III.3 Ausdrücke

Mittwoch, 18. Januar 2017 09:00

Zentrales Konzept der fkt. Programmierung.
Ausdruck wird zunächst auf seinen Typ überprüft. Wenn er korrekt getypt ist, wird er anschließend ausgewertet.
In GHCi:

```
:t exp   ← berechnet nur den Typ des Ausdrucks exp

exp      ← überprüft erst den Typ und wertet Ausdruck dann aus
```

Führe jetzt Syntaxdiagramm für Ausdrücke. Gib jeweils Typ des Ausdrucks an u. beschreibe seine Auswertung.

- Var (Variable).
  z.B. x, square
  (Strings, die mit Kleinbuchst.

beginnen)

- Constr (Datenkonstruktoren)

z.B. True, False, [], :

(normalerweise Strings, die mit Großbuchst. beginnen).

Dienen zum Aufbau der Objekte einer Datenstr., werden nicht weiter ausgewertet.

True, False haben Typ Bool

x : xs Wenn x einen Typ a hat, dann hat xs den Typ [a] und x : xs hat ebenfalls den Typ [a].

- integer (ganze Zahlen)

0, 1, -1, 2, -2, ... sind Ausdrücke vom Typ Int, die nicht weiter ausgewertet werden.

- float (Gleitkommazahlen)

-2.5 oder 3.4 e+23

sind Ausdrücke vom Typ Float.

- char (Zeichen)

  'a', '3', '$', ...

  sind Ausdrücke vom Typ Char

- [exp$_1$, .., exp$_n$] , $n \geq 0$

  (Listen)

  [0,1,2] ist Ausdruck vom Typ [Int],

  ist Abkürzung für

  0:1:2:[]

Bei [exp$_1$, .., exp$_n$] müssen

exp$_1$, .., exp$_n$ den gleichen Typ a haben. Dann hat [exp$_1$, .., exp$_n$] den Typ [a].

- string "hallo" ist für Haskell eine Abkürzung für ['h', 'a', 'l', 'l', 'o'].

  Typ String ist identisch mit [Char].

'x' : "hallo" ergist

"xhallo"

- $(exp_1, .., exp_n)$ mit $n \geq 0$

ist ein Tupel von n Ausdrücken.

Wenn exp_1 den Typ a_1 hat, ..,

exp_n — n — a_n hat,

dann hat $(exp_1, .., exp_n)$ den

Typ $(a_1, .., a_n)$.

Bsp: (True, 2) hat den

Typ (Bool, Int).

Einstellige Tupel: (exp)

sind für Haskell identisch

zu exp

Nullstellige Tupel: ()

ist das einzige Objekt des

Typs ().

- $(exp_1 \ldots exp_n)$, $n \geq 2$

Funktionsanwendung.

Steht für

$((exp_1\ exp_2)\ exp_3) \ldots exp_n$.

Bsp: plus 2 3

- if exp_1 then exp_2 else exp_3

  exp_1: Typ Bool

  exp_2, exp_3: müssen denselben Typ haben

  Es wird erst exp_1 ausgewertet und je nach Ergebnis erhält man exp_2 oder exp_3.

- let letdecls in exp

  analog zu "where"

- \ pat_1 ... pat_n -> exp , $n \geq 1$

  Lambda-Ausdruck, denn \ steht für λ.
  Steht für die Funktion, die die Argumente $pat_1, \ldots, pat_n$ auf das Ergebnis exp abbildet.

$(\backslash x \rightarrow 2 * x)\ 5 = 2 * 5 = 10$

Verdopplungsfunktion, die jedes x auf 2*x abbildet

$(\backslash x\ y \rightarrow x + y)\ 2\ 3 = 2 + 3 = 5$

Lambda-Ausdruck beschreibt eine Funktion ohne Namen (anonyme Funktion).

```
( \ (x,y) -> x+y )   (2,3)   = 2+3 = 5
```

Statt

```
plus :: Int -> Int -> Int
plus x y = x+y
```

Könnte man auch folgendes definieren:

```
plus x = \ y -> x+y
```

oder

```
plus = \ x y -> x+y
```