The Untyped Lambda Calculus

Manuel Eberl
eberlm@cs.tum.edu

August 20, 2011

What is the Lambda calculus?

- developed by Alonzo Church in the 1930s
- a formal system for definition and analysis of functions
- as powerful as the Turing machine
- forms the basis of functional programming languages

Basic idea

Everything is a function and functions are defined and applied in one expression

Example

Consider these functions:

- *sqr*: expects a number and returns its square
- *twice*: expects a function f and returns $f \circ f$

- $sqr\ 3 = 9$

Basic idea

Everything is a function and functions are defined and applied in one expression

Example

Consider these functions:

- *sqr*: expects a number and returns its square
- *twice*: expects a function f and returns $f \circ f$

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

Formal Definition

The Lambda Calculus has three kinds of expressions:

Definition

Lambda expressions are:

- Variables e.g. a, b, c, $\ldots$
- Abstractions e.g. $\lambda x \,.\, T$ where x is a variable and T is a Lambda expression
- Applications e.g. $S\ T$ where S and T are Lambda expressions

Additionally: parentheses for grouping

Abstractions

Abstraction

$\lambda x.\, T$ - defines a function that maps x to T. T is a Lambda expression which usually contains x (but not necessarily)

Example

- $\lambda x.\, x$ (the identity function id)
- $\lambda x.\, a$ (a function that always returns a)

Applications

Application

$S\ \ T$ - applies the function S to the expression T.

Definition (β Reduction)

If S is of the shape $(\lambda x. R)\ T$ (so-called redex for reducible expression): Reduction by replacing all x in R by T

$$(\lambda \mathbf{x}. \mathbf{x}\ y\ \mathbf{x})\ T \longrightarrow T\ y\ T$$

Example

$$(\lambda \mathbf{x}. 42\ \mathbf{x}\ y\ \mathbf{x}\ 1337)\ \mathbf{wuppdi}$$

$$\downarrow \beta$$

$$42\ \mathit{wuppdi}\ \ y\ \ \mathit{wuppdi}\ 1337$$

Applications

Examples

- $id\ a$

Applications

Examples

- *id a* - yields *a*
- *id id*

Applications

Examples

- *id a* - yields *a*
- *id id* - yields *id*
- *sqr* 3

Applications

Examples

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

Applications

Examples

- *id a* - yields *a*
- *id id* - yields *id*
- *sqr* 3 - yields 9
- *twice sqr* - yields a “to the 4^{th} power” function
- *twice sqr* 3

Applications

Examples

- $id\ a$ - yields a
- $id\ id$ - yields id
- $sqr\ 3$ - yields 9
- $twice\ sqr$ - yields a "to the 4th power" function
- $twice\ sqr\, 3$ - yields 81

Caution: applications are left-associative. $f\ f\ x$ means $(f\ f)\ x$, not $f\ (f\ x)$

Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define $add\ a\ b = a + b$?

Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define $add\ a\ b = a + b$?

Solution: nesting functions - $add(a)$ returns a function $add_a(b)$ with constant a, which maps b to $a + b$.
This nesting is called "Currying".

Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define $add\ a\ b = a + b$?

Solution: nesting functions - $add(a)$ returns a function $add_a(b)$ with constant a, which maps b to $a + b$.
This nesting is called "Currying".

Example 1 - Addition

- Let $+$ be a function that adds its two parameters

Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define $add\ a\ b = a + b$?

Solution: nesting functions - $add(a)$ returns a function $add_a(b)$ with constant a, which maps b to $a + b$.
This nesting is called "Currying".

Example 1 - Addition

- Let $+$ be a function that adds its two parameters
- $+\ 4 = +_4$ ist an "Increase by 4" function

Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define $add\ a\ b = a + b$?

Solution: nesting functions - $add(a)$ returns a function $add_a(b)$ with constant a, which maps b to $a + b$.
This nesting is called "Currying".

Example 1 - Addition

- Let $+$ be a function that adds its two parameters
- $+\ 4 = +_4$ ist an "Increase by 4" function
- $+\ 4\ 2 = +_{4,2}$ is equivalent to 6

Functions with more than one parameter - Currying

Example 2- reverse application function

Definition: *reverse* x y returns y x

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

Functions with more than one parameter - Currying

Example 2- reverse application function

Definition: *reverse* x y returns y x

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

$reverse\ a\ b \equiv$

Functions with more than one parameter - Currying

Example 2- reverse application function

Definition: *reverse* x y returns y x

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

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

Functions with more than one parameter - Currying

Example 2- reverse application function

Definition: *reverse* x y returns y x

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

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

Functions with more than one parameter - Currying

Example 2- reverse application function

Definition: *reverse* x y returns y x

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

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

Shorthand for nested functions: Instead of $\lambda x.\lambda y.\lambda z\ T$ write $\lambda xyz.\, T$.

