III.5 Typen und Datenstrukturen

Freitag, 20. Januar 2017 17:00

Jeder Ausdruck hat einen Typ.

Beispiele für Typen: Bool, Int, Char, Float,
[Int], [[Int]],
Int -> Bool, [Int] -> [Bool], [Int -> Bool],
Int -> (Int -> Int), (Int -> Int) -> Int, ...
(Bool, Int), (Bool, Int -> Int, [Int]), ...

Man kann Typkonstruktoren auf Typen anwenden, um neue Typen zu bekommen.

- tyconstr type$_1$ ... type$_n$, wobei tyconstr ein n-stelliger Typkonstruktor

  Bsp. für vordefinierte Typkonstruktoren der Stelligkeit 0:
  Bool, Int, Char, Float, ...

  Typkonstruktoren sind in Haskell (normalerweise) Strings, die mit Großbuchstaben beginnen.
  Sehen also aus wie Datenkonstruktoren, aber bedeuten etwas anderes:

```
und :: Bool -> Bool -> Bool
und True y  = y
und _    _  = False
```

```
und :: True -> Bool -> Bool
und Bool y = y
```

- [type]

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

Typ der Listen, bei denen alle Argumente den Typ type haben.

- $type_1 \to type_2$

  d.h.: $\to$ ist vordef. 2-stelliger Typkonstruktor

  Typ der Funktionen von $type_1$ nach $type_2$

- $(type_1, \ldots, type_n)$ mit $n \geq 0$

  d.h. (...) ist vordef. Typkonstruktor beliebiger Stelligkeit

  Typ der n-stelligen Tupel, bei denen die 1te Komponente den Typ $type_1$ hat, ..., die n-te Komponente den Typ $type_n$ hat.

  $(type_1)$ ist dasselbe wie $type_1$.

- Var

Typvariable: nötig für parametrische Polymorphie

2 Arten von Polymorphie:

- ad hoc-Polymorphie:
  verschiedene Implementierungen einer Funktion. Es hängt von Typen d. Argumente ab, welche Implem.

ausgeführt wird.

- parametrische Polymorphie:
  Dieselbe Implementierg. einer Funktion wird für Argumente versch. Typen angewendet.

Typisch für OO-Sprachen: ad-hoc Polymorphie

Typisch für fkt. Sprachen: param. Polymorphie.

Aber: Viele moderne Sprachen haben beides (Java + Haskell).

Bsp: Man sollte len so implementieren, dass die gleiche Implementierung für alle Arten v. Listen verwendet werden kann.
Hier: a ist eine Typvar., kann mit bel. Typ instantiiert werden:

```
len [1,2,3]
len [True, False]
len [1, True]
```

Mehrfache Vorkommen der gleichen Typvar. in einem Typ

müssen gleich instantiiert werden.
Wenn man die Typdeklaration weglässt, dann inferiert Haskell automatisch den allgemeinsten Typ.

`[a] -> [a] -> [a]` ist verschieden von `[a] -> [b] -> [a]`

Allgemein darf man eine Funktion vom Typ $type_1 \to type_2$ auf Argument vom Typ $type$ anwenden, falls es eine Instantiierung σ ← sigma der Typvariablen gibt (eine "Substitution"), so dass $\sigma(type) = \sigma(type_1)$ ist.
Dann hat das Ergebnis den Typ $\sigma(type_2)$.

```
app [True] []
```

`[a]` ← von app erwartete Typen
`[a]` `[a]`
`[Bool]` `[b]` ← Typen der aktuellen Argumente

Man kann die aktuellen und die erwarteten Typen gleich machen:
$\sigma(a) = Bool$, $\sigma(b) = Bool$
Ergebnis hat also den Typ
$\sigma([a]) = [Bool]$

Weitere Beispiele zur Typberechnung

`[] :: [a]`

Weitere Beispiele zur Typberechnung

$[\,] :: [a]$

$(:) :: a \to [a] \to [a]$

$$f\ x = x : [x]$$

Expected types (above): x : a, $[x]$: $[a]$, $x : [x]$: $[a]$

Actual types (below): x : b, x : b, $[x]$: $[b]$

Man kann den erwarteten und den aktuellen Typ gleich machen durch $\sigma(b) = a$, $\sigma(a) = a$.

f hat also den Typ $\underbrace{a}_{\sigma(b)} \to \underbrace{[a]}_{\sigma([a])}$

$$g\ x = x \mathbin{++} x$$

Expected types (above): x : $[a]$, x : $[a]$, $x \mathbin{++} x$: $[a]$

Actual types (below): x : b, x : b, x : b

Die Funktion ++ entspricht "app" und ist in Haskell vordefiniert.

$(++) :: [a] \to [a] \to [a]$

Gilt mit $\sigma(b) = [a]$, $\sigma(a) = a$.

