

Programming Language Foundations

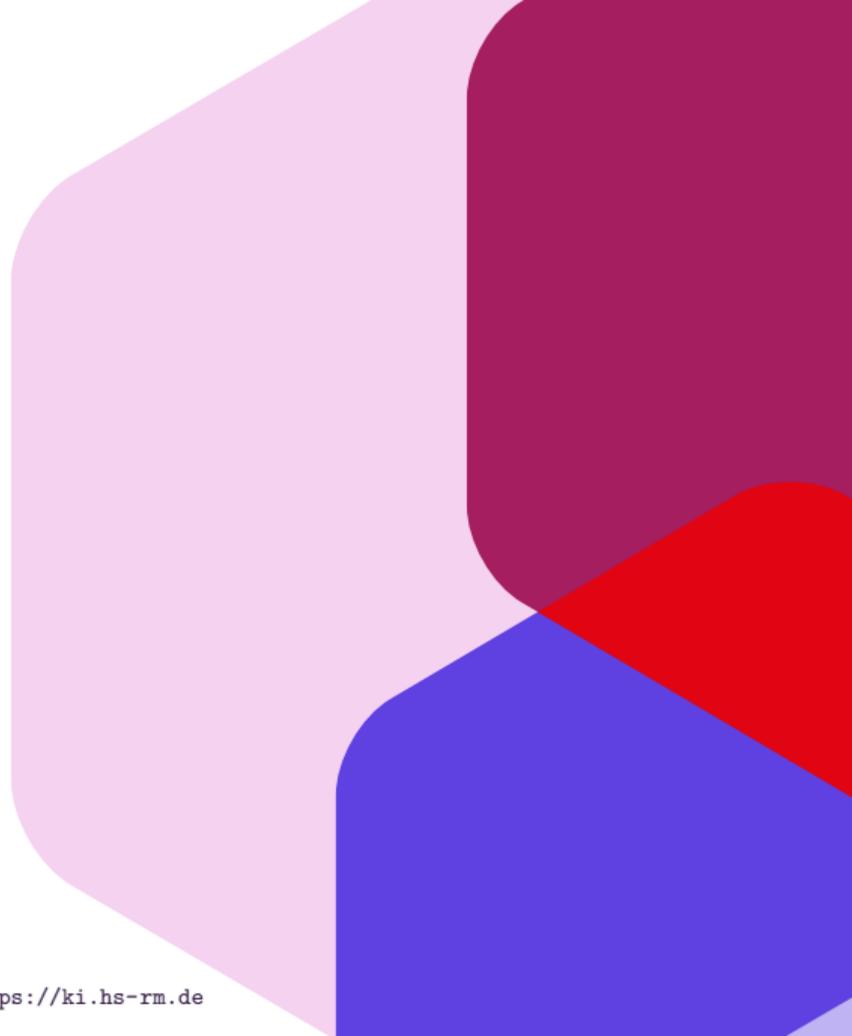
03 Lambda Calculus

Prof. Dr. David Sabel

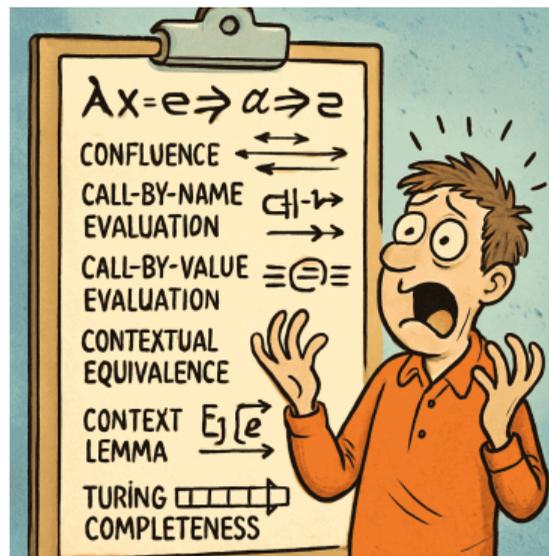
Winter term 2025/26

Last update: December 17, 2025

Image credits: All images are self-drawn or generated with AI from OpenAI at <https://ki.hs-rm.de>



- Syntax of the Lambda Calculus
- α -Renaming and β -Reduction
- Confluence and the Church-Rosser-Theorem
- Call-by-Name Evaluation
- Call-by-Value Evaluation
- Call-by-Need Evaluation
- Contextual Equivalence
- Context Lemma
- Turing Completeness



The untyped lambda calculus

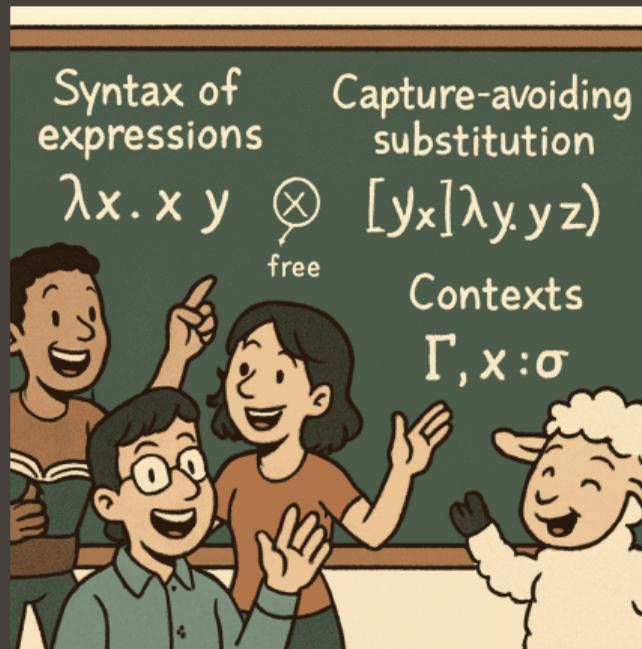
- is a foundational **model of computation**
- is the core of **functional programming** languages

Lambda notation is used in **other settings** too:

- **non-functional programming** languages like Java or Python have introduced functional concepts and lambda expressions
- in **mathematics**, lambda notation is used to represent function
- we use it later for **denotational semantics** of an imperative programming language

SYNTAX OF THE LAMBDA CALCULUS

- Syntax of expressions
- Free and bound variables
- Capture-avoiding substitution
- Contexts



Expressions
$$\mathbf{Expr} ::= V \mid \lambda V. \mathbf{Expr} \mid (\mathbf{Expr} \ \mathbf{Expr})$$

where V is a non-terminal for variables

Explanations:

- $\lambda x. s$ is an **abstraction** = an anonymous function, λx binds x in body s
- $(s \ t)$ is an **application** = expression s is applied to argument t

Examples:

- $\lambda x. x$ is the identity function, like $id(x) = x$, but anonymous
- $((\lambda x. x) \ z)$ represents $id(z)$
- $((\lambda x. x) \ (\lambda x. x))$ represents $id(id)$
- $\lambda x. \lambda y. x$ represents $f(x, y) = x$

To omit parentheses, we use the following conventions:

- application is left-associative:

$s t r$ means $((s t) r)$ and not $(s (t r))$

- the body of an abstraction extends as far as possible:

$(\lambda x.s t$ means $\lambda x.(s t)$ and not $((\lambda x.s) t)$)

- abbreviation:

We write $\lambda x_1, \dots, x_n.t$ for the nested abstractions $\lambda x_1.(\lambda x_2. \dots (\lambda x_n.t) \dots)$.

Some prominent expressions:

$I := \lambda x.x$	(identity)
$K := \lambda x.\lambda y.x$	(projection to first argument)
$K_2 := \lambda x.\lambda y.y$	(projection to second argument)
$\Omega := (\lambda x.(x x)) (\lambda x.(x x))$	(diverging expression)
$Y := \lambda f.(\lambda x.(f (x x))) (\lambda x.(f (x x)))$	(call-by-name fixpoint combinator)
$Z := \lambda f.(\lambda x.(f \lambda z.(x x) z)) (\lambda x.(f \lambda z.(x x) z))$	(call-by-value fixpoint combinator)
$S := \lambda x.\lambda y.\lambda z.(x z) (y z)$	(S-combinator)

$$FV(x) = x$$

$$FV(\lambda x.s) = FV(s) \setminus \{x\}$$

$$FV(s t) = FV(s) \cup FV(t)$$

$$BV(x) = \emptyset$$

$$BV(\lambda x.s) = BV(s) \cup \{x\}$$

$$BV(s t) = BV(s) \cup BV(t)$$

Example: $s = (\lambda x.\lambda y.\lambda w.(x y z)) x$

Free variables $FV(s) = \{x, z\}$ and $BV(s) = \{x, y, w\}$.

$$FV(x) = x$$

$$FV(\lambda x.s) = FV(s) \setminus \{x\}$$

$$FV(s t) = FV(s) \cup FV(t)$$

$$BV(x) = \emptyset$$

$$BV(\lambda x.s) = BV(s) \cup \{x\}$$

$$BV(s t) = BV(s) \cup BV(t)$$

Example: $s = (\lambda x.\lambda y.\lambda w.(x y z)) x$

Free variables $FV(s) = \{x, z\}$ and $BV(s) = \{x, y, w\}$.

Closed and open expressions:

- t is **closed** (or a **program**) if $FV(t) = \emptyset$
- otherwise t is **open**

$$FV(x) = x$$

$$BV(x) = \emptyset$$

$$FV(\lambda x.s) = FV(s) \setminus \{x\}$$

$$BV(\lambda x.s) = BV(s) \cup \{x\}$$

$$FV(s t) = FV(s) \cup FV(t)$$

$$BV(s t) = BV(s) \cup BV(t)$$

Example: $s = (\lambda x.\lambda y.\lambda w.(x y z)) x$

Free variables $FV(s) = \{x, z\}$ and $BV(s) = \{x, y, w\}$.

Closed and open expressions:

- t is **closed** (or a **program**) if $FV(t) = \emptyset$
- otherwise t is **open**

Occurrence x in t is **bound** if it is in scope of a binder λx , otherwise it called **free**

Example: $((\lambda x.\lambda y.\lambda w.(\underbrace{x}_{\text{bound}} \underbrace{y}_{\text{bound}} \underbrace{z}_{\text{free}})) \underbrace{x}_{\text{free}})$

Definition (Capture-Avoiding Substitution)

If $BV(s) \cap FV(t) = \emptyset$, then $s[t/x]$ is s where all free occurrences of x are replaced by t :

$$x[t/x] = t$$

$$y[t/x] = y, \text{ if } x \neq y$$

$$(\lambda y.s)[t/x] = \begin{cases} \lambda y.(s[t/x]) & \text{if } x \neq y \\ \lambda y.s & \text{if } x = y \end{cases}$$

$$(s_1 s_2)[t/x] = (s_1[t/x] s_2[t/x])$$

$$(s_1 s_2)[t/x] = (s_1[t/x] s_2[t/x])$$

Example: $(\lambda x.z x)[(\lambda y.y)/z] = (\lambda x.((\lambda y.y) x))$.

Without the side condition: $(\lambda x.z x)[\lambda y.x/z]$ would lead to $\lambda x.((\lambda y.x) x)$

Contexts C : Expressions with one hole $[\cdot]$

$$\mathbf{Ctx} ::= [\cdot] \mid \lambda V. \mathbf{Ctx} \mid (\mathbf{Ctx} \mathbf{Expr}) \mid (\mathbf{Expr} \mathbf{Ctx})$$

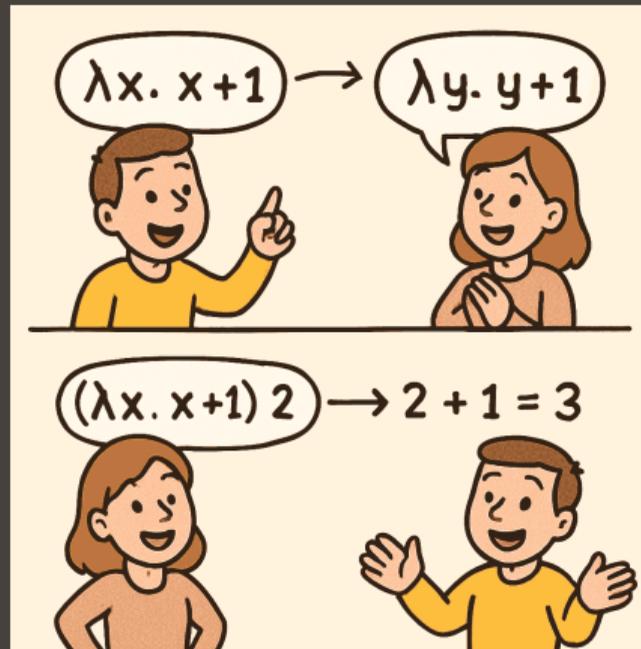
$C[s]$ is an expression: it is C where the hole is replaced by s

This may capture variables, e.g. for context $C = \lambda x. [\cdot]$ and expression $\lambda y. x$:

$$C[\lambda y. x] = \lambda x. (\lambda y. x).$$

α -Renaming and β -Reduction

- Renaming of variables
- Distinct variable convention
- Substitution (with renaming)
- β -reduction, contextual closure



A single α -renaming-step is of the form:

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \quad \text{if } y \notin BV(C[\lambda x.s]) \cup FV(C[\lambda x.s])$$

The reflexive-transitive closure of $\xrightarrow{\alpha} \cup \xleftarrow{\alpha}$ is called α -equivalence and written $s =_{\alpha} t$.

We identify α -equivalent expressions (and write $s = t$ also if $s =_{\alpha} t$)

To avoid naming conflicts, we assume the following convention:

Distinct Variable Convention (DVC)

In any expression s , **bound and free** variables are disjoint, i.e. $BV(s) \cap FV(s) = \emptyset$, and all **variables on binders** are pairwise distinct.

The convention can be obeyed by using α -renamings.

Example

$(y (\lambda y.((\lambda x.(x \lambda x.x)) (x y))))$ violates the DVC

Example

$(y (\lambda y.((\lambda x.(x \lambda x.x)) (x y))))$ violates the DVC

since x and y occur **free** and **bound** and x occurs **twice at a binder**

Example

$(y (\lambda y.((\lambda x.(x \lambda x.x)) (x y))))$ violates the DVC

since x and y occur free and bound and x occurs twice at a binder

apply α -renamings to satisfy the DVC:

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

Example

$(y (\lambda y.((\lambda x.(x \lambda x.x)) (x y))))$ violates the DVC

since x and y occur free and bound and x occurs twice at a binder

apply α -renamings to satisfy the DVC:

$$\begin{aligned} & (y (\lambda y.((\lambda x.(x \lambda x.x)) (x y)))) \\ \xrightarrow{\alpha} & (y (\lambda y_1.((\lambda x.(x \lambda x.x)) (x y_1)))) \end{aligned}$$

Example

$(y (\lambda y.((\lambda x.(x \lambda x.x)) (x y))))$ violates the DVC

since x and y occur free and bound and x occurs twice at a binder

apply α -renamings to satisfy the DVC:

$$\begin{aligned}
 & (y (\lambda y.((\lambda x.(x \lambda x.x)) (x y)))) \\
 \xrightarrow{\alpha} & (y (\lambda y_1.((\lambda x.(x \lambda x.x)) (x y_1)))) \\
 \xrightarrow{\alpha} & (y (\lambda y_1.((\lambda x_1.(x_1 \lambda x.x)) (x y_1))))
 \end{aligned}$$

Example

$(y (\lambda y.((\lambda x.(x \lambda x.x)) (x y))))$ violates the DVC

since x and y occur free and bound and x occurs twice at a binder

apply α -renamings to satisfy the DVC:

$$\begin{aligned} & (y (\lambda y.((\lambda x.(x \lambda x.x)) (x y)))) \\ \xrightarrow{\alpha} & (y (\lambda y_1.((\lambda x.(x \lambda x.x)) (x y_1)))) \\ \xrightarrow{\alpha} & (y (\lambda y_1.((\lambda x_1.(x_1 \lambda x.x)) (x y_1)))) \\ \xrightarrow{\alpha} & (y (\lambda y_1.((\lambda x_1.(x_1 \lambda x_2.x_2)) (x y_1)))) \end{aligned}$$

Substitution $s[t/x]$ without side condition:

Definition (Substitution)

If $BV(s) \cap FV(t) = \emptyset$, then $s[t/x]$ is the capture-avoiding substitution.

Otherwise, let $s' =_{\alpha} s$ such that s' fulfills the DVC.

Then $BV(s') \cap FV(t) = \emptyset$ holds.

Then let $s[t/x] = s'[t/x]$ using capture-avoiding substitution for $s'[t/x]$.

Note: $s[t/x]$ may not satisfy the DVC.

Better: Used α -renamed copies for each t

β -Reduction

The most important **reduction rule** of the lambda calculus:

Definition

The (direct) (β) -reduction is defined as

$$(\beta) \quad (\lambda x.s) t \xrightarrow{\beta} s[t/x]$$

If $r_1 \xrightarrow{\beta} r_2$, then we say r_1 **directly reduces** to r_2 .