Example: $reverse := \lambda xy.\, y\ x$

Free variables and bound variables

A variable in an abstraction is called bound if it was defined by a lambda.
Non-bound variables are free. (cf. $\exists$ and $\forall$)

Example

- $\lambda xy.x\ y$ - x and y are bound
- $\lambda x.x\ y$ - x is bound, y is free
- $(\lambda x.x)\, x$ - inner x is bound, outer x is free
- $\lambda x.\lambda x.x$ - x is bound and "hides" the outer x

α conversion

What is the difference between $\lambda x. x$ and $\lambda y. y$?

α conversion

What is the difference between $\lambda x.\, x$ and $\lambda y.\, y$?
Answer: none at all! Names of bound variables are irrelevant.
Renaming bound variables is called α conversion.

α conversion

What is the difference between $\lambda x.\, x$ and $\lambda y.\, y$?
Answer: none at all! Names of bound variables are irrelevant.
Renaming bound variables is called α conversion.

Definition (α conversion)

A λ abstraction $\lambda x.\, A$ is equivalent to $\lambda y.\, A[x := y]$

where $A[x := y]$ means: all occurences of x in A are replaced with y (but: take variable hiding into account)

α conversion

What is the difference between $\lambda x.\, x$ and $\lambda y.\, y$?
Answer: none at all! Names of bound variables are irrelevant.
Renaming bound variables is called α conversion.

Definition (α conversion)

A λ abstraction $\lambda x.\, A$ is equivalent to $\lambda y.\, A[x := y]$

where $A[x := y]$ means: all occurences of x in A are replaced with y (but: take variable hiding into account)

Expressions that can be converted to one another by α conversion are called α-equivalent.

Collision of variable names

Caution: When replacing variables or applying β reduction, variables defined on deeper levels must not be overwritten!

Example - collision after α-Konversion

$\lambda xy.\,x$ must not be α converted to $\lambda y.\lambda y.\,y$

Collision of variable names

Caution: When replacing variables or applying β reduction, variables defined on deeper levels must not be overwritten!

Example - collision after α-Konversion

$\lambda xy.\, x$ must not be α converted to $\lambda y.\lambda y.\, y$

Solution: replace colliding variable as well:
$\lambda xy.\, x \;\equiv\; \lambda xz.\, x \;\equiv\; \lambda yz.\, y$

Collision of variable names

Caution: When replacing variables or applying β reduction, variables defined on deeper levels must not be overwritten!

Example - collision after α-Konversion

$\lambda xy.\, x$ must not be α converted to $\lambda y.\lambda y.\, y$

Solution: replace colliding variable as well:
$\lambda xy.\, x \;\equiv\; \lambda xz.\, x \;\equiv\; \lambda yz.\, y$

Example - collision after β reduction

$(\lambda ab.\, a)\ b$ - first b bound, second b free.

Collision of variable names

Caution: When replacing variables or applying β reduction, variables defined on deeper levels must not be overwritten!

Example - collision after α-Konversion

$\lambda xy.\, x$ must not be α converted to $\lambda y.\lambda y.\, y$

Solution: replace colliding variable as well:
$\lambda xy.\, x \;\equiv\; \lambda xz.\, x \;\equiv\; \lambda yz.\, y$

Example - collision after β reduction

$(\lambda ab.\, a)\; b$ - first b bound, second b free.

After β reduktion:
$\lambda b.\, b$ - bound b is returned - different result!

Collision of variable names

Caution: When replacing variables or applying β reduction, variables defined on deeper levels must not be overwritten!

Example - collision after α-Konversion

$\lambda xy.\,x$ must not be α converted to $\lambda y.\lambda y.\,y$

Solution: replace colliding variable as well:
$\lambda xy.\,x \;\equiv\; \lambda xz.\,x \;\equiv\; \lambda yz.\,y$

Example - collision after β reduction

$(\lambda ab.\,a)\; b$ - first b bound, second b free.

