1.1.2 Expressions

Dienstag, 19. April 2016 13:00

Expressions = central concept of functional programming

Every expression has a type. Interpreter first checks whether expression is well typed. If yes, then the expression is evaluated afterwards.

`:t exp` just computes the type of `exp` in GHCi.

We now introduce the different forms of expressions (with their type and the value that it evaluates to).

Slide 11

- Var (strings with lower-case symbol)
- Constr (data constructors, strings with upper-case symbol). Are not evaluated, but used to represent objects of data types (e.g., True, False, [], :, ...)
- integer (0, 1, -1, ...) have type Int
- float (-2.5, 3.4e+23, ...) have the type Float
- char ('a', .., 'z', 'A', .., 'Z', '0', .., '9', ' ', '\n', ...) have the type Char ('\n' = newline)
- [exp1, .., expn] stands for the list of the expressions exp1, .., expn, i.e., for exp1 : exp2 : .... : expn : []

exp1, ..., expn must all have the same type τ,

then $[exp_1, \ldots, exp_n]$ has type $[\tau]$

$[1,2,3]$ has type $[Int]$

$[True, True, False]$ has type $[Bool]$

$[square, +2]$ has type $[Int \to Int]$

$Int \to Int$ $\quad$ $Int \to Int$

$[[1,2], [], [3,4,5]]$ has type $[[Int]]$

~~$[1, True]$~~ not well typed

- String : List of characters, i.e., expression of type
  $[Char] = String$.
  $['h', 'e', 'l', 'l', 'o'] = "hello"$

- $(exp_1, \ldots, exp_n)$ where $n \geq 0$
  stands for the tuple of n expressions

  If exp_1 has type τ_1, ..., exp_n has type τ_n,
  then $(exp_1, \ldots, exp_n)$ has type $(\tau_1, \ldots, \tau_n)$.

  $(1, True, [1,2,3])$ has type $(Int, Bool, [Int])$

- $(exp_1 \ldots exp_n)$, $n \geq 2$
  means that: $(\ldots((exp_1\ exp_2)\ exp_3) \ldots\ exp_n)$

  i.e.: function application

plus 2 3 means (plus 2) 3

- if exp_1 then exp_2 else exp_3 is an expression of type τ

  exp_1: type Bool; exp_2, exp_3: same type τ

  Here, one first evaluates exp_1 and depending on its result, one obtains exp_2 or exp_3.

```
maxi (x,y) = if x >= y then x else y
```

- let decls in exp is used to define a local declaration block decls, which is only valid in exp.

```
roots a b c = let d = sqrt (...)
                  e = 2 * a
              in ((-b-d)/e, (-b+d)/e)
```

- case exp of {$pat_1 \to exp_1$; .... ; $pat_n \to exp_n$}

  Here, one tries to match pat_1 to exp. If this succeeds, then the result is exp_1, etc.

```
und True y = y
und False y = False
```

```
und x y = case x
          of True -> y
             False -> False
```

- $\backslash pat_1 \ldots pat_n \rightarrow exp \quad , \quad n \geq 1$

  ↑ stands for "λ"

  Lambda-expression

stands for the function that takes n arguments $pat_1, \ldots, pat_n$ and returns the result exp.

This allows to define "anonymous" functions by just a single expression.

$\backslash x \rightarrow 2 * x$

double-function

stands for the function that takes an argument x and returns $2 * x$

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

Type of $(\backslash pat_1 \ldots pat_n \rightarrow exp) : \tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$

pat_1 / type τ_1 — pat_n | type τ_n — exp \ type τ

$\backslash x \rightarrow 2 * x$ has type $Int \rightarrow Int$

x: Int — $2 * x$: Int

$\backslash (x, y) \rightarrow x + y$ has type $(Int, Int) \rightarrow Int$

(x, y): Int, Int — $x + y$: Int

Instead of `plus x y = x + y`

we could define `plus = \ x y -> x + y`

or

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