

Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

25. Oktober 2007

- Syntax
- Typen
- Auswertung
- Programmierung

Als erweiterte Kernsprache: **KFPTSP**

Syntax:

$$T ::= V \mid (TC \ T_1 \dots T_n) \mid (T_1 \rightarrow T_2)$$

wobei V Typvariable,
 T_i Typen
 TC (parametrisierter) Typkonstruktoren

Beispiele: zu Typkonstruktoren:

Bool

List a bzw. $[a]$.

Parameter a : Typ der Elemente der Liste.

```
data [a] = [] | (a : [a])
```

```
data Vector a = Vectordaten [a]
```

True :: Bool
False :: Bool
&& :: Bool → Bool → Bool
|| :: Bool → Bool → Bool

Cons :: $a \rightarrow [a] \rightarrow [a]$ *a* ist Typvariable
Nil :: $[a]$ *a* ist Typvariable
(:) :: $a \rightarrow [a] \rightarrow [a]$ *a* ist Typvariable
[] :: $[a]$ *a* ist Typvariable

- $$\frac{f :: a \rightarrow b; \quad s :: a}{(f \ s) :: b}$$

- $$\frac{s :: T}{s :: T'}$$

wobei $T' = \sigma(T)$ und σ eine Einsetzung von Typen für Typvariablen ist.

- $$\frac{s :: T; \quad t_1 : a, \dots, t_n : a}{(\text{case}_T \ s \ \text{of} \ \{pat_1 \rightarrow t_1; \dots\}) :: a}$$

(für nullstellige Typkonstruktoren)

Es fehlen: pattern, guards, list comprehensions, ...

Typ des Ausdrucks `True && False` unter Verwendung der Regeln:

$$\frac{\frac{\&\& :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}; \quad \text{True} : \text{Bool}}{(\&\& \text{True}) :: \text{Bool} \rightarrow \text{Bool}}; \quad \text{False} : \text{Bool}}{(\&\& \text{True} \text{ False}) :: \text{Bool}}$$

$$\frac{(\cdot) :: \alpha \rightarrow [\alpha] \rightarrow [\alpha], \quad 1 : \text{Int}; \quad [] :: [\alpha']}{(1 \cdot) :: [\text{Int}] \rightarrow [\text{Int}]}; \quad \frac{}{(1 : []) ::: [\text{Int}]}$$

Typvariablen sind zu instanziiieren!

Das Verfahren berechnet auch speziellere Typen,
nicht nur die allgemeinsten

Muss man manchmal **beide** Typen
 $f : \tau_1 \rightarrow \tau_2$ und $s :: \tau_3$ instanziiieren,
so dass die Regel anwendbar wird?

$$\frac{f :: \tau_1 \rightarrow \tau_2; \quad s :: \tau_3}{(f \ s) :: \tau_4}$$

`concatMap :: (a → [b]) → [a] → [b]`

`reverse :: [c] → [c]`

`(concatMap reverse) :: [[a]] → [a]`

Besonderheiten der Typisierung in Haskell.

- beschränkte ganze Zahlen (`Int`),
- unbeschränkte ganze Zahlen (`Integer`),
- Gleitkommazahlen (`Float`),
- doppelt genaue Gleitkommazahlen (`Double`),
- rationale Zahlen (`Ratio α`).

```
2 * 2                = 4
2.0/3.0              = 0.666666667
(1%2)/(3%2)          = (1%3) :: Ratio Integer
(123456789 :: Int)*
  (987654321 :: Int) = -67153019 :: Int
123456789 * 987654321 = 121932631112635269 :: Integer
```

Typklasse `Num` der numerischen Typen

`+`, `-`, `*` auf allen Objekten mit Typ aus der Typklasse `Num`

`(+)`: `Num a => a -> a -> a`

`Fractional`: Typen, für die `/` erlaubt (z.B. nicht für `Int`)

Implementierung von `Integer` in KFPTS

Peanozahlen: Typ `Pint` aufgebaut mit zwei Konstruktoren:

`S` einstellig
`0` nullstellig

$0, (S\ 0), S(S\ 0), S(S(S\ 0)), \dots$

implementieren die nicht-negativen ganzen Zahlen $0, 1, 2, 3, \dots$

(Haskell-Variante der KFPTS Definitionen)

```
data Pint = Zero | Succ Pint
  deriving (Eq, Show)
```

```
istZahl x = case x of {Zero -> True; Succ y -> istZahl y}
```

```
peanoPlus x y = if istZahl x && istZahl y
  then pplus x y else bot
```

```
pplus x y = case x of Zero -> y
  Succ z -> Succ (pplus z y)
```

Der Haskell-Typ der Funktionen ist:

- `istZahl :: Pint → Bool`
- `peanoPlus :: Pint → Pint → Pint`
- `pplus :: Pint → Pint → Pint`

`istZahl` bewirkt das korrekte Terminierungsverhalten

Z.B. `(+ bot 1)` als auch `(+ 1 bot)` terminieren nicht.

D.h. auch `(peanoPlus bot (Succ Zero))` und
`(peanoPlus (Succ Zero) bot)`
dürfen nicht terminieren.

Beachte: `(pplus (Succ Zero) bot)` terminiert

für nichtnegative ganze Zahlen:

```
peanoleq x y =  
  (istZahl x) &&  
  (istZahl y) &&  
  (case x of  
    {Zero -> True;  
     Succ xx ->  
       case y of  
         {Zero -> False;  
          Succ yy -> peanoleq xx yy}})
```

Die Fakultätsfunktion auf Peanozahlen:

```
pmal x y = case x of Zero -> Zero  
          Succ z -> pplus (pmal z y) y
```

```
pfak x = case x of Zero -> Succ Zero  
          Succ z -> (pmal x (pfak z))
```

Wir können annehmen, dass alle arithmetischen Funktionen

in KFPTS definiert sind.

Beispiel

5- Tupel: (1, 2, 3, 4, 5).

Übersetzung nach KFPT als Typ mit 5-stelligem Konstruktor `Tuple5`

$$\text{Tuple5} :: a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow (a_1, a_2, a_3, a_4, a_5)$$

Es gibt ein null-stelliges Tupel, geschrieben `()`.

kein einstelliges Tupel.

Selektoren liefern die einzelnen Komponenten eines Tupels

In Haskell mit Pattern realisiert:

```
fst x = case x of {(u, v) -> u}
```

```
snd x = case x of {(u, v) -> v}
```

oder einfacher in Haskell, aber äquivalent:

```
fst (u, v) = u
```

```
snd (u, v) = v
```

Kodierung in KFPTS erfordert die Angabe des Typs:

```
fst x = case_Tupel2 x of {Tupel2 u v -> u}  
snd x = case_Tupel2 x of {Tupel2 u v -> v}
```

Beispiel-Definitionen und Verwendung

```
letztes_element []      = error "Liste leer"  
letztes_element (x:xs) = if xs == [] then x  
                        else  letztes_element  xs
```

Zur Info: diese Funktion ist end-rekursiv

`[1..10] ----> [1,2,3,4,5,6,7,8,9,10]`

`[1..] ----> [1,2,3,4,5,6,7,8,9,10,11,.....]`

interne Definitionen: (Nach KFPTS übersetzbar)

```
upto m n = if m > n then []  
          else m : (upto (m+1) n)
```

```
from m = m : (from (m+1))
```

```
map f xs = case xs of {Nil -> Nil; Cons h t -> Cons (f h) (map f t)}
```

Haskell-Definition:

```
map    :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs)  = (f x) : (map f xs)
```

Die Funktion **append**, Infix geschrieben als ++:

```
++ xs ys = case xs of {Nil -> ys; Cons h t -> Cons h (t ++ ys )}
```

Die Haskell-Definition ist:

```
(++) :: [a] -> [a] -> [a]  
[] ++ ys      = ys  
(x:xs) ++ ys = x:(xs ++ ys)
```

Länge einer Liste:

```
length xs = case xs of {Nil -> 0; Cons h t -> (1 + (length t))}
```

