

Der untypisierte Lambda-Kalkül

Manuel Eberl
eberlm@cs.tum.edu

20. August 2011

Was ist der Lambda-Kalkül?

- Entwickelt von Alonzo Church in den 1930er Jahren
- Ist ein formales System zur Definition und Untersuchung von Funktionen
- Ist genauso mächtig wie die Turingmaschine
- Bildet die Grundlage für funktionale Programmiersprachen

Grundlegende Idee

Alles ist eine Funktion und Funktionen werden in einem Ausdruck gleichzeitig definiert und angewandt.

Beispiel

Wir haben zwei Funktionen:

- *sqr*: Erwartet eine Zahl und gibt ihr Quadrat zurück
- *twice*: Erwartet eine Funktion f und gibt $f \circ f$ zurück

Dann ist:

- $sqr\ 3 = 9$

Grundlegende Idee

Alles ist eine Funktion und Funktionen werden in einem Ausdruck gleichzeitig definiert und angewandt.

Beispiel

Wir haben zwei Funktionen:

- *sqr*: Erwartet eine Zahl und gibt ihr Quadrat zurück
- *twice*: Erwartet eine Funktion f und gibt $f \circ f$ zurück

Dann ist:

- $\text{sqr } 3 = 9$
- $(\text{twice } \text{sqr}) 3 = \text{sqr } (\text{sqr } 3) = 81$

Drei Arten von Ausdrücken im Lambda-Kalkül:

Definition

Lambda-Ausdrücke sind:

Variablen etwa a, b, c, \dots

Abstraktionen wie $\lambda x. T$, wobei x eine Variable und T ein Lambda-Ausdruck ist

Applikationen wie $S T$, wobei S und T Lambda-Ausdrücke sind

Zusätzlich: Klammern zur Gruppierung

Abstraktion

$\lambda x. T$ - Definiert eine Funktion, die x auf T abbildet. T ist ein Lambda-Ausdruck, der - normalerweise, aber nicht zwingend - x enthält.

Beispiel

- $\lambda x. x$ (die Identitätsfunktion *id*)
- $\lambda x. a$ (eine Funktion, die konstant a zurückgibt)

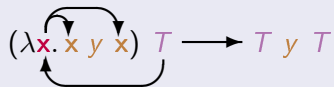
Die Applikation

Applikation

S T - Wendet die Funktion S auf den Ausdruck T an.

Definition (β -Reduktion)

Falls S die Form $(\lambda x. R)$ T hat (sog. Redex für reducible expression): Reduktion durch Ersetzung aller x in R durch T

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


Beispiel

$(\lambda x. 42 x y x 1337)$ wuppd

$\downarrow \beta$

42 wuppd y wuppd 1337

Beispiele

- $id\ a$

Beispiele

- $id\ a$ - liefert a
- $id\ id$

Beispiele

- $id\ a$ - liefert a
- $id\ id$ - liefert id
- $sqr\ 3$

Beispiele

- *id a* - liefert *a*
- *id id* - liefert *id*
- *sqr 3* - liefert 9
- *twice sqr*

Beispiele

- $id\ a$ - liefert a
- $id\ id$ - liefert id
- $sqr\ 3$ - liefert 9
- $twice\ sqr$ - liefert eine „hoch 4“-Funktion
- $twice\ sqr\ 3$

Beispiele

- $id\ a$ - liefert a
- $id\ id$ - liefert id
- $sqr\ 3$ - liefert 9
- $twice\ sqr$ - liefert eine „hoch 4“-Funktion
- $twice\ sqr\ 3$ - liefert 81

Achtung: Die Applikation ist linksassoziativ. $f\ f\ x$ ist $(f\ f)\ x$, nicht etwa $f\ (f\ x)$

Funktionen mit mehreren Parametern - Currying

Problem: Nur ein Parameter pro Abstraktion erlaubt. Wie definiert man z.B. $add\ a\ b = a + b$?

Funktionen mit mehreren Parametern - Currying

Problem: Nur ein Parameter pro Abstraktion erlaubt. Wie definiert man z.B. $add\ a\ b = a + b$?

Überlegung: Funktionen schachteln - $add(a)$ liefert Funktion $add_a(b)$ mit konstantem a , die b auf $a + b$ abbildet.
Diese Schachtelung nennt man „Currying“.

Funktionen mit mehreren Parametern - Currying

Problem: Nur ein Parameter pro Abstraktion erlaubt. Wie definiert man z.B. $add\ a\ b = a + b$?