After β reduktion:
$\lambda b.\,b$ - bound b is returned - different result!$\Rightarrow$ Rename bound variable: $\lambda b'.\,b$

Data types?

Problem: no data types such as numbers, booleans, lists, strings - there are only functions. But: Lambda calculus is Turing complete, so it has to be possible to e.g. do calculations with numbers. But how?

Solution: encode all data types as functions!

Definition of true and false

true and *false* as functions of two parameters: $\mathit{true}\ a\ b \equiv a$ and $\mathit{false}\ a\ b \equiv b$.

Definition

$\mathit{true} := \lambda ab.\, a$
$\mathit{false} := \lambda ab.\, b$

Booleans can be used to make "branches".
"if A then B else C" where A is a boolean can be expressed as $A\ B\ C$.

Boolean operations

Boolean operations

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

Boolean operations

Boolean operations

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

Boolean operations

Boolean operations

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

Boolean operations

Boolean operations

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

Boolean operations

Boolean operations

- *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*; -

Boolean operations

Boolean operations

- *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; - $\lambda a.\lambda b.\, a\ true\ b$

Boolean operations

Boolean operations

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

Boolean operations

Boolean operations

- *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; - $\lambda a.\lambda b.\, a\ (not\ b)\ b$

Definition of pairs of values

Useful for more complex calculations and data structures: encoding two values (expressions) in a pair

Definition of pairs of values

Useful for more complex calculations and data structures: encoding two values (expressions) in a pair

A pair (a, b) is a function that expects a function z and applies z to a and b

Definition of pairs of values

Useful for more complex calculations and data structures: encoding two values (expressions) in a pair
A pair (a, b) is a function that expects a function z and applies z to a and b

Definition

A pair (a, b) is encoded as: $\lambda z.\, z\ a\ b$

Definition of pairs of values

Useful for more complex calculations and data structures: encoding two values (expressions) in a pair
A pair (a, b) is a function that expects a function z and applies z to a and b

Definition

A pair (a, b) is encoded as: $\lambda z.\, z\ a\ b$

Accessing pair elements with *first* and *second*

Definition

$\mathit{first} := \lambda p.\, p\ \mathit{true}$
$\mathit{second} := \lambda p.\, p\ \mathit{false}$
(cf. Definition of *true* and *false*)

Examples of pairs

Beispiele

Let p be the pair $(Karl, Ranseier)$.
$\Rightarrow p := \lambda z.\, z \; Karl \; Ranseier$
$first \; p = p \; true = Karl$
$second \; p = p \; false = Ranseier$

Definition of Church numerals

How can we encode numbers from $\mathbb{N}_0$?

Definition of Church numerals

How can we encode numbers from $\mathbb{N}_0$?
Solution: 0 is a function that applies another function 0 times, 1 applies it 1 times and so on

Definition of Church numerals

How can we encode numbers from $\mathbb{N}_0$?
Solution: 0 is a function that applies another function 0 times, 1 applies it 1 times and so on

Definition

The Church numeral for the number n maps f to f^n.

Definition of Church numerals

How can we encode numbers from $\mathbb{N}_0$?
Solution: 0 is a function that applies another function 0 times, 1 applies it 1 times and so on

Definition

The Church numeral for the number n maps f to f^n.
$\Rightarrow n \hat{=} \lambda fx.\, \underbrace{f\ (f \ldots (f}_{n \, \text{mal}}\ x) \ldots) = \lambda fx.\, f^n\ x$

Definition of Church numerals

How can we encode numbers from $\mathbb{N}_0$?
Solution: 0 is a function that applies another function 0 times, 1 applies it 1 times and so on

Definition

The Church numeral for the number n maps f to f^n.
$\Rightarrow n \hat{=} \lambda fx.\ \underbrace{f\ (f \ldots (f}_{n \,\mathrm{mal}}\ x) \ldots) = \lambda fx.\ f^n\ x$

Examples

- $0 := \lambda fx.\ x$ (maps all functions to id)
- $1 := \lambda fx.\ f\ x$
- $2 := \lambda fx.\ f\ (f\ x)$ ($\equiv$ *twice*)
- $\ldots$

Arithmetic operations - Addition

Successor function: *succ*

Maps n to $n+1$, i.e. *succ* n applies f once more than n.