Contextual closure

The contextual closure of β -reduction is $\xrightarrow{C,\beta}$ defined as

$$C[s] \xrightarrow{C,\beta} C[t] \text{ iff } C \text{ is a context and } s \xrightarrow{\beta} t.$$

Examples

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

Examples

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x]$$

Examples

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

Examples

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

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

Examples

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

$$(\lambda y.y y y) (x z) \xrightarrow{\beta} (y y y)[(x z)/y]$$

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

$$(\lambda y.y y y) (x z) \xrightarrow{\beta} (y y y)[(x z)/y] = (x z) (x z) (x z)$$

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

$$(\lambda y.y y y) (x z) \xrightarrow{\beta} (y y y)[(x z)/y] = (x z) (x z) (x z)$$

To obey the DVC after a β -reduction: Apply α -renaming, e.g.

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

$$(\lambda y.y y y) (x z) \xrightarrow{\beta} (y y y)[(x z)/y] = (x z) (x z) (x z)$$

To obey the DVC after a β -reduction: Apply α -renaming, e.g.

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

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

$$(\lambda y.y y y) (x z) \xrightarrow{\beta} (y y y)[(x z)/y] = (x z) (x z) (x z)$$

To obey the DVC after a β -reduction: Apply α -renaming, e.g.

$$(\lambda x.(x x)) (\lambda y.y) \xrightarrow{\beta} \underbrace{(\lambda y.y) (\lambda y.y)}_{\text{violates the DVC}}$$

$$(\lambda x.x) (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

$$(\lambda y.y y y) (x z) \xrightarrow{\beta} (y y y)[(x z)/y] = (x z) (x z) (x z)$$

To obey the DVC after a β -reduction: Apply α -renaming, e.g.

$$(\lambda x.(x x)) (\lambda y.y) \xrightarrow{\beta} \underbrace{(\lambda y.y) (\lambda y.y)}_{\text{violates the DVC}} =_{\alpha} \underbrace{(\lambda y_1.y_1) (\lambda y_2.y_2)}_{\text{satisfies the DVC}}$$

To evaluate an expression, β -reduction has to be applied to subexpressions, e.g.

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

Those subexpressions are called a **redex** (reducible expression).

To evaluate an expression, β -reduction has to be applied to subexpressions, e.g.

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

Those subexpressions are called a **redex** (reducible expression).

To evaluate an expression, β -reduction has to be applied to subexpressions, e.g.

$$((\lambda x.x) (\lambda y.y)) (\lambda z.z) \xrightarrow{C,\beta} (\lambda y.y) (\lambda z.z)$$

Those subexpressions are called a **redex** (reducible expression).

To evaluate an expression, β -reduction has to be applied to subexpressions, e.g.

$$((\lambda x.x) (\lambda y.y)) (\lambda z.z) \xrightarrow{C,\beta} (\lambda y.y) (\lambda z.z)$$

Those subexpressions are called a **redex** (reducible expression).

Using $\xrightarrow{C,\beta}$ is not deterministic, e.g. $((\lambda x.x x) ((\lambda y.y) (\lambda z.z)))$ has two redexes:

- $((\lambda x.x x) ((\lambda y.y) (\lambda z.z))) \xrightarrow{C,\beta} ((\lambda y.y) (\lambda z.z)) ((\lambda y.y) (\lambda z.z))$
- $((\lambda x.x x) ((\lambda y.y) (\lambda z.z))) \xrightarrow{C,\beta} ((\lambda x.x x) (\lambda z.z)).$

To evaluate an expression, β -reduction has to be applied to subexpressions, e.g.

$$((\lambda x.x) (\lambda y.y)) (\lambda z.z) \xrightarrow{C,\beta} (\lambda y.y) (\lambda z.z)$$

Those subexpressions are called a **redex** (reducible expression).

Using $\xrightarrow{C,\beta}$ is not deterministic, e.g. $((\lambda x.x x) ((\lambda y.y) (\lambda z.z)))$ has two redexes:

- $((\lambda x.x x) ((\lambda y.y) (\lambda z.z))) \xrightarrow{C,\beta} ((\lambda y.y) (\lambda z.z)) ((\lambda y.y) (\lambda z.z))$
- $((\lambda x.x x) ((\lambda y.y) (\lambda z.z))) \xrightarrow{C,\beta} ((\lambda x.x x) (\lambda z.z))$.

Fixing the position where to apply the reduction is called a **reduction strategy**.

We do it soon, but first we consider arbitrary $\xrightarrow{C,\beta}$ -steps.

CHURCH-ROSSER-THEOREM

- The diamond property
- Confluence
- \rightarrow_1 -reduction
- Proof of the Church-Rosser-Theorem



For a binary relation $\rightarrow \subseteq (M \times M)$, we denote with

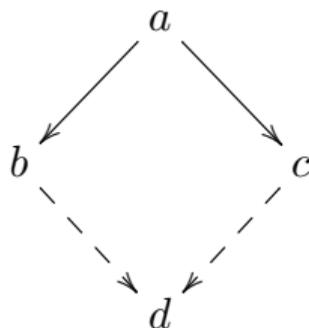
- \leftrightarrow the **symmetric closure** of \rightarrow (i.e. $a \leftrightarrow b$ iff $a \rightarrow b$ or $b \rightarrow a$).
 - \xrightarrow{i} the **i -fold composition** of \rightarrow
 $(a \xrightarrow{0} a$ for all $a \in M$ and for $i > 0$: $a \xrightarrow{i} b$, if $\exists b' \in M : a \rightarrow b'$ and $b' \xrightarrow{i-1} b$).
 - $\xrightarrow{i \vee j}$ is the **union** of the i -fold and the j -fold composition
 $(a \xrightarrow{i \vee j} b$ iff $a \xrightarrow{i} b$ or $a \xrightarrow{j} b$).
- In particular, $\xrightarrow{0 \vee 1}$ is the reflexive-closure of \rightarrow .
- $\xrightarrow{*}$ the **reflexive-transitive closure** of \rightarrow ($a \xrightarrow{*} b$ iff $\exists i \in \mathbb{N}_0 : a \xrightarrow{i} b$).
 - $\xleftrightarrow{*}$ the **reflexive-transitive closure** of \leftrightarrow .
 - $\xrightarrow{+}$ the **transitive closure** of \rightarrow ($a \xrightarrow{+} b$ iff $\exists i \in \mathbb{N} : a \xrightarrow{i} b$).

The Diamond Property and Confluence

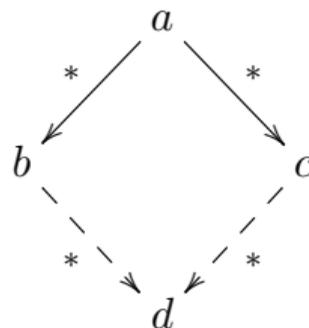
Definition

A binary relation $\rightarrow \subseteq M \times M$

- has the **diamond property** iff
whenever $a \rightarrow b$ and $a \rightarrow c$ there exists $d \in M$ such that $b \rightarrow d$ and $c \rightarrow d$.
- is **confluent** iff $\xrightarrow{*}$ has the diamond property.



\rightarrow has the diamond property

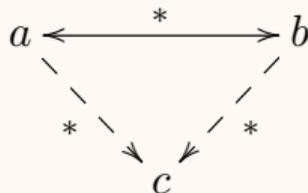


\rightarrow is confluent

- our goal is to show that $\xrightarrow{C,\beta}$ is confluent
- if confluence holds, then normal forms are **unique**:
if we reduce all $\xrightarrow{C,\beta}$ -redexes, then we get the same expression (up to α -renaming)
independently from the order and positions where the reductions were applied

Lemma 3.3.2

If reduction relation \rightarrow is confluent, then $a \overset{*}{\leftrightarrow} b$ implies $\exists c : a \overset{*}{\rightarrow} c \wedge b \overset{*}{\rightarrow} c$



Proof. By induction on $a \overset{i}{\leftrightarrow} b$.

- Base case: if $i = 0$, then $a = b$ and the claim holds
- ...

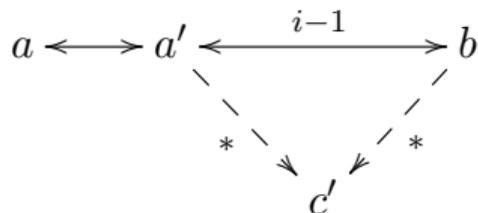
Proof (Cont'd)

If $i > 0$, then $\exists a' : a \leftrightarrow a' \overset{i-1}{\longleftrightarrow} b$.

$$a \longleftrightarrow a' \overset{i-1}{\longleftrightarrow} b$$

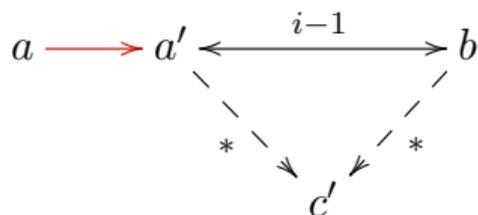
Proof (Cont'd)

If $i > 0$, then $\exists a' : a \leftrightarrow a' \xrightarrow{i-1} b$. By induction hypothesis $\exists c' : a' \xrightarrow{*} c'$ and $b \xrightarrow{*} c'$.



Proof (Cont'd)

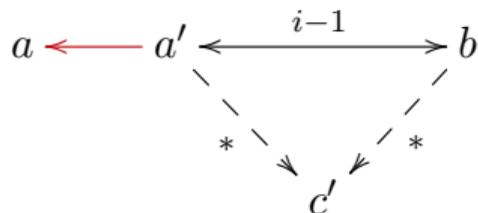
If $i > 0$, then $\exists a' : a \leftrightarrow a' \xrightarrow{i-1} b$. By induction hypothesis $\exists c' : a' \xrightarrow{*} c'$ and $b \xrightarrow{*} c'$.



- If $a \rightarrow a'$, then $a \rightarrow a' \xrightarrow{*} c'$ and thus $a \xrightarrow{*} c'$. Since also $b \xrightarrow{*} c'$, the claim holds.

Proof (Cont'd)

If $i > 0$, then $\exists a' : a \leftrightarrow a' \xrightarrow{i-1} b$. By induction hypothesis $\exists c' : a' \xrightarrow{*} c'$ and $b \xrightarrow{*} c'$.

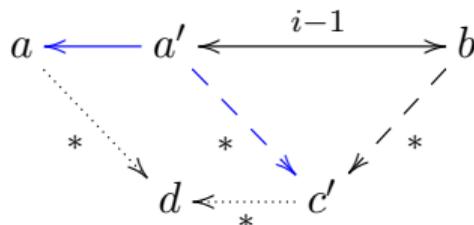


- If $a \rightarrow a'$, then $a \rightarrow a' \xrightarrow{*} c'$ and thus $a \xrightarrow{*} c'$. Since also $b \xrightarrow{*} c'$, the claim holds.
- If $a' \rightarrow a$, then $a' \xrightarrow{*} a$.



Proof (Cont'd)

If $i > 0$, then $\exists a' : a \leftrightarrow a' \xrightarrow{i-1} b$. By induction hypothesis $\exists c' : a' \xrightarrow{*} c'$ and $b \xrightarrow{*} c'$.



- If $a \rightarrow a'$, then $a \rightarrow a' \xrightarrow{*} c'$ and thus $a \xrightarrow{*} c'$. Since also $b \xrightarrow{*} c'$, the claim holds.
- If $a' \rightarrow a$, then $a' \xrightarrow{*} a$.

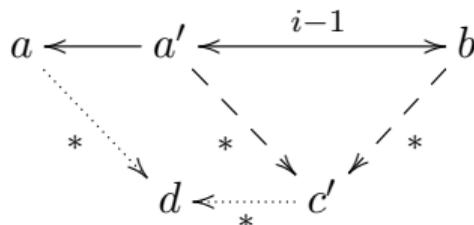
Since \rightarrow is confluent, $\xrightarrow{*}$ has the diamond property and thus:

from $a' \xrightarrow{*} a$ and $a' \xrightarrow{*} c'$, we obtain d with $a \xrightarrow{*} d$ and $c' \xrightarrow{*} d$.

□

Proof (Cont'd)

If $i > 0$, then $\exists a' : a \leftrightarrow a' \xrightarrow{i-1} b$. By induction hypothesis $\exists c' : a' \xrightarrow{*} c'$ and $b \xrightarrow{*} c'$.



- If $a \rightarrow a'$, then $a \rightarrow a' \xrightarrow{*} c'$ and thus $a \xrightarrow{*} c'$. Since also $b \xrightarrow{*} c'$, the claim holds.
- If $a' \rightarrow a$, then $a' \xrightarrow{*} a$.
Since \rightarrow is confluent, $\xrightarrow{*}$ has the diamond property and thus:
from $a' \xrightarrow{*} a$ and $a' \xrightarrow{*} c'$, we obtain d with $a \xrightarrow{*} d$ and $c' \xrightarrow{*} d$.
Since $b \xrightarrow{*} c' \xrightarrow{*} d$, the claim holds. □

Diamond Property: Inheritance from \rightarrow to \rightarrow^*

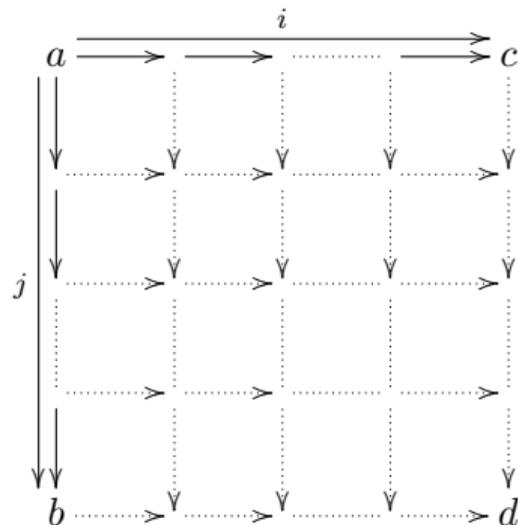
Lemma 3.3.3

Let \rightarrow be a binary relation and \rightarrow^* be its reflexive-transitive closure.
If \rightarrow has the diamond property, then \rightarrow^* has the diamond property.

Proof: By induction on (i, j) one can show that:

If $a \xrightarrow{i} b$ and $a \xrightarrow{j} c$ then $\exists d: b \xrightarrow{i} d$ and $c \xrightarrow{j} d$.

The inner square diagrams follow from the diamond property of \rightarrow .



Closures of $\xrightarrow{\beta}$

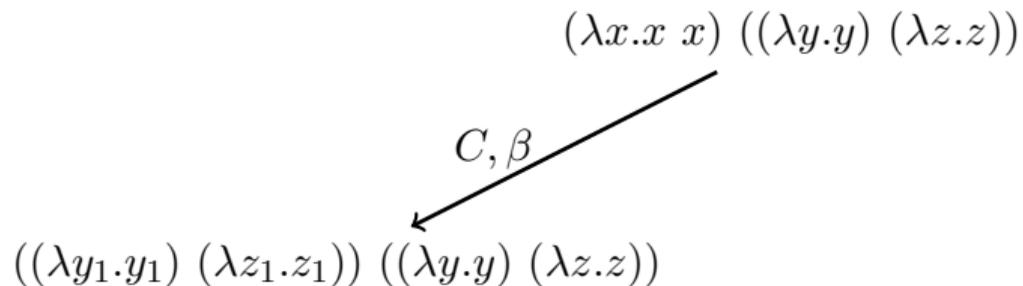
- With $s \xleftrightarrow{C,\beta} t$ we denote the **symmetric closure** of $\xrightarrow{C,\beta}$
(i.e. $s \xleftrightarrow{C,\beta} t$ iff $s \xrightarrow{C,\beta} t$ or $t \xrightarrow{C,\beta} s$)
- With $s \xrightarrow{C,\beta,*} t$ we denote the **reflexive-transitive closure** of $\xrightarrow{C,\beta}$
- With $s \xleftrightarrow{C,\beta,*} t$ we denote the **reflexive-transitive closure** of $\xleftrightarrow{C,\beta}$

The relation $\xleftrightarrow{C,\beta,*}$ is sometimes also called β -equivalence or also convertibility.