Überlegung: Funktionen schachteln - $add(a)$ liefert Funktion $add_a(b)$ mit konstantem a , die b auf $a + b$ abbildet. Diese Schachtelung nennt man „Currying“.

Beispiel 1 - Addition

- $+$ sei die Funktion, die zwei Parameter addiert

Funktionen mit mehreren Parametern - Currying

Problem: Nur ein Parameter pro Abstraktion erlaubt. Wie definiert man z.B. $add\ a\ b = a + b$?

Überlegung: Funktionen schachteln - $add(a)$ liefert Funktion $add_a(b)$ mit konstantem a , die b auf $a + b$ abbildet. Diese Schachtelung nennt man „Currying“.

Beispiel 1 - Addition

- $+$ sei die Funktion, die zwei Parameter addiert
- $+ 4 = +_4$ ist eine „Addiere 4“-Funktion

Funktionen mit mehreren Parametern - Currying

Problem: Nur ein Parameter pro Abstraktion erlaubt. Wie definiert man z.B. $add\ a\ b = a + b$?

Überlegung: Funktionen schachteln - $add(a)$ liefert Funktion $add_a(b)$ mit konstantem a , die b auf $a + b$ abbildet. Diese Schachtelung nennt man „Currying“.

Beispiel 1 - Addition

- $+$ sei die Funktion, die zwei Parameter addiert
- $+ 4 = +_4$ ist eine „Addiere 4“-Funktion
- $+ 4\ 2 = +_{4,2}$ ist äquivalent zu 6

Funktionen mit mehreren Parametern - Currying

Beispiel 2 - Reverse Application-Funktion

Definition: *reverse* x y liefert als Ergebnis y x

$reverse := \lambda x. \lambda y. y x$

Funktionen mit mehreren Parametern - Currying

Beispiel 2 - Reverse Application-Funktion

Definition: *reverse* x y liefert als Ergebnis y x

$reverse := \lambda x. \lambda y. y$ x

$reverse$ a $b \equiv$

Funktionen mit mehreren Parametern - Currying

Beispiel 2 - Reverse Application-Funktion

Definition: *reverse* x y liefert als Ergebnis y x

$reverse := \lambda x. \lambda y. y$ x

$reverse$ a $b \equiv \underline{(\lambda x. \lambda y. y$ $x)}$ a b

Funktionen mit mehreren Parametern - Currying

Beispiel 2 - Reverse Application-Funktion

Definition: $reverse\ x\ y$ liefert als Ergebnis $y\ x$

$reverse := \lambda x. \lambda y. y\ x$

$reverse\ a\ b \equiv \underline{(\lambda x. \lambda y. y\ x)\ a}\ b \equiv reverse_a\ b \equiv \underline{(\lambda y. y\ a)\ b}$

Funktionen mit mehreren Parametern - Currying

Beispiel 2 - Reverse Application-Funktion

Definition: *reverse* x y liefert als Ergebnis y x

$reverse := \lambda x. \lambda y. y$ x

$reverse$ a $b \equiv \underline{(\lambda x. \lambda y. y$ $x)}$ a $b \equiv reverse_a$ $b \equiv \underline{(\lambda y. y$ $a)}$ $b \equiv b$ a

Bei verschachtelten Funktionsdefinitionen: Statt $\lambda x. \lambda y. \lambda z$ T
vereinfachend $\lambda xyz.$ T .

Beispiel: $reverse := \lambda xy. y$ x

Freie und gebundene Variablen

Variable in einer Abstraktion heißt gebunden, wenn sie durch ein Lambda definiert wurde.

Nicht gebundene Variablen heißen frei. (vgl. Existenz-/Allquantor)

Beispiel

- $\lambda xy. x y$ - x und y gebunden
- $\lambda x. x y$ - x gebunden, y frei.
- $(\lambda x. x) x$ - inneres x gebunden, äußeres x frei
- $\lambda x. \lambda x. x$ - x gebunden und „verdeckt“ äußeres x

Frage: Was ist der Unterschied zwischen $\lambda x. x$ und $\lambda y. y$?

α -Konversion

Frage: Was ist der Unterschied zwischen $\lambda x. x$ und $\lambda y. y$?

Antwort: Es gibt keinen! Namen gebundener Variablen sind irrelevant.

Den Austausch gebundener Variablennamen bezeichnet man als α -Konversion.

