IA014: Advanced Functional Programming

2. Untyped Lambda Calculus

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Formal development

Syntax

The set of *λ-terms* is defined by the following BNF grammar:

$$\begin{array}{rcll} M & ::= & x & \text{variable} \\ & | & MM' & \text{application} \\ & | & \lambda x.M & \text{abstraction} \end{array}$$

- where $x, y, z, \ldots$ are *variables* from a countable set Var
- we use uppercase letters $M, N, \ldots$ to denote λ-terms
- $(,)$ are used whenever the meaning is not clear

examples

$$\boldsymbol{I} \equiv \lambda x.x \qquad \boldsymbol{K} \equiv \lambda x.(\lambda y.x)$$

$$\boldsymbol{\omega} \equiv \lambda x.(x\ x) \qquad \boldsymbol{S} \equiv \lambda x.(\lambda y.(\lambda z.((x\ z)(y\ z))))$$

$$\boldsymbol{\Omega} \equiv \boldsymbol{\omega}\ \boldsymbol{\omega} \equiv (\lambda x.(x\ x))(\lambda x.(x\ x))$$

Syntactic conventions

- $\lambda x_1 x_2 \ldots x_n.M$ means $\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots))$
- $M_1 M_2 M_3 \ldots M_n$ means $(\ldots((M_1 M_2)M_3)\ldots)$
  application associates to the left
- $\lambda x.x\ y$ means $\lambda x.(x\ y)$
  function application takes precedence
- spaces have no meaning

Examples

	simplified
$\lambda x.(x\ y\ z)$	$\lambda x.x\ y\ z$
$(\lambda x.x)\ y$	$(\lambda x.x)\ y$
$\lambda x.(\lambda y.x)$	$\lambda xy.x$
$\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z))))$	$\lambda xyz.(x\ z)\ (y\ z)$

Variable binding

In a term $\lambda x.M$ the variable x is *bound* in the *body* M.

- λ is an *abstraction operator* and M is its *scope*.
- *occurrence* of a variable is *free* if it is not bound

Formally The set of *free variables,* $FV(M)$, in a λ-term M is defined as:

$$\begin{aligned} FV(x) &= x \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \end{aligned}$$

If $FV(M) = \emptyset$, then M is called a *closed term* or a *combinator*.

Binding examples

term	free	bound
$\lambda x.x\ y$	y	x
$(\lambda x.x\ y)\ x$	x, y	x
$\lambda xy.x$	$--$	x, y
$(\lambda x.x\ x)(\lambda x.x\ x)$	$--$	x

Convention we will use: the names of free and bound variables will always be different.

Substitution

$M[x := N]$ – the *substitution* of N for the free occurrences of x in M, is defined as:

$$
\begin{aligned}
x[x := N] &= N \\
y[x := N] &= y && \text{if } y \neq x \\
(M_1 M_2)[x := N] &= (M_1[x := N])(M_2[x := N]) \\
(\lambda x.M)[x := N] &= \lambda x.M \\
(\lambda y.M)[x := N] &= \lambda y.(M[x := N]) && \text{if } x \neq y \\
& && \text{provided } y \notin FV(N)
\end{aligned}
$$

In the last case, if $y \notin FV(N)$ we say that y is *fresh* for N.

Substitution II

comments

The freshness requirement is absolutely crucial:

$$\lambda y.x[x := y] \neq \lambda y.y \qquad \textit{name capture}$$

Our substitution is *capture-avoiding*.

Alternative definition of substitution (only the last case):

$$(\lambda y.M)[x := N] = \lambda z.M[y := z][x := N]$$

provided $x \neq y$ and $z \notin FV(M) \cup FV(N) \cup \{x\}$

examples

$$(\lambda y.x)[y := x] = \lambda y.x$$
$$(\lambda y.x)[x := y] = \lambda z.y$$
$$(\lambda x.(\lambda y.y\ z)\ (\lambda w.w)\ z\ x)[y := z] = \ldots$$

α-conversion and equivalence

Observation: The terms $\lambda x.x$ and $\lambda y.y$ are, for all practical purposes, equivalent.

The *α-equivalence relation*, $=_\alpha$, is defined by the following rules:

$$\frac{M[x := z] =_\alpha N[y := z] \qquad z \notin FV(M) \cup FV(N)}{\lambda x.M =_\alpha \lambda y.N}$$

$$\frac{}{x =_\alpha x} \qquad \frac{M_1 =_\alpha M_2 \qquad N_1 =_\alpha N_2}{M_1 N_1 =_\alpha M_2 N_2}$$

Renaming bound variables is often called *α-conversion*.