Arithmetic operations - Addition

Successor function: *succ*

Maps n to $n+1$, i.e. $succ\ n$ applies f once more than n.
$succ := \lambda nfx.\, f\ (n\ f\ x)$

Arithmetic operations - Addition

Successor function: *succ*

Maps n to $n+1$, i.e. $succ\ n$ applies f once more than n.
$succ := \lambda nfx.\, f\ (n\ f\ x)$

Addition: $+$

In order to add m and n: put $n\ f$ as parameter into $m\ f$:

Arithmetic operations - Addition

Successor function: *succ*

Maps n to $n+1$, i.e. $succ\ n$ applies f once more than n.
$succ := \lambda nfx.\, f\ (n\ f\ x)$

Addition: $+$

In order to add m and n: put $n\ f$ as parameter into $m\ f$:
$(m+n) \hat{=} \lambda fx.f^m\ (f^n\ x)$

Arithmetic operations - Addition

Successor function: *succ*

Maps n to $n+1$, i.e. $succ\ n$ applies f once more than n.
$succ := \lambda nfx.\, f\ (n\ f\ x)$

Addition: $+$

In order to add m and n: put $n\ f$ as parameter into $m\ f$:
$(m+n) \widehat{=} \lambda fx.f^m\ (f^n\ x)$

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

Arithmetic operations - Addition

Example: *succ*

$succ\ 1 \equiv succ\ (\lambda fx.\, f\ x) \equiv$

Arithmetic operations - Addition

Example: *succ*

$$succ\ 1 \equiv succ\ (\lambda fx.\, f\ \ x) \equiv \underline{(\lambda nfx.\, f\ \ (n\ \ f\ \ x))}\ \underline{(\lambda fx.\, f\ \ x)} \equiv$$

Arithmetic operations - Addition

Example: *succ*

$succ\ 1 \equiv succ\ (\lambda fx.\ f\ \ x) \equiv \underline{(\lambda nfx.\ f\ \ (n\ \ f\ \ x))}\ \underline{(\lambda fx.\ f\ \ x)} \equiv$
$\lambda fx.\ f\ \ (\underline{(\lambda f'x'.\ f'\ \ x')}\ \underline{f}\ \ \underline{x}) \equiv$

Arithmetic operations - Addition

Example: *succ*

$succ\ 1 \equiv succ\ (\lambda fx.\, f\ x) \equiv \underline{(\lambda nfx.\, f\ (n\ f\ x))}\ \underline{(\lambda fx.\, f\ x)} \equiv$
$\lambda fx.\, f\ (\underline{(\lambda f'x'.\, f'\ x')}\ \underline{f}\ \underline{x}) \equiv \lambda fx.\, f\ (f\ x) = 2$

Arithmetic operations - Addition

Example: *succ*

$succ\ 1 \equiv succ\ (\lambda fx.\, f\ \ x) \equiv \underline{(\lambda nfx.\, f\ \ (n\ f\ \ x))}\ \underline{(\lambda fx.\, f\ \ x)} \equiv$
$\lambda fx.\, f\ \ (\underline{(\lambda f'x'.\, f'\ \ x')}\ \underline{f}\ \underline{x}) \equiv \lambda fx.\, f\ \ (f\ \ x) = 2$

Example: $+$

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

Arithmetic operations - Addition

Example: *succ*

$succ\ 1 \equiv succ\ (\lambda fx.\, f\ \ x) \equiv \underline{(\lambda nfx.\, f\ \ (n\ f\ \ x))}\ \underline{(\lambda fx.\, f\ \ x)} \equiv$
$\lambda fx.\, f\ \ (\underline{(\lambda f'x'.\, f'\ \ x')}\ \underline{f}\ \underline{x}) \equiv \lambda fx.\, f\ \ (f\ \ x) = 2$

Example: $+$

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

Arithmetic operations - Addition

Example: *succ*

$succ\ 1 \equiv succ\ (\lambda fx.\, f\ \ x) \equiv \underline{(\lambda nfx.\, f\ \ (n\ \ f\ \ x))}\ \underline{(\lambda fx.\, f\ \ x)} \equiv$
$\lambda fx.\, f\ \ (\underline{(\lambda f'x'.\, f'\ \ x')}\ \underline{f}\ \ \underline{x}) \equiv \lambda fx.\, f\ \ (f\ \ x) = 2$