Frage: Was ist der Unterschied zwischen $\lambda x. x$ und $\lambda y. y$?

Antwort: Es gibt keinen! Namen gebundener Variablen sind irrelevant.

Den Austausch gebundener Variablennamen bezeichnet man als α -Konversion.

Definition (α -Konversion)

Eine λ -Abstraktion $\lambda x. A$ ist äquivalent zu $\lambda y. A[x := y]$ überführt werden.

Bei $A[x := y]$: Alle Vorkommnisse von x in A werden durch y ersetzt (aber: Verdeckung beachten)

Frage: Was ist der Unterschied zwischen $\lambda x. x$ und $\lambda y. y$?

Antwort: Es gibt keinen! Namen gebundener Variablen sind irrelevant.

Den Austausch gebundener Variablennamen bezeichnet man als α -Konversion.

Definition (α -Konversion)

Eine λ -Abstraktion $\lambda x. A$ ist äquivalent zu $\lambda y. A[x := y]$ überführt werden.

Bei $A[x := y]$: Alle Vorkommnisse von x in A werden durch y ersetzt (aber: Verdeckung beachten)

Ausdrücke, die sich durch α -Konversion ineinander überführen lassen, heißen α -äquivalent.

Kollision von Variablennamen

Achtung: Man darf bei einer Ersetzung von Variablen und bei der β -Reduktion nicht tiefer definierte Variablen „überschreiben“!

Beispiel - Kollision bei α -Konversion

$\lambda xy. x$ darf z.B. nicht zu $\lambda y. \lambda y. y$ α -konvertiert werden.

Kollision von Variablennamen

Achtung: Man darf bei einer Ersetzung von Variablen und bei der β -Reduktion nicht tiefer definierte Variablen „überschreiben“!

Beispiel - Kollision bei α -Konversion

$\lambda xy.x$ darf z.B. nicht zu $\lambda y.\lambda y.y$ α -konvertiert werden.

Lösung: Ersetze kollidiere Variable ebenfalls:

$\lambda xy.x \equiv \lambda xz.x \equiv \lambda yz.y$

Kollision von Variablennamen

Achtung: Man darf bei einer Ersetzung von Variablen und bei der β -Reduktion nicht tiefer definierte Variablen „überschreiben“!

Beispiel - Kollision bei α -Konversion

$\lambda xy. x$ darf z.B. nicht zu $\lambda y. \lambda y. y$ α -konvertiert werden.

Lösung: Ersetze kollidiere Variable ebenfalls:

$\lambda xy. x \equiv \lambda xz. x \equiv \lambda yz. y$

Beispiel - Kollision bei β -Reduktion

$(\lambda ab. a) b$ - erstes b gebunden, zweites b frei.

Kollision von Variablennamen

Achtung: Man darf bei einer Ersetzung von Variablen und bei der β -Reduktion nicht tiefer definierte Variablen „überschreiben“!

Beispiel - Kollision bei α -Konversion

$\lambda xy. x$ darf z.B. nicht zu $\lambda y. \lambda y. y$ α -konvertiert werden.

Lösung: Ersetze kollidiere Variable ebenfalls:

$\lambda xy. x \equiv \lambda xz. x \equiv \lambda yz. y$

Beispiel - Kollision bei β -Reduktion

$(\lambda ab. a) b$ - erstes b gebunden, zweites b frei.

Bei β -Reduktion:

$\lambda b. b$ - gebundenes b wird zurückgegeben - völlig anderes Ergebnis!

Kollision von Variablennamen

Achtung: Man darf bei einer Ersetzung von Variablen und bei der β -Reduktion nicht tiefer definierte Variablen „überschreiben“!

Beispiel - Kollision bei α -Konversion

$\lambda xy. x$ darf z.B. nicht zu $\lambda y. \lambda y. y$ α -konvertiert werden.

Lösung: Ersetze kollidiere Variable ebenfalls:

$\lambda xy. x \equiv \lambda xz. x \equiv \lambda yz. y$

Beispiel - Kollision bei β -Reduktion

$(\lambda ab. a) b$ - erstes b gebunden, zweites b frei.

Bei β -Reduktion:

$\lambda b. b$ - gebundenes b wird zurückgegeben - völlig anderes Ergebnis! \Rightarrow Umbenennung der gebundenen Variable: $\lambda b'. b$

Datentypen?