Compare with the previous slide!

β-reduction

The idea of *function application* is expressed by the following equational axiom:

$$(\lambda x.M)N = M[x := N] \qquad (\beta)$$

In computational context, this can be though of as a single computation step, called *β-reduction step*:

$$(\lambda x.M)N \to M[x := N]$$

In this context, $(\lambda x.M)N$ is called a *redex* (reducible expression).

Full β-reduction

- a redex can appear anywhere in a term M
- *Full β-reduction* is then given by the following set of rules:

$$\frac{}{(\lambda x.M)N \to_\beta M[x := N]} \qquad \frac{M_1 \to_\beta M_2}{\lambda x.M_1 \to_\beta \lambda x.M_2}$$

$$\frac{M_1 \to_\beta M_2}{M_1 N \to_\beta M_2 N} \qquad \frac{N_1 \to_\beta N_2}{M N_1 \to_\beta M N_2}$$

We also define a *reflexive transitive closure* $\to_\beta^*$ of $\to_\beta$:
$M \to_\beta^* N$ iff either $M = N$ or there is a sequence

$$M \to_\beta M_1 \to_\beta M_2 \to_\beta \ldots \to_\beta N$$

- We will often drop the index β from $\to_\beta$.
- We will work modulo α-equivalence.

β-reduction – examples

$$(\lambda x.\lambda y.yx)(\lambda z.u) \to \lambda y.y(\lambda z.u)$$
$$(\lambda x.xx)(\lambda z.u) \to (\lambda z.u)(\lambda z.u)$$
$$(\lambda y.y\ a)((\lambda x.x)(\lambda z.(\lambda u.u)\ z)) \to (\lambda y.y\ a)(\lambda z.(\lambda u.u)\ z)$$
$$(\lambda y.y\ a)((\lambda x.x)(\lambda z.(\lambda u.u)\ z)) \to (\lambda y.y\ a)((\lambda x.x)(\lambda z.z))$$
$$(\lambda y.y\ a)((\lambda x.x)(\lambda z.(\lambda u.u)\ z)) \to ((\lambda x.x)(\lambda z.(\lambda u.u)z))a$$

$\boldsymbol{\Omega} \equiv (\lambda x.x\ x)(\lambda x.x\ x) \to (\lambda x.x\ x)(\lambda x.x\ x) \to \ldots$

$\boldsymbol{K}a\boldsymbol{\Omega} \equiv (\lambda xy.x)\ a\ \boldsymbol{\Omega}$

$\boldsymbol{K}a\boldsymbol{\Omega} \to a$

$\boldsymbol{K}a\boldsymbol{\Omega} \to \boldsymbol{K}a\boldsymbol{\Omega} \to a$

$\boldsymbol{K}a\boldsymbol{\Omega} \to \boldsymbol{K}a\boldsymbol{\Omega} \to \boldsymbol{K}a\boldsymbol{\Omega} \to a$

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

$(\lambda x.x\ x)((\lambda y.y)z) \to ((\lambda y.y)z)((\lambda y.y)z) \to^2 z\ z$

Normal form, Questions

Normal form

Term M is in *β-normal form* iff it contains no β-redexes.

Q1: Can a term have more than one normal form?

Q2: Does a β-reduction always terminate?

Q3: Does the order of evaluation matter?

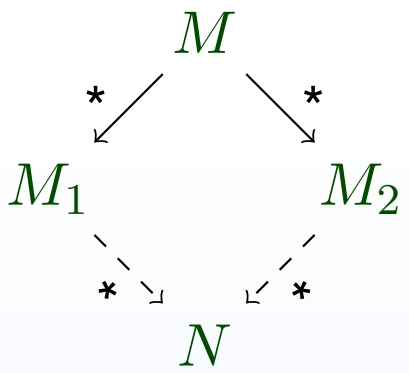
Q4: In which order should we select the redexes?

Confluence

Q1: Can a term have more than one normal form?

Theorem (Church-Rosser)

Let M be a λ-term. If $M \to^* M_1$ and $M \to^* M_2$, then there is N such that $M_1 \to^* N$ and $M_2 \to^* N$



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

$(\lambda x.x\ x)((\lambda y.y)z) \to ((\lambda y.y)z)((\lambda y.y)z) \to^2 z\ z$

Non-termination

Q2: Does a β-reduction always terminate?

