Programmierparadigmen
Wintersemester 2019/2020

Zusammenfassung

Autor: Frieder Haizmann

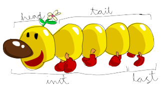
Inhaltsverzeichnis

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 [1,2,3] => [1,2], last [1,2,3] => 3
take n l -- erste n Elemente von l
drop n l  -- l ohne erste n Elemente
```

```
takewhile bedingung liste
dropwhile bedingung liste
span b l == (takewhile b l, dropwhile b l)
```

```
null[1,2,3] => False
length
isIn
```

```
a ++ b
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]` $=>^+$ `[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 |` $\underbrace{\texttt{x<-[0..n]}}_{\text{Generator}}$, $\underbrace{\texttt{x `mod `2 == 0}}_{\text{Tester}}$`]`

Unendliche Listen: `odds = 1 : map (+2) odds`

```
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`	`Circle :: Double -> Shape`
`  | Rectangle Double Double`	`Rectangle :: Double -> Double -> Shape`

Aufzählungstypen

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

Polymorphe Datentypen (Optionale Werte)

```haskell
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

```haskell
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
        ⋮
```

Fehlende Implementierungen: Default implementiert

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

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

1.6 Monaden

2 λ-Kalkül

2.1 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 $t_1 \stackrel{\alpha}{=} t_2$, 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 ($\lambda x.f\ x \stackrel{\eta}{=} f$) 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 $(\lambda x.t_2)\ t_1$

2.2 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	$exp = \lambda m.\lambda n.n\ m$ $\stackrel{\eta}{=} \lambda m.\lambda n.\lambda s.\lambda z.n\ m\ s\ z$

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

Rekursionsoperator $Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$
$f\ (Y\ f) \stackrel{\eta}{=} Y\ f$
$Y\ f$ ist Fixpunkt von f

Church-Rosser: Wenn $t \Rightarrow t_1$ und $t_1 \stackrel{*}{\Rightarrow} t_2$, dann gibt es t' mit $t_1 \stackrel{*}{\Rightarrow} t'$ und $t_2 \stackrel{*}{\Rightarrow} t'$

Call-by-name	Reduziere linkesten äußersten Redex aber nicht, falls von einem λ umgeben → Reduziere Argumente erst, wenn benötigt
Call-by-Value	Reduziere linkesten Redex, der nicht von einem λ umgeben und dessen Argument ein Wert → Argument vor Funktionsaufruf ist auszuwerten

2.3 Regelsysteme

Fregescher Schlussstrich

$$\frac{\varphi_1 \quad \varphi_2 \quad \varphi_3 \quad \cdots \quad \varphi_n}{\varphi}$$

Jede Regel stellt Implikation dar, φ_i Voraussetzungen, φ Konklusion

	Introduktionsregel	**Eliminationsregel**
Konjunktion	$\wedge I \dfrac{\psi \ \varphi}{\psi \wedge \varphi}$	$\wedge E_1 \dfrac{\psi \wedge \varphi}{\psi} \quad \wedge E_2 \dfrac{\psi \wedge \varphi}{\varphi}$
All-Quantor	$\forall I \dfrac{P(y)\ y\ ist\ frei\ in\ P}{\forall x.P(x)}$	$\forall E \dfrac{\forall x.P(x)}{P(y)}$
Implikation	$\begin{array}{c} \psi \\ \vdots^1 \\ \rightarrow I \dfrac{\varphi}{\psi \longrightarrow \varphi} \end{array}$	$\text{MP} \dfrac{\psi \longrightarrow \varphi \quad \psi}{\varphi}$

Alternative Notation

$$\frac{\Gamma_1 \vdash \varphi_1 \quad \dots \quad \Gamma_n \vdash \varphi_n}{\vdash \varphi}$$

Regelsysteme alternative Notation

Introduktionsregeln **Eliminationsregeln**

$$\wedge I \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}$$

$$\to I \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \longrightarrow \psi} \quad \textsc{AssmI} \frac{}{\Gamma, \varphi \vdash \varphi}$$

$$\wedge I_1 \frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \wedge \psi} \quad \cdots$$

$$\wedge E_1 \frac{\Gamma \vdash \psi \wedge \psi}{\Gamma \vdash \varphi} \quad \wedge E_2 \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\mathrm{MP} \frac{\Gamma \vdash \varphi \longrightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$

$$\mathrm{VE} \frac{\Gamma \vdash \varphi \wedge \psi \quad \Gamma, \varphi \vdash \omega \quad \Gamma, \psi \vdash \omega}{\Gamma \vdash \omega}$$

[1] Herleitung gemäß Regeln verbindet Implikation der Regeln mit prädikatenlogischer Implikation

2.4 Typsysteme

Funktionstypen rechtsassoziativ Typsystem $\Gamma \vdash t : \tau$ im Typkontext Γ hat Term t Typ τ. Γ ordnet freien Variablen x ihren Typ $\Gamma(x)$ zu.

