III.5 Typen und Datenstrukturen

Dienstag, 19. Dezember 2017 09:30

Typen in Haskell:

Bool, Int, Float, Char, ...

(Int, Bool), (Int, Int, Int), ...

[Int], [(Int, Bool)], ...

Int -> Bool, Int -> Int -> Int, ...

Typen werden also mit Hilfe von Typkonstruktoren gebildet:

- nullstellige Typkonstruktoren
  Int, Bool, ...
- einstellige Typkonstruktoren
  [...]
- zweistellige Typkonstr.:
  ->
- beliebig-stelliger Typkonstr.:
  (...)

Darüber hinaus kann man als Programmierer neue eigene Typkonstruktoren definieren (z.B. für Bäume, Graphen, ...)
-> nach Weihnachten.

Grammatik

- tyconstr type$_1$ ... type$_n$,
  wobei tyconstr ein Typkon-

stuktor der Stelligkeit n ist.
z.B. Bool, Int, ...
sind Typkonstruktoren d.
Stelligkeit 0.

Strings, die mit Großbuch-
staben beginnen.

Achtung: unterscheide
Typkonstruktoren und
Datenkonstruktoren.

Bool ist Typkonstruktor

True ist Datenkonstr.

f :: Int → ~~True~~

f :: Int → Bool

---

g True = 5

g ~~Bool~~ = 5

- [type]
  Typ der Listen, bei denen
  alle Argumente den Typ
  "type" haben.
- $type_1 \to type_2$
  Typ d. Funktionen von
  $type_1$ nach $type_2$

- $(type_1, \ldots, type_n)$, $n \geq 0$
  Typ der Tupel mit n Komponenten, wobei die i-te Komponente Typ $type_i$ hat.
- Typvariable
  nötig für parametrische Polymorphie

  Polymorphie: eine Fkt. kann auf Argumente versch. Typen angewendet werden

  - param. Polymorphie: (typisch für FP)
    die gleiche Fkt-Implem. wird für alle Typen von Argumenten benutzt
  - ad-hoc Polymorphie: (typisch für OO)
    versch. Implementierungen - Typen d. Argumente bestimmen, welche Impl. ausgeführt wird.

  Java + Haskell haben beide Arten v. Polymorphie.
  Hier: Zeige nur param. Polymorphie in Haskell.

  Bsp: len sollte für alle Arten v. Listen verwendbar

Sein ⇒ nur 1 Implem. von len

Lösung: Verwende Typvariable, die mit bel. Typen instantiiert werden kann.

Mehrfache Vork. der gleichen Typvariable in einem Typ müssen gleich instantiiert werden.

len [True, False]

len [1,2,3]

len [True, 1]

app :: [a] → [a] → [a]
ist in Haskell als Infix-Fkt. ++ vordefiniert.

[1,2] ++ [3] ergibt [1,2,3]

[True] ++ [False] ergibt
[True, False]

Eine Funktion vom Typ
$type_1 \to type_2$

kann auf ein Argument vom Typ
type
angewendet werden, falls es eine (allgemeinste) Ersetzung σ

der Typvariablen gibt, so dass

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

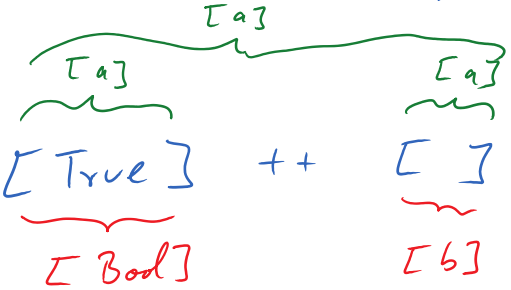
Ergebnis der Fkt-Anwendung hat den Typ $\sigma(type_2)$.

Unifikation: finde Instantiierung der Variablen, so dass 2 Ausdrücke gleich werden.

Diese Instantiierung heißt Unifikator. Wir suchen nach dem allgemeinsten Unifikator (most general unifier, MGU).

Bsp:
(++) :: [a] -> [a] -> [a],
d.h. ++ erwartet 2 Argumente vom Typ [a].

Aber [True] hat den Typ [Bool] und [] hat den Typ [b].



Unif-Problem: Finde eine Inst. σ der

Typvariablen a, b, so dass

$\sigma([a]) = \sigma([Bool])$ und

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

Lösung: a / Bool (instant. a mit Bool)

b / Bool

Der Ausdruck

[True] ++ []

ist also korrekt getypt und

er hat den Typ $\sigma([a]) =$

[Bool].