- This is called *strong normalization property.*
- Untyped λ-calculus *is not* strongly normalizing:

$$\boldsymbol{\Omega} \equiv (\lambda x.x\ x)(\lambda x.x\ x) \to (\lambda x.x\ x)(\lambda x.x\ x) \to (\lambda x.x\ x)(\lambda x.x\ x) \to \ldots$$

- Simply-typed λ-calculus *is* strongly normalizing.
  - What does that mean?
  - . . . it is decidable whether program halts . . .
  - Then it is not *Turing complete*! (computationally universal)

Nondeterminism

Q3: Does the order of evaluation matter?

Compare

$$(\lambda x.z)((\lambda w.www)(\lambda w.www)) \rightarrow$$
$$(\lambda x.z)((\lambda w.www)(\lambda w.www)(\lambda w.www)) \rightarrow$$
$$(\lambda x.z)((\lambda w.www)(\lambda w.www)(\lambda w.www)(\lambda w.www)) \rightarrow$$
$$\ldots$$

with

$$(\lambda x.z)((\lambda w.www)(\lambda w.www)) \rightarrow z$$

The choice of evaluation (reduction) strategy matters!

Evaluation Strategies

Q4: In which order should we select the redexes?

- Full β-reduction
  - Any redex can be selected.
  - As defined on slide 11
- Applicative order
  - form of strict/eager evaluation
  - *leftmost innermost*
  - evaluate arguments (left to right) before applying function
- Normal order
  - form of non-strict evaluation
  - *leftmost outermost*
  - applying function before evaluating arguments
  - *complete* – if there is a normal form, it would be eventually reached

Call-By-Value semantics of λ

- strict/eager evaluation strategy
- unlike applicative order, does not reduce the body of the function before applying the function
- used, e.g., by ML

$M ::=$	x	variable
	$\mid MM'$	application
	$\mid \lambda x.M$	abstraction
$V ::=$	$\lambda x.M$	abstraction *value*

$$\frac{}{(\lambda x.M)V \to_\beta M[x := V]}$$

$$\frac{M_1 \to_\beta M_2}{M_1 N \to_\beta M_2 N} \qquad \frac{N_1 \to_\beta N_2}{V N_1 \to_\beta V N_2}$$

β-equivalence

β-reduction is directional. However, we can also use it "bidirectionally"

β-equivalence

Two terms M and N are said to be β-equivalent (written as $M =_\beta N$) if either:

1. $M \equiv N$, or
2. there is a sequence of terms $M = M_0, M_1, \ldots M_k = N$ s.t. for all $1 \leq i \leq k$ either
   - $M_{i-1} \to M_i$, or
   - $M_i \to M_{i-1}$

(i.e. $=_\beta$ is a reflexive, symmetric and transitive closure of $\to_\beta$)

From Church-Rosser, two normalizing terms are β-equivalent iff their normal forms are equal.

η-conversion (reduction)

Let us assume we have the following term:

$$N \equiv \lambda x.M\ x$$

and that x is not free in M.

Observation: $(\lambda x.M\ x)N \to_\beta MN$

η-reduction rule: $$\frac{x \notin FV(M)}{(\lambda x.M\ x) \to_\eta M}$$

- removes redundant λ-abstractions
- we can define $\to_{\beta\eta}$-reduction
- $\to_{\beta\eta}$-reduction is (again) confluent

Encoding Mathematics in λ-calculus

Booleans

truth values

- true $:= \lambda x.\lambda y.x$
- false $:= \lambda x.\lambda y.y$

true $t\,f$

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

$\rightarrow (\lambda y.t)f$

$\rightarrow t$

false $t\,f$

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

$\rightarrow (\lambda y.y)f$

$\rightarrow f$

conditional statement

- if $:= \lambda xyz.xyz$

Boolean operators

truth values

- true $:= \lambda x.\lambda y.x$
- false $:= \lambda x.\lambda y.y$

operators

- and $:= \lambda xy.x\ y\ x$
- or $:= \lambda xy.x\ x\ y$
- not $:= \lambda xyz.x\ z\ y$

Check that the operators behave as expected!

Pairs

Desired behaviour:

$$\text{fst}(\text{pair } x\ y) \to^*_\beta x$$
$$\text{snd}(\text{pair } x\ y) \to^*_\beta y$$

Idea: use Booleans for projections

- pair $:= \lambda xyf.f\ x\ y$
- fst $:= \lambda p.p$ true
- snd $:= \lambda p.p$ false

Check that the operators behave as expected!

