III.5 Typen

Dienstag, 13. Januar 2015 16:00

Typen

Typen sind Mengen v. gleichartigen Werten.

Vordef. Typen:

Int, Bool, Char, ...

(Int, Bool),

[Int],

Int -> Int

(Int -> Int, [Int]),

...

Ziel: eigene neue Datenstrukturen

Typen

- (tyconstr $type_1$ ... $type_n$), $n \geq 0$

Bsp für Typkonstr.:
Int, Bool, ...
(0-stellig)
Strings mit Großbuchstaben
(wie Datenkonstr.)
Datenkonstr: True, False

and True False ✓
~~and True Bool~~ ↯

- [type]

d.h.: [...] ist ein 1-stelliger Typkonstruktor

Bsp: [Int], [[Int]]

- $type_1 \rightarrow type_2$

d.h. → ist ein 2-stelliger Typkonstr.

Bsp: [Int] → Int

- (ty_1, ..., ty_n), $n \geq 0$

(...) sind beliebig stelliger Typkonstr.

Bsp: (Int, [Bool], Int → Bool)

- Var

Typvariable,

nötig für parametrische Polymorphie

Polymorphie

Eine Fkt. kann auf Argumente versch. Typen angewendet werden.

- ad-hoc Polym.

vom Typ der Argumente hängt ab, welche Implementierung der Fkt. ausgeführt wird.

- parametr. Polymorphie

für versch. Typen v. Argumenten wird immer die gleiche Implementierung ausgeführt.

(≤ generische Typen in Java)

Typvariablen (z.B. a) können mit bel. Typ instantiiert werden.

Mehrfache Vorkommen der gleichen Typvar. in einem Typ müssen gleich instantiiert werden.

Bsp: app ist in Haskell unter dem Namen ++ vordefiniert.

[1,2] ++ [3,4,5] ergibt [1,2,3,4,5].

Anwendung polymorpher Funktionen:

Fkt. vom Typ

$type_1 \to type_2$

darf auf Argument vom Typ

type

angewendet werden, falls es eine Instantiierung σ der Typvariablen gibt, so dass

$\sigma(type_1) = \sigma(type)$

ist.

σ heißt allgemeinster Unifikator von $type_1$ und type.

(most general unifier, MGU)

Ergebnis der Funktionsanwendg. hat Typ

$\sigma(type_2)$.

Bsp: [True] ++ []

[Bool] [b]

Erwartet:

[a] [a]

Lösung:

$\sigma(a) = Bool$

$\sigma(b) = Bool$

Ergebnis hat Typ:

$\sigma([a]) =$

[Bool]

Weitere Beispiele zur Typinferenz

Bsp1 `f x = x : [x]`

b b [b]

Erwartet: a [a]

Typ von (:) :: a -> [a] -> [a]

[a]

Lösung: $\sigma(b) = a$ ↝ f :: a → [a]

$\sigma(b)$

Bsp 2: g x = x ++ x

b b b

Typ von ++:

(++) :: [a] → [a] → [a]

Erwartet [a] [a]

[a]

Lösung: $\sigma(b) = [a]$ ↝ g :: [a] → [a]

$\sigma(b)$

Bsp 3: h x = x : [1,2]

b b [Int]

(:) :: a → [a] → [a]

1 :: Int

2 :: Int

Erwartet a [a]

[a]

Vermutung:

h :: Int → [Int]

$\sigma(b)$ $\sigma([a])$

Lösung: $\sigma(a) = Int$, $\sigma(b) = Int$

Bsp 4: i x = \y → [x] : [y]

b c [b] [c]

Erwartet: a [a]

[a]

Lösung: $\sigma(a) = [b]$, $\sigma(c) = [b]$

Also: i :: b → [b] → [[b]]

$\sigma(b)$ $\sigma(c)$ $\sigma([a])$

Bsp 5 j x = x : x

b b b

Erwartet a [a]

[a]

← Dieses Unifikationsproblem ist unlösbar. Es ex. kein σ mit $\sigma(b) = \sigma(a) = \sigma([a])$

Topdecl: Deklarationen, die nur auf oberster Ebene möglich sind (nicht in lokalen Decl.) erlaubt and

Bsp Color, MyBool
Sind neue 0-stlge
Typkonstruktoren.
Syntax ≙ Regeln in EBNF
Sind wie Enum-Typen in
Java (endl. viele Objekte
in einem Datentyp).
Pattern Matching wie bei
vordef. Typen. Patterns ≙
Ausdrücke aus Variablen u.
Datenkonstruktoren
(wie Red, Yellow, Green).
Damit Werte auf Bildschirm
ausgegeben werden können
musses Fkt show geben,
die Werte in Strings
konvertiert.
1. Verbesserung:
Datentypen mit ∞
vielen Werten.
Nach jedem Datenkonstruk-
tor werden die Typen
der Argumente angegeben.
Zero :: Nats
Succ :: Nats → Nats
Objekte des Typs Nats:
Zero ≙ 0
Succ Zero ≙ 1
Succ (Succ Zero) ≙ 2
z.B.:
plus (SZ) (SZ)
wertet aus zu
S (S Z)
Datenkonstruktoren wie
Succ und Zero werden
nicht weiter ausgewertet.
Bsp:
mul :: Nats → Nats → Nats
mul Zero y = Zero

```
mul y Zero = Zero

inf :: Nats
inf = Succ inf

mul inf Zero =
mul (Succ inf) Zero =
      Zero
```

```
mul :: Int -> Int -> Int
mul 0 y = 0
mul y 0 = 0

inf :: Int
inf = inf + 1

mul inf 0 =
mul (inf + 1) 0 =
mul (inf + 1 + 1) 0 = ...
```

terminiert nicht

2. Verbesserung:

Definiere eigene Typkonstruktoren mit Stelligkeit > 0.

Wenn a ein Typ ist, dann ist List a auch ein Typ.

Bsp: List Int, List Nats, ...

2 Datenkonstruktoren

```
Nil :: List a
Cons :: a -> List a -> List a
```

entspricht

```
[]  :: [a]
(:) :: a -> [a] -> [a]
```

Auf analoge Weise

lassen sich Datenstruk-
turen für Bäume, Graphen
etc definieren.