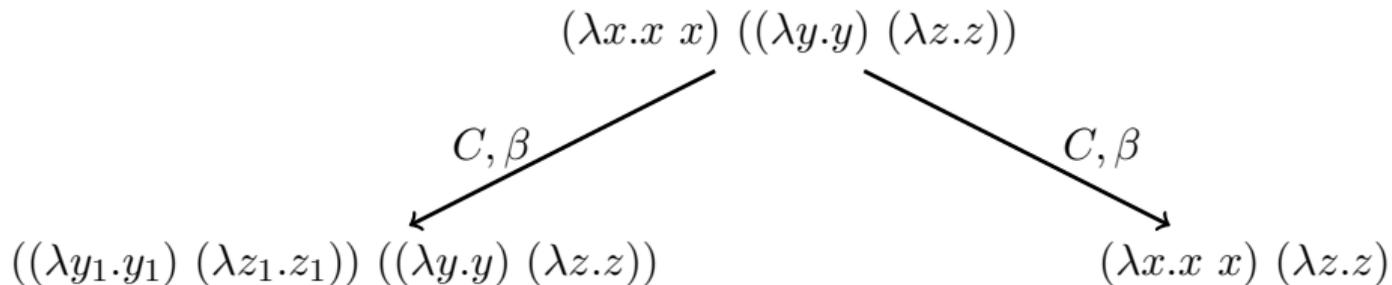
$\xrightarrow{C, \beta}$ does not have the diamond property

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

$\xrightarrow{C, \beta}$ does not have the diamond property

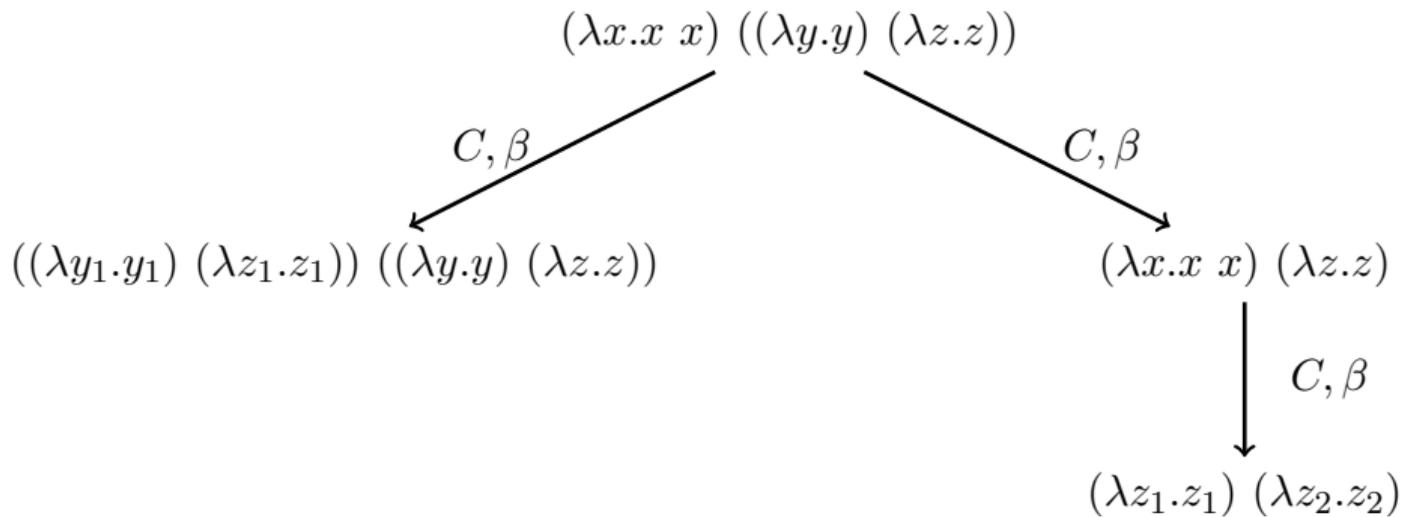


$\xrightarrow{C, \beta}$ does not have the diamond property



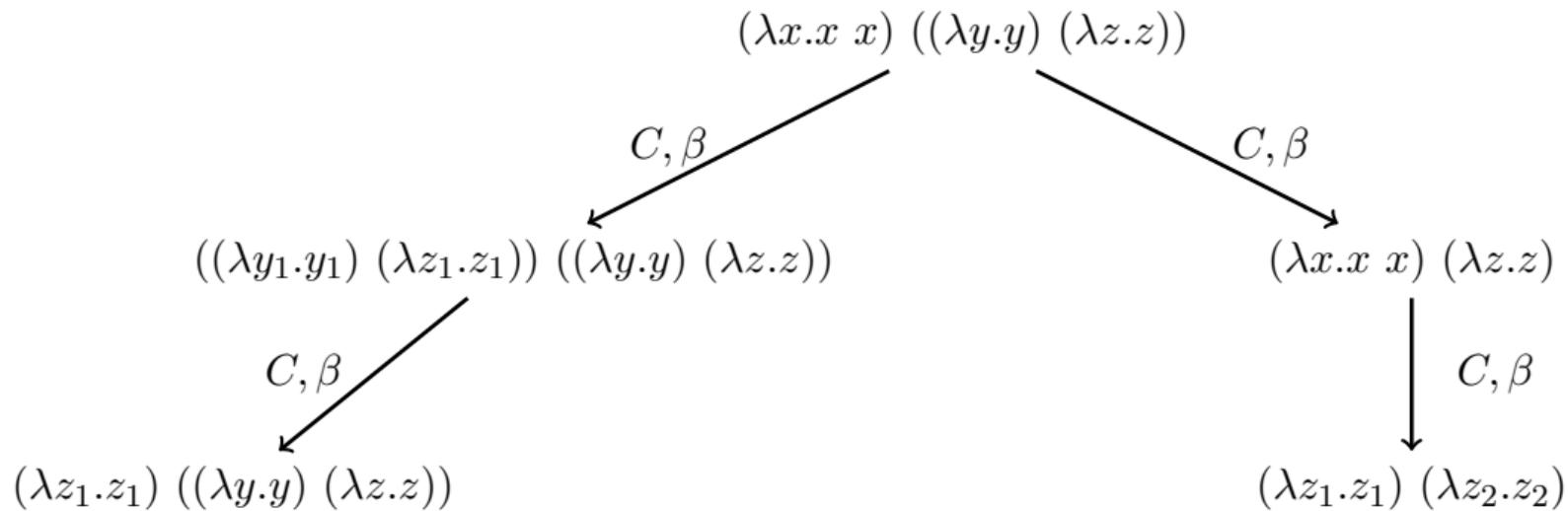
$\xrightarrow{C, \beta}$ does not have the diamond property

hsrm*



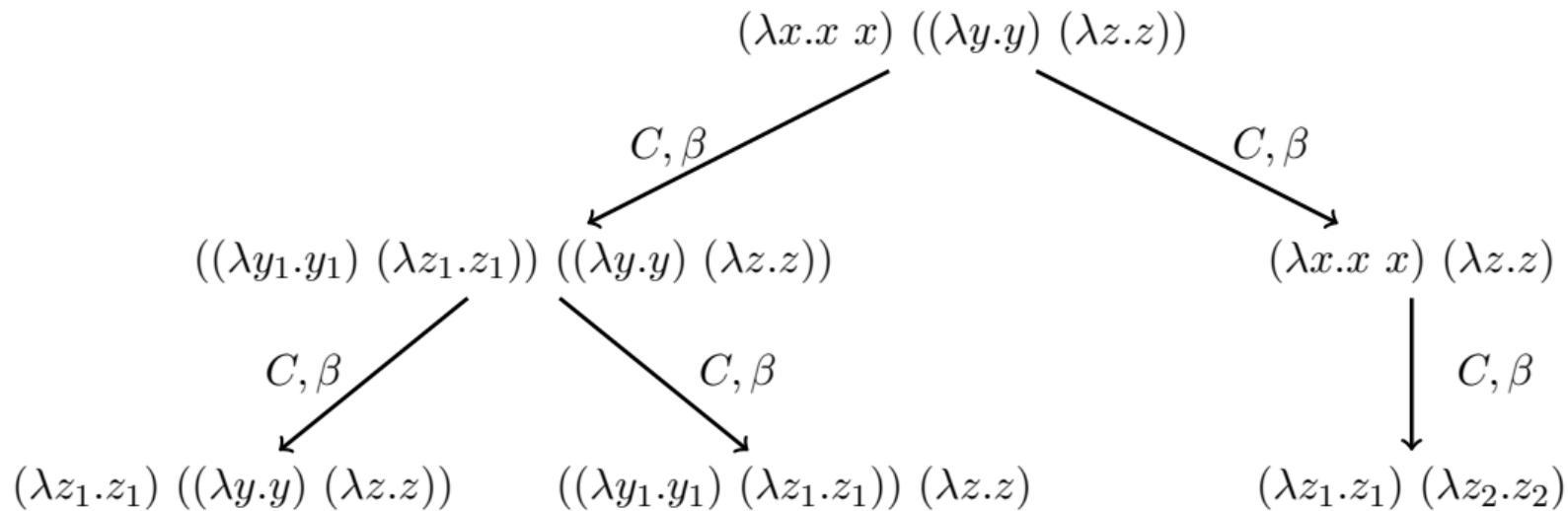
$\xrightarrow{C, \beta}$ does not have the diamond property

hsrm*



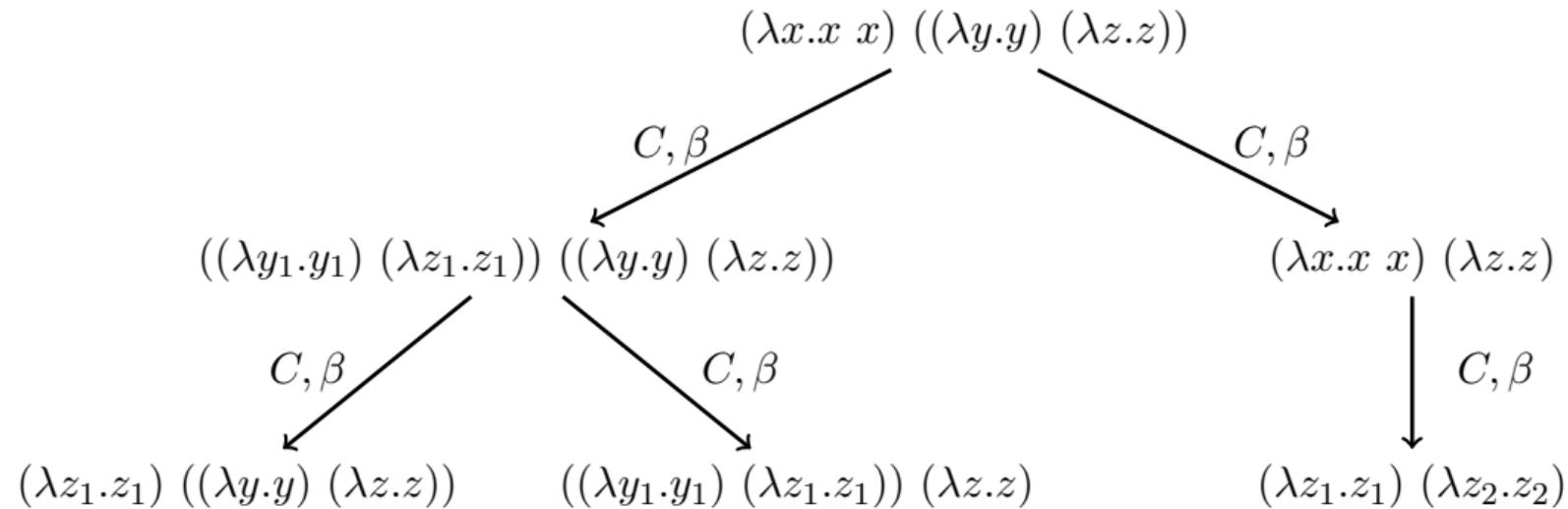
$\xrightarrow{C, \beta}$ does not have the diamond property

hsrm*



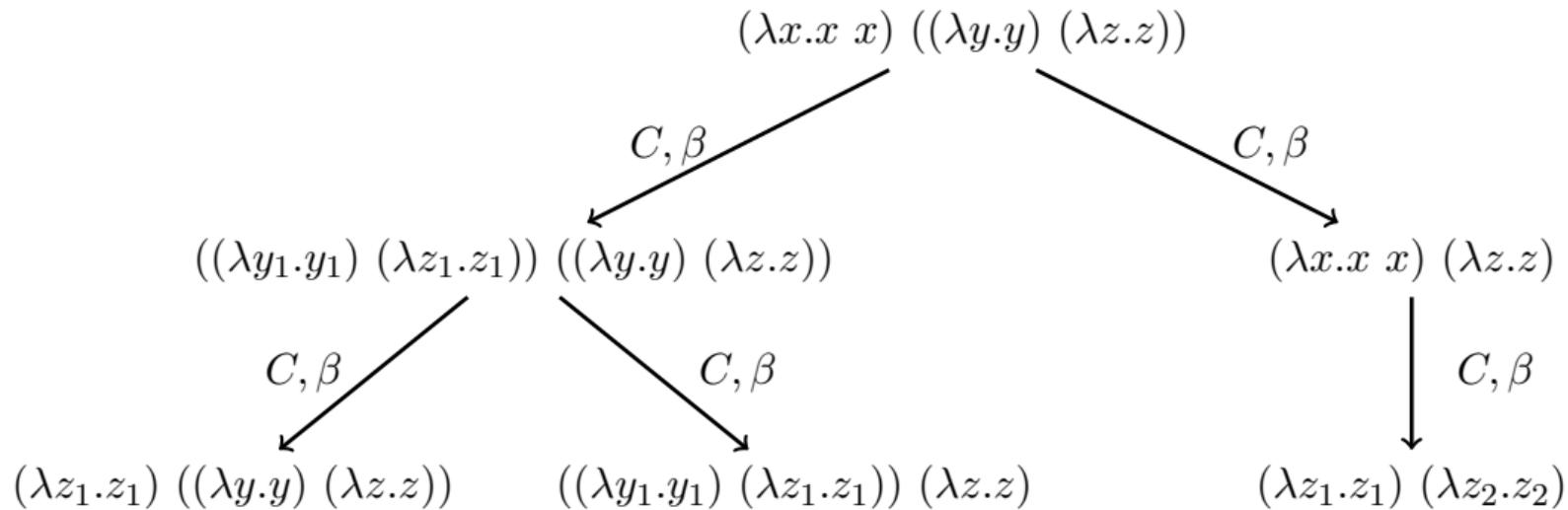
$\xrightarrow{C, \beta}$ does not have the diamond property

hsrm*



→ Confluence of $\xrightarrow{\beta}$ cannot be proved by the previous lemma

$\xrightarrow{C,\beta}$ does not have the diamond property



→ Confluence of $\xrightarrow{\beta}$ cannot be proved by the previous lemma

→ Idea:

- Another reduction relation \rightarrow_1 with $\xrightarrow{C,\beta,0\vee 1} \subseteq \rightarrow_1 \subseteq \xrightarrow{C,\beta,*}$ and $\xrightarrow{*}_1 = \xrightarrow{C,\beta,*}$
- Prove diamond-property of \rightarrow_1 . This implies diamond-property of $\xrightarrow{*}_1 = \xrightarrow{C,\beta,*}$

Definition (Parallel Reduction \rightarrow_1)

The relation $\rightarrow_1 \subseteq (\mathbf{Expr} \times \mathbf{Expr})$ is inductively defined by:

- 1 $s \rightarrow_1 s$ for all expressions s .
- 2 if $s_1 \rightarrow_1 s_2$ and $t_1 \rightarrow_1 t_2$, then $(s_1 t_1) \rightarrow_1 (s_2 t_2)$.
- 3 if $s_1 \rightarrow_1 s_2$ and $t_1 \rightarrow_1 t_2$, then $((\lambda x.s_1) t_1) \rightarrow_1 s_2[t_2/x]$.
- 4 if $s \rightarrow_1 t$, then $\lambda x.s \rightarrow_1 \lambda x.t$.

Lemma 3.3.7

$$\frac{C, \beta}{\longrightarrow} \subseteq \rightarrow_1$$

Proof. Let $C[(\lambda x.s) t] \xrightarrow{C, \beta} C[s[t/x]]$. We show $C[(\lambda x.s) t] \rightarrow_1 C[s[t/x]]$ by structural induction on C .

If $C = [\cdot]$, then $(\lambda x.s) t \rightarrow_1 s[t/x]$ by ③, since $s \rightarrow_1 s$ and $t \rightarrow_1 t$ by ①.

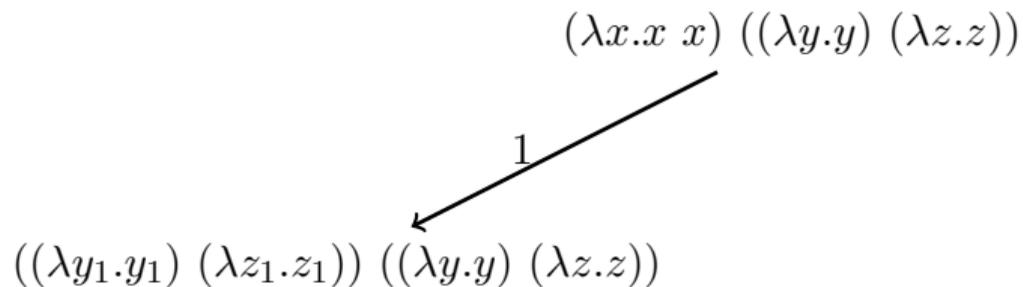
If $C \neq [\cdot]$ we use as IH that $C'[(\lambda x.s) t] \rightarrow_1 C'[s[t/x]]$ where C' is a proper subcontext of context C .