Church numerals

How do we construct natural numbers?

- $0, succ(0), succ(succ(0)), \ldots$
- the same idea is behind the *Church numerals*
- function, which takes 0 and $succ$ as parameters

- $\underline{0} := \lambda f.\lambda x.x$
- $\underline{1} := \lambda f.\lambda x.f\ x$
- $\underline{2} := \lambda f.\lambda x.f\ (f\ x)$
- $\underline{3} := \lambda f.\lambda x.f\ (f\ (f\ x))$
- $\ldots$
- $\underline{n} := \underbrace{f(f \ldots (f}_{\text{n-times}}(x) \ldots) = \lambda f.\lambda x.f^n(x)$

Arithmetic operations

$$\underline{n} := \lambda f.\lambda x.f^n(x)$$

- successor
  succ $:= \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$
- addition
  plus $:= \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$ or
  plus $:= \lambda m.\lambda n.m$ succ n
- multiplication
  times $:= \lambda m.\lambda n.\lambda f.m\ (n\ f)$ or
  using plus (exercise)
- exponentiation
  (exercise)
- subtraction (assuming a predecessor)
  minus $:= \lambda m.\lambda n.n$ pred m

Predecessor function

$$\underline{n} := \lambda f.\lambda x.f^n(x)$$

- To compute predecessor, you have to remove one f.
- But there is no "empty" λ-term!

Wisdom tooth trick

- you count up to n, *remembering also the previous number*
- *pairs* are ideal:
  $(0,0),(0,1),(1,2),(2,3),(3,4)\ldots$

λ-calculus encoding

$$\text{step} := \lambda p.\,\text{pair}\ (\text{snd}\ p)\ (\text{succ}(\text{snd}\ p))$$

$$\text{pred} := \lambda n.\,\text{fst}(n\ \text{step}\ (\text{pair}\ \underline{0}\ \underline{0}))$$

Lists – exercise

Define lists and functions operating on them:

- nil – empty list
- null – test for emptiness
- cons – prepends element to a list
- hd – head of the list
- tl – tail of the list

Desired behaviour:

$$\text{null}\ \ \text{nil} \to^* \text{true} \qquad \text{hd}(\text{cons}\ x\ l) \to^* x$$
$$\text{null}\ (\text{cons}\ x\ l) \to^* \text{false} \qquad \text{tl}(\text{cons}\ x\ l) \to^* l$$

Recursion

Recursive functions

- As we have seen, many functions are λ-definable.
- What about the factorial?

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

- Problem: in λ-calculus, functions are anonymous:

$$\text{fact} \equiv \lambda n.\, \text{if}\, n = 0\, \text{then}\, 1\, \text{else}\, n * \text{fact}(n-1)$$

Q: No self-reference in λ-calculus?

Luckily not: $\boldsymbol{\omega}\ y = (\lambda x.x\ x)\ y \to y\ y$

Manual recursion

Idea: recursive function f takes a definition of itself as first argument, and passes it to the subsequent calls

$$G := \lambda f.\lambda n.\, \text{if}\, n = 0\, \text{then}\, 1\, \text{else}\, n * (f\ f\ (n-1))$$

$$\text{fact} := G\ G = (\lambda x.x\ x)\ G$$

Works as intended (TRY IT!)

problem: every recursive call needs to be rewritten as self-application

More natural recursion

$$G := \lambda f.\lambda n.\, \text{if}\, n = 0\, \text{then}\, 1\, \text{else}\, n * (f\ f\ (n-1))$$

$$\text{fact} := G\ G = (\lambda x.x\ x)\ G$$

problem: every recursive call needs to be rewritten as self-application

Our goal: write just

$$G := \lambda f.\lambda n.\, \text{if}\, n = 0\, \text{then}\, 1\, \text{else}\, n * (f\ (n-1))$$

- Naturally, we want $Gfx = fx$ to hold.
- I.e. $Gf = f$ . . . we are looking for a *fixed point* F of G!
- We would like to do this *automatically*.

Find a function $\mathbf{FIX}$ such that $F = G(\mathbf{FIX}\, G) = \mathbf{FIX}\, G$

Does such a function exist?