Die Haskell-Definition ist:

```
length :: [a] -> Int  
length []      = 0  
length (_:t)   = 1 + (length t)
```

```
zip :: [a] -> [b] -> [(a,b)]
zip [] []      = []
zip [] xs     = []
zip xs []     = []
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)

unzip :: [(a,b)] -> ([a],[b])
unzip []          = ([],[])
unzip ((x,y):xs) = (x:x1, y:y1)
                  where (x1,y1) = unzip xs
```

concatMap:

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f [] = []
concatMap f (x:xs) = (f x) ++ concatMap f xs

concatMap f xs entspricht concat (map f xs)
```

```
reverse :: [a] -> [a]
```

```
reverse xs = case xs of {Nil -> Nil; Cons h t -> ((reverse t) ++ [h])}
```

Z.B. `reverse [1, 2, 3] → [3, 2, 1]`

Die Haskell-Definition ist:

```
reverse [] = []
```

```
reverse (h:t) = (reverse t) ++ [h]
```

Laufzeit: $O(n^2)$.

Die effizientere Methode ist die Verwendung eines Stacks:

```
reverse x          = rev_accu x []
rev_accu xs stack = case xs of {[] -> stack;
                               h:t  -> (rev_accu t (h:stack))}
```

rev_accu ist endrekursiv (tail-recursive)

Transpose: einer Matrix als Liste von Listen.

$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ wird als `[[1,2],[3,4]]` dargestellt.

```
hd xs = case xs of {h:t -> h; [] -> bot}
```

```
tl xs = case xs of {h:t -> t; [] -> bot}
```

```
transpose xss = case xss of
  {[] -> bot;
   h:t ->
     case h of {[] -> [];
                h:t -> (map hd xss)
                : (transpose (map tl xss))}}
```

Haskell-Definition von `transpose`

```
transpose ([] : rest) = []  
transpose x           = (map hd x) : (transpose (map tl x))
```

Die Typen sind:

```
hd :: [a] → a
```

```
tl :: [a] → [a]
```

```
transpose :: [[a]] → [[a]]
```

`transpose [[1, 2], [3, 4], [5, 6]]` ergibt bei der Auswertung nacheinander:

`[1, 3, 5] : transpose [[2], [4], [6]]`
→ `[[1, 3, 5], [2, 4, 6]]`

Beachte: es gibt in Haskell den Datentyp `Array` (siehe Handbuch)

```
vectoradd_1 xs ys = map vadd (transpose [xs, ys])  
vadd [x,y]      = x + y
```

optimierte Version:

```
zipWith :: ( a-> b-> c) -> [a] -> [b] -> [c]  
vectoradd_2          = zipWith (+)
```

let, where sind rekursive Bindungsoperatoren

$$\text{let } x_1 = t_1; \dots; x_n = t_n \text{ in } Exp$$

entspricht

$$Exp \text{ where } x_1 = t_1; \dots; x_n = t_n$$

Beachte, dass im Haskell-Report die Verwendung des `where` eingeschränkt ist.

Ein nicht-rekursives `let` (`where`) ist einfach darstellbar:

$$\text{let } x_1 = t_1; \dots \quad ; x_n = t_n \text{ in } Exp$$

entspricht dann (in etwa)

$$(\backslash x_1 \dots \quad x_n \rightarrow Exp) t_1 \dots t_n$$

Unendliche Liste mit gleichen Elementen:

```
let x = 10 : x in x
```

[10, 10, 10, ...

rekursive Definition von length:

```
let length = (\xs -> case xs of {[] -> 0; (y:ys) -> 1+(length ys)})  
in (length [1..10])
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter pred [] = []
filter pred (h:t) = if pred h then h:rest
                    else rest
                    where rest = filter pred t
```

```
remove p xs = filter (\x -> not (p(x))) xs
```

Oder kürzer geschrieben:

```
remove p = filter (not . p)
```

Geschrieben als Punkt: .

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$
$$f \cdot g = \lambda x \rightarrow f (g x)$$

Der neue Haskell-Standard wird vermutlich ein anderes Zeichen empfehlen.