- If $C = (C' r)$, then $r \rightarrow_1 r$ by ①, $C'[(\lambda x.s) t] \rightarrow_1 C'[s[t/x]]$ by the IH and thus $C[(\lambda x.s) t] = (C'[(\lambda x.s) t] r) \rightarrow_1 (C'[s[t/x]] r) = C[s[t/x]]$ by ②.
- The case $C = (r C')$ is completely analogous to the previous one.
- If $C = \lambda y.C'$, then $C'[(\lambda x.s) t] \rightarrow_1 C'[s[t/x]]$ by the IH and thus $C[(\lambda x.s) t] = \lambda y.C'[(\lambda x.s) t] \rightarrow_1 \lambda y.C'[s[t/x]] = C[s[t/x]]$ by ④. □

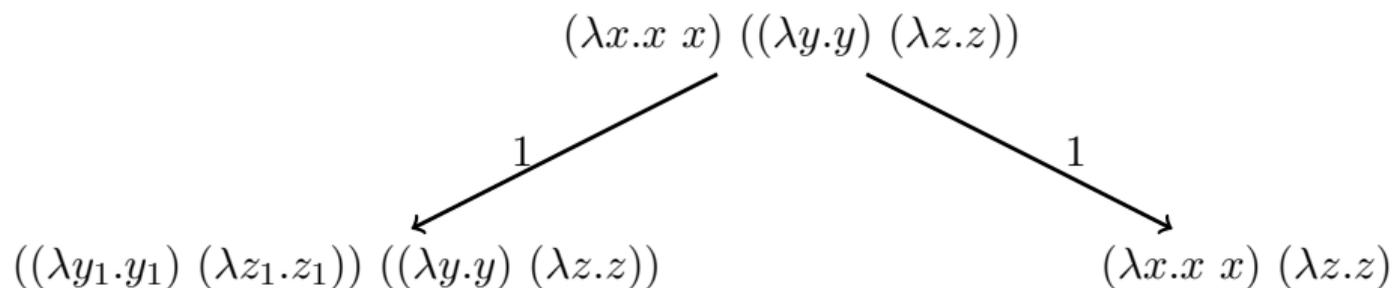
Examples for \rightarrow_1

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

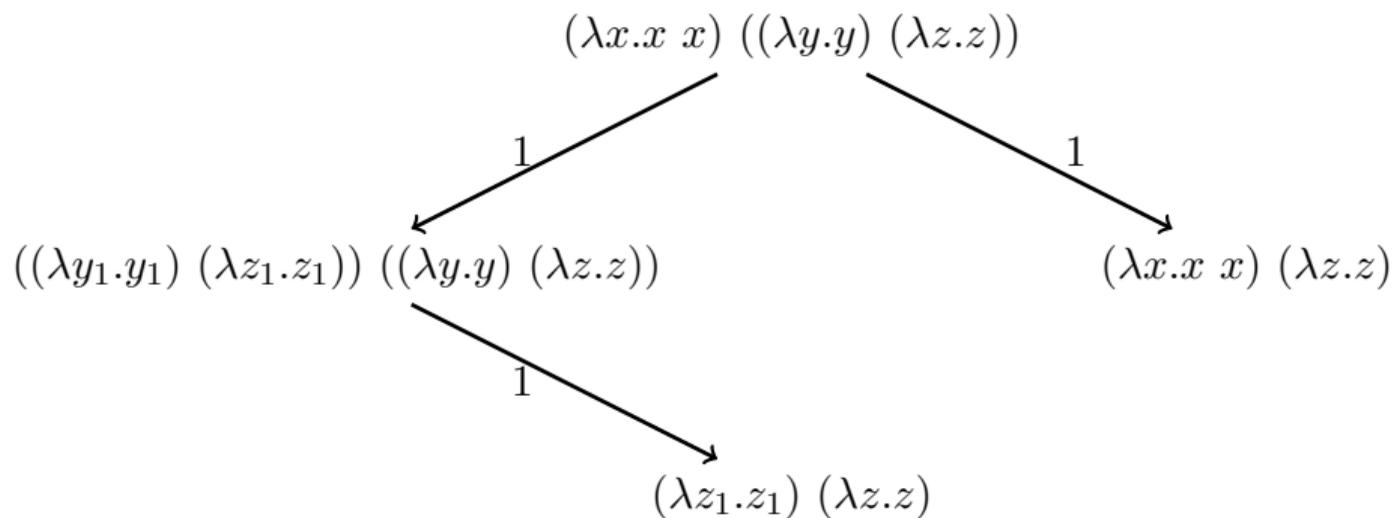
Examples for \rightarrow_1



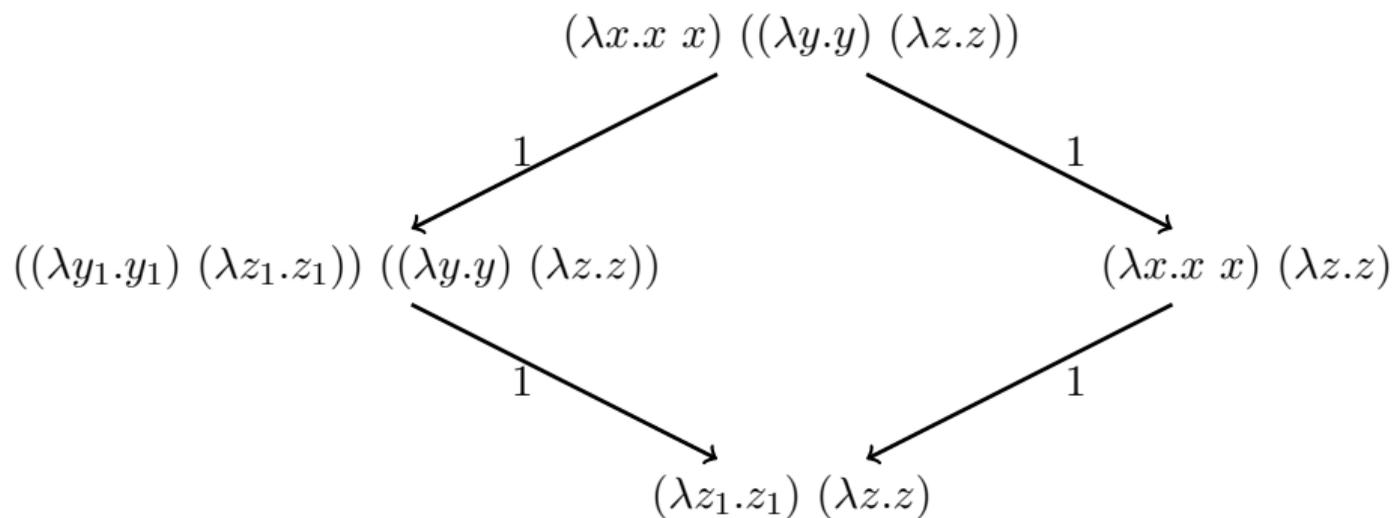
Examples for \rightarrow_1



Examples for \rightarrow_1



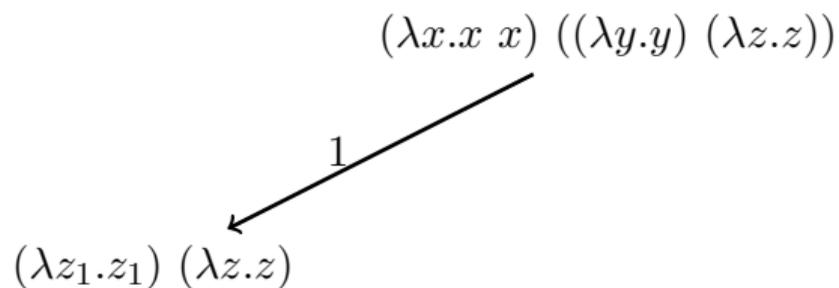
Examples for \rightarrow_1



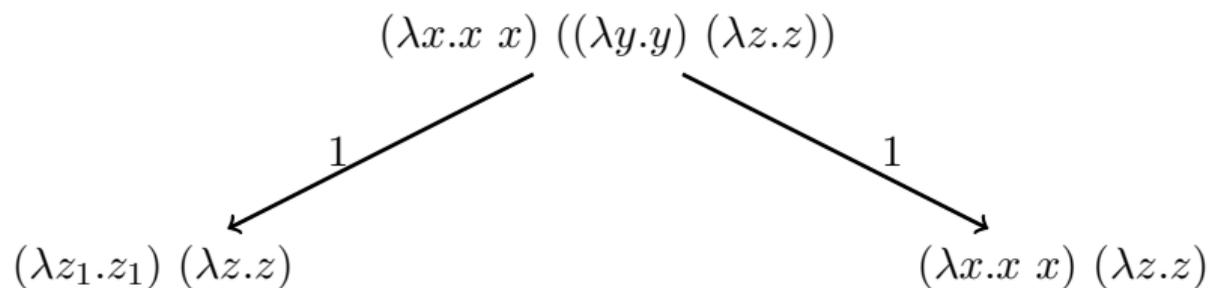
Examples for \rightarrow_1 (2)

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

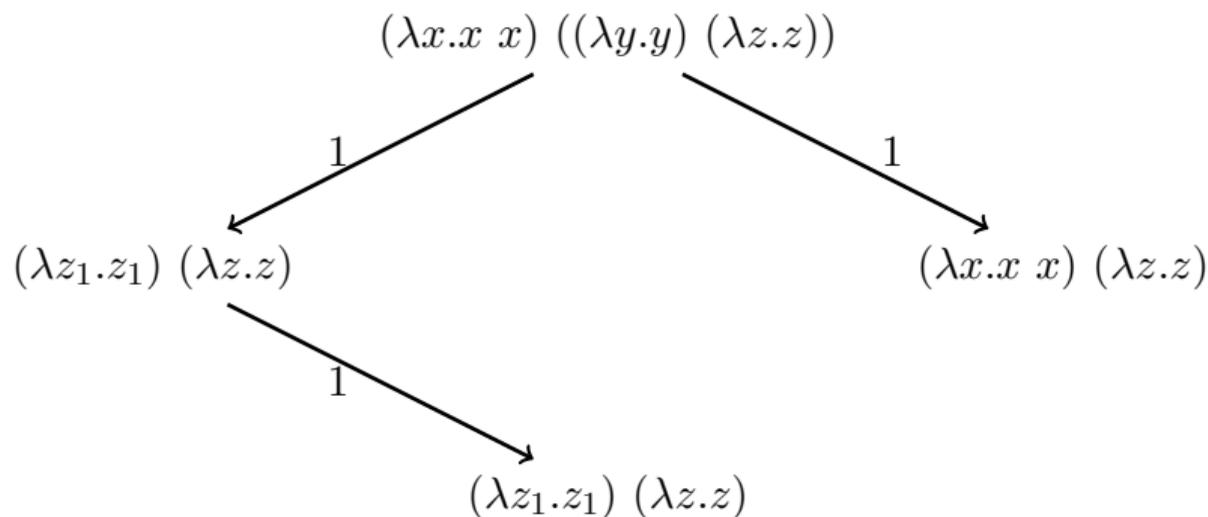
Examples for \rightarrow_1 (2)



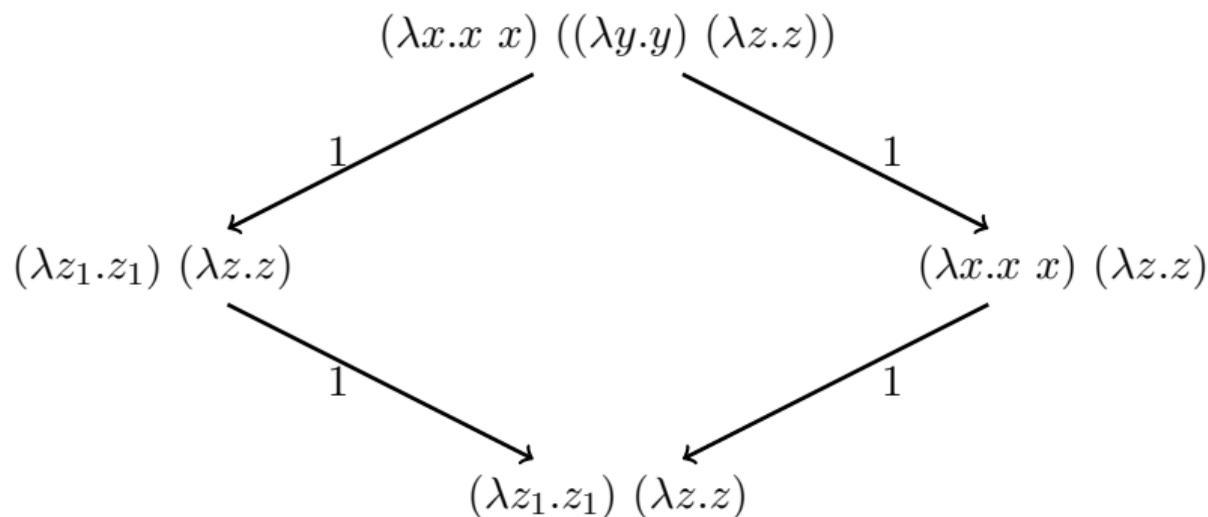
Examples for \rightarrow_1 (2)



Examples for \rightarrow_1 (2)



Examples for \rightarrow_1 (2)



Lemma 3.3.10

If $s \rightarrow_1 r$ and $t \rightarrow_1 u$ then $s[t/x] \rightarrow_1 r[u/x]$.

Proof. This can be shown by induction on $s \rightarrow_1 r$.

The base case $s = r$ can be shown by structural induction on s .

For the induction step, we only show one exemplary case:

- If $s = (s_1 s_2) \rightarrow_1 (r_1 r_2) = r$ with $s_i \rightarrow_1 r_i$ for $i = 1, 2$, the IH shows $s_i[t/x] \rightarrow_1 r_i[u/x]$ for $i = 1, 2$ and thus $(s_1[t/x] s_2[t/x]) \rightarrow_1 (r_1[u/x] r_2[u/x])$. Since $s[t/x] = (s_1 s_2)[t/x]$ and $(r_1 r_2)[u/x] = r[u/x]$, the claim holds.

Lemma 3.3.11

Relation \rightarrow_1 has the diamond property.

Proof. We show that whenever $s \rightarrow_1 t$ then for all r with $s \rightarrow_1 r$ there exists r' with $t \rightarrow_1 r'$ and $r \rightarrow r'$.

We use induction on the definition of \rightarrow_1 in $s \rightarrow_1 t$.

Base case: $t = s$, i.e. $s \rightarrow_1 s$. Then choose $r' = r$ and the claim holds.

For the induction step, all other cases of the definition of \rightarrow_1 have to be considered.
We show one exemplary case.

Diamond Property for \rightarrow_1 (Cont'd)

If $s = ((\lambda x.s_1) s_2)$ and $t = t_1[t_2/x]$ where $s_i \rightarrow_1 t_i$, then for $s \rightarrow_1 r$ there are two cases:

- $s = ((\lambda x.s_1) s_2) \rightarrow_1 ((\lambda x.r_1) r_2)$ with $s_1 \rightarrow_1 r_1$ and $s_2 \rightarrow_1 r_2$.
Applying the IH to $s_1 \rightarrow_1 t_1$ and $s_1 \rightarrow_1 r_1$ and also to $s_2 \rightarrow_1 t_2$ and $s_2 \rightarrow_1 r_2$ shows that there exists r'_1 and r'_2 such that: $t_i \rightarrow_1 r'_i$, $r_i \rightarrow_1 r'_i$ for $i = 1, 2$.
This shows that $t_1[t_2/x] \rightarrow_1 r'_1[r'_2/x]$ and $((\lambda x.r_1) r_2) \rightarrow_1 r'_1[r'_2/x]$ (using the previous lemma). Thus the diamond property holds.
- If $s = ((\lambda x.s_1) s_2)$ and $r = r_1[r_2/x]$ where $s_1 \rightarrow_1 r_1$ and $s_2 \rightarrow_1 r_2$. Applying the IH to $s_1 \rightarrow_1 t_1$ and $s_1 \rightarrow_1 r_1$ and also to $s_2 \rightarrow_1 t_2$ and $s_2 \rightarrow_1 r_2$ shows that there exists r'_1 and r'_2 such that: $t_i \rightarrow_1 r'_i$, $r_i \rightarrow_1 r'_i$ for $i = 1, 2$. This shows that $t_1[t_2/x] \rightarrow_1 r'_1[r'_2/x]$ and $r_1[r_2/x] \rightarrow_1 r'_1[r'_2/x]$ (using the previous lemma). Thus the diamond property holds.

Coincidence of $\xrightarrow{C,\beta,*}$ and $\xrightarrow{*}_1$

Lemma 3.3.12

$$\xrightarrow{C,\beta,*} = \xrightarrow{*}_1$$

Proof (Sketch).

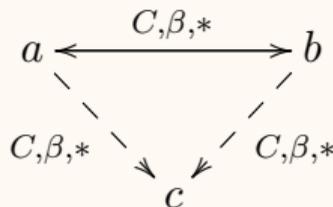
Since $\xrightarrow{C,\beta} \subseteq \rightarrow_1$ (Lemma 3.3.7), $\xrightarrow{C,\beta,*} \subseteq \xrightarrow{*}_1$ also holds.

$\rightarrow_1 \subseteq \xrightarrow{C,\beta,*}$ can be proved by inspecting the different cases of the inductive definition of \rightarrow_1 .

Finally, $\xrightarrow{*}_1 \subseteq \xrightarrow{C,\beta,*}$ holds, since $\xrightarrow{*}_1 \subseteq \left(\xrightarrow{C,\beta,*}\right)^*$ and $\xrightarrow{C,\beta,*}^* = \xrightarrow{C,\beta,*}$.

