

Übung 9

Lambda, Church

Lambda

Church-Booleans

Church-Zahlen

Lambda in Haskell



- Der **Lambda-Kalkül** ist ein Termersetzungssystem.
- Terme:
 - Alle Variablen v sind Terme
 - Wenn v Variable und t Term, so ist auch $(\lambda v.(t))$ Term
 - Wenn t_1, t_2 Terme, so ist auch $(t_1 t_2)$ Term
- Termersetzungen
 - **α -Reduktion:** $(\lambda v.(t)) \equiv^\alpha (\lambda v'.(\underline{t} [v'/v]))$ *v nicht frei*
 - wenn Substitution $[v'/v]$ in t zulässig
 - Umbenennung von Parametern
 - **β -Reduktion:** $(\lambda v.(\underline{t})) x \equiv^\beta t [x/v]$
 - wenn Substitution $[x/v]$ in t zulässig
 - Funktionsaufruf
 - **η -Reduktion:** $(\lambda \underline{v}.(\underline{t} \underline{v})) \equiv^\eta t$
 - wenn v nicht frei in t
 - Funktionale Abstraktion (kein ~~Parameter~~ *Arg.* nötig)



- Bekannt: Typ Boolean in Haskell

- Speichert 1 Bit (eine binäre Entscheidung)

- `data Bool = False | True`

Prelude

- `deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)`

- Jetzt: Booleans nach Church im Lambda-Kalkül

- Es gibt nur Funktionen und deren Anwendung

- **Church-Boolean** ist Funktion, die eines ihrer Argumente zurückgibt

- True: erstes, False: zweites

`true = (λ x.(λ y.(x)))`

`false = (λ x.(λ y.(y)))`

- Wird selbst als Argument an andere Funktionen übergeben



▪ Imperatives Programm:

```
• Boolean x=...;           // Werte und Rechnen darauf
  Boolean y=!x;           // Fallunterscheidung
  if(x) {
    a();
  } else {
    b();
  }
```

```
▪ if(a || b)
  • c()
```

```
▪ or a,b,tmp           cmp 0, a
▪ cmp 0, tmp           jnz c
▪ jnz c                cmp 0, b
                       jnz c
```



▪ Imperatives Programm:

```

• Boolean x=...;           // Werte und Rechnen darauf
Boolean y=1x;
  if(x) {                 // Fallunterscheidung
    a();
  } else {
    b();
  }

```

▪ In Lambda:

• $\lambda x.(\lambda a.(\lambda b.(x a b)))$

• Anwendung auf true = $(\lambda x.(\lambda y.(x)))$:

▫ $\lambda x.(\lambda a.(\lambda b.(x a b))) (\lambda x.(\lambda y.(x))) \equiv^{\alpha}$

$\lambda x.(\lambda a.(\lambda b.(x a b))) (\lambda z.(\lambda y.(z))) \equiv^{\beta}$

$\lambda a.(\lambda b.((\lambda z.(\lambda y.(z))) a b)) \equiv^{\beta}$

$\lambda a.(\lambda b.(\lambda y.(a) b)) \equiv^{\beta}$

$\lambda a.(\lambda b.(a b))$

$\lambda x.(\lambda y.(x))$



- Fallunterscheidung bekannt
- Wie rechne ich damit?

- Beispiel Negation

- $\text{neg} = (\lambda x.(x \text{ false true}))$

- Anwendung auf false = $(\lambda x.(\lambda y.(y)))$:

- $(\lambda x.(x \text{ false true})) (\lambda x.(\lambda y.(y))) \equiv^\alpha$

- $(\lambda x.(x \text{ false true})) (\lambda z.(\lambda y.(y))) \equiv^\beta$

- $(\lambda z.(\lambda y.(y))) \text{ false true} \equiv^\beta$

- $(\lambda z.(\lambda y.(y))) \text{ false true} \equiv^\beta$

- $\text{id} = (\lambda y.(y)) \text{ true} \equiv^\beta$

true



- Logisches Und:

X	Y	False	True
False	False	False	False
True	False	False	True

- Umsetzung in Lambda?

$$\text{and} = \lambda x. \lambda y. (x \ y \ \text{false})$$

$$\text{nand} = \text{neg and}$$



- Imperative Sprache:

- `int n=...; // Wert und Rechnen darauf`
`n++;`
`for(; n<end; n++) { // Iteration`
`x=! (x);`
`}`



▪ Imperatives Programm

```

• int n=...;           // Wert und rechnen darauf
  n++;
  for(; n<end; n++) { // Iteration
    x=! (x);
  }

```

▪ In Lambda:

- $(\lambda n. (\lambda x. (n \text{ neg } x)))$
- Anwendung auf 2:

$$\begin{aligned}
 & (\lambda n. (\lambda x. (n \text{ neg } x))) (\lambda f. (\lambda x. (f (f x)))) \equiv^\alpha \\
 & (\lambda n. (\lambda x. (n \text{ neg } x))) (\lambda f. (\lambda y. (f (f y)))) \equiv^\beta \\
 & (\lambda x. ((\lambda f. (\lambda y. (f (f y)))) \text{ neg } x)) \equiv^\beta \\
 & (\lambda x. ((\lambda y. (\text{neg } (\text{neg } y)))) x) \equiv^\beta \\
 & (\lambda x. (\text{neg } (\text{neg } x))) \equiv \lambda \text{ neg } \text{ neg}
 \end{aligned}$$



- Iteration bekannt
- Wie rechne ich damit?