Example: $+$

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

Arithmetic operations - Addition

Example: *succ*

$succ\ 1 \equiv succ\ (\lambda fx.\, f\ \ x) \equiv \underline{(\lambda nfx.\, f\ \ (n\ f\ \ x))}\ \underline{(\lambda fx.\, f\ \ x)} \equiv$
$\lambda fx.\, f\ \ (\underline{(\lambda f'x'.\, f'\ \ x')}\ \underline{f}\ \underline{x}) \equiv \lambda fx.\, f\ \ (f\ \ x) = 2$

Example: $+$

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

Arithmetic operations - Addition

Example: *succ*

$succ\ 1 \equiv succ\ (\lambda fx.\, f\ x) \equiv \underline{(\lambda nfx.\, f\ (n\ f\ x))}\ \underline{(\lambda fx.\, f\ x)} \equiv$
$\lambda fx.\, f\ (\underline{(\lambda f'x'.\, f'\ x')}\ \underline{f}\ \underline{x}) \equiv \lambda fx.\, f\ (f\ x) = 2$

Example: $+$

$+\ 2\ 3 \equiv \underline{(\lambda mnfx.\, m\ f (n\ f\ x))}\ \underline{2}\ \underline{3} \equiv$
$\equiv \lambda fx.\, 2\ f\ (3\ f\ x) \equiv$
$\equiv \lambda fx.\underline{(\lambda f'x'.\, f'\ (f'\ x'))}\ \underline{f}\ (\underline{(\lambda f'x'.\, f'\ (f'\ (f'\ x')))}\ \underline{f}\ \underline{x}) \equiv$
$\equiv \lambda fx.\underline{(\lambda x'.\, f\ (f\ x'))}\underline{(f\ (f\ (f\ x)))} \equiv$
$\equiv \lambda fx.\, f\ (f\ (f\ (f\ (f\ x)))) = 5$

Arithmetic operations - Multiplication

Multiplication: $*$

For numbers m and n: what happens when n is applied to f and m is applied to the result?

Arithmetic operations - Multiplication

Multiplication: $*$

For numbers m and n: what happens when n is applied to f and m is applied to the result?
$\Rightarrow$ $n\ f$ is f chained n times. Therefore, $m\ (n\ f)$ is f chained $m \cdot n$ times.

Arithmetic operations - Multiplication

Multiplication: $*$

For numbers m and n: what happens when n is applied to f and m is applied to the result?
$\Rightarrow$ $n\ f$ is f chained n times. Therefore, $m\ (n\ f)$ is f chained $m \cdot n$ times.

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

Arithmetic operations - Subtraction

We need the opposite of *succ*: a predecessor function.
0 has no predecessor, therefore saturated subtraction:
$pred(n) := n \dot{-} 1$, i.e. predecessor of 0 defined as 0.

Arithmetic operations - Subtraction

We need the opposite of *succ*: a predecessor function.
0 has no predecessor, therefore saturated subtraction:
$pred(n) := n \dot{-} 1$, i.e. predecessor of 0 defined as 0.

Predecessor function: *pred*

Helper function Φ, that maps $p := (a, b)$ to $(a + 1, a)$:

Arithmetic operations - Subtraction

We need the opposite of *succ*: a predecessor function.
0 has no predecessor, therefore saturated subtraction:
$pred(n) := n \dot{-} 1$, i.e. predecessor of 0 defined as 0.

Predecessor function: *pred*

Helper function Φ, that maps $p := (a, b)$ to $(a + 1, a)$:
$\Phi := \lambda pz.\, z\ (succ\ (first\ p))\ (first\ p)$

Arithmetic operations - Subtraction

We need the opposite of *succ*: a predecessor function.
0 has no predecessor, therefore saturated subtraction:
$pred(n) := n \dot{-} 1$, i.e. predecessor of 0 defined as 0.

Predecessor function: *pred*

Helper function Φ, that maps $p := (a, b)$ to $(a + 1, a)$:
$\Phi := \lambda pz.\, z\ (succ\ (first\ p))\ (first\ p)$

$\Phi\ n$ applied to $(0, 0)$ n times yields: $(n, n \dot{-} 1)$
$\Rightarrow\ pred := \lambda n.\, second(n\ \Phi\ (\lambda z.\, z\ 0\ 0))$

Arithmetic operations - Subtraction

Predecessor of 0 is 0.
$\Rightarrow$ Only saturated subtraction possible: $m \dot{-} n = max(m - n, 0)$

Arithmetic operations - Subtraction

Predecessor of 0 is 0.
$\Rightarrow$ Only saturated subtraction possible: $m \dot{-} n = max(m - n, 0)$