Factorial using FIX

Let us assume we have such a function $\mathbf{FIX}$ and take

$$G := \lambda f.\lambda n.\,\text{if}\, n = 0\,\text{then}\, 1\,\text{else}\, n * (f\ (n-1))$$

$$\text{fact} := \mathbf{FIX}\ \ G = G\ (\mathbf{FIX}\ \ G)$$

Let's compute the factorial of 2:

$\text{fact}\ \ 2 = (\mathbf{FIX}\, G)2 = G(\mathbf{FIX}\, G)2 =$

$\lambda f.\lambda n.\,\text{if}\, n = 0\,\text{then}\, 1\,\text{else}\, n * (f(n-1))(\mathbf{FIX}\, G)2 \to_\beta$

$\lambda n.\,\text{if}\, n = 0\,\text{then}\, 1\,\text{else}\, n * ((\mathbf{FIX}\, G)(n-1))2 \to_\beta$

$\text{if}\, 2 = 0\,\text{then}\, 1\,\text{else}\, 2 * ((\mathbf{FIX}\, G)(2-1)) \to_\delta$

$\text{if}\, 2 = 0\,\text{then}\, 1\,\text{else}\, 2 * ((\mathbf{FIX}\, G)1) \to_\delta$

$2 * ((\mathbf{FIX}\, G)1) = 2 * (G(\mathbf{FIX}\, G)1) =$

$2 * (\lambda f.\lambda n.\,\text{if}\, n = 0\,\text{then}\, 1\,\text{else}\, n * (f(n-1))(\mathbf{FIX}\, G)1) \to_\beta$

$2 * (\lambda n.\,\text{if}\, n = 0\,\text{then}\, 1\,\text{else}\, n * ((\mathbf{FIX}\, G)(n-1))1) \to_\beta$

$2 * (\text{if}\, 1 = 0\,\text{then}\, 1\,\text{else}\, 1 * ((\mathbf{FIX}\, G)(1-1))) \to_\delta$

$2 * (\text{if}\, 1 = 0\,\text{then}\, 1\,\text{else}\, 1 * ((\mathbf{FIX}\, G)0)) \to_\delta$

$2 * (1 * ((\mathbf{FIX}\, G)0))$

Y combinator (Church)

$$\boldsymbol{Y} := \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$$

Theorem
$\boldsymbol{Y}\, g = g(\boldsymbol{Y}\, g)$

Proof.

$$\begin{aligned}
\boldsymbol{Y}\ g &= (\lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))\ g \\
&\to (\lambda x.g(x\ x))\ (\lambda x.g(x\ x)) \\
&\to g((\lambda x.g(x\ x))(\lambda x.g(x\ x))) \\
&\leftarrow g(\lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))\ g) \\
&= g(\boldsymbol{Y}\, g)
\end{aligned}$$

□

Note: $\boldsymbol{Y}\ g \not\to^* g(\boldsymbol{Y}\ g)$

Θ combinator (Turing)

$$\Theta := (\lambda x f.f(x\ x\ f))(\lambda x f.f(x\ x\ f))$$

Theorem
$\Theta\ g \to^* g(\Theta\ g)$

Proof.

$$\begin{aligned}
\Theta\ g &= (\lambda x f.f(x\ x\ f))\ (\lambda x f.f(x\ x\ f))\ g \\
&\to (\lambda h.h\ ((\lambda x f.f(x\ x\ f))\ (\lambda x f.f(x\ x\ f))\ h))\ g \\
&\to g((\lambda x f.f(x\ x\ f))\ (\lambda x f.f(x\ x\ f))\ g) \\
&= g(\Theta\ g)
\end{aligned}$$

□

From λ-calculus to functional programming I

Applied λ-calculi

applied λ-calculus = λ-calculus + constants (+ operations)

But we have just seen that we can do everything with pure λ-calculus!?

Why applied λ-calculi?

- efficiency
- reliability
- convenience

Formally

syntax

The set of *lambda terms with constants* $\Lambda(\mathbb{C})$ is defined by:

$$M ::= x \mid M\ M' \mid \lambda x.M \mid C$$

Where $C \in \mathbb{C}$ for some *set of constants* $\mathbb{C}$.

Different applied λ-calculi arise by a different choice of the set $\mathbb{C}$.

δ-reduction

Let

- $X \subseteq \Lambda(\mathbb{C})$ be a set of closed normal forms (usually we take $X \subseteq \mathbb{C}$)
- $\delta \in \mathbb{C}$ a special constant
- $f : X \to \Lambda(\mathbb{C})$ *externally defined* function

Then the following δ-contraction rules are added to those of the (pure) λ-calculus:

$$\delta M_1 \dots M_k \to f(M_1 \dots M_k)$$

for $M_1, \dots M_k$ in X.