- Beispiel Nachfolger

- $\text{succ} = (\lambda n.(\lambda f.(\lambda x.(f (n f x))))$

$f^n(x)$

- Anwendung auf 3:

$$\begin{aligned}
 & (\lambda n.(\lambda f.(\lambda x.(f (n f x)))) (\lambda f.(\lambda x.(f (f (f x)))) \equiv^\alpha \\
 & (\lambda n.(\lambda f.(\lambda x.(f (n f x)))) (\lambda g.(\lambda y.(g (g (g y)))) \equiv^\beta \\
 & (\lambda f.(\lambda x.(f ((\lambda g.(\lambda y.(g (g (g y)))) f x)))) \equiv^\beta \\
 & (\lambda f.(\lambda x.(f ((\lambda y.(f (f (f y)))) x)))) \equiv^\beta \\
 & (\lambda f.(\lambda x.(f (f (f (f x)))))) = 4
 \end{aligned}$$



- Addition im Buch S.217 falsch
- Was tut $mystery = \lambda m.(\lambda n.(m n))$?

• Anwendung auf 2 3:

$$(\lambda m.(\lambda n.(m n))) 2 3 \equiv^{\beta}$$

$$(\lambda n.(2 n)) 3 \equiv^{\beta}$$

$$2 3 =$$

$$(\lambda f. (\lambda x.(f (f x)))) (\lambda f. (\lambda x.(f (f (f x)))) \equiv^{\alpha}$$

$$(\lambda t. (\lambda x.(t (f x)))) (\lambda g. (\lambda y.(g (g (g y)))) \equiv^{\beta}$$

$$(\lambda x.((\lambda g. (\lambda y.(g (g (g y)))) ((\lambda g. (\lambda y.(g (g (g y)))) x))) \equiv^{\beta}$$

$$(\lambda x.((\lambda g. (\lambda y.(g (g (g y)))) ((\lambda y.(x (x (x y)))))) \equiv^{\eta}$$

$$(\lambda x.((\lambda g. (\lambda y.(g (g (g y)))) (x (x (x)))) \equiv^{\alpha}$$

$$(\lambda f.((\lambda g. (\lambda y.(g (g (g y)))) (f . f . f))) \equiv^{\beta}$$

$$(\lambda f. (\lambda y.(f.f.f (f.f.f (f.f.f y)))) \equiv^{\alpha}$$

$$(\lambda f. (\lambda x.(f^9 (x))) = 9$$

exp = $\lambda b.\lambda e.(e b)$

- Vermutung: $mystery x y = y^x$

- Lambda-Kalkül ist Grundlage aller funktionalen Sprachen
 - Alle rein funktionalen Sprachkonstrukte können in Lambda ausgedrückt werden
- Umgekehrt kann Haskell Lambda auch direkt notieren
 - $(\lambda v.(t))$ wird notiert als:
 $(\backslash v \rightarrow t)$
 - $(\lambda v_1.(\lambda v_2.(... (\lambda v_n.(t)) ...)))$ wird notiert als:
 $(\backslash v1 v2 \dots vn \rightarrow t)$
- t kann beliebiger Haskell-Ausdruck sein
 - also auch Operatoren, Datentypen, Bibliotheken nutzen
 - Nützlich z.B. um Hilfsfunktionen zu definieren:
 - `map (\x -> x**2+1) [1..10]`

$x^2 + 1$

~~(**2)~~
succ | (+1)



- Typinferenz produziert Typen zu niedriger Ebene:

- $\text{csucc} = \lambda n f x \rightarrow f (n f x)$

- `:t csucc`

- $\text{csucc} :: ((a \rightarrow b) \rightarrow c \rightarrow a) \rightarrow (a \rightarrow b) \rightarrow c \rightarrow b$

- Lambda-Ausdruck hat drei Lambdas, ist aber eine einstellige Funktion auf Church-Zahlen

- Wie erreichen wir:

- $\text{csucc} :: \text{ChurchZahl} \rightarrow \text{ChurchZahl}$

- Mit benutzerdefinierten Typen:

- `type ChurchNumeral a = (a -> a) -> a -> a`

- `csucc :: ChurchNumeral a -> ChurchNumeral a`

- $\text{csucc} = \lambda n f x \rightarrow f(n f x)$

- `:t csucc`

- $\text{csucc} :: \text{ChurchNumeral } a \rightarrow \text{ChurchNumeral } a$



- Denke nicht objektorientiert oder imperativ!
- Diese Typisierung wird fehlschlagen:
 - `type ChurchBoolean a = a -> a -> a`
 - `cand :: ChurchBoolean a -> ChurchBoolean a -> ChurchBoolean a`
- Haskell inferiert Typen aus der Verwendung der Argumente
 - Betrachte genau die unterschiedliche Verwendung des 1. und 2. Arguments
 - Die durch den Interpreter gelieferte Typisierung kann nützlich sein.
 - Sie ist **keine** gültige Lösung der Aufgabe
 - Platzhalter für Typen sind ausschließlich als Argument an `ChurchBoolean` oder `ChurchNumeral` zulässig.

