Freie Universität Berlin

ALP I

λ–Kalkül

Teil 2

WS 2012/2013

Prof. Dr. Margarita Esponda

Lokale Variablennamen

Haskell:

```
let x = exp1 in exp2
```

Lambda:

λ exp1 . exp2

Einfache Regel:
Der Geltungsbereich eines Lambda-Ausdrucks erstreckt sich soweit wie möglich nach rechts.

$$\lambda m.\lambda a.\lambda b.aa(bm) \quad => \quad (\lambda m.(\lambda a.(\lambda b.aa(bm))))$$

Lambda-Ausdrücke sind rechtsassoziativ

β-Reduktion

Die Auswertung bzw. Funktionsapplikation von Lambda-Ausdrücken wird auch β-Reduktion genannt.

$$(\lambda x.E1)\,(E2) \rightarrow_{\beta} E1\,[E2\backslash x]$$

Redex (Reduzierbarer Ausdruck)

Funktionsapplikation ist linksassoziativ

α-Konversion

$$(\lambda x\,.\,x\,z\,y) \to_{\alpha} (\lambda t\,.\,t\,z\,y)$$

Haskell:

```
u =  let  a = 2
          b = a + ( let  a = 3
                         in  a+b )
      in  b*b
```

äquivalent zu

```
u =  let  a = 2
          b = a + ( let  x = 3
                         in  x+b )
      in  b*b
```

β-Konversion und α-Konversion

$$(\lambda x.E1)\ (E2) \rightarrow_{\beta} E1\ [E2\backslash x]$$

Was passiert, wenn x in E2 vorkommt?

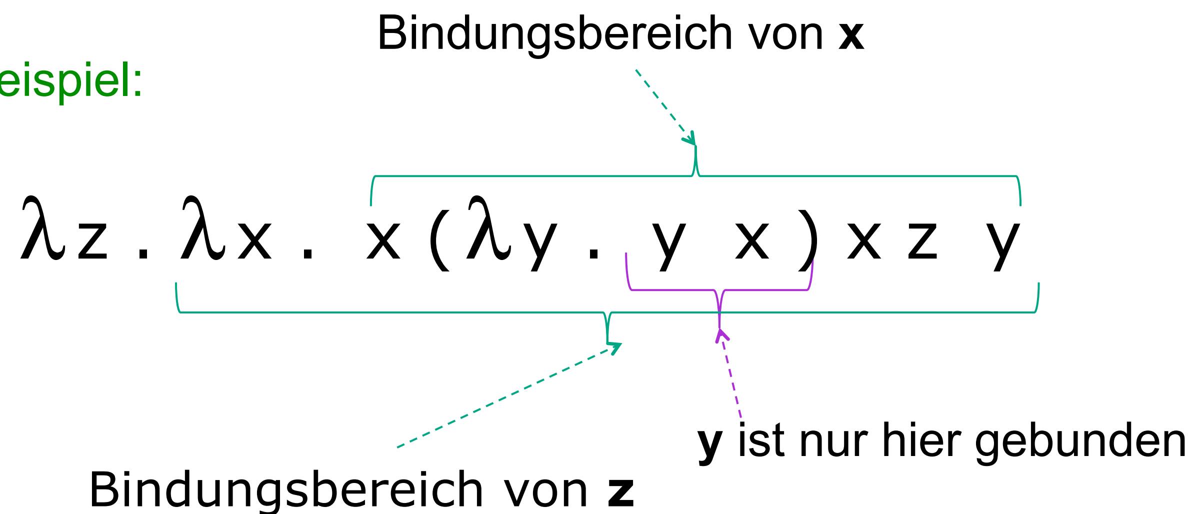
Haskell:

```
u  n =  let  a = 2*n
             b = a + ( let  a = 3
                            b = c
                       in   a+b )
             c = a^2
        in   b*b
```

Bindungsbereich von Variablennamen

Eine Lambda-Abstraktion **λx .E** bindet alle Vorkommen von **x** innerhalb des Ausdrucks **E**.

Mit anderen Worten: **E** ist der Geltungsbereich der Variablennamen **x**.



