The `lc-visualizer` package

Florian Sihler

https://github.com/EagleoutIce/latex-lambda-calculus-visualizer

Version v1.0.0 – 2026/11/21

`lc-visualizer` typesets λ-terms with support for coloring variables (bind-sensitive) and dynamic abstraction ranges. We also support reducing terms and converting them to their normal form step by step (if a normal-form exists), and to convert them to de Bruijn indices.

```
\LC{(\\a.\\b.\a (\b \\b.\b \a)) a c}
```

(λa.λb.a (b λb.b a)) a c

```
\lcReduce[textcolor]{(\\f.\\x.\f \x) \x}
```

$$
\begin{aligned}
&(\lambda f.\lambda x.f\ x)\ x \\
=_{\alpha}\ &(\lambda f.\lambda x'.f\ x')\ x \\
\to_{\beta}\ &\lambda x'.x\ x' \\
\to_{\eta}\ &x
\end{aligned}
$$

Contents

1 Writing terms 2

2 Customization 2

3 Marks 5

4 Rewriting rules 5
- 4.1 Predefined rules . . . 6

5 Reductions 8
- 5.1 Steps . . . 8
- 5.2 Semantics . . . 9
- 5.3 Commands . . . 10
- 5.4 Options . . . 11
- 5.5 Animations . . . 17
- 5.6 Caching and generated code . . . 17

A The reduction in the source 18

B Benchmarking 20

Index of commands and keys 21

1 Writing terms

`\LC{<term>}` typesets a term. It changes the category codes of `(`, `)`, `.` and `[`, so it only works in running text, not within the argument of another macro, a beamer frame without `[fragile]`, `align`, a footnote or a macro definition. Terms are not broken across lines. `\RLC{<term>}` works everywhere, but needs `\(`, `\)` and `\.` instead. In moving arguments like `\section` or `\caption`, give a short version for the table of contents (`\section[...]{...}`). The optional argument of both is the start of the color cycle (default 0) or `plain` for a black term without ranges.

Input	Meaning	Example
`\\x.<body>`	abstraction $\lambda x.\,body$	λx.x
`\\{x,y}.<body>`	several abstractions	λx y.y x
`\x` (`\a` to `\z`)	variable, colored like its abstraction	x y
`\\{foo}.<body>`, `\\{x'}.<body>`	abstraction over another name	λfoo.foo
`\lcVar{foo}`, `\x'`	variable with another name	λx'.x' foo
`\\[red]{x}[m]`, `\x[m]`	color of a binder, mark named `m`	λx.x
`(<term>)`, `\(<term>\)`	grouping	(λx.x) y
`<term> <term>`	application (left-associative)	f x y
`\Y`, `\equiv`	a bold Y and ≡	**Y** ≡ **Y**
`yes`, `42`	plain words and numbers are single atoms	yes 42
`[<term>, <term>]`	a list of the `list` module	(*cons* a (*cons* b *nil*))

The shortcuts `\a` to `\z` stand for the variables of those letters. A longer name is written `\\{name}.` as a binder and `\lcVar{name}` as a variable, and `\lcEnableChar{`⟨*name*⟩`}` adds a shortcut for it, so after `\lcEnableChar{foo}` the macro `\foo` is the variable `foo` as well.

```
\LC{\\{fun,arg}.\lcVar{fun} \lcVar{arg}}

\lcEnableChar{fun}\lcEnableChar{arg}
\lcReduce{(\\{fun,arg}.\fun \arg) \lcVar{id} \
↪lcVar{val}}
```

λfun arg.fun arg
(λfun.λarg.fun arg) id val
$\to_\beta$ (λarg.id arg) val
$\to_\beta$ id val

```
\RLC{\Y \equiv \\f\.\(\\x\.\f \(\x \x\)\) \(\\x\.\f \(\x \x\)\)}
```

Y ≡ λf.(λx.f (x x)) (λx.f (x x))

```
\LC[3]{(\\a.\a) \\b.\b \a}

\LC[plain]{(\\a.\a) \\b.\b \a}
```

(λa.a) λb.b a
(λa.a) λb.b a

2 Customization

All settings are local to the current group. Every one of them has a macro and a key. `\lcSetup{`⟨*keys*⟩`}` sets the keys as the defaults for everything that follows, and `\LC[<keys>]`, `\RLC[<keys>]` and the reduction commands take the same keys for one term. A bare number stays the start of the color cycle, so `\LC[3]`

and `\LC[color-start=3]` are the same. There is one key set, so the keys of the reductions (Section 5.4) work in `\lcSetup` as well and `\lcReduceSetup` is another name for it.

```
\lcSetup{colors={teal,orange},lambda-ranges=false}
\LC{(\\x.\\y.\x) \y} \quad \
↪LC[node-background=false]{(\\x.\\y.\x) \y}
```