$$\textsc{Const}\frac{c \in \mathit{Const}}{\Gamma \vdash c : \tau_2} \qquad \textsc{Var}\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\textsc{Abs}\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \longrightarrow \tau_2} \qquad \textsc{App}\frac{\Gamma \vdash t_1 : \tau_2 \longrightarrow \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$$

Typisierung von λ-Term t: Paar (Γ, τ), sodass $\Gamma \vdash t : \tau$ herleitbar
β-Reduktion: Substitution von x $(\lambda x.t_1)\ t_2 \Rightarrow t_1[x \mapsto t_2]$

Substitutionslemma	Wenn $\Gamma, x : \tau_2 \vdash t_1 : \tau_1$ und $\Gamma \vdash t_2 : \tau_2$ dann $\Gamma \vdash t_1[x \mapsto t_2] : \tau_1$
Typerhaltungstheorem	Wenn $\Gamma \vdash t : \tau$ und $t \Rightarrow t'$ dann $\Gamma \vdash t' : \tau$
Typschema	Typ der Gestalt $\forall\alpha_1.\forall\alpha_2.\cdots\forall\alpha_n$ heißt Typschema. Es bindet freie Typvariablen $\alpha_1 \ldots \alpha_n$ in τ
Instanziierung eines Typschemas	Für nicht-Schema-Typen τ_2 ist der Typ $\tau[\alpha \mapsto \tau_2]$ eine Instanzieierung vom Typschema $\forall\alpha.\tau$ Schreibweise $(\forall\alpha.\tau) \succeq \tau[\alpha \rightarrow \tau_2]$
Beispiele	$\forall\alpha.\alpha \rightarrow \alpha \succeq int \rightarrow int$ $\forall\alpha.\alpha \rightarrow \alpha \succeq (int \rightarrow int) \rightarrow (int \rightarrow int) \quad int \succeq int$ $\alpha \rightarrow \alpha \not\succeq int \rightarrow int \quad \alpha \not\succeq bool$ $\forall\alpha.\alpha \rightarrow \alpha \not\succeq bool$

2.5 Typinferenz

$\textsc{Var}\dfrac{\Gamma(x) = \sigma \quad \sigma \succeq \iota}{\Gamma \vdash x : \tau}$ Constraint: $\{\iota = \tau\}$	$\textsc{App}\dfrac{\Gamma \vdash f : \xi \quad \Gamma \vdash x : \varphi}{\Gamma \vdash f\ x : \alpha}$ Constraint: $\{\xi = \varphi \to \alpha\}$
$\textsc{Abs}\dfrac{\Gamma, p : \pi \vdash b : \beta}{\Gamma \vdash \lambda p.b : \alpha}$ Constraint: $\{\alpha = \pi \to \beta\}$	$\textsc{Let}\dfrac{\Gamma \vdash y : \pi \quad \Gamma' \vdash b : \beta}{\Gamma \vdash let\ x = y\ in\ b : \tau}$ Constraints: Siehe unten

$ta(\tau, \Gamma)$ bindet alle in Γ freien Typvariablen mit einem $\forall$ in τ
Bsp.: $ta(\alpha \to \beta, x : \beta, y : \delta) = \forall \alpha.\alpha \to \beta$

$$\textsc{Let}\frac{\Gamma \vdash y : \pi \quad \Gamma' \vdash b : \beta}{\Gamma \vdash let\ x = y\ in\ b : \tau}$$

Constraints: Finde Unifikator σ_{LET} und allg. Typ π für y

1. Sei C_0 die bisherige Constraintmenge inklusive $\{\tau = \beta\}$
2. Sammle Constraints aus linken Teilbaum in C_{LET}
3. Berechne *mgu* σ_{LET} von C_{LET}
4. Berechne $\Gamma' := \sigma_{\text{LET}}(\Gamma), x : ta(\sigma_{\text{LET}}(\pi), \sigma_{\text{LET}}(\Gamma))$
5. Benutze Γ' in rechten Teilbaum, sammle Constraints in C_1
6. Ergebnisscontraints sind $C_0 \cup C'_{\text{LET}} \cup C_1$
   mit $C'_{\text{LET}} := \{\alpha_i = \sigma_{\text{LET}}(\alpha_i) | \sigma_{\text{LET}} \text{ definiert für } \alpha_i\}$

Typen sind Rechstassoziativ also wenn suche nach mgu und

$$\alpha_1 = \alpha_3 \to \alpha_4 \to \alpha_5 = (\alpha_3 \to (\alpha_4 \to \alpha_5))$$
$$\alpha_2 = \alpha_6 \to \alpha_7$$

dann

$$\alpha_6 \Rightarrow \alpha_3$$
$$\alpha_7 \Rightarrow \alpha_4 \to \alpha_5$$