III.3 Ausdrücke

Dienstag, 12. Dezember 2017 11:00

Eingabe eines Ausdrucks in Interpreter bewirkt

- Typüberprüfung des Ausdrucks
- Auswertung, falls Ausdr typkorrekt

Reine Typüberprüfung mit :t

:t square

ergibt Int -> Int

Zu jeder Art v. Ausdr geben wir jetzt den Typ u. das Resultat an, zu dem er auswertet.

Arten v. Ausdrücken

- Variablen (z.B. x, square, ...).
  Strings, die mit Kleinbuchst. beginnen.
- Datenkonstruktoren (z.B. True, False, [], :, ...)
  meist Strings, die mit Großbuchstaben beginnen.

Datenkonstr. dienen zum Aufbau der Objekte v. Datenstrukturen. Sie werden nicht weiter ausgewertet.

z.B. x : xs stelt für die Liste, die aus xs durch Einfügen v. x entsteht.

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

- integer: 0, 1, -1, ...
  vom Typ Int
- float: -2.5, 3.4e+23, ...
  vom Typ Float
- char: 'a', 'A', '5', '\n'
  vom Typ Char (\n: newline)
- [exp$_1$, ..., exp$_n$] für $n \geq 0$
  steht für die Liste der n Ausdrücke exp$_1$, .., exp$_n$.
  Hierbei ist [0, 1, 2] eine Abkürzung für 0:1:2:[].
  Hierbei müssen exp$_1$, ..., exp$_n$ den gleichen Typ α haben. Dann hat [exp$_1$, .., exp$_n$] den Typ [α].
- string: Ein String ist in Haskell eine Liste von Char.
  Kurzschreibweise:

"hallo" = ['h', 'a', 'l', 'l', 'o']

Typ String = [Char]

's' : "hallo" = "shallo"

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

Tupel von n Ausdrücken.

Wenn exp_1 den Typ a_1 hat,

exp_n den Typ a_n hat,

dann hat $(exp_1, ..., exp_n)$ den Typ $(a_1, .., a_n)$.

Einelementige Tupel werden mit ihrer Komponente identifiziert: (5) = 5

Nullelem. Tupel () hat den Typ ().

Die Listen [1,2] [3,4,5] haben beide den Typ [Int].

Der Tupel (1,2) hat den Typ (Int,Int), der Tupel (3,4,5) hat den Typ (Int,Int, Int).

- $(exp_1\ exp_2 \ldots exp_n)$, $n \geq 2$ steht für die Funktionsanwendung von Ausdrücken. Funktionsanwendung assoziiert nach links: steht für $((exp_1\ exp_2)\ exp_3) \ldots exp_n)$

plus 2 3 steht für

(plus 2) 3 ← ergibt 5

Funktion, die Zahlen um 2 erhöht

- if exp_1 then exp_2 else exp_3

  exp_1: Typ Bool

  exp_2, exp_3: müssen den gleichen Typ haben

Je nach Ergebnis d. Auswertung von exp_1 ergibt sich exp_2 oder exp_3.

- let locdecls in exp

  locdecls: Lokale Deklarationen, die nur im Ausdruck exp gelten.

Analog zu lokaler Dekl. mit "where".

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

  \ stellt für λ

Lambda-Ausdruck beschreibt

die Funktion, die die Argumente $pat_1 \dots pat_n$ auf exp abbildet.

Wenn pat_1 den Typ a_1 hat, ...,
pat_n -"- a_n hat und
exp -"- a hat,

dann hat $\backslash pat_1 \dots pat_n \to exp$ den

Typ $a_1 \to a_2 \to \dots \to a_n \to a$

stellt für $a_1 \to (a_2 \to \dots \to (a_n \to a))$

Bsp: $\backslash x \to 2 * x$

ist die Funktion, die ein Argument x erwartet und $2 * x$ zurückliefert. D.h: es ist die Verdopplungsfunktion.

$(\backslash x \to 2 * x)\ 5$ ergibt 10

Hiermit kann man "unbenannte Funktionen" direkt als Aus-

durch hinschreiben.

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
```

Die Argumente von λ dürfen bel. Patterns sein.

z.B: `\ (x,y) -> x-y`

`\ (x:xs) -> x`