Church-Rosser-Theorem

For the lambda calculus the following holds: If $a \xleftrightarrow{C,\beta,*} b$, then there exists c , such that $a \xrightarrow{C,\beta,*} c$ and $b \xrightarrow{C,\beta,*} c$



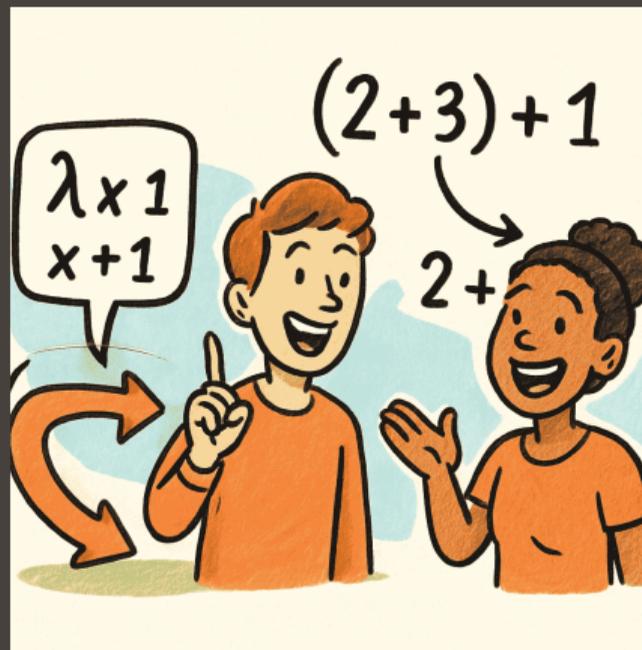
Proof.

Applying Lemma 3.3.3 for \rightarrow_1 (using Lemma 3.3.11) shows that \rightarrow_1 is confluent and that $\xrightarrow{*}_1$ has the diamond property.

With the equation of Lemma 3.3.12, we have that $\xrightarrow{C,\beta,*}$ has the diamond property and thus $\xrightarrow{C,\beta}$ is confluent. Finally, Lemma 3.3.2 then shows the claim. □

CALL-BY-NAME EVALUATION

- Reduction contexts
- Alternative definition with labeling
- Convergence
- Standardisation-Theorem



Ideas:

- do not reduce below λ
- reduce the leftmost-outermost β -redex
- in $(\lambda x.s) t$ pass t to the function body without evaluating t

Definition

Reduction contexts R are built by the following grammar:

$$\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt} \mathbf{Expr})$$

If $r_1 \xrightarrow{\beta} r_2$ and R is a reduction context,
then $R[r_1] \xrightarrow{\text{name}} R[r_2]$ is a **call-by-name reduction step**.

Example

$$\begin{aligned} & (\lambda w.w) ((\lambda u.u) (\lambda v.v)) (\lambda x.((\lambda y.y) (\lambda z.z))) \\ \xrightarrow{\text{name}} & (\lambda u.u) (\lambda v'.v') (\lambda x.((\lambda y.y) (\lambda z.z))) \\ \xrightarrow{\text{name}} & (\lambda v.v) (\lambda x.((\lambda y.y) (\lambda z.z))) \\ \xrightarrow{\text{name}} & \lambda x.((\lambda y.y) (\lambda z.z)) \\ \xrightarrow{\text{name}} & \end{aligned}$$

Alternative Definition of Call-by-Name Reduction

Use a labeling algorithm to mark the redex:

- For expression s , start with s^* .
- Apply the label shifting as long as possible:

$$(s_1 s_2)^* \Rightarrow (s_1^* s_2)$$

The result is of the form $(s_1^* s_2 \dots s_n)$, where s_1 is not an application.

- If s_1 is an abstraction $\lambda x.s'_1$ and $n \geq 2$, then reduce as follows:

$$(\lambda x.s'_1) s_2 \dots s_n \xrightarrow{\text{name}} (s'_1[s_2/x] \dots s_n)$$

- If s_1 is an abstraction and $n = 1$, then no call-by-name reduction is applicable (since the whole expression is an abstraction)
- If s_1 is a variable, then no call-by-name reduction is applicable (since a free variable has been detected)

Example

$$\left(\left((\lambda w. w) \left((\lambda u. u) (\lambda v. v) \right) \right) (\lambda x. ((\lambda y. y) (\lambda z. z))) \right)$$

Example

$$\left(\left((\lambda w.w) ((\lambda u.u) (\lambda v.v)) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)^*$$

Example

$$\left(\left((\lambda w.w) \left((\lambda u.u) (\lambda v.v) \right) \right)^* (\lambda x.((\lambda y.y) (\lambda z.z))) \right)$$

Example

$$\left(\left((\lambda w.w)^* ((\lambda u.u) (\lambda v.v)) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)$$

Example

$$\xrightarrow{\text{name}} \left(\left((\lambda w.w) ((\lambda u.u) (\lambda v.v)) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)$$

Example

$$\xrightarrow{\text{name}} \left(\left((\lambda w.w) ((\lambda u.u) (\lambda v.v)) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)$$
$$\left(((\lambda u.u) (\lambda v.v)) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)^*$$

Example

$$\xrightarrow{\text{name}} \left(\left((\lambda w.w) ((\lambda u.u) (\lambda v.v)) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)$$
$$\left(((\lambda u.u) (\lambda v.v))^* (\lambda x.((\lambda y.y) (\lambda z.z))) \right)$$

Example

$$\xrightarrow{\text{name}} \left(\left((\lambda w.w) ((\lambda u.u) (\lambda v.v)) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)$$
$$\left(((\lambda u.u)^* (\lambda v.v)) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)$$

Example

$$\begin{aligned} & \left(\left((\lambda w.w) \left((\lambda u.u) (\lambda v.v) \right) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left(\left((\lambda u.u) (\lambda v.v) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left((\lambda v.v) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \end{aligned}$$

Example

$$\begin{aligned} & \left(\left((\lambda w.w) ((\lambda u.u) (\lambda v.v)) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left(((\lambda u.u) (\lambda v.v)) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left((\lambda v.v) (\lambda x.((\lambda y.y) (\lambda z.z))) \right)^* \end{aligned}$$

Example

$$\begin{aligned} & \left(\left((\lambda w.w) \left((\lambda u.u) (\lambda v.v) \right) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left(\left((\lambda u.u) (\lambda v.v) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left((\lambda v.v)^* (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \end{aligned}$$

Example

$$\begin{aligned} & \left(\left((\lambda w.w) \left((\lambda u.u) (\lambda v.v) \right) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left(\left((\lambda u.u) (\lambda v.v) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left((\lambda v.v) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & (\lambda x.((\lambda y.y) (\lambda z.z))) \end{aligned}$$

Example

$$\begin{aligned} & \left(\left((\lambda w.w) \left((\lambda u.u) (\lambda v.v) \right) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left(\left((\lambda u.u) (\lambda v.v) \right) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & \left((\lambda v.v) (\lambda x.((\lambda y.y) (\lambda z.z))) \right) \\ \xrightarrow{\text{name}} & (\lambda x.((\lambda y.y) (\lambda z.z)))^* \end{aligned}$$

- Call-by-name reduction is deterministic: if $s \xrightarrow{\text{name}} t$ and $s \xrightarrow{\text{name}} t' \implies t = t'$
- No call-by-name reduction is applicable iff $s = R[x]$ or if s is an abstraction (called an **FWHNF** (functional weak head normal form))
- Reaching an FWHNF means success
- $\xrightarrow{\text{name},+}$ and $\xrightarrow{\text{name},*}$ are the transitive and reflexive-transitive closure of $\xrightarrow{\text{name}}$.

Definition

Expression s (call-by-name) **converges** $s \downarrow$ iff \exists abstraction $v : s \xrightarrow{\text{name},*} v$.

If s does not converge, we write $s \uparrow$ and say s **diverges**.

Standardisation (Preparations)

- Goal: show that call-by-name evaluation is optimal w.r.t. convergence
- Technique: \rightarrow_1 -reduction, also on contexts:
 C is treated like an expression with constant $[\cdot]$
- If $R \rightarrow_1 R'$ for a reduction context R , then R' is also a reduction context.
- If $C \rightarrow_1 C'$, $s \rightarrow_1 s'$, where C is a context, then $C[s] \rightarrow_1 C[s']$

Definition ($\xrightarrow{\text{name}}_1$ and $\xrightarrow{\text{int}}_1$)

If $R \rightarrow_1 R'$, $s \rightarrow_1 s'$, $t \rightarrow_1 t'$, and R is a reduction context, then

$$R[(\lambda x.s) t] \xrightarrow{\text{name}}_1 R'[s'[t'/x]]$$

Let $\xrightarrow{\text{int}}_1 := \rightarrow_1 \setminus \xrightarrow{\text{name}}_1$ be the **internal** $\xrightarrow{1}$ -reduction.

Note that: $\xrightarrow{\text{name}} \subset \xrightarrow{\text{name}}_1 \subset \rightarrow_1$

Counting the contracted redexes:

Definition

Define the measure $\phi : \rightarrow_1 \rightarrow \mathbb{N}_0$ inductively as

$$\phi(x \rightarrow_1 x) = 0, \text{ if } x \text{ is a variable}$$

$$\phi(\lambda x.s \rightarrow_1 \lambda x.s') = \phi(s \rightarrow s')$$

$$\phi(((\lambda x.s) t) \rightarrow_1 s'[t'/x]) = 1 + \phi(s \rightarrow_1 s') + k \cdot \phi(t \rightarrow_1 t'), \text{ where } k \text{ is the number of free occurrences of } x \text{ in } s$$

$$\phi((s t) \rightarrow_1 (s' t')) = \phi(s \rightarrow_1 s') + \phi(t \rightarrow_1 t')$$

Measure ϕ is defined for every \rightarrow_1 -step and it is well-founded.

Lemma 3.4.10

If $s \rightarrow_1 t$, then $s \xrightarrow{\text{name},*} s' \xrightarrow{\text{int}}_1 t$.

Proof. If $s \xrightarrow{\text{int}}_1 t$, then the claim holds.

Otherwise, $s \xrightarrow{\text{name}}_1 t$, i.e. $s = R[(\lambda x.r) u]$, $r \rightarrow_1 r'$, $u \rightarrow_1 u'$, $R \rightarrow_1 R'$, $t = R'[r'[u'/x]]$.

Then $s \xrightarrow{\text{name}} R[r[u/x]] \rightarrow_1 R'[r'[u'/x]]$ and this can be iterated. If the iteration stops, the demanded reduction sequence is constructed.

For termination, we verify $\phi(s \rightarrow_1 t) > \phi(R[r[u/x]] \rightarrow_1 R'[r'[u'/x]])$:

$$\begin{aligned} \phi(R[(\lambda x.r) u] \rightarrow_1 R'[r'[u'/x]]) &= \phi(R \rightarrow_1 R') + \phi((\lambda x.r) u \rightarrow_1 r'[u'/x]) \\ &= \phi(R \rightarrow_1 R') + 1 + \phi(r \rightarrow_1 r') + k\phi(u \rightarrow_1 u') \\ \phi(R[r[u/x]] \rightarrow_1 R'[r'[u'/x]]) &= \phi(R \rightarrow_1 R') + \phi(r \rightarrow_1 r') + k\phi(u \rightarrow_1 u') \end{aligned}$$

where k is the number of free occurrences of x in r . □

Lemma 3.4.11

Let $s \rightarrow_1 t \xrightarrow{\text{name}} r$, then there exists u such that $s \xrightarrow{\text{name},+} u \xrightarrow{\text{int}}_1 r$.

$$\begin{array}{ccc}
 s & \xrightarrow{\quad} & t \\
 \text{name},+ \downarrow & & \downarrow \text{name} \\
 u & \xrightarrow{\text{int}}_1 & r
 \end{array}$$

Proof.

By Lemma 3.4.10 $s \rightarrow_1 t \xrightarrow{\text{name}} r$ can be written as $s \xrightarrow{\text{name},*} t' \xrightarrow{\text{int}}_1 t \xrightarrow{\text{name}} r$.

Since $t \xrightarrow{\text{name}} r$, we can assume that $t = R[(\lambda x.t_0) t_1] \xrightarrow{\text{name}} R[t_0[t_1/x]] = r$.

Since $t' \xrightarrow{\text{int}}_1 t$ is internal, $t' = R'[(\lambda x.t'_0) t'_1]$ where $R' \rightarrow_1 R$, $t'_0 \rightarrow_1 t_0$, and $t'_1 \rightarrow_1 t_1$.

Then $t' = R'[(\lambda x.t'_0) t'_1] \xrightarrow{\text{name}} R'[t'_0[t'_1/x]] \rightarrow_1 R[t_0[t_1/x]] = r$ holds.

Applying Lemma 3.4.10 to $R'[t'_0[t'_1/x]] \rightarrow_1 R[t_0[t_1/x]] = r$ shows the claim. \square

Shifting $\xrightarrow{\text{name},*}$ over \rightarrow_1

Applying Lemma 3.4.11 iteratively shows:

Lemma 3.4.12

If $s \rightarrow_1 t \xrightarrow{\text{name},*} v$, v is an FWHNF, then $s \xrightarrow{\text{name},*} v' \xrightarrow{\text{int}}_1 v$ where v' is an FWHNF.

The induction is on the number of $\xrightarrow{\text{name}}$ -steps.

For the base case, observe that $\xrightarrow{\text{int}}_1$ steps do not transform non-FWHNFs into FWHNFs.

For the induction step, apply Lemma 3.4.11 and use the induction hypothesis.

Call-by-name evaluation is an optimal strategy w.r.t. termination:

Standardisation-Theorem

If $s \xrightarrow{C,\beta,*} v$ where v is a FWHNF, then $s \downarrow$.

Proof. The given sequence is also a sequence of \rightarrow_1 -reductions, since $\xrightarrow{C,\beta} \subseteq \rightarrow_1$.

We show by induction on n : if $s \xrightarrow{n} v$ where v is an FWHNF, then $s \downarrow$.

If $n = 0$, then the claim holds. The induction step:

$$s \xrightarrow[1]{} s' \xrightarrow[1]{n-1} v$$



Call-by-name evaluation is an optimal strategy w.r.t. termination:

Standardisation-Theorem

If $s \xrightarrow{C, \beta, *} v$ where v is a FWHNF, then $s \Downarrow$.

Proof. The given sequence is also a sequence of \rightarrow_1 -reductions, since $\xrightarrow{C, \beta} \subseteq \rightarrow_1$.

We show by induction on n : if $s \xrightarrow{n} v$ where v is an FWHNF, then $s \Downarrow$.

If $n = 0$, then the claim holds. The induction step:

$$\begin{array}{c}
 s \xrightarrow{1} s' \xrightarrow[n-1]{1} v \\
 \quad \quad \quad \downarrow \text{name,*} \\
 \quad \quad \quad v'
 \end{array}$$

Dashed steps follow from the IH,

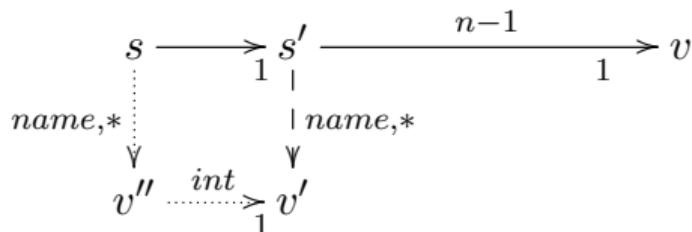


Call-by-name evaluation is an optimal strategy w.r.t. termination:

Standardisation-Theorem

If $s \xrightarrow{C,\beta,*} v$ where v is a FWHNF, then $s \Downarrow$.

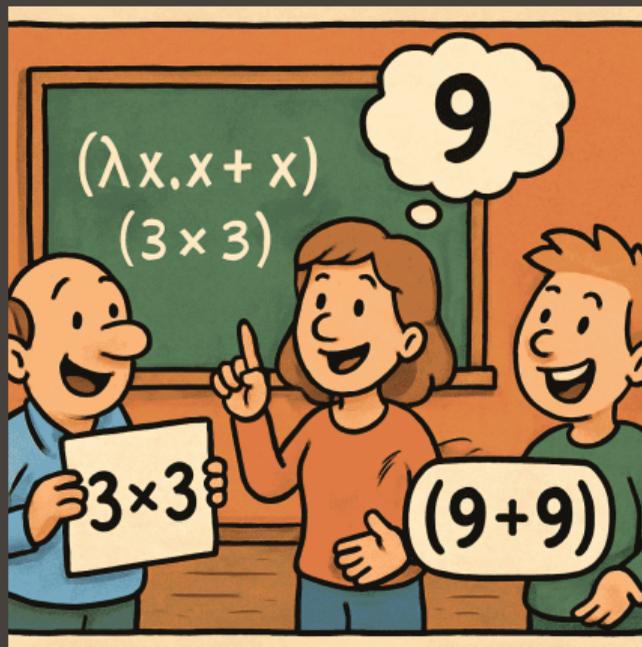
Proof. The given sequence is also a sequence of \rightarrow_1 -reductions, since $\xrightarrow{C,\beta} \subseteq \rightarrow_1$. We show by induction on n : if $s \xrightarrow{n} v$ where v is an FWHNF, then $s \Downarrow$. If $n = 0$, then the claim holds. The induction step:



Dashed steps follow from the IH, dotted steps follow by Lemma 3.4.12



CALL-BY-VALUE EVALUATION



- Used in strict functional programming languages like ML, F#, ...
- Difference to call-by-name: β -reduction is only permitted if the argument is a value (an abstraction)

Definition

The (direct) (β_{value}) -reduction is defined as

$$(\lambda x.s) v \xrightarrow{\beta_{value}} s[v/x] \text{ where } v \text{ is a variable or an abstraction.}$$

We write $\xrightarrow{C, \beta_{value}}$ for the contextual closure of $\xrightarrow{\beta_{value}}$.

Call-by-value evaluation requires to evaluate parameters before calling the function!