Freie und Gebundene Variablennamen

Definition eines Hilfsoperators, der die Elemente von zwei Listen ohne Verdopplungen konkateniert.

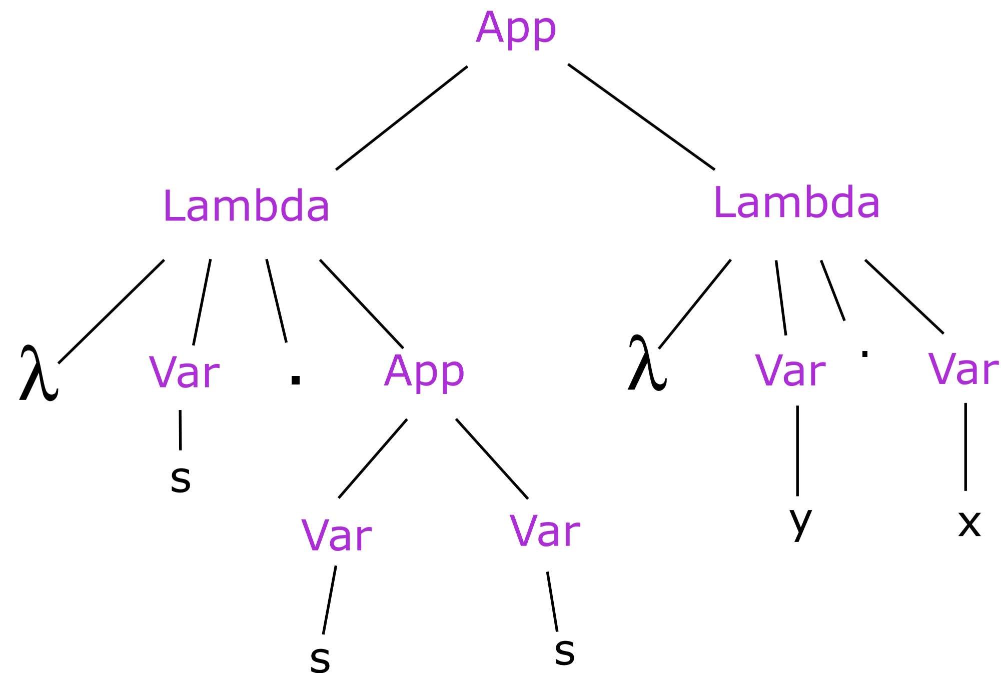
Definition als Operator

```
(+++) :: [String] -> [String] -> [String]
(+++) [] ys        = ys
(+++) (x:xs) ys  | not (elementOf x ys)  = x : (+++) xs ys
                 | otherwise             = (+++) xs ys
```

Syntaxbaum

```
data Expr = Var String | App Expr Expr | Lambda String Expr | Nil
                                         deriving ( Eq )
```

(λs. ss)(λy.x)



Freie Universität Berlin

Freie und Gebundene Variablennamen

Wir können die Menge **FV** der ungebundenen Variablen mit Hilfe der folgenden Funktion berechnen:

```
free  ::  Expr -> [String]
free  (Var x)           =  [x]
free ( App e1 e2)       =  (free e1) +++ (free e2)
free (Lambda x  e)      =  remove x (free e)
```

α-Konversion

Ein Lambda-Ausdruck E wird als geschlossen bezeichnet, wenn keine freie Variablen in E beinhaltet sind.

$$FV(E) = \emptyset$$

$$\lambda x.E \;=\; \lambda y.(E[y\backslash x]) \quad \text{wenn} \quad y \notin FV(E)$$

Wir können Variablen immer umbenennen, wenn die neuen Namen nicht als freie Variablennamen innerhalb des Lambda-Ausdrucks vorkommen.

