III.3 Ausdrücke

Mittwoch, 12. Dezember 2018 08:30

Eingabe eines Ausdrucks in Interpreter bewirkt:

- Typüberprüfung
- Auswertung (falls Ausdruck typkorrekt)

Mit :t exp

wird nur der Typ des Ausdrucks exp bestimmt.

Zu jeder Art von Ausdrücken geben wir Typ und Ergebnis der Auswertung an.

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

Funktionssymbole zum Aufbau einer Datenstruktur, werden nicht weiter ausgewertet.

True, False haben Typ Bool

x : xs hat Typ [a]
↑ ↑
Typ a Typ [a]

- Ganze Zahlen:

  0, 1, -1, 2, -2, ...

  vom Typ Int

- Gleitkommazahlen:

  -2.5, 3.4e-23, ...

  Vom Typ Float

- Zeichen

  'a', 'A', '0', '1', ...,

  '\n', ...

  Vom Typ Char

- $[exp_1, .., exp_n]$ mit $n \geq 0$ steht für die Liste der Ausdrücke $exp_1, .., exp_n$. Alle diese Ausdrücke müssen denselben Typ a haben. Dann hat $[exp_1, .., exp_n]$ den Typ $[a]$.

  [1,2,3] ist Abkürzung für

  1 : 2 : 3 : []

  (= 1 : (2 : (3 : [])))

- Strings sind Listen von Char. Hierzu existiert Kurzschreibweise:

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

  's' : "hallo" = "shallo"

Datentyp String = [Char]

- $(exp_1, .., exp_n)$ mit $n \geq 0$ ist der Tupel aus den n Teilausdrücken $exp_1, .., exp_n$.
  Wenn exp_1 den Typ a_1 hat, ..., exp_n den Typ a_n hat, dann hat $(exp_1, .., exp_n)$ den Typ $(a_1, .., a_n)$.

  (1,2) hat Typ (Int, Int)

  (1,2,3) hat Typ (Int, Int, Int)

  [1,2] hat Typ [Int]

  [1,2,3] hat Typ [Int]

Einelementige Tupel werden mit ihrer einzigen Komponente identifiziert:

(Int) = Int

$(5) = 5$

Der Typ $()$ hat nur einen einzigen Wert: $()$

- $(exp_1\ exp_2 \ldots exp_n)$ mit $n \geq 2$ stellt für die Funktionsanwendung:

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

z.B. square (Int->Int) 5 (Int)

plus (Int->Int->Int) 2 (Int) 3 (Int)

- if exp_1 then exp_2 else exp_3

exp_1: Typ Bool

exp_2, exp_3: Ausdrücke desselben Typs a

hat den Typ a.

Je nach Wert von exp1
ergibt sich exp2 oder exp3.

- Lokale Deklarationen:
  mit where oder let.
  let: Voranstellen der
  lokalen Dekl.
  where: Nachstellen der
  lokalen Dekl.
- Lambda-Ausdrücke:

\ pat1 ... patn -> exp

↑ steht für λ

\ pat -> exp steht
für die Funktion,
die Argument pat
auf das Ergebnis exp
abbildet.

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

Funktion, die Zahlen verdoppelt

$= 2 * 5 = 10$

Lambda-Ausdrücke sind "anonyme" Funktionen ohne Namen, in denen die Funktionsdefinition direkt im Ausdruck steht.

$\backslash pat_1 \ldots pat_n \to exp$

$a_1 \quad a_n \quad a$

hat den Typ: $a_1 \to a_2 \to \ldots \to a_n \to a$

$\backslash x \to 2 * x$ hat Typ Int → Int

Int Int

Folgende Funktionsdefinitionen sind möglich:

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

oder

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

oder

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

Argumente von λ können beliebige Patterns sein:

```
(\(x:xs) -> x) [1,2,3] = 1
```

↑ Funktion, die das erste Element einer Liste liefert

$(\backslash(x,y) \to y)\ (1, True) = True$