Definition

Call-by-value reduction contexts E are built as follows:

$$\mathbf{ECtxt} ::= [\cdot] \mid (\mathbf{ECtxt} \mathbf{Expr}) \mid ((\lambda V. \mathbf{Expr}) \mathbf{ECtxt})$$

If $r_1 \xrightarrow{\beta_{value}} r_2$ and E is a call-by-value reduction context, then

$$E[r_1] \xrightarrow{value} E[r_2]$$

is a call-by-value reduction.

Example

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

Example

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

Example

$$\begin{array}{l} (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \end{array}$$

Example

$$\begin{array}{l} (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \end{array}$$

Example

$$\begin{aligned} & (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \end{aligned}$$

Example

$$\begin{aligned} & (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \end{aligned}$$

Example

$$\begin{aligned} & (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & (\lambda z_2.z_2) ((\lambda z_3.z_3) (\lambda z_4.z_4)) \end{aligned}$$

Example

$$\begin{aligned} & (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & (\lambda z_2.z_2) ((\lambda z_3.z_3) (\lambda z_4.z_4)) \end{aligned}$$

Example

$$\begin{aligned} & (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & (\lambda z_2.z_2) ((\lambda z_3.z_3) (\lambda z_4.z_4)) \\ \xrightarrow{\text{value}} & (\lambda z_2.z_2) (\lambda z_4.z_4) \end{aligned}$$

Example

$$\begin{aligned} & (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & (\lambda z_2.z_2) ((\lambda z_3.z_3) (\lambda z_4.z_4)) \\ \xrightarrow{\text{value}} & (\lambda z_2.z_2) (\lambda z_4.z_4) \end{aligned}$$

Example

$$\begin{aligned} & (\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) ((\lambda z_1.z_1)(\lambda z_2.z_2)) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & (\lambda z_2.z_2) ((\lambda z_3.z_3) (\lambda z_4.z_4)) \\ \xrightarrow{\text{value}} & (\lambda z_2.z_2) (\lambda z_4.z_4) \\ \xrightarrow{\text{value}} & (\lambda z_4.z_4) \end{aligned}$$

- For s , start with s^\star .
- Exhaustively apply the rules:

$$(s_1 s_2)^\star \Rightarrow (s_1^\star s_2)$$

$$(v^\star s) \Rightarrow (v s^\star) \quad \text{if } v \text{ is an abstraction and } s \text{ is not an abstraction or a variable}$$

Three cases

- s is labeled with \star and s is an abstraction: no reduction applicable
- $s = E[x^\star]$, i.e. a free variable in reduction position: no reduction applicable
- $s = E[(\lambda x.t)^\star v]$ where v is an abstraction or variable:

$$\text{Then } E[(\lambda x.t) v] \xrightarrow{\text{value}} E[t[v/x]]$$

Example

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

Example

$((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z)))^*$

Example

$$((\lambda x.(x (x x)))^* ((\lambda y.y y) (\lambda z.z)))$$

Example

$$((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))^*)$$

Example

$$((\lambda x.(x (x x))) ((\lambda y.y y)^* (\lambda z.z)))$$

Example

$$\begin{array}{l} ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \end{array}$$

Example

$$\begin{array}{l} ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2)))^* \end{array}$$

Example

$$\begin{array}{l} ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} ((\lambda x.(x (x x)))^* ((\lambda z_1.z_1) (\lambda z_2.z_2))) \end{array}$$

Example

$$\begin{array}{l} ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2)))^* \end{array}$$

Example

$$\begin{array}{l} ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} ((\lambda x.(x (x x))) ((\lambda z_1.z_1)^* (\lambda z_2.z_2))) \end{array}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \star \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x)))^* (\lambda z_2.z_2) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) ((\lambda z_4.z_4) (\lambda z_5.z_5))) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & \left((\lambda z_3.z_3) \left((\lambda z_4.z_4) (\lambda z_5.z_5) \right) \right)^* \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3)^* ((\lambda z_4.z_4) (\lambda z_5.z_5))) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) ((\lambda z_4.z_4) (\lambda z_5.z_5))^*) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) ((\lambda z_4.z_4)^* (\lambda z_5.z_5))) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) ((\lambda z_4.z_4) (\lambda z_5.z_5))) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) (\lambda z_5.z_5)) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) ((\lambda z_4.z_4) (\lambda z_5.z_5))) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) (\lambda z_5.z_5))^* \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) ((\lambda z_4.z_4) (\lambda z_5.z_5))) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) \star (\lambda z_5.z_5)) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) ((\lambda z_4.z_4) (\lambda z_5.z_5))) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) (\lambda z_5.z_5)) \\ \xrightarrow{\text{value}} & (\lambda x_5.x_5) \end{aligned}$$

Example

$$\begin{aligned} & ((\lambda x.(x (x x))) ((\lambda y.y y) (\lambda z.z))) \\ \xrightarrow{\text{value}} & ((\lambda x.(x (x x))) ((\lambda z_1.z_1) (\lambda z_2.z_2))) \\ \xrightarrow{\text{value}} & (\lambda x.(x (x x))) (\lambda z_2.z_2) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) ((\lambda z_4.z_4) (\lambda z_5.z_5))) \\ \xrightarrow{\text{value}} & ((\lambda z_3.z_3) (\lambda z_5.z_5)) \\ \xrightarrow{\text{value}} & (\lambda x_5.x_5)^* \end{aligned}$$

Note that call-by-value reduction is deterministic.

Definition

Expression s **converges for call-by-value evaluation**:

$$s \downarrow_{value} \text{ iff } \exists \text{abstraction } v : s \xrightarrow{value,*} v.$$

If $\neg s \downarrow_{value}$, then we write $s \uparrow_{value}$ (s diverges for call-by-value evaluation).

Standardisation-Theorem immediately shows:

Corollary

For all expressions s : $s \downarrow_{value} \implies s \downarrow$

Standardisation-Theorem immediately shows:

Corollary

For all expressions s : $s \downarrow_{value} \implies s \downarrow$

The converse does **not** hold:

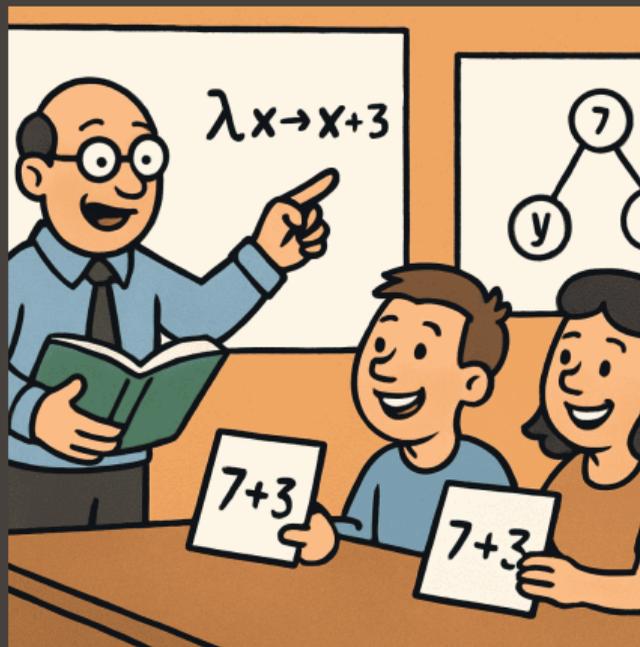
- $\Omega := (\lambda x.x x) (\lambda x.x x)$.
- $\Omega \xrightarrow{name} \Omega$ and also $\Omega \xrightarrow{value} \Omega$
- $(\lambda x.\lambda y.y) \Omega \xrightarrow{name} \lambda y.y$, thus $(\lambda x.\lambda y.y) \Omega \downarrow$
- $(\lambda x.\lambda y.y) \Omega \xrightarrow{value} (\lambda x.\lambda y.y) \Omega$, thus $(\lambda x.\lambda y.y) \Omega \uparrow_{value}$

Call-by-Name and Call-by-Value (Cont'd)

Consider $f\ s_1\ s_2\ s_3$

- In call-by-value evaluation: first s_1 , then s_2 , then s_3 , then the application
→ Evaluation order is predictable
- In call-by-name evaluation: first the application.
When (if it all) s_1, s_2, s_3 are evaluated depends on the definition of f !
→ Evaluation order is not predictable
- This is relevant, if $s_i = \text{print } i$
- Main reason why strict functional languages (ML, Ocaml, ...) permit direct side-effects, but non-strict functional languages (Haskell) forbid them.

CALL-BY-NEED EVALUATION



- Also called: **lazy evaluation with sharing**
- It optimizes the call-by-name evaluation: by avoiding duplicated evaluations:
 - Consider $(\lambda x. \dots x \dots x) t$.
 - In call-by-name evaluation t is **copied** and perhaps evaluated several times!
 - Idea of call-by-need: **share** the result of evaluating t
 - We use a new construct for this sharing `let $x = t$ in ...`

Expressions with `let`

Expr ::= V | $\lambda V.$ Expr | (Expr Expr) | `let $V =$ Expr in Expr`

Note that `let` in Haskell is recursive, here we have a non-recursive `let`.

Reduction contexts are \mathbf{R}_{need} :

$$\mathbf{R}_{need} ::= \mathbf{LR}[\mathbf{A}] \mid \mathbf{LR}[\mathbf{let } x = \mathbf{A} \text{ in } \mathbf{R}_{need}[x]]$$

$$\mathbf{A} ::= [\cdot] \mid (\mathbf{A} \mathbf{Expr})$$

$$\mathbf{LR} ::= [\cdot] \mid \mathbf{let } V = \mathbf{Expr} \text{ in } \mathbf{LR}$$

- $\mathbf{A} \hat{=}$ left into the application
- $\mathbf{LR} \hat{=}$ right into the `let`

Call-by-need reduction step $\xrightarrow{\text{need}}$, defined by:

$$(l\beta) \quad R_{\text{need}}[(\lambda x.s) t] \xrightarrow{\text{need}} R_{\text{need}}[\text{let } x = t \text{ in } s]$$

$$(cp) \quad LR[\text{let } x = \lambda y.s \text{ in } R_{\text{need}}[x]] \xrightarrow{\text{need}} LR[\text{let } x = \lambda y.s \text{ in } R_{\text{need}}[\lambda y.s]]$$

$$(llet) \quad LR[\text{let } x = (\text{let } y = s \text{ in } t) \text{ in } R_{\text{need}}[x]] \\ \xrightarrow{\text{need}} LR[\text{let } y = s \text{ in } (\text{let } x = t \text{ in } R_{\text{need}}[x])]$$

$$(lapp) \quad R_{\text{need}}[(\text{let } x = s \text{ in } t) r] \xrightarrow{\text{need}} R_{\text{need}}[\text{let } x = s \text{ in } (t r)]$$

- $(l\beta)$ and (cp) replace (β) ,
- $(lapp)$ and $(llet)$ adjust lets

- Labels: \star, \diamond, \odot
- $\star \vee \diamond$ means \star or \diamond
- For s , start with s^\star .

Shifting-Rules:

- (1) $(\text{let } x = s \text{ in } t)^\star \Rightarrow (\text{let } x = s \text{ in } t^\star)$
- (2) $(\text{let } x = C_1[y^\diamond] \text{ in } C_2[x^\odot]) \Rightarrow (\text{let } x = C_1[y^\diamond] \text{ in } C_2[x])$
- (3) $(\text{let } x = s \text{ in } C[x^{\star \vee \diamond}]) \Rightarrow (\text{let } x = s^\diamond \text{ in } C[x^\odot])$
- (4) $(s t)^{\star \vee \diamond} \Rightarrow (s^\diamond t)$

where (2) is preferred over (3)

(lbeta) $((\lambda x.s)^\diamond t) \rightarrow \text{let } x = t \text{ in } s$

(cp) $\text{let } x = (\lambda y.s)^\diamond \text{ in } C[x^\odot] \rightarrow \text{let } x = \lambda y.s \text{ in } C[\lambda y.s]$

(llet) $\text{let } x = (\text{let } y = s \text{ in } t)^\diamond \text{ in } C[x^\odot]$
 $\rightarrow \text{let } y = s \text{ in } (\text{let } x = t \text{ in } C[x])$

(lapp) $((\text{let } x = s \text{ in } t)^\diamond r) \rightarrow \text{let } x = s \text{ in } (t r)$

Example

$(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y) x))^*$

Example

$(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y) x)^*)$

Example

$(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y) \diamond x))$

Example

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x))$$
$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } (\text{let } y = x \text{ in } y))^\star$$

Example

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x))$$
$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } (\text{let } y = x \text{ in } y)^\star)$$

Example

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x))$$
$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } (\text{let } y = x \text{ in } y^\star))$$

Example

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x))$$

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } (\text{let } y = x^\diamond \text{ in } y^\circ))$$

Example

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x))$$
$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } (\text{let } y = x^\diamond \text{ in } y))$$

Example

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = ((\lambda u. u) (\lambda w. w)) \diamond \text{ in } (\text{let } y = x \odot \text{ in } y))$$

Example

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x))$$

$$\xrightarrow{\text{need, lbeta}} (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y))$$

Example

$$\begin{array}{l}
 (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u) \text{ in } (\text{let } y = x \text{ in } y))^*
 \end{array}$$

Example

$$\begin{array}{l}
 (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u) \text{ in } (\text{let } y = x \text{ in } y)^\star)
 \end{array}$$

Example

$$\begin{aligned}
 & (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u) \text{ in } (\text{let } y = x^\diamond \text{ in } y^\circ))
 \end{aligned}$$

Example

$$\begin{array}{l}
 (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u) \text{ in } (\text{let } y = x^\diamond \text{ in } y))
 \end{array}$$

Example

$$\begin{array}{l}
 (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y))
 \end{array}$$

Example

$$\begin{aligned}
& (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y) \diamond x)) \\
\frac{\text{need,lbeta}}{\rightarrow} & (\text{let } x = ((\lambda u.u) \diamond (\lambda w.w)) \text{ in } (\text{let } y = x \odot \text{ in } y)) \\
\frac{\text{need,lbeta}}{\rightarrow} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u) \diamond \text{ in } (\text{let } y = x \odot \text{ in } y)) \\
\frac{\text{need,lllet}}{\rightarrow} & (\text{let } u = \lambda w.w \text{ in } (\text{let } x = u \text{ in } (\text{let } y = x \text{ in } y)))^*
\end{aligned}$$

Example

$$\begin{aligned}
 & (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,llet}} & (\text{let } u = \lambda w.w \text{ in } (\text{let } x = u \text{ in } (\text{let } y = x \text{ in } y))^\star)
 \end{aligned}$$

Example

$$\begin{aligned}
 & (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,llet}} & (\text{let } u = \lambda w.w \text{ in } (\text{let } x = u \text{ in } (\text{let } y = x \text{ in } y)^\star))
 \end{aligned}$$

Example

$$\begin{aligned}
 & (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,llet}} & (\text{let } u = \lambda w.w \text{ in } (\text{let } x = u \text{ in } (\text{let } y = x \text{ in } y^*)))
 \end{aligned}$$

Example

$$\begin{array}{l}
 (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,llet}} (\text{let } u = \lambda w.w \text{ in } (\text{let } x = u \text{ in } (\text{let } y = x^\diamond \text{ in } y^\circ)))
 \end{array}$$

Example

$$\begin{array}{l}
 (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,llet}} (\text{let } u = \lambda w.w \text{ in } (\text{let } x = u \text{ in } (\text{let } y = x^\diamond \text{ in } y)))
 \end{array}$$

Example

$$\begin{aligned}
 & (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
 \xrightarrow{\text{need,llet}} & (\text{let } u = \lambda w.w \text{ in } (\text{let } x = u^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)))
 \end{aligned}$$

Example

$$\begin{aligned}
& (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\frac{\text{need,lbeta}}{\rightarrow} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
\frac{\text{need,lbeta}}{\rightarrow} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
\frac{\text{need,llet}}{\rightarrow} & (\text{let } u = \lambda w.w \text{ in } (\text{let } x = u^\diamond \text{ in } (\text{let } y = x \text{ in } y)))
\end{aligned}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \end{array}
\end{array}$$

$$\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y)))
\end{array}$$

Example

$$\begin{aligned}
& (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\frac{\text{need,lbeta}}{\rightarrow} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
\frac{\text{need,lbeta}}{\rightarrow} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
\frac{\text{need,llet}}{\rightarrow} & (\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
\frac{\text{need,cp}}{\rightarrow} & (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = x \text{ in } y)))^*
\end{aligned}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}
\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = x \text{ in } y))^\star)
\end{array}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}$$

$$\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = x \text{ in } y)^\star))
\end{array}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}$$

$$\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = x \text{ in } y^\star)))
\end{array}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}$$

$$\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = x^\diamond \text{ in } y^\circ)))
\end{array}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}$$

$$\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = x^\diamond \text{ in } y)))
\end{array}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}$$

$$\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)))
\end{array}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}$$

$$\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w) \text{ in } y)))^*
\end{array}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}$$

$$\begin{array}{l}
(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y))) \\
(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w) \text{ in } y))^\star)
\end{array}$$

Example

$$\begin{aligned}
& (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\frac{\text{need,lbeta}}{\longrightarrow} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
\frac{\text{need,lbeta}}{\longrightarrow} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
\frac{\text{need,llet}}{\longrightarrow} & (\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
\frac{\text{need,cp}}{\longrightarrow} & (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y))) \\
\frac{\text{need,cp}}{\longrightarrow} & (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w) \text{ in } y)^\star))
\end{aligned}$$