Beispiel:

$$(\lambda x.\ y\ x)\ x \rightarrow_{\alpha} (\lambda a.\ y\ a)\ x \rightarrow_{\beta} y\ x$$

α-Konversion

Formale Definition der freien Variablen Substitution:

$x\ [e\backslash x] = e$

$y\ [e\backslash x] = y \qquad$ wenn $y \neq x$

$(e1\ e2)\ [e\backslash x] = (e1[e\backslash x]\ \ e2[e\backslash x])$

$(\lambda x.e1)[e\backslash x] = (\lambda x.e1)$

$(\lambda y.e1)[e\backslash x] = (\lambda y.(e1[e\backslash x])$ wenn $y\neq x$ und $y\notin FV(e1)$

Vorgänger-Funktion

Um die Vorgänger-Funktion zu berechnen, erzeugen wir zuerst Zahlenpaare der Form (n, n-1), und dann wählen wir das zweite Element des Tupels:

$(\lambda t.\ t\ x\ y)$ stellt das Tupel (x,y) dar

Das erste Element des Tupels ist:

$$(\lambda t.\ t\ x\ y)\ T \Rightarrow T\ x\ y \Rightarrow x$$

Das zweite Element des Tupels ist:

$$(\lambda t.\ t\ x\ y)\ F \Rightarrow F\ x\ y \Rightarrow y$$

Vorgänger-Funktion

Die Funktion **H** erzeugt bei Eingabe des Tupels (n, n-1) das Tupel (n+1, n)

$$H \equiv (\lambda pz.z\ (\ S(pT)\ )(\ pT\ ))$$

Wenn $p = (n, n-1)$, dann ist pT gleich dem ersten Element des Tupels.

Vorgänger-Funktion

H angewendet an **p** = (2,1)

H (λt. t 2 1)

=> (λpz.z(S(pT))(pT)) (λt. t 2 1)

=> (λz.z(S((λt. t 2 1) T))((λt. t 2 1) T))

=> (λz.z (S(T 2 1))(T 2 1))

=> (λz.z (S (2)) 2)

=> (λz.z 3 2)

Vorgänger-Funktion

Der Vorgänger der Zahl **n** wird dann berechnet, indem die Funktion **H** **n**-mal auf das Tupel $(\lambda t.t00)$ angewendet wird, und dann nur das zweite Element des Ergebnis-Tupels gewählt wird.

$$P \equiv (\lambda n.nH(\lambda z.z00)F)$$

Die Berechnung der Vorgänger von 0 ist 0.

Vorgänger-Funktion

Beispiel Vorgänger von 1:

$$P\,1 \Rightarrow (\lambda n.nH(\lambda z.z00)F)\ 1$$

$$\Rightarrow (1H(\lambda z.z00)F)$$

$$\Rightarrow (\lambda z.z10)\ F$$

$$\Rightarrow (F\ 10)$$

$$\Rightarrow\ 0$$

Vergleichsfunktionen

Wenn die Vorgänger-Funktion **x**-mal angewendet auf **y** zu Null führt, dann ist **x >= y**

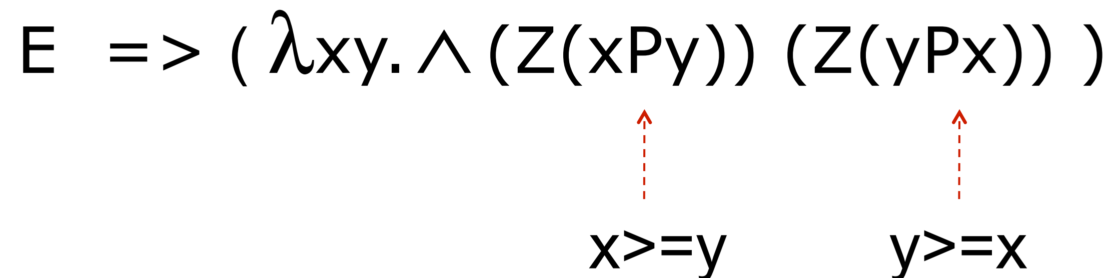
(>=) Vorgänger-Funktion

$$G \equiv (\lambda xy.Z(xPy))$$

Test ob gleich Null

Gleichheit

Die Gleichheit-Funktion kann dann definiert werden, indem getestet wird, dass **x>=y** und **y>=x** ist.