Problem: Keine Datentypen wie Zahlen, Wahrheitswerte, Listen oder Strings - es gibt nur Funktionen. Aber: Gleichmächtig zur Turing-Maschine - es muss also möglich sein, etwa mit Zahlen zu rechnen. Aber wie?

Lösung: Man kodiert jeden Datentypen als eine Funktion!

Definition von true und false

true und *false* als Funktionen mit zwei Parametern: $true\ a\ b \equiv a$
und $false\ a\ b \equiv b$.

Definition

$true := \lambda ab. a$

$false := \lambda ab. b$

Hiermit lassen sich sehr einfach „Verzweigungen“ realisieren.
„if *A* then *B* else *C*“, mit Wahrheitswert *A* ist einfach $A\ B\ C$.

Boole'sche Operationen

Boole'sche Operationen

- *not* - **if** *a* **then** *false* **else** *true*; -

Boole'sche Operationen

Boole'sche Operationen

- *not* - **if** *a* **then** *false* **else** *true*; - $\lambda a. a \text{ false } true$

Boole'sche Operationen

Boole'sche Operationen

- *not* - **if a then $false$ else $true$** ; - $\lambda a. a \text{ false } true$
- *and* - **if a then b else $false$** ; -

Boole'sche Operationen

Boole'sche Operationen

- *not* - **if a then $false$ else $true$** ; - $\lambda a. a \ false \ true$
- *and* - **if a then b else $false$** ; - $\lambda a. \lambda b. a \ b \ false$

Boole'sche Operationen

Boole'sche Operationen

- *not* - **if a then $false$ else $true$** ; - $\lambda a. a\ false\ true$
- *and* - **if a then b else $false$** ; - $\lambda a. \lambda b. a\ b\ false$
- *or* - **if a then $true$ else b** ; -

Boole'sche Operationen

Boole'sche Operationen

- *not* - **if a then $false$ else $true$** ; - $\lambda a. a \ false \ true$
- *and* - **if a then b else $false$** ; - $\lambda a. \lambda b. a \ b \ false$
- *or* - **if a then $true$ else b** ; - $\lambda a. \lambda b. a \ true \ b$

Boole'sche Operationen

Boole'sche Operationen

- *not* - **if a then $false$ else $true$** ; - $\lambda a. a \ false \ true$
- *and* - **if a then b else $false$** ; - $\lambda a. \lambda b. a \ b \ false$
- *or* - **if a then $true$ else b** ; - $\lambda a. \lambda b. a \ true \ b$
- *xor* - **if a then $not \ b$ else b** ; -

Boole'sche Operationen

Boole'sche Operationen

- *not* - **if** *a* **then** *false* **else** *true*; - $\lambda a. a \text{ false } true$
- *and* - **if** *a* **then** *b* **else** *false*; - $\lambda a. \lambda b. a \text{ } b \text{ } false$
- *or* - **if** *a* **then** *true* **else** *b*; - $\lambda a. \lambda b. a \text{ } true \text{ } b$
- *xor* - **if** *a* **then** *not b* **else** *b*; - $\lambda a. \lambda b. a \text{ } (not \text{ } b) \text{ } b$

Definition von Wertepaaren

Nützlich für komplexere Berechnungen und Datenstrukturen:
Kodierung zweier Werte (Ausdrücke) als Paar

Definition von Wertepaaren

Nützlich für komplexere Berechnungen und Datenstrukturen:

Kodierung zweier Werte (Ausdrücke) als Paar

Ein Paar (a, b) ist eine Funktion, die eine Funktion z erwartet z auf a und b anwendet

Definition von Wertepaaren

Nützlich für komplexere Berechnungen und Datenstrukturen:

Kodierung zweier Werte (Ausdrücke) als Paar

Ein Paar (a, b) ist eine Funktion, die eine Funktion z erwartet z auf a und b anwendet

Definition

Ein Paar (a, b) wird kodiert als: $\lambda z. z a b$

Definition von Wertepaaren

Nützlich für komplexere Berechnungen und Datenstrukturen:
Kodierung zweier Werte (Ausdrücke) als Paar
Ein Paar (a, b) ist eine Funktion, die eine Funktion z erwartet z auf a und b anwendet

Definition

Ein Paar (a, b) wird kodiert als: $\lambda z. z a b$

Zugriff auf Paarelemente mit *first* und *second*

Definition

$first := \lambda p. p true$

$second := \lambda p. p false$

(vgl. Definition von *true* und *false*)

Beispiele zu Wertepaaren

Beispiele

p sei das Paar $(Karl, Ranseier)$.