δ-rule examples I

Booleans

$$\mathbb{C} = \mathbf{true}, \mathbf{false}, \mathbf{not}, \mathbf{and}, \mathbf{or}, \mathbf{ite}$$

δ-rules

$$\begin{array}{rr} \mathbf{not\ true} \to \mathbf{false} & \mathbf{ite\ true} \to \mathsf{true} (\equiv \lambda xy.x) \\ \mathbf{not\ false} \to \mathbf{true} & \mathbf{ite\ false} \to \mathsf{false} (\equiv \lambda xy.y) \\ \mathbf{and\ true\ true} \to \mathbf{true} & \mathbf{or\ true\ true} \to \mathbf{true} \\ \mathbf{and\ true\ false} \to \mathbf{false} & \mathbf{or\ true\ false} \to \mathbf{true} \\ \mathbf{and\ false\ true} \to \mathbf{false} & \mathbf{or\ false\ true} \to \mathbf{true} \\ \mathbf{and\ false\ false} \to \mathbf{false} & \mathbf{or\ false\ false} \to \mathbf{false} \end{array}$$

δ-rule examples II

Integers

$$\mathbb{C} = \{\mathbf{n} | n \in \mathbb{Z}\} \cup \mathbf{plus}, \mathbf{minus}, \mathbf{times}, \mathbf{divide}, \mathbf{equal}$$

δ-rule schemas

$$\begin{aligned}
\mathbf{plus\ m\ n} &\to m + n \\
\mathbf{minus\ m\ n} &\to m - n \\
\mathbf{times\ m\ n} &\to m * n \\
\mathbf{divide\ m\ n} &\to m \div n && \text{for } n \neq 0 \\
\mathbf{divide\ m\ 0} &\to error \\
\mathbf{equal\ m\ m} &\to \mathbf{true} \\
\mathbf{equal\ m\ n} &\to \mathbf{false} && \text{for } m \neq n
\end{aligned}$$

$\beta\delta$-reduction

The "combined" reduction of the "new" calculus is called *$\beta\delta$-reduction*, written as $\rightarrow_{\beta\delta}$ $(\rightarrow^*_{\beta\delta})$.

Theorem

Let f be a function on closed normal forms. Then the resulting notion of reduction $\rightarrow^*_{\beta\delta}$ satisfies the Church-Rosser property.

The notion of normal form also generalizes.

Syntactic sugar

Definition (Syntactic sugar)

Programming language construct, which *can be removed* from the language without any effect on:

- functionality
- expressive power

Why do we need sugar?

To make language *sweeter* for humans. Can e.g.

- improve readability
- be more concise
- more natural to some

Sugar I – functions

- binary arithmetic operators ($+, -, *, \ldots$) in *infix* notation
  $4 \; + \; 5 \implies \; + \; 4\,5 \implies \mathbf{plus}\; 4\,5$
- comparison operators ($<, \leq, =, \neq, \ldots$) in *infix* notation
  $4 \; < \; 5 \implies \; < \; 4\,5$
- if-then-else
  if p then x else $y \implies \mathbf{ite}\; p\; x\; y$

Sugar II – local declarations

$$\text{let } x = M \text{ in } N \Longrightarrow (\lambda x.N)M$$

recursive declarations

$$\text{letrec } x = M \text{ in } N \Longrightarrow (\lambda x.N)\,(\boldsymbol{Y}\ \lambda x.M)$$

example:

```
letrec f = lambda n.if n=1 then 1
                   else n * (f(n-1))
in f 5
```

is desugared to

$$(\lambda f.f\ 5)\ (\boldsymbol{Y}\ \lambda f.\lambda n.\text{if } (=\ n\ 1)1(*\ n\ (f\ (-\ n\ 1))))$$

Sugar III – function declarations

let $f\ x_1\ \ldots\ x_n = M$ in N

$\Longrightarrow$ let $f \lambda x_1 \ldots \lambda x_n.M$ in N

$\Longrightarrow (\lambda f.N)(x_1 \ldots \lambda x_n.M)$

and similarly

letrec $f\ x_1\ \ldots\ x_n = M$ in N

$\Longrightarrow$ letrec $f \lambda x_1 \ldots \lambda x_n.M$ in N