Gleichheit und Ungleichheit

Beispiel:

$$E\ 1\ 0 \Rightarrow (\lambda xy.\wedge(Z(xPy))(Z(yPx)))\ 1\ 0$$

$$\Rightarrow \wedge(Z(1P0))\ (Z(0P1)))$$

$$\Rightarrow \wedge(Z(0))\ (Z(1)))$$

$$\Rightarrow \wedge(T)\ (F)$$

$$\Rightarrow (F)$$

Normal-Form

Ein Lambda-Ausdruck befindet sich in normaler Form, wenn nicht weiter reduziert werden kann.

Beispiele:

$\lambda x \,.\, x$

$\lambda x \,.\, \lambda z \,.\, y$

Nicht alle Lambda-Ausdrücke haben eine Normal-Form

Beispiele:

Wenn $A = \lambda x \,.\, x\ x$ dann

$A\ A \Rightarrow (\lambda x.x\ x)\ (\lambda x\ .x\ x) \Rightarrow (\lambda x.x\ x)\ (\lambda x\ .x\ x)$

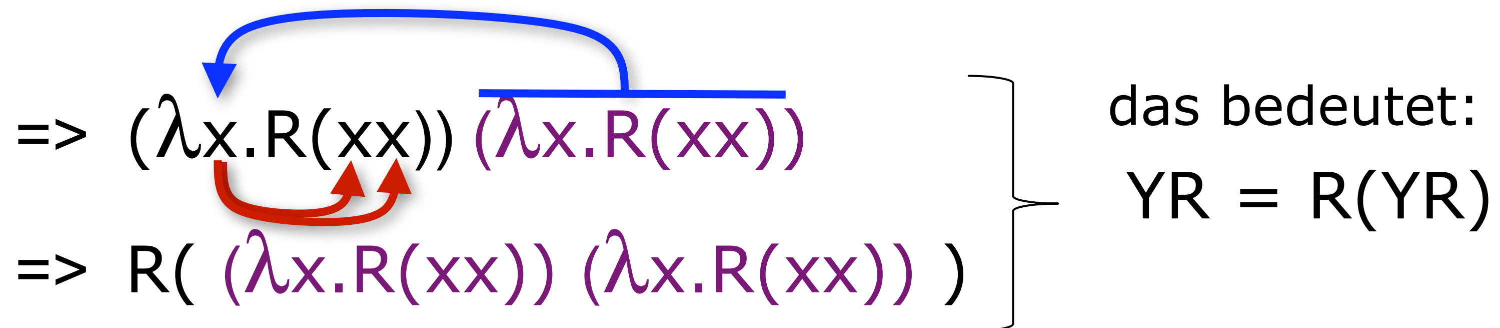
Rekursive Funktionen

Rekursive Funktionen werden mit Hilfe der Funktion **Y** definiert, die die Eigenschaft hat, sich selber zu reproduzieren.

$$Y \equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$$

Angewendet an eine Funktion R, ergibt das:

$$YR \equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ R$$



Rekursive Funktionen

Beispiel:

Wir möchten die Summe der ersten **n** natürlichen Zahlen rekursiv berechnen.

Haskell:

```
sum 0 = 0
sum n = n + sum (n-1)
```

Lambda-Kalkül:

Die Nachfolger-Funktion wird **n**-mal angewendet

$$R \equiv (\lambda r\ n\ .\ Z\ n\ 0\ (\ n\ S\ (\ r\ (\ P\ n\ ))))$$

Test **n** gegen **0**

rekursiver Aufruf mit (**n-1**)

Rekursive Funktionen

Wir müssen den **Y**-Operator verwenden, um die Funktion **R** rekursiv zu machen.

$$R \equiv (\lambda r\ n\ .\ Z\ n\ 0\ (\ n\ S\ (\ r\ (\ P\ n\ ))))$$

Beispiel für die Summe der Zahlen von 0 bis 3 ist:

YR3 = R(YR)3 = Z 3 0 (3S((YR)(P3)))

. . .

3 S (Y R 2)

. . .

3 S (2 S (1 S 0))

. . .

6

Schöne Feiertage
und
ein gesundes neues Jahr
2013!