$\Rightarrow p := \lambda z. z \text{ Karl Ranseier}$

$first\ p = p\ true = Karl$

$second\ p = p\ false = Ranseier$

Definition der Church-Numerale

Gesucht: Eine Kodierung der Zahlen aus \mathbb{N}_0 .

Definition der Church-Numerale

Gesucht: Eine Kodierung der Zahlen aus \mathbb{N}_0 .

Lösung: 0 als Funktion, die eine andere Funktion 0 mal anwendet, 1 eine, die sie 1 mal anwendet usw.

Definition der Church-Numerale

Gesucht: Eine Kodierung der Zahlen aus \mathbb{N}_0 .

Lösung: 0 als Funktion, die eine andere Funktion 0 mal anwendet, 1 eine, die sie 1 mal anwendet usw.

Definition

Church-Numeral zur Zahl n bildet f auf f^n abbildet.

Definition der Church-Numerale

Gesucht: Eine Kodierung der Zahlen aus \mathbb{N}_0 .

Lösung: 0 als Funktion, die eine andere Funktion 0 mal anwendet, 1 eine, die sie 1 mal anwendet usw.

Definition

Church-Numeral zur Zahl n bildet f auf f^n abbildet.

$$\Rightarrow n \hat{=} \lambda f x. \underbrace{f (f \dots (f x) \dots)}_{n \text{ mal}} = \lambda f x. f^n x$$

Definition der Church-Numerale

Gesucht: Eine Kodierung der Zahlen aus \mathbb{N}_0 .

Lösung: 0 als Funktion, die eine andere Funktion 0 mal anwendet, 1 eine, die sie 1 mal anwendet usw.

Definition

Church-Numeral zur Zahl n bildet f auf f^n abbildet.

$$\Rightarrow n \hat{=} \lambda f x. \underbrace{f (f \dots (f x) \dots)}_{n \text{ mal}} = \lambda f x. f^n x$$

Beispiele

- $0 := \lambda f x. x$ (bildet jede Funktion auf *id* ab)
- $1 := \lambda f x. f x$
- $2 := \lambda f x. f (f x)$ (\equiv *twice*)
- ...

Nachfolgerfunktion: *succ*

Bildet n auf $n + 1$ ab, d.h. *succ* n wendet f ein mal mehr an als n .

Nachfolgerfunktion: *succ*

Bildet n auf $n + 1$ ab, d.h. *succ* n wendet f ein mal mehr an als n .
 $succ := \lambda n f x. f (n f x)$

Arithmetische Operationen - Addition

Nachfolgerfunktion: *succ*

Bildet n auf $n + 1$ ab, d.h. *succ* n wendet f ein mal mehr an als n .
 $succ := \lambda n f x. f (n f x)$

Addition: $+$

Für Addition von m und n addieren: $n f$ als Parameter in $m f$ einsetzen:

Arithmetische Operationen - Addition

Nachfolgerfunktion: *succ*

Bildet n auf $n + 1$ ab, d.h. *succ* n wendet f ein mal mehr an als n .
 $succ := \lambda n f x. f (n f x)$

Addition: $+$

Für Addition von m und n addieren: $n f$ als Parameter in $m f$ einsetzen:

$$(m + n) \hat{=} \lambda f x. f^m (f^n x)$$

Arithmetische Operationen - Addition

Nachfolgerfunktion: *succ*

Bildet n auf $n + 1$ ab, d.h. *succ* n wendet f ein mal mehr an als n .

$succ := \lambda n f x. f (n f x)$

Addition: $+$

Für Addition von m und n addieren: $n f$ als Parameter in $m f$ einsetzen:

$(m + n) \hat{=} \lambda f x. f^m (f^n x)$

$\Rightarrow + := \lambda m n f x. m f (n f x)$

Arithmetische Operationen - Addition

Beispiel: *succ*

succ 1 \equiv *succ* ($\lambda f x. f x$) \equiv

Arithmetische Operationen - Addition

Beispiel: *succ*

$$\mathit{succ} \ 1 \equiv \mathit{succ} \ (\lambda f x. f \ x) \equiv \underline{(\lambda n f x. f \ (n \ f \ x))} \ \underline{(\lambda f x. f \ x)} \equiv$$

Arithmetische Operationen - Addition

Beispiel: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') \underline{f} \underline{x}) \equiv \end{aligned}$$