Also: $g :: \underbrace{[a]}_{\sigma(b)} \to \underbrace{[a]}_{\sigma([a])}$

$$h\ x = x : [1, 2]$$

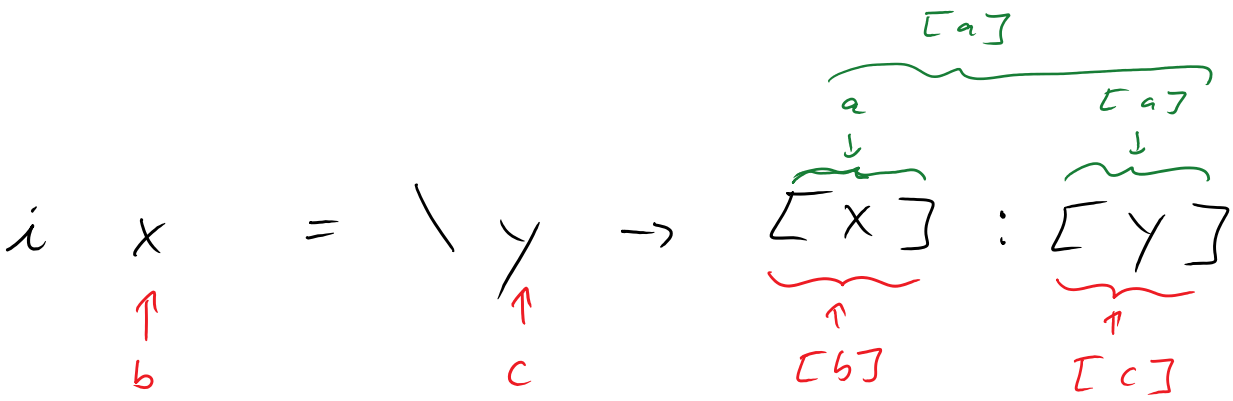
Expected types (above): x : a, $[1, 2]$: $[a]$, $x : [1,2]$: $[a]$

Actual types (below): x : b, x : b, $[1, 2]$: $[Int]$

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

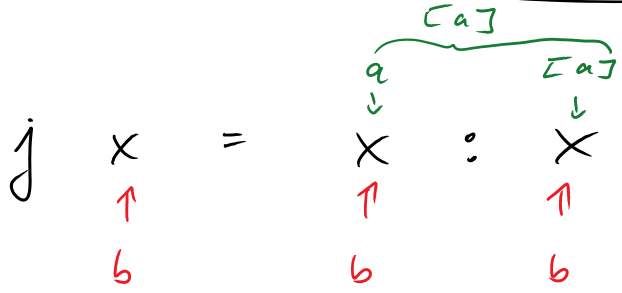
Lösung: $\sigma(a) = Int$, $\sigma(b) = Int$

Also: $g :: \underbrace{Int}_{\sigma(b)} \to \underbrace{[Int]}_{\sigma([a])}$



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

Also: $i :: \underbrace{b}_{\sigma(b)} \to \underbrace{[b]}_{\sigma(c)} \to \underbrace{[[b]]}_{\sigma([a])}$



Es existiert keine Substitution σ mit $\sigma(b) = \sigma(a)$ und $\sigma(b) = \sigma([a])$. $\Rightarrow$ nicht korrekt getypt

Unifikation: Man sucht nach einer Instantiierung der Variablen, so dass 2 Ausdrücke gleich werden. Braucht man an versch. Stellen in der Informatik

(z.B. Typchecking bei polymorphen Typen, Auswerten von Logikprogrammen, ...)

Deklaration neuer Datentypen

D.h: Definition neuer Typkonstruktoren. Dies darf nur auf der obersten Ebene d. Programms geschehen.

Haskell-Prog = Folge von linksbündig untereinander stehenden topdecl's.

Schlüsselwort data erlaubt es, neue Typkonstruktoren durch Angabe einer EBNF-artigen Grammatik einzuführen.

```
square :: Color -> Color
          Int
Int       Int

square x = x * x
```

Haskell berechnet hier

square :: Int -> Int

⇒ Typfehler.

Um einen Wert auf dem Bildschirm auszugeben, ruft Haskell eine Funktion show auf, um Werte bel. Typen in Strings zu konvertieren (wie toString in Java). show ist vordefiniert für viele vordef. Typen, aber existiert nicht für selbstdef. Typen.

2 Lösungen:

1. show für eigene Typen selbst implementieren (benötigt Konzept des Überschreibens in Haskell)
2. Lasse eine Standardimplementierung von show automatisch erzeugen.

Bsp: Datentyp für nat. Zahlen (im Unterschied zu Color, MyBool hat dieser Typ unendl. viele Objekte).

Nats: 0-stelliger Typkonstr.
Hat 2 Datenkonstruktoren

Zero :: Nats

Succ :: Nats -> Nats

Nats enthält jetzt die folgenden Objekte:

Zero, Succ Zero, Succ (Succ Zero), ...
≙ 0 ≙ 1 ≙ 2

Bsp: Typ für Listen

1-stelliger Typkonstruktor List.

dh: List Int
List Nat
List (List Bool)
} sind Typen

Datenkonstruktoren von List a:

Nil :: List a ≙ []

Cons :: a -> List a -> List a ≙ :