldplus :: (Num a) => [a] -> a
-- Addiert alle Elemente einer Liste.
-- Listenelemente muessen alle vom gleichen Zahltyp sein
foldplus [] = 0
foldplus (x:xs) = x + foldplus xs
```

©2010

TU Kaiserslautern

462

Typregeln in Haskell:

Wir nennen eine Ausdruck A *typannotiert*, wenn A und allen Teilausdrücken von A ein Typ zugeordnet ist.

Beispiel:

Die Typnotationen von `abs (-2)` ist
`((abs :: Int -> Int) (-2 :: Int)) :: Int`

Die Typregeln legen fest, wann ein typannotierter Ausdruck typkorrekt ist. In Haskell muss gelten:

- Die Typen der formalen Parameter müssen gleich den Typen der aktuellen Parameter sein.
- Bedingte Ausdrücke müssen korrekt typisiert sein.

©2010

TU Kaiserslautern

463

Begriffserklärung: (Typinferenz)

Typinferenz bedeutet das Ableiten der Typnotation für Ausdrücke aus den gegebenen Deklarationsinformationen.

Bemerkung:

In Programmiersprachen mit parametrischem Typsystem kann Typinferenz sehr komplex sein.

©2010

TU Kaiserslautern

464

Beispiele:

1. Leere Liste:

```
x = [ ]
```

Inferierter Typ für x :: [a]

2. Enthalten sein in Liste:

```
enthalten p1 [] = False
enthalten p1 (x:xs)
  | p1 == x = True
  | otherwise = enthalten p1 xs
```

Inferierter Typ für enthalten :: (Eq a) => a -> [a] -> Bool

©2010

TU Kaiserslautern

465

Beispiele: (2)

3. Einsortieren in geordnete Liste mit Vergleichsfunktion:

```
einsortieren vop p1 [] = [p1]
einsortieren vop p1 (x:xs)
  | p1 `vop` x = p1:x:xs
  | otherwise = x : (einsortieren vop p1 xs)
```

Inferierter Typ für einsortieren ::

```
(t -> t -> Bool) -> t -> [t] -> [t]
```

Bemerkung:

Bei der Typinferenz versucht man immer den allgemeinsten Typ herauszufinden.

©2010

TU Kaiserslautern

466

Parametrische markierte Binärbaumtypen:

1. Alle Markierungen sind vom gleichen Typ:

```
data BBAum a =
  Blatt a
  | Zweig a (BBAum a) (BBAum a)
```

2. Blatt- und Zweigmarkierungen sind möglicherweise von unterschiedlichen Typen:

```
data BBAum a b =
  Blatt a
  | Zweig b (BBAum a b) (BBAum a b)
```

©2010

TU Kaiserslautern

467

Parametrische markierte Binärbaumtypen: (2)

3. Und was passiert hier?

```
data BBAum a b =
  Blatt a
  | Zweig b (BBAum b a) (BBAum a b)
```

©2010

TU Kaiserslautern

468