Programmierparadigmen

Zusammenfassung

Autor: Frieder Haizmann

Inhaltsverzeichnis

1 Haskell

Linear Rekursiv: In jeden Definitionszweig kommt nur ein rekursiver Aufruf vor.
Endrekursiv: Linear rekursiv in jeden Zweig ist der rekursive Aufruf nicht in andere Aufrufe eingebettet.

1.1 Listen

`[]`, `x:xs`, `head[1,2,3] => 1`, `tail[1,2,3] => [2,3]`, `init xs -- Alle elemente bis auf das l`
`null[1,2,3] => False`, `length`,`isIn`, `a ++ b`, `take n l -- erste n Elemente von l`,
`drop n l -- l ohne erste n Elemente`, `xs !! n -- Nte Listenelement`,

```
reverse xs
map f (x:xs) -- wendet f auf alle Listenelemente an
filter pred (x:xs) -- behalte alle Elemente, die Prädikat erfüllen
```

1.2 Funktionsanwendungen

Funktionskomposition $f \circ g$: `comp f g = (\x -> f (g x))` Infix: `f.g`
n-Fache Funktionsanwendung f^n: `iter f n`
Funktionstypen sind rechts-assoziativ, Funktionsanwendung ist links-assoziativ
`f x = y + x` Variable x Gebunden, Variable y frei.

1.3 Lokale Namensbindung

```
energy m = let c = 299
               square = x = x * x
           in m * (square c)
```

```
energy m = m * (square c)
    where c = 299
          square x = x * x
```

Einrückung hat semantische Bedeutung.
Bei Schachtelung: Inneres `let` bindet stärker.

1.4 Folds

```
foldr operator initial [] = initial
foldr operator initial (x:xs) = operator x (foldr operator initial xs)
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f`...(xn `f` z)...)
foldl op i [] i
foldl op i (x:xs) = foldl op (op i x) xs
foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...)`f` xn
```

Beispiel: `length ~~list~~ = foldr (\x n -> n + 1) 0 ~~list~~`

```
concat [xs, ys, zs] = xs ++ ys ++ zs concat == foldr app []

zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs ys = []
zip = zipWith (,) -- zip[1,2,3],[7,8,9] = [(1,7),(2,8),(3,9)]
```

Kurznotation Intervalle `[a..b]` $\Rightarrow^+$ `[a, a+1, a+2, ..., b]`
List Comprehensions `[e | q1, ..., qm]`, q_i Tests oder Generatoren der Form $p \leftarrow list$ mit Muster p und Listenausdruck *list*
Beispiel: `squares n = [x * x | x <- [0..n]]`
Im Muster `p` gebundene Variablen können in `e` und in `qi` verwendet werden
Beispiel: `evens n = [x |` $\underbrace{\texttt{x<-[0..n]}}_{\texttt{Generator}}$`,` $\underbrace{\texttt{x `mod `2 == 0]}}_{\texttt{Tester}}$ **Unendliche Listen**: `odds = 1 : map (+2`

```
iterate f a = a : iterate f (f a)
odds = iterate (+2)  1
iterate f x !! 23 -- führt "Schleife" 23 mal aus
```

1.5 Typen

```
type neuerName = [Integer]
               = String
               = (a, b)
               = ...
```

Algebraische Datentypen	Konstruktoren
`data Shape = Circle Double` `| Rectangle Double Double`	`Circle :: Double -> Shape` `Rectangle :: Double -> Double -> Shape`

Aufzählungstypen

```
data Season = Spring | Summer | Autumn | Winter
```

Polymorphe Datentypen (Optionale Werte)

```
data Maybe t = Nothing | Just t
data Either s t = Left s | Right t
data Matrix t = Dense [[t]] -- Liste von Zeilen
        | Sparse [(Integer, Integer t)] t -- Einträge (i,j,v) und Defaultwert
data Stack t = Empty | Stacked t (Stack t)
```

Polymorphe Funktion mit Typeinschränkung

```
qsort :: Ord t => [t] -> [t]
qsort [] = []
qsort (p:ps) = ...
```

Instanzen von Ord implementieren `<=, <, >, >=, ...`

Typklassen-Definitionen

```haskell
instance Eq Bool where
    True == True = True
    True == False = False
          ⋮

data shape = Circle Double
        | Rectangle Double Double
        deriving Eq -- Keine eigene (==) Funktion mehr notwendig
```

Fehlende Implementierungen: Default implementiert

Automatische Instanziierung auch für `Show`,`Ord`,`Enum`

1.6 Monaden

2 untypisiertes λ-Kalkül

	Notation	Beispiele	
Variablen	x	x	y
Abstraktion	$\lambda x.t$	$\lambda y.0$	$\lambda f.\lambda x.\lambda y.f\ y\ x$
Funktionsanwendung	$t_1 t_2$	$f42$	$\lambda x.x + 5)\ 7$

Funktionsanwendung linksassoziativ, bindet stärker als Abstraktion

α-Äquivalenz	t_1 und t_t heißen α-äquivalent $t_1 \overset{\alpha}{=} t_2$, wenn t_1 in t_2 durch konsistente Umbenennung der λ-gebundenen Variablen überführt werden kann
η-Äquivalenz	Terme $\lambda x.f\ x$ und f heißen η-äquivalent ($\lambda x.f\ x \overset{\eta}{=} f$) falls x nicht freie Variable vob f
Redex	Ein λ-Term der Form $(\lambda x.t_1)\ t_2$ heißt Redex
β-Reduktion	β-Reduktion entspricht der Ausführung der Funktionsanwendung auf einen Redex $$(\lambda x.t_1)t_2 \Rightarrow t_1[x \mapsto t_2]$$
Substitution	$t_1[x \mapsto t_2]$ erhält man aus den Term t_1, wenn man alle freien Vorkommen von x durch t_2 ersetzt
Normalform	Ein Term, der nicht weiter reduziert werden kann
Volle β-Reduktion	Jeder Redex kann jederzeit reduziert werden
Normalreihenfolge	Immer der linkeste äußerste Redex wird Reduziert

$let\ x = t_1\ in\ t_2$ wird zu $(\lambda x.t_2)\ t_1$

2.1 Church-Zahlen

Eine (natürliche) Zahl drückt aus, wie oft die funktion s angewendet wurde

$$\begin{aligned} c_0 &= \lambda s.\lambda z.z \\ c_1 &= \lambda s.\lambda z.s\ z \\ c_2 &= \lambda s.\lambda z.s\ (s\ z) \\ &\vdots \\ c_n &= \lambda s.\lambda z\ s^n\ z \end{aligned}$$

Nachfolgefunktion

$$succ = \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$$

wobei n Church Zahl

Addition	$plus = \lambda m.\lambda n.\lambda z.m\ s\ (n\ s\ z)$
Multiplikation	$times = \lambda m.\lambda n.\lambda s.n(m\ s) \stackrel{\eta}{=} \lambda m.\lambda n.\lambda s.\lambda z.n(m\ s)\ z$
Potenzieren	$exp = \lambda m.\lambda n.n\ m \stackrel{\eta}{=} \lambda m.\lambda n.\lambda s.\lambda z.n\ m\ s\ z$

$c_{true} = \lambda t.\lambda f.t \quad c_{false} = \lambda t.\lambda f.f$

$isZero = \lambda n.n\ (\lambda x.c_{false})\ c_{true}$