Arithmetische Operationen - Addition

Beispiel: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f \ x) \equiv (\lambda n f x. f \ (n \ f \ x)) \ (\lambda f x. f \ x) \equiv \\ &\lambda f x. f \ (\underline{(\lambda f' x'. f' \ x') \ f \ x}) \equiv \lambda f x. f \ (f \ x) = 2 \end{aligned}$$

Arithmetische Operationen - Addition

Beispiel: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') \underline{f} \underline{x}) \equiv \lambda f x. f (f x) = 2 \end{aligned}$$

Beispiel: +

$$+ \ 2 \ 3 \equiv (\lambda m n f x. m f (n f x)) \underline{2} \ \underline{3} \equiv$$

Arithmetische Operationen - Addition

Beispiel: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda fx. f x) \equiv (\lambda nfx. f (n f x)) (\lambda fx. f x) \equiv \\ &\lambda fx. f ((\lambda f'x'. f' x') f x) \equiv \lambda fx. f (f x) = 2 \end{aligned}$$

Beispiel: +

$$\begin{aligned} + 2 3 &\equiv (\lambda mnfx. m f (n f x)) 2 3 \equiv \\ &\equiv \lambda fx. 2 f (3 f x) \equiv \end{aligned}$$

Arithmetische Operationen - Addition

Beispiel: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') f x) \equiv \lambda f x. f (f x) = 2 \end{aligned}$$

Beispiel: +

$$\begin{aligned} + 2 3 &\equiv (\lambda m n f x. m f (n f x)) 2 3 \equiv \\ &\equiv \lambda f x. 2 f (3 f x) \equiv \\ &\equiv \lambda f x. (\lambda f' x'. f' (f' x')) f ((\lambda f' x'. f' (f' (f' x')))) f x \equiv \end{aligned}$$

Arithmetische Operationen - Addition

Beispiel: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') f x) \equiv \lambda f x. f (f x) = 2 \end{aligned}$$

Beispiel: +

$$\begin{aligned} + 2 3 &\equiv (\lambda m n f x. m f (n f x)) 2 3 \equiv \\ &\equiv \lambda f x. 2 f (3 f x) \equiv \\ &\equiv \lambda f x. (\lambda f' x'. f' (f' x')) f ((\lambda f' x'. f' (f' (f' x')))) f x \equiv \\ &\equiv \lambda f x. (\lambda x'. f (f x')) (f (f (f x))) \equiv \end{aligned}$$

Arithmetische Operationen - Addition

Beispiel: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') f x) \equiv \lambda f x. f (f x) = 2 \end{aligned}$$

Beispiel: +

$$\begin{aligned} + 2 3 &\equiv (\lambda m n f x. m f (n f x)) 2 3 \equiv \\ &\equiv \lambda f x. 2 f (3 f x) \equiv \\ &\equiv \lambda f x. (\lambda f' x'. f' (f' x')) f ((\lambda f' x'. f' (f' (f' x')))) f x \equiv \\ &\equiv \lambda f x. (\lambda x'. f (f x')) (f (f (f x))) \equiv \\ &\equiv \lambda f x. f (f (f (f (f x)))) = 5 \end{aligned}$$

Arithmetische Operationen - Multiplikation

Multiplikation: *

Für zwei Zahlen m und n : Was passiert, wenn man n auf f und dann m auf das Ergebnis?

Multiplikation: *

Für zwei Zahlen m und n : Was passiert, wenn man n auf f und dann m auf das Ergebnis?

$\Rightarrow n f$ ist f n -mal verkettet. $m (n f)$ ist dann f $m \cdot n$ mal verkettet.

Arithmetische Operationen - Multiplikation

Multiplikation: $*$

Für zwei Zahlen m und n : Was passiert, wenn man n auf f und dann m auf das Ergebnis?

$\Rightarrow n f$ ist f n -mal verkettet. $m (n f)$ ist dann f $m \cdot n$ mal verkettet.

$\Rightarrow * := \lambda mnf. m (n f)$

Arithmetische Operationen - Subtraktion

Gegenstück zur Nachfolgerfunktion *succ* benötigt:

Vorgängerfunktion *pred*.

0 hat eigentlich keine Vorgänger, daher saturierte Subtraktion:

$pred(n) := n \dot{-} 1$, d.h. Vorgänger von 0 ist 0.

Arithmetische Operationen - Subtraktion

Gegenstück zur Nachfolgerfunktion *succ* benötigt:

Vorgängerfunktion *pred*.