Weitere Beispiele zur Typinferenz

[] :: [a]

(:) :: a → [a] → [a]

```
f x = x : [x]
```

x: b, x: b, [x]: [b], x: [a], x: a

Lösung σ: a / b (instant. a mit b)

↝ f hat den Typ $\sigma(b) \to \sigma([a])$

d.h. $b \to [b]$

(Dies ist die gesuchte allgemeinste Lösung.
a / Int, b / Int wäre auch eine Lösung,
aber nicht die allgemeinste.)

```
g x = x ++ x
```

x: b, x: b, x: b, x: [a], x: [a], [a]

Allgem. Lösung σ: b / [a]

b b b

Also hat g den Typ: $\sigma(b) \to \sigma([a])$,

d.h. $[a] \to [a]$

```
g x = x : [1,2]
```

x: b, x: b, [1,2]: [Int]

Allgem. Lösung σ: a/Int, b/Int

Also hat h den Typ $\sigma(b) \to \sigma([a])$,

d.h. $Int \to [Int]$

```
i x = \y -> [x] : [y]
```

x: b, y: c, [x]: [b], [y]: [c], [x] : [y]: [a]

Allgem. Lösung σ: a/[b], c/[b]

Also hat i den Typ: $\sigma(b) \to \sigma(c) \to \sigma([a])$,

d.h. $b \to [b] \to [[b]]$

```
j x = x : x
```

x: b, x: b, x: b, x : x: [a]

Man sucht nach einer Inst. σ, so dass $\sigma(a) = \sigma(b)$ und $\sigma([a]) = \sigma(b)$ gilt.

Dies ist unmöglich,

a und [a] sind nicht unifizierbar.

=> Nicht typkorrekt.

Deklaration neuer Typkonstruktoren

topdecl : Deklarationen auf oberster Prog-Ebene.

data-Deklarationen sind nur auf oberster Ebene möglich, nicht in lokalen Deklarationen.

Programm : Folge von linksbündig untereinander stehenden topdecl-Deklarationen.

data : führt einen neuen Typkonstruktor und die dazugehörigen Datenkonstruktoren ein.

Bsp: Color, MyBool sind Aufzählungstypen (≙ enums in Java).
Syntax ähnlich zu EBNF-Grammatik.

Pattern Matching kann auch

für selbstdef. Typen benutzt werden (Pattern ≙ linearer Ausdruck aus Datenkonstruktoren u. Variablen).

Java: um Ausdrücke auf dem Bildschirm ausgeben zu können, müssen sie in Strings transformiert werden.
⇒ toString-Methode muss überschrieben werden, um Objekte eigener Klassen auszugeben.

Haskell: analog dazu werden Werte mit einer Funktion show in Strings umgewandelt, um sie ausgeben zu können.
Man muss show "überschreiben", um Objekte eigener Typen ausgeben zu können.
Einfache Lösung:
"deriving Show" erzeugt eine Standard-Implementierung der show-Funktion für unseren Typ.

Datentypen mit Datenkonstruktoren, die Argumente haben: Nats

(erlaubt die Repräsentation der unendl. vielen nat. Zahlen)

```
Zero :: Nats
Succ :: Nats -> Nats
```

Nats ist also der Datentyp mit den Objekten

Zero $\hat{=}$ 0

Succ Zero $\hat{=}$ 1

Succ (Succ Zero) $\hat{=}$ 2

Succ (... Succ (Zero)) $\hat{=}$ 1000

1000 Succs

```
mult :: Nats -> Nats -> Nats
mult Zero y = Zero
mult x Zero = Zero
  :
```

```
infinity :: Nats
infinity = Succ infinity
```

Dann:

```
mult infinity zero =
mult (Succ infinity) zero = zero
```

Dagegen:

```
mult :: Int -> Int -> Int
mult 0 y = 0
mult x 0 = 0
        :
infinity :: Int
infinity = 1 + infinity
```

Dann:

```
mult infinity 0 =
mult (1+infinity) 0 =
mult (1+(1+infi)) 0 = ...
```

terminiert nicht

Man kann nicht nur
0-stellige Typkonstruktoren
definieren.

Bsp: List ist 1-stelliger
Typkonstruktor, d.h.
List a ist ein Typ.

Datenkonstruktoren:

```
Nil :: List a
```

Cons :: a → (list a) → (List a)

List a ≙ [a]

Nil ≙ [ ] ← Namen aus der Sprache Lisp

Cons ≙ :