Example

$$\begin{array}{l}
 (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y) \diamond x)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = ((\lambda u.u) \diamond (\lambda w.w)) \text{ in } (\text{let } y = x \odot \text{ in } y)) \\
 \xrightarrow{\text{need,lbeta}} (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u) \diamond \text{ in } (\text{let } y = x \odot \text{ in } y)) \\
 \xrightarrow{\text{need,llet}} (\text{let } u = (\lambda w.w) \diamond \text{ in } (\text{let } x = u \odot \text{ in } (\text{let } y = x \text{ in } y))) \\
 \xrightarrow{\text{need,cp}} (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \diamond \text{ in } (\text{let } y = x \odot \text{ in } y))) \\
 \xrightarrow{\text{need,cp}} (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w) \text{ in } y^{\star})))
 \end{array}$$

Example

$$\begin{aligned}
& (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y) \diamond x)) \\
\frac{\text{need,lbeta}}{\longrightarrow} & (\text{let } x = ((\lambda u.u) \diamond (\lambda w.w)) \text{ in } (\text{let } y = x \odot \text{ in } y)) \\
\frac{\text{need,lbeta}}{\longrightarrow} & (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u) \diamond \text{ in } (\text{let } y = x \odot \text{ in } y)) \\
\frac{\text{need,llet}}{\longrightarrow} & (\text{let } u = (\lambda w.w) \diamond \text{ in } (\text{let } x = u \odot \text{ in } (\text{let } y = x \text{ in } y))) \\
\frac{\text{need,cp}}{\longrightarrow} & (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \diamond \text{ in } (\text{let } y = x \odot \text{ in } y))) \\
\frac{\text{need,cp}}{\longrightarrow} & (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w) \diamond \text{ in } y \odot)))
\end{aligned}$$

Example

$$\begin{array}{l}
(\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \\ \xrightarrow{\text{need,cp}} \\ \xrightarrow{\text{need,cp}} \end{array}
\end{array}$$

$(\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y))$
 $(\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y))$
 $(\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y)))$
 $(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)))$
 $(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w)^\diamond \text{ in } y^\circ)))$
 $(\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w) \text{ in } (\lambda w.w))))$

Example

$$\begin{aligned}
& (\text{let } x = (\lambda u.u) (\lambda w.w) \text{ in } ((\lambda y.y)^\diamond x)) \\
\begin{array}{l} \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,lbeta}} \\ \xrightarrow{\text{need,llet}} \\ \xrightarrow{\text{need,cp}} \\ \xrightarrow{\text{need,cp}} \\ \xrightarrow{\text{need,cp}} \end{array} & (\text{let } x = ((\lambda u.u)^\diamond (\lambda w.w)) \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
& (\text{let } x = (\text{let } u = \lambda w.w \text{ in } u)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y)) \\
& (\text{let } u = (\lambda w.w)^\diamond \text{ in } (\text{let } x = u^\circ \text{ in } (\text{let } y = x \text{ in } y))) \\
& (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w)^\diamond \text{ in } (\text{let } y = x^\circ \text{ in } y))) \\
& (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w)^\diamond \text{ in } y^\circ))) \\
& (\text{let } u = (\lambda w.w) \text{ in } (\text{let } x = (\lambda w.w) \text{ in } (\text{let } y = (\lambda w.w) \text{ in } (\lambda w.w))))
\end{aligned}$$

- The final expression is a call-by-need **FWHNF**
- **Call-by-need FWHNF**: expression of the form $LR[\lambda x.s]$, i.e.

$$\begin{aligned}
& \text{let } x_1 = s_1 \text{ in} \\
& \quad (\text{let } x_2 = s_2 \text{ in} \\
& \quad \quad (\dots \\
& \quad \quad \quad (\text{let } x_n = s_n \text{ in } \lambda x.s)))
\end{aligned}$$

Definition

Expression s **converges for call-by-need evaluation**:

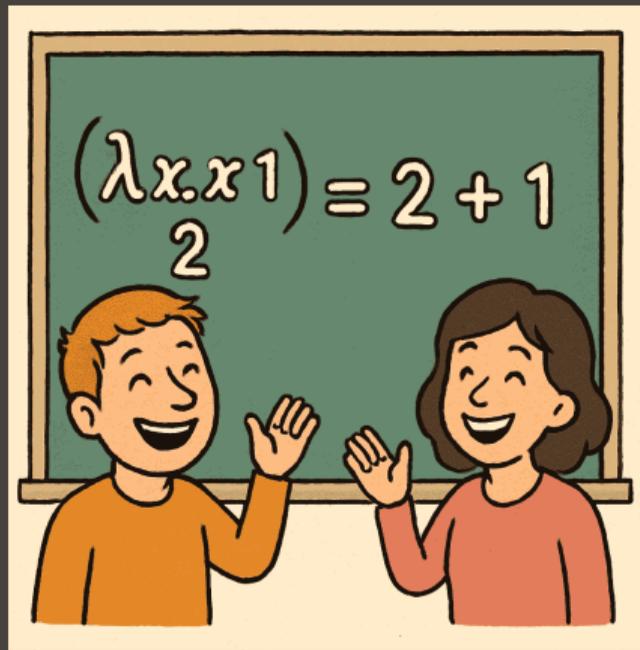
$$s \downarrow_{need} \iff \exists \text{FWHNF } v : s \xrightarrow{need,*} v$$

Proposition

Let s be (let-free) expression, then $s \downarrow \iff s \downarrow_{need}$.

→ Call-by-name and call-by-need are equal w.r.t. convergence

CONTEXTUAL EQUIVALENCE



Equality in the Lambda Calculus

Up to now, we used three notions of equality:

- Syntactic equality
- α -equivalence
- β -convertibility $\xrightarrow{C, \beta, *}$

All of them are quite restrictive

→ We introduce a **semantic equivalence**, called **contextual equivalence**

Idea of Contextual Equivalence

Leibniz' law of the identity of indiscernibles:

- if objects o_1 and o_2 have the same property for all properties, then o_1 is identical to o_2 .
- equality thus means: in every context, we can exchange o_1 by o_2 , but no difference is observable.

For program calculi like the lambda calculus:

Expressions s and t are equal iff
 their behaviour cannot be distinguished independently in which context they are used.

convergence $\forall C : C[s] \text{ and } C[t] \dots$

Definition

For the call-by-name lambda calculus:

contextual approximation \leq_c and **contextual equivalence** \sim_c are defined as

- $s \leq_c t$ iff $\forall C : C[s] \downarrow \implies C[t] \downarrow$
- $s \sim_c t$ iff $s \leq_c t$ und $t \leq_c s$

For the call-by-value lambda calculus:

contextual approximation $\leq_{c,value}$ and **contextual equivalence** $\sim_{c,value}$ are defined as:

- $s \leq_{c,value} t$ iff $\forall C : \text{If } C[s], C[t] \text{ are closed and } C[s] \downarrow_{value}, \text{ then also } C[t] \downarrow_{value}$
- $s \sim_{c,value} t$ iff $s \leq_{c,value} t$ and $t \leq_{c,value} s$

We omit the call-by-need lambda calculus.

- call-by-name: no difference if all, or only closing contexts are used in \sim_c
- call-by-value: there is a difference:
 $x \sim_{c,value} \lambda y.(x y)$ holds, but would not hold, if all contexts are used

	Variables represent
call-by-name	any expression
call-by-value	any value

Remark:

- the transformation $s \rightarrow \lambda x.(s x)$ is called **eta-expansion**
- the inverse transformation $\lambda x.(s x) \rightarrow s$ is called **eta-reduction**

Contextual equivalence is the coarsest equivalence that distinguishes obviously different expressions.

An important property is:

Proposition

- \sim_c and $\sim_{c,value}$ are **congruences**, i.e. they are equivalence relations (i.e. reflexive, symmetric & transitive) and compatible with contexts ($s \sim t \implies C[s] \sim C[t]$).
- \leq_c and $\leq_{c,value}$ are **precongruences**, i.e. they are preorders (i.e. reflexive & transitive) and compatible with contexts ($s \leq t \implies C[s] \leq C[t]$).

Proof: We only consider the precongruences, since the congruences follows by symmetry.

(next slide)

\leq_c is a Precongruence

- reflexivity: for all contexts and expressions $C[s] \Downarrow \implies C[s] \Downarrow$, thus $s \leq_c s$
- transitivity: Let $r \leq_c s$ and $s \leq_c t$ and $C[r] \Downarrow$.

We have to show $C[t] \Downarrow$:

From $r \leq_c s$ we have $C[s] \Downarrow$.

From $s \leq_c t$ we also have $C[t] \Downarrow$.

- compatibility: Let $s \leq_c t$ and C be a context.

We have to show $C[s] \leq_c C[t]$

Let C' be a context such that $C'[C[s]] \Downarrow$.

Since $C'[C[\cdot]]$ is also a context, $s \leq_c t$ shows $C'[C[t]] \Downarrow$.

$\leq_{c,value}$ is a Precongruence

- reflexivity and compatibility: similar to \leq_c
- transitivity: let $r \leq_{c,value} s$ and $s \leq_{c,value} t$

Let C be a context such that $C[r]$ and $C[t]$ are closed and $C[r] \downarrow_{value}$.

We have to show $C[t] \downarrow_{value}$. If $C[s]$ is also closed, the reasoning is as for \leq_c .

Otherwise, assume $FV(C[s]) = \{x_1, \dots, x_n\}$.

Let v_1, \dots, v_n be arbitrary closed values and $D = (\lambda x_1, \dots, x_n. [\cdot]) v_1 \dots v_n$.

Since $C[r]$ and $C[t]$ are closed:

- $D[C[r]] \xrightarrow{value,*} C[r]$ and $D[C[t]] \xrightarrow{value,*} C[t]$
- Thus: $D[C[r]] \downarrow_{value} \iff C[r] \downarrow_{value}$ and $D[C[t]] \downarrow_{value} \iff C[t] \downarrow_{value}$

Since $C[r] \downarrow_{value}$, we have $D[C[r]] \downarrow_{value}$.

From $r \leq_{c,value} s$, we have $D[C[s]] \downarrow$.

From $s \leq_{c,value} t$, we have $D[C[t]] \downarrow_{value}$ and thus also $C[t] \downarrow_{value}$. □

- Program transformation $s \rightarrow t$ is correct iff $s \sim_c t$ holds
- Congruence property shows that **local transformations** preserve “global” equivalence: if $s \sim_c t$ then $C[s] \sim_c C[t]$
- Proving correctness is **usually hard**, because of the universal quantification on all contexts
- Decision problems $s \stackrel{?}{\sim}_c t$ or $s \stackrel{?}{\not\sim}_c t$ are undecidable, since:
 $s \stackrel{?}{\not\sim}_c \Omega$ can be used to encode the halting problem
(and since the lambda calculus is Turing complete)

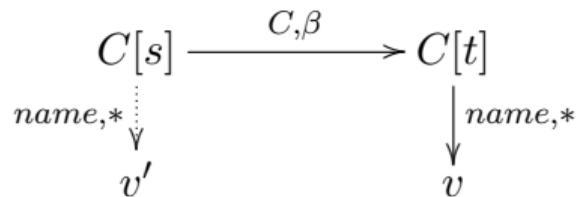
(β) is Correct in the Call-by-Name Lambda Calculus

Proposition

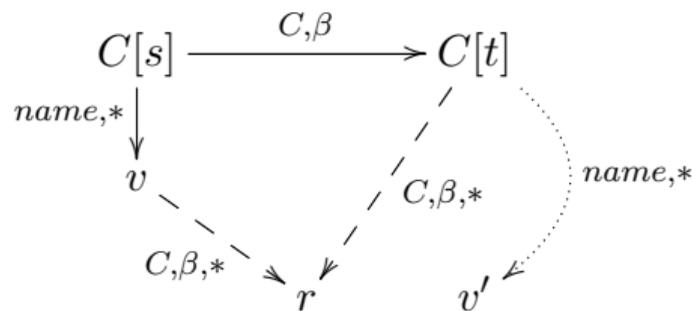
In the call-by-name lambda calculus: if $s \xrightarrow{\beta} t$, then $s \sim_c t$.

Proof. Let $s \xrightarrow{\beta} t$ and C be a context.

$$C[t] \downarrow \implies C[s] \downarrow :$$



$$C[s] \downarrow \implies C[t] \downarrow :$$



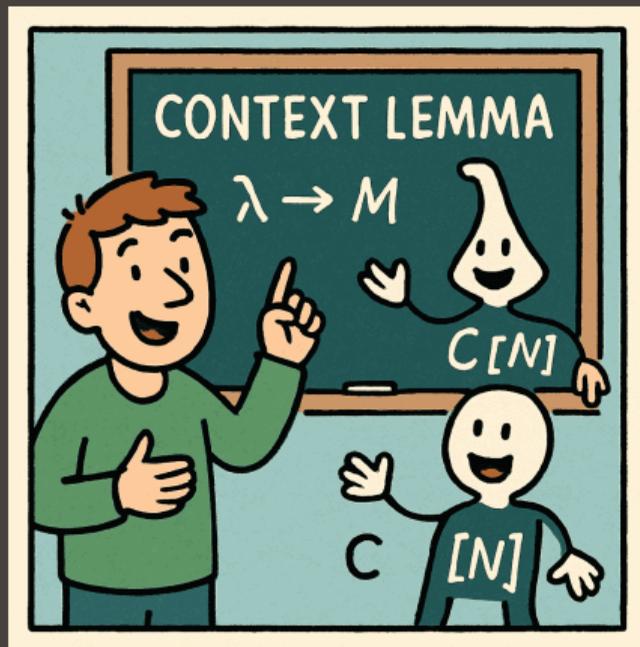
\longrightarrow given reductions

$- - \triangleright$ follows from the Church-Rosser-Theorem

v, r, v' are WHNFs.

$\cdots \triangleright$ follows from the Standardisation-Theorem

CONTEXT LEMMA



- In general, a context lemma states that
it suffices to check a subset of all contexts to conclude contextual equivalence
- We only consider the case of the call-by-name lambda calculus
- We require multi-contexts: Contexts with several (or no) holes
We write $M[\cdot_1, \dots, \cdot_n]$ for a multi-context with n holes
We write $M[s_1, \dots, s_n]$ if the hole \cdot_i of M is replaced by s_i (for $i = 1, \dots, n$).

Context Lemma

Let s and t be closed expressions. If for all reductions contexts R , the implication $R[s] \downarrow \implies R[t] \downarrow$ holds, then also $s \leq_c t$ holds.

Proof. We prove the more general claim using multi-contexts:

If for all closed expressions s_i, t_i and for $i = 1, \dots, n$: for all reduction contexts R the implication $R[s_i] \downarrow \implies R[t_i] \downarrow$ holds, then for all multi-contexts M the implication $M[s_1, \dots, s_n] \downarrow \implies M[t_1, \dots, t_n] \downarrow$ holds.

The context lemma follows with $n = 1$.

If for all closed expressions s_i, t_i and for $i = 1, \dots, n$: for all reduction contexts R the implication $R[s_i] \downarrow \implies R[t_i] \downarrow$ holds, then for all multi-contexts M the implication $M[s_1, \dots, s_n] \downarrow \implies M[t_1, \dots, t_n] \downarrow$ holds.

Assume, the preconditions hold and $M[s_1, \dots, s_n] \xrightarrow{\text{name}, m} v$ where v is a WHNF. We use induction on the following pair, ordered lexicographically;

- 1 The number m of call-by-name reductions from $M[s_1, \dots, s_n]$ to a WHNF.
- 2 The number n of holes of M .

Base case:

- Let $n = 0$ and m arbitrary: Then M is an expression and the claim holds.

Induction step:

- Let $n > 0$, i.e. M has holes
- We split into two cases (next slide)

Case: There exists a hole i , s.t. $M[s_1, \dots, s_{i-1}, \cdot_i, s_{i+1}, \dots, s_n]$ is a reduction context

- Then there exists a hole j , such that $M[r_1, \dots, r_{j-1}, \cdot_j, r_{j+1}, \dots, r_n]$ is a reduction context for any expressions r_1, \dots, r_n .
- Let $M' = M[\cdot_1, \dots, \cdot_{j-1}, s_j, \cdot_{j+1}, \dots, \cdot_n]$
- Since $M'[s_1, \dots, s_{j-1}, s_{j+1}, \dots, s_n] = M[s_1, \dots, s_n]$, they have the same call-by-name evaluation
- M' has $n - 1$ holes, i.e. we can apply the IH, showing $M'[t_1, \dots, t_{j-1}, t_{j+1}, t_n] \downarrow$.
- $C_s = M[s_1, \dots, s_{j-1}, \cdot_j, s_{j+1}, \dots, s_n]$ and $C_t = M[t_1, \dots, t_{j-1}, \cdot_j, t_{j+1}, \dots, t_n]$ are both reduction contexts, $M'[t_1, \dots, t_{j-1}, t_{j+1}, t_n] = C_t[s_j]$ and $C_t[s_j] \downarrow$. Thus the precondition shows that $C_t[t_j] \downarrow$.
- Since $C_t[t_j] = M[t_1, \dots, t_n]$, this shows the claim.

Proof of the Context Lemma (Cont'd)

Case $n > 0$ and no hole is a reduction context.