0 hat eigentlich keine Vorgänger, daher saturierte Subtraktion:

$pred(n) := n \dot{-} 1$, d.h. Vorgänger von 0 ist 0.

Vorgängerfunktion: *pred*

Hilfsfunktion Φ , die das Paar $p := (a, b)$ auf das Paar $(a + 1, a)$ abbildet:

Arithmetische Operationen - Subtraktion

Gegenstück zur Nachfolgerfunktion *succ* benötigt:

Vorgängerfunktion *pred*.

0 hat eigentlich keine Vorgänger, daher saturierte Subtraktion:

$pred(n) := n \dot{-} 1$, d.h. Vorgänger von 0 ist 0.

Vorgängerfunktion: *pred*

Hilfsfunktion Φ , die das Paar $p := (a, b)$ auf das Paar $(a + 1, a)$ abbildet:

$\Phi := \lambda pz. z (succ (first p)) (first p)$

Arithmetische Operationen - Subtraktion

Gegenstück zur Nachfolgerfunktion *succ* benötigt:

Vorgängerfunktion *pred*.

0 hat eigentlich keine Vorgänger, daher saturierte Subtraktion:

$pred(n) := n \dot{-} 1$, d.h. Vorgänger von 0 ist 0.

Vorgängerfunktion: *pred*

Hilfsfunktion Φ , die das Paar $p := (a, b)$ auf das Paar $(a + 1, a)$ abbildet:

$\Phi := \lambda p z. z (succ (first p)) (first p)$

Φ n mal auf Paar $(0, 0)$ angewandt gibt: $(n, n \dot{-} 1)$

$\Rightarrow pred := \lambda n. second(n \Phi (\lambda z. z 0 0))$

Arithmetische Operationen - Subtraktion

Vorgänger von 0 ist hier 0

⇒ Nur saturierte Subtraktion $m \dot{-} n = \max(m - n, 0)$ möglich

Arithmetische Operationen - Subtraktion

Vorgänger von 0 ist hier 0

⇒ Nur saturierte Subtraktion $m \dot{-} n = \max(m - n, 0)$ möglich

Saturierte Subtraktion: $\dot{-}$

Für $m \dot{-} n$: Vorgängerfunktion $pred$ wird n mal auf m angewandt.

⇒ $\dot{-} := \lambda mn. n \text{ pred } m$

Negative Zahlen im Lambda-Kalkül auch möglich (z.B. Paar aus Church-Numeral und Vorzeichen-Boolean).

Vergleichsoperationen

Wichtigster Vergleich: Test auf 0

Vergleichsoperationen

Wichtigster Vergleich: Test auf 0

Test auf 0: *iszero*

Hilfsfunktion h , die konstant *false* zurückgibt: $h := \lambda x. \text{false}$

Vergleichsoperationen

Wichtigster Vergleich: Test auf 0

Test auf 0: *iszero*

Hilfsfunktion h , die konstant *false* zurückgibt: $h := \lambda x. false$

$h^n = id$ wenn $n = 0$, sonst gibt h^n immer *false* zurück

Vergleichsoperationen

Wichtigster Vergleich: Test auf 0

Test auf 0: *iszero*

Hilfsfunktion h , die konstant *false* zurückgibt: $h := \lambda x. false$

$h^n = id$ wenn $n = 0$, sonst gibt h^n immer *false* zurück

$\Rightarrow n h true$ ist *true* gdw. $n = 0$, sonst *false*.

$\Rightarrow iszero := \lambda n. n (\lambda x. false) true$

Vergleichsoperationen

Kleiner/gleich: \leq

$$m \leq n \Leftrightarrow m - n \leq 0 \Leftrightarrow m \dot{-} n = 0.$$

$$\Rightarrow \leq := \lambda mn. \text{iszero } (\dot{-} \ m \ n)$$

Vergleichsoperationen

Kleiner/gleich: \leq

$$m \leq n \Leftrightarrow m - n \leq 0 \Leftrightarrow m \dot{-} n = 0.$$
$$\Rightarrow \leq := \lambda mn. \text{iszero } (\dot{-} \ m \ n)$$

Größer/gleich: \geq

$$m \geq n \Leftrightarrow n - m \leq 0 \Leftrightarrow n \dot{-} m = 0.$$
$$\Rightarrow \geq := \lambda mn. \text{iszero } (\dot{-} \ n \ m)$$