Saturated subtraction: $\dot{-}$

For $m \dot{-} n$: predecessor function *pred* is applied n times to m.
$\Rightarrow \dot{-} := \lambda mn.\, n\ pred\ \ m$

Negative numbers can be expressed in Lambda calculus. (e.g. pair of Church numeral and sign boolean)

Comparison operators

Most important comparison: test if number is 0

Comparison operators

Most important comparison: test if number is 0

iszero

Helper function h, that always returns *false*: $h := \lambda x.\mathit{false}$

Comparison operators

Most important comparison: test if number is 0

iszero

Helper function h, that always returns *false*: $h := \lambda x.\, \mathit{false}$

$h^n = \mathit{id}$ if $n = 0$; if $n \neq 0$, h^n returns *false*

Comparison operators

Most important comparison: test if number is 0

iszero

Helper function h, that always returns *false*: $h := \lambda x.\, \mathit{false}$

$h^n = \mathit{id}$ if $n = 0$; if $n \neq 0$, h^n returns *false*
$\Rightarrow$ $n\ h\ \mathit{true}$ is *true* iff $n = 0$, *false* otherwise.

$\Rightarrow$ $\mathit{iszero} := \lambda n.\, n\ (\lambda x.\, \mathit{false})\ \mathit{true}$

Comparison operators

Less or equal: $\leq$

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

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

Comparison operators

Less or equal: $\leq$

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

Greater or equal: $\geq$

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

Comparison operators

Less or equal: $\leq$

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

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

Greater or equal: $\geq$

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

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

Equal: $=$

$\Rightarrow = := \lambda mn.\, and\ (\leq\ \ m\ n)\ (\geq\ \ m\ n)$

Comparison operators

Less/greater: $<$ und $>$

$m < n \Leftrightarrow \neg(m \geq n)$
$\Rightarrow < := \lambda mn.\, not\ (\geq\ \ m\ n)$

$m > n \Leftrightarrow \neg(m \leq n)$
$\Rightarrow > := \lambda mn.\, not\ (\leq\ \ m\ n)$

Recursion

"Real" recursions with terminatio conditions are somewhat complicated in Lambda calculus

Recursion

"Real" recursions with terminatio conditions are somewhat complicated in Lambda calculus

Example - the factorial *fac*

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

Recursion

“Real” recursions with terminatio conditions are somewhat complicated in Lambda calculus

Example - the factorial *fac*

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

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

Recursion

“Real” recursions with terminatio conditions are somewhat complicated in Lambda calculus

Example - the factorial *fac*

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

Intuitive solution: $fac := \lambda n.\,(iszero\ n)\ 1\ (*\ n\ (fac\ (pred\ n)))$
But: *fac* has to be plugged into *fac* again, expression grows infinitely

Recursion

Example - the factorial *fac*

One possible solution:
Helper function fac_rec receives parameter n and itself as f, then calls f.

Recursion

Example - the factorial *fac*

One possible solution:

Helper function fac_rec receives parameter n and itself as f, then calls $f.fac\_rec := \lambda fn.(iszero\ \ n)\ 1\ (*n\,(f\ f\ (pred\ n)))$

Recursion

Example - the factorial *fac*

One possible solution:
Helper function fac_rec receives parameter n and itself as f, then calls f.$fac\_rec := \lambda fn. (iszero\ \ n)\ 1\ (*n\,(f\ f\ (pred\ n)))$
$fac := \lambda n.\, fac\_rec\ \ fac\_rec\ \ n$

Alternatively: recursion with fixed point operator

Advantages and disadvantages

Advantages

- very simple definition
- is both a “programming language” and a mathematical construct about which mathematical statements can be made
- no side effects

Advantages and disadvantages

Advantages

- very simple definition
- is both a "programming language" and a mathematical construct about which mathematical statements can be made
- no side effects

Disadvantages

- no "type safety" - functions for booleans can be applied to numbers, pairs, $\ldots \Rightarrow$ mistakes can produce results that are difficult to comprehend
- even "simple" operations (subtraction, equality) need many rewriting operations

Conclusion

- Very useful for mathematical/theoretical purposes (verification, computability and so on)
- Unsuitable as an actual programming language
  But: usable with typing and syntactic sugar (cf. LISP, Haskell, ML) and elements of Lambda calculus have found their way into imperative/object oriented languages (Python, C#,. . .)