- If $m = 0$, then $M[s_1, \dots, s_n]$ is a WHNF and $M[t_1, \dots, s_n]$ must be a WHNF too.
- Otherwise, $M[s_1, \dots, s_n] \xrightarrow{\text{name}} s' \xrightarrow{\text{name}, m-1} v$
- Inspect what can happen with the subexpressions s_1, \dots, s_n in M
- Since no hole of M is in a reduction context they can only change their position and may be duplicated or removed.
- Since s_1, \dots, s_n are closed no other expression can be copied inside any s_i .
- Thus: There exists M' with u holes, such that
 - $s' = M'[s_{f(1)}, \dots, s_{f(u)}]$ where $f : \{1, \dots, u\} \rightarrow \{1, \dots, n\}$.
 - $M[r_1, \dots, r_n] \xrightarrow{\text{name}} M'[r_{f(1)}, \dots, r_{f(u)}]$ for any expressions r_1, \dots, r_n
 - in particular, $M[t_1, \dots, t_n] \xrightarrow{\text{name}} M'[t_{f(1)}, \dots, t_{f(u)}] = t'$.

Since $s' \xrightarrow{\text{name}, m-1} v$ and the precondition holds for all pairs $s_{f(i)}, t_{f(i)}$ for $i = 1, \dots, m$ we can apply the IH to s' and t' showing $t' \downarrow$ and thus also $t \downarrow$. □

Proposition 3.8.2

Let s and t be expressions with free variables x_1, \dots, x_n . Then $s \leq_c t$ iff for all closed expressions t_1, \dots, t_n : $s[t_1/x_1, \dots, t_n/x_n] \leq_c t[t_1/x_1, \dots, t_n/x_n]$

Proof.

- “ \Rightarrow ”: This follows since \leq_c is a precongruence and since (β) is correct.
- “ \Leftarrow ”: This can be shown by the context lemma and an induction on the number of variables.

In the call-by-value lambda calculus:

- $(\beta_{value}) \subseteq \sim_{c,value}$ (without a proof)
- $(\beta) \not\subseteq \sim_{c,value}$, since $((\lambda x. (\lambda y. y)) \Omega) \uparrow_{value}$ and $\lambda y. y \downarrow_{value}$

The contextual equivalences w.r.t. call-by-name and call-by-value evaluation are not related:

$$\sim_c \not\subseteq \sim_{c,value} \text{ and } \sim_{c,value} \not\subseteq \sim_c$$

E.g. $((\lambda x. (\lambda y. y)) \Omega) \sim_{c,value} \Omega$ but $((\lambda x. (\lambda y. y)) \Omega) \not\sim_c \Omega$.

Proposition

All closed diverging expressions are least elements w.r.t. \leq_c and $\leq_{c,value}$.
For instance $\Omega \leq_c s$ and also $\Omega \leq_{c,value} s$ for all expressions s .

With $K := \lambda x. \lambda y. x$, $Y := \lambda f. (\lambda x. (f (x x))) (\lambda x. (f (x x)))$, and
 $Z := \lambda f. (\lambda x. (f \lambda z. (x x) z)) (\lambda x. (f \lambda z. (x x) z))$:

- $Y K$ is a greatest element of \leq_c , i.e. $\forall s : s \leq_c Y K$
- $Z K$ is a greatest element of $\leq_{c,value}$, i.e. $\forall s : s \leq_{c,value} Z K$

eteness

We only prove the proposition for the call-by-name lambda calculus.

- Let \perp be a closed diverging expression and s be an arbitrary closed expression.
- Let R be an arbitrary reduction context, then $R[\perp]$ cannot converge, i.e. $R[\perp] \uparrow$.
- The context lemma now immediately shows $\perp \leq_c s$.
- Since \perp is closed, Proposition 3.8.2 shows $\perp \leq_c s$ for any (perhaps also open) expression s

Greatest Elements w.r.t. \leq_c

- $Y := \lambda f.(\lambda x.(f (x x))) (\lambda x.(f (x x)))$
- Let $r_y = (\lambda x.K (x x))$.
- Then $Y K \xrightarrow{C,\beta} r_y r_y \xrightarrow{C,\beta} K (r_y r_y)$
- $(Y K) s_1 \dots s_n \xrightarrow{C,\beta,*} K (r_y r_y) s_1 \dots s_n \xrightarrow{C,\beta,*} (r_y r_y) s_2 \dots s_n \xrightarrow{C,\beta,*} (r_y r_y) \xrightarrow{C,\beta} K (r_y r_y) \xrightarrow{C,\beta} \lambda x.(r_y r_y)$
- The Standardisation-Theorem shows that for all R : $R[(Y K)]\downarrow$.
- Since $(Y K)$ is closed, the context lemma shows $s \leq_c (Y K)$ for every closed expression s
- Proposition 3.8.2 shows $s \leq_c (Y K)$ for any (perhaps also open) expression s .

The Z-Combinator and Call-By-Value

We explain the call-by-value evaluation of $(Z K)$:

- $Z = \lambda f.(\lambda x.(f \lambda z.(x x) z)) (\lambda x.(f \lambda z.(x x) z))$
- Let $r_z = (\lambda x.(K \lambda z.(x x) z))$
- $Z K \xrightarrow{\text{value}} r_z r_z \xrightarrow{\text{value}} K \lambda z.((r_z r_z) z) \xrightarrow{\text{value}} \lambda y.\lambda z.(r_z r_z) z$
- For values v_1, \dots, v_n : $(Z K) v_1 \dots v_n \xrightarrow{\text{value},*} (r_z r_z) v_1 \dots v_n \xrightarrow{\text{value},*}$
 $(\lambda y.\lambda z.(r_z r_z) z) v_1 \dots v_n \xrightarrow{\text{value}} (\lambda z.(r_z r_z) z) v_2 \dots v_n \xrightarrow{\text{value}}$
 $(r_z r_z) v_2 v_3 \dots v_n \xrightarrow{\text{value},*} (r_z r_z) \xrightarrow{\text{value},*} \lambda y.\lambda z.(r_z r_z) z$

TURING COMPLETENESS OF THE LAMBDA CALCULUS



- We do not provide a formal proof
- In the lecture notes, a Haskell-implementation of Turing machines can be found
- We argue that the program constructs used in the program, can be encoded in the lambda calculus

Constructs:

- Named functions
- Recursion
- Data (booleans, numbers, pairs, lists)

Function Definitions

Non-recursive Haskell-function $f\ x_1 \dots x_n = e$ can be represented by $\lambda x_1, \dots, x_n. e$

Recursive functions can be encoded by the fixpoint combinator:

For simplicity, let us assume, that e only calls f , but no other functions.

Non-recursive Haskell-function $f\ x_1 \dots x_n = e$ can be represented by $\lambda x_1, \dots, x_n. e$

Recursive functions can be encoded by the fixpoint combinator:

For simplicity, let us assume, that e only calls f , but no other functions.

Then f can be encoded by $Y (\lambda f. \lambda x_1 \dots x_n. e)$:

Non-recursive Haskell-function $f\ x_1 \dots x_n = e$ can be represented by $\lambda x_1, \dots, x_n. e$

Recursive functions can be encoded by the fixpoint combinator:

For simplicity, let us assume, that e only calls f , but no other functions.

Then f can be encoded by $Y (\lambda f. \lambda x_1 \dots x_n. e)$:

- Let $F = (\lambda f. \lambda x_1 \dots x_n. e)$ and $r_y = (\lambda x. F (x\ x))$.

Non-recursive Haskell-function $f\ x_1 \dots x_n = e$ can be represented by $\lambda x_1, \dots, x_n. e$

Recursive functions can be encoded by the fixpoint combinator:

For simplicity, let us assume, that e only calls f , but no other functions.

Then f can be encoded by $Y (\lambda f. \lambda x_1 \dots x_n. e)$:

- Let $F = (\lambda f. \lambda x_1 \dots x_n. e)$ and $r_y = (\lambda x. F (x\ x))$.
- Then $Y\ F \xrightarrow{C, \beta} r_y\ r_y = (\lambda x. F (x\ x))\ r_y \xrightarrow{C, \beta} F (r_y\ r_y) \xleftarrow{C, \beta} F (Y\ F)$, i.e. $Y\ F \sim_c F (Y\ F)$.

Non-recursive Haskell-function $f\ x_1 \dots x_n = e$ can be represented by $\lambda x_1, \dots, x_n. e$

Recursive functions can be encoded by the fixpoint combinator:

For simplicity, let us assume, that e only calls f , but no other functions.

Then f can be encoded by $Y (\lambda f. \lambda x_1 \dots x_n. e)$:

- Let $F = (\lambda f. \lambda x_1 \dots x_n. e)$ and $r_y = (\lambda x. F (x\ x))$.
- Then $Y\ F \xrightarrow{C, \beta} r_y\ r_y = (\lambda x. F (x\ x))\ r_y \xrightarrow{C, \beta} F (r_y\ r_y) \xleftarrow{C, \beta} F (Y\ F)$, i.e. $Y\ F \sim_c F (Y\ F)$.
- Thus $Y\ F \sim_c F^i (Y\ F)$ where F^i is the i -fold application of F and also $Y\ F \sim_c F (Y\ F) \sim_c \lambda x_1, \dots, x_n. e[(Y\ F)/f]$.

Non-recursive Haskell-function $f\ x_1 \dots x_n = e$ can be represented by $\lambda x_1, \dots, x_n. e$

Recursive functions can be encoded by the fixpoint combinator:

For simplicity, let us assume, that e only calls f , but no other functions.

Then f can be encoded by $Y\ (\lambda f. \lambda x_1 \dots x_n. e)$:

- Let $F = (\lambda f. \lambda x_1 \dots x_n. e)$ and $r_y = (\lambda x. F\ (x\ x))$.
- Then $Y\ F \xrightarrow{C, \beta} r_y\ r_y = (\lambda x. F\ (x\ x))\ r_y \xrightarrow{C, \beta} F\ (r_y\ r_y) \xleftarrow{C, \beta} F\ (Y\ F)$, i.e. $Y\ F \sim_c F\ (Y\ F)$.
- Thus $Y\ F \sim_c F^i\ (Y\ F)$ where F^i is the i -fold application of F and also $Y\ F \sim_c F\ (Y\ F) \sim_c \lambda x_1, \dots, x_n. e[(Y\ F)/f]$.

For mutual recursive functions, the encoding is a bit more complicated, but still possible.

- The Haskell-program uses data types and selectors
- It suffices to represent booleans, tuples, lists of arbitrary length and natural numbers to encode the program
- We sketch the so-called Church encoding of numbers, booleans, pairs and lists.

Encoding:

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

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

$b\ s\ t$ for $b \in \{true, false\}$ behaves like if b then s else t .

Idea:

- Number i is represented by the i -fold function composition.
- $0 = f^0 = id$, $1 = f^1$, $2 = f^2$, ...

Encoding:

$$0 := \lambda f. \lambda x. x$$

$$i := \lambda f. \lambda x. f^i x, \text{ if } i > 0$$

Church Encoding of Numbers (Cont'd)

Addition:

$$plus = \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)$$

Example:

$$plus \ 3 \ 2 = \underbrace{(\lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x))}_{plus} \underbrace{(\lambda f. \lambda x. (f \ (f \ (f \ x))))}_3 \underbrace{(\lambda f. \lambda x. (f \ (f \ x)))}_2$$

Church Encoding of Numbers (Cont'd)

Addition:

$$plus = \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)$$

Example:

$$\begin{aligned} plus \ 3 \ 2 &= \underbrace{(\lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x))}_{plus} \underbrace{(\lambda f. \lambda x. (f \ (f \ (f \ x))))}_3 \underbrace{(\lambda f. \lambda x. (f \ (f \ x)))}_2 \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda f. \lambda x. (f \ (f \ (f \ x)))) \ f \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \end{aligned}$$

Church Encoding of Numbers (Cont'd)

Addition:

$$plus = \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)$$

Example:

$$\begin{aligned} plus \ 3 \ 2 &= \underbrace{(\lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x))}_{plus} \underbrace{(\lambda f. \lambda x. (f \ (f \ (f \ x))))}_3 \underbrace{(\lambda f. \lambda x. (f \ (f \ x)))}_2 \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda f. \lambda x. (f \ (f \ (f \ x)))) \ f \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda x. (f \ (f \ (f \ x)))) \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \end{aligned}$$

Church Encoding of Numbers (Cont'd)

Addition:

$$plus = \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)$$

Example:

$$\begin{aligned} plus \ 3 \ 2 &= \underbrace{(\lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x))}_{plus} \underbrace{(\lambda f. \lambda x. (f \ (f \ (f \ x))))}_3 \underbrace{(\lambda f. \lambda x. (f \ (f \ x)))}_2 \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda f. \lambda x. (f \ (f \ (f \ x)))) \ f \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda x. (f \ (f \ (f \ x)))) \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (f \ (f \ (f \ (n \ f \ x)))) \ (\lambda f. \lambda x. (f \ (f \ x))) \end{aligned}$$

Church Encoding of Numbers (Cont'd)

Addition:

$$plus = \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)$$

Example:

$$\begin{aligned} plus \ 3 \ 2 &= \underbrace{(\lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x))}_{plus} \underbrace{(\lambda f. \lambda x. (f \ (f \ (f \ x))))}_3 \underbrace{(\lambda f. \lambda x. (f \ (f \ x)))}_2 \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda f. \lambda x. (f \ (f \ (f \ x)))) \ f \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda x. (f \ (f \ (f \ x)))) \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (f \ (f \ (f \ (n \ f \ x)))) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} \lambda f. \lambda x. (f \ (f \ (f \ ((\lambda f. \lambda x. (f \ (f \ x))) \ f \ x)))) \end{aligned}$$

Church Encoding of Numbers (Cont'd)

Addition:

$$plus = \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)$$

Example:

$$\begin{aligned} plus \ 3 \ 2 &= \underbrace{(\lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x))}_{plus} \underbrace{(\lambda f. \lambda x. (f \ (f \ (f \ x))))}_3 \underbrace{(\lambda f. \lambda x. (f \ (f \ x)))}_2 \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda f. \lambda x. (f \ (f \ (f \ x)))) \ f \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (\lambda x. (f \ (f \ (f \ x)))) \ (n \ f \ x)) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} (\lambda n. \lambda f. \lambda x. (f \ (f \ (f \ (n \ f \ x)))) \ (\lambda f. \lambda x. (f \ (f \ x))) \\ &\xrightarrow{C, \beta} \lambda f. \lambda x. (f \ (f \ (f \ ((\lambda f. \lambda x. (f \ (f \ x))) \ f \ x)))) \\ &\xrightarrow{C, \beta, 2} \lambda f. \lambda x. (f \ (f \ (f \ (f \ (f \ x)))) = 5 \end{aligned}$$

Church Encoding of Numbers (Cont'd)

Successor:

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

Predecessor (complicated and $pred\ 0 = 0$):

$$pred = \lambda n. \lambda f. \lambda x. n (\lambda g. \lambda h. h (g f)) (\lambda z. x) (\lambda u. u)$$

Encoding:

$$\mathit{pair} := \lambda x. \lambda y. \lambda z. z \ x \ y$$

The first two arguments are the arguments of the pair, the third one is for the selector.

Examples:

$$\begin{aligned} \mathit{first} &= \lambda p. p \ K \\ \mathit{second} &= \lambda p. p \ K2 \end{aligned}$$

Note that $\mathit{first} (\mathit{pair} \ s \ t) \sim_c s$ and $\mathit{second} (\mathit{pair} \ s \ t) \sim_c t$.

Lists

Non-empty lists can be encoded by pairs p :

- the first component of p is the element
- the second component of p is the tail of the list

Lists

Non-empty lists can be encoded by pairs p :

- the first component of p is the element
- the second component of p is the tail of the list

With the empty list: additional pair $pair\ flag\ p$ where $flag$ is *true* or *false* and

- $pair\ true\ s$ means the empty list (independent of s)
- $(false, p)$ is a non-empty list.

Encoding:

$$nil \quad :=\ pair\ true\ true$$

$$cons \quad :=\ \lambda h.\lambda t.pair\ false\ (pair\ h\ t)$$

Lists

Non-empty lists can be encoded by pairs p :

- the first component of p is the element
- the second component of p is the tail of the list

With the empty list: additional pair *pair flag p* where *flag* is *true* or *false* and

- *pair true s* means the empty list (independent of s)
- $(false, p)$ is a non-empty list.

Encoding:

$nil := pair\ true\ true$

$cons := \lambda h.\lambda t.pair\ false\ (pair\ h\ t)$

Examples:

$isNil = first$

$head = \lambda l.first\ (second\ l)$

$tail = \lambda l.second\ (second\ l)$

- Church encoding does not distinguish between data and functions
- Also different data is encoded in the same way (e.g. 0 and *false*)

Types?

- Haskell only allows well-typed programs
- the lambda calculus has no types
- This is not a restriction for Turing completeness, since the typed encoding can also be used as an untyped one

This concludes our sketch on Turing completeness

- the lambda calculus is too small to really program in it
- also for a core language it is too difficult to express data and recursion
- equivalences in the lambda calculus do not necessarily hold in Haskell, since different data is mapped to the same lambda expressions
- Haskell has `seq`, the lambda calculus cannot simulate this
- In the next chapter: We extend the lambda calculus to a real core language of functional programming