Vergleichsoperationen

Kleiner/gleich: \leq

$$m \leq n \Leftrightarrow m - n \leq 0 \Leftrightarrow m \dot{-} n = 0.$$
$$\Rightarrow \leq := \lambda mn. \text{iszero } (\dot{-} m n)$$

Größer/gleich: \geq

$$m \geq n \Leftrightarrow n - m \leq 0 \Leftrightarrow n \dot{-} m = 0.$$
$$\Rightarrow \geq := \lambda mn. \text{iszero } (\dot{-} n m)$$

Gleichheit: $=$

$$\Rightarrow = := \lambda mn. \text{and } (\leq m n) (\geq m n)$$

Kleiner und größer: $<$ und $>$

$$m < n \Leftrightarrow \neg(m \geq n)$$

$$\Rightarrow < := \lambda mn. \text{not } (\geq m n)$$

$$m > n \Leftrightarrow \neg(m \leq n)$$

$$\Rightarrow > := \lambda mn. \text{not } (\leq m n)$$

„Echte“ Rekursionen mit Abbruchbedingungen im Lambda-Kalkül etwas umständlich.

„Echte“ Rekursionen mit Abbruchbedingungen im Lambda-Kalkül etwas umständlich.

Beispiel - die Fakultätsfunktion *fac*

$$fac(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot fac(n - 1) & \text{sonst} \end{cases}$$

„Echte“ Rekursionen mit Abbruchbedingungen im Lambda-Kalkül etwas umständlich.

Beispiel - die Fakultätsfunktion *fac*

$$fac(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot fac(n - 1) & \text{sonst} \end{cases}$$

Naheliegend: $fac := \lambda n. (iszero\ n)\ 1\ (*\ n\ (fac\ (pred\ n)))$

„Echte“ Rekursionen mit Abbruchbedingungen im Lambda-Kalkül etwas umständlich.

Beispiel - die Fakultätsfunktion *fac*

$$fac(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot fac(n - 1) & \text{sonst} \end{cases}$$

Naheliegend: $fac := \lambda n. (iszero\ n)\ 1\ (*\ n\ (fac\ (pred\ n)))$

Aber: *fac* muss wieder in *fac* eingesetzt werden, wird unendlich lang.

Beispiel - die Fakultätsfunktion fac

Lösungsmöglichkeit:

Hilfsfunktion fac_rec erhält Parameter n und sich selbst als f und ruft f auf.

Beispiel - die Fakultätsfunktion *fac*

Lösungsmöglichkeit:

Hilfsfunktion *fac_rec* erhält Parameter *n* und sich selbst als *f* und ruft *f* auf. $fac_rec := \lambda fn. (iszero\ n)\ 1\ (*n\ (f\ f\ (pred\ n)))$

Beispiel - die Fakultätsfunktion *fac*

Lösungsmöglichkeit:

Hilfsfunktion *fac_rec* erhält Parameter *n* und sich selbst als *f* und

ruft *f* auf. $fac_rec := \lambda fn. (iszero\ n)\ 1\ (*n\ (f\ f\ (pred\ n)))$

$fac := \lambda n. fac_rec\ fac_rec\ n$

Alternativ: Rekursion mit Fixpunktoperator

Vorteile

- sehr einfache Definition
- ist gleichzeitig eine „Programiersprache“, aber auch ein mathematisches Konstrukt, über das einfach mathematische Aussagen getroffen werden können
- keine Seiteneffekte

Vor- und Nachteile

Vorteile

- sehr einfache Definition
- ist gleichzeitig eine „Programmiersprache“, aber auch ein mathematisches Konstrukt, über das einfach mathematische Aussagen getroffen werden können
- keine Seiteneffekte

Nachteile

- keine „Typsicherheit“ - Funktionen für boole'sche Operanden können auch z.B. auf Zahlen, Paare, ... angewandt werden \Rightarrow schwer nachvollziehbare Fehler
- selbst „einfache“ Operationen (Subtraktion, Gleichheit) sind mit sehr komplexen Umformungen verbunden

- Gut geeignet für mathematische/theoretische Zwecke (Korrektheitsverifikation, Berechenbarkeit, usw.)
- Ungeeignet als Programmiersprache
Aber: mit Typisierung und syntaktischem Zucker gut verwendbar (vgl. LISP, Haskell, ML) und Elemente aus ihm haben auch Einzug in imperative/objektorientierte Sprachen gehalten (Python, C#,...)