III.4 Patterns

Mittwoch, 12. Dezember 2018 09:00

Patterns: auf linken Seiten von Fkt-deklarationen. Sie beschreiben die Form der erwarteten Argumente.

Im Prinzip:

Patterns $\hat{=}$ Ausdrücke aus Datenkonstruktoren und Variablen

Bsp:

```
f :: [Int] -> Bool
f []     = False
f (5:xs) = True
f  xs    = False
```

untersucht, ob Liste mit 5 anfängt.

Weiteres Bsp: app (append)
für Konkatenation von Listen.
Ist in Haskell als Infix-
Funktion ++ vordefiniert:

[1,2] ++ [3,4,5] =

[1,2,3,4,5]

Auswertung des Pattern Matchings:
Leftmost Outermost Strategie.
Wenn man nicht entscheiden
kann, welche definierende
Gleichung des äußersten
Funktionssymbols anwendbar
ist, dann müssen Argumente
ausgewertet werden, aber
nur so lange, bis klar ist,
welche def. Gleichung des
äußeren Fkt.-Symbols passt.

len (app [1] [2])
        1:[]

= len (1 : app [] [2])

= 1 + len (app [] [2])

= 1 + len [2]
        2:[]

= 1 + (1 + len [])

= 1 + (1 + 0)

= 1 + 1

= 2

werte app aus, bis man weiß, ob Pattern [] passt

vordef. arithm. Operationen wie + können nur ausgewertet werden, wenn Argumente Zahlen sind

Weiteres Bsp:

```
zeros :: [Int]
zeros = 0 : zeros

f :: [Int] -> [Int] -> [Int]
f [] ys = []
f xs [] = []
```

Auswertung von:

f [] zeros = [] termi-

nicht

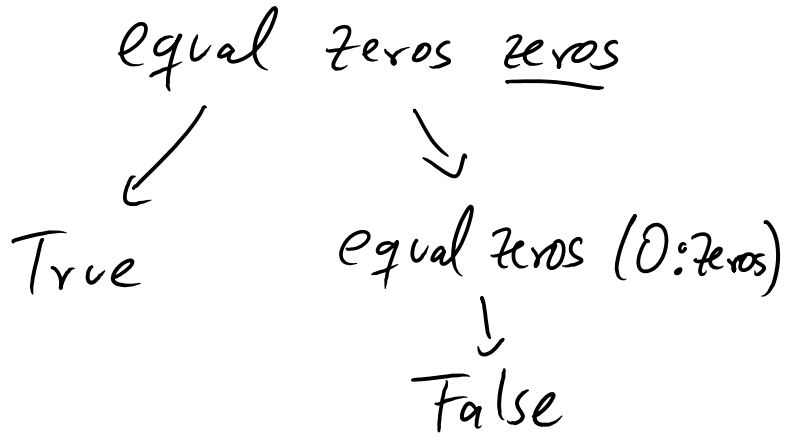
```
f zeros []
=f (0:zeros) []
=[]
```

terminiert auch

↳ werte zeros so lange aus, bis man entscheiden kann, welcher Pattern passt

Patterns müssen linear sein, d.h: auf der linken Seite einer definierenden Gleichung darf keine Variable mehrfach vorkommen.
Grund: Sonst kann das Ergebnis der Auswertung von der Auswertungsstrategie abhängen:

```
zeros :: [Int]
zeros = 0 : zeros
```



Zeige nun bei jedem Pattern:

- Auf welche Ausdrücke passt der Pattern?
- Wie werden die Variablen des Patterns beim Pattern Matching instantiiert?

Arten von Pattern:

- Variable

```
square :: Int -> Int
square x = x * x
```

passt auf jeden Wert, Variable wird beim Pattern Matching mit dem Wert belegt.

- Underscore = Joker-Pattern

```
und :: Bool -> Bool -> Bool
und True y = y
und _    _ = False
```

passt auf jeden Wert, aber es erfolgt keine Variablenbelegung. Falls _ mehrfach auftritt, darf er mit unterschiedlichen Werten belegt werden.

- integer, float, char, string-Werte können als Pattern benutzt. Sie passen nur auf sich selbst, keine Variablenbelegung.

```
is_5 :: Int -> Bool
is_5  5 = True
is_5  _ = False
```

- Constr $pat_1 \ldots pat_n$

  ↑

  n-stelliger Datenkonstruktor

  passt auf alle Werte, die mit dem Datenkonstruktor Constr gebildet wurden und bei denen $pat_1, \ldots, pat_n$ auf die n Argumente passen.
- $[pat_1, \ldots, pat_n]$

  passt auf alle n-elementigen Listen, bei denen $pat_1, \ldots, pat_n$ auf die Elemente passen
- $(pat_1, \ldots, pat_n)$ analog