(λx.λy.x) y (λx.λy.x) y

A *style* is a named group of settings. `style=<name>` applies it, `\lcNewStyle{⟨name⟩}{⟨code⟩}` defines one, and the code is anything that could stand in `\lcSetup`. The package brings `plain` (black, without ranges and shading) and `plain-math` (`plain` in the text font with a math λ).

```
\lcNewStyle{mine}{colors=red,font=\itshape}
\LC[style=plain]{(\\x.\x) \y} \quad
\LC[style=plain-math]{(\\x.\x) \y} \quad
\LC[style=mine]{(\\x.\x) \y}
```

(λx.x) y (λx.x) y (λx.x) y

Key	Meaning
style, **plain**	a named style, see `\lcNewStyle`
colors, **color-start**, **color-by-name**	the color cycle, where it starts, and a color per name instead of per position
node-background, **textcolor**	the TikZ node behind a variable, and the plain form without it
text-mixin	the color of a name that has no node
color-mixin, **duplicate-mixin**	that node's fill and the shade of a repeated name
shade-same	shade a name that is bound again
lambda, **font**	the λ symbol and the font of a term
lambda-ranges, **paren-depth**	ranges of abstractions and of parentheses, both off by default
glow, **single-lambdas**	glowing variables and one λ per binder
rule-depth, **name-strut**	how deep rules are expanded and the strut of their labels
inner-strut, **strut-line**, **box**	space in rule boxes, line height, and the box around a term
node-style, **bracket-color**	TikZ options of that node and the color of the parentheses
arrow-from	where an arrow leaves its term (`0` left, `.5` middle, `1` right)

lambda-ranges: draw the range of each abstraction (default `false`, as the ranges need room above and below a line).

```
\lcSetup{lambda-ranges}
\LC{((\\a.\a) \b \a)}
```

((λa.a) b a)

paren-depth: draw the ranges of parentheses up to that depth (default `0`).

```
\lcSetup{paren-depth=2}
\LC{((\\a.\a) \b \a)}
```

((λa.a) b a)

colors, **color-start**, **color-by-name**: the color cycle, where it starts, and whether every name keeps its own color. Reductions always color by name.

```
\lcSetup{colors={orange,purple}}
\LC{((\\a.\a) \\b.\b \a)}
```

((λa.a) λb.b a)

text-mixin: how the name of a variable is colored without **node-background**, `#1` is the color of its binder (default `#1!80!black`).

```
\lcSetup{node-background=false,
↪text-mixin=#1!45!black}
\LC{((\\a.\a) \\b.\b \a)}
```

((λa.a) λb.b a)

node-background: put every variable in a TikZ node with a colored background (default `true`), **color-mixin** is the fill of that node, `#1` is the color. Without it a variable is only its name in its color, which **text-mixin** sets. **textcolor** is the other name of `node-background=false`, so `\LC[textcolor]` colors the names and draws no node.

```
\lcSetup{color-mixin=#1!55!white}
\LC{((\\a.\a) \\b.\b \a)}

\lcSetup{textcolor}
\LC{((\\a.\a) \\b.\b \a)}
```

((λa.a) λb.b a)
((λa.a) λb.b a)

shade-same: shade a variable whose name is bound again instead of giving it a new color (default `true`), **duplicate-mixin** is that shade.

```
\LC{(\\a.\a) (\\a.\a) (\\a.\a)}

\lcSetup{shade-same=false}
\LC{(\\a.\a) (\\a.\a) (\\a.\a)}

\lcSetup{shade-same, duplicate-mixin=#1!40!green}
\LC{(\\a.\a) (\\a.\a) (\\a.\a)}
```

(λa.a) (λa.a) (λa.a)
(λa.a) (λa.a) (λa.a)
(λa.a) (λa.a) (λa.a)

glow: color the text inside a variable node as well, so it glows. Pass a value to control how the text color mixes with the binding color (default `90`). It needs **node-background**.

```
\lcSetup{glow}
\LC{(\\a.\a (\\b.\b \c)) \a \c}
```

(λa.a (λb.b c)) a c

single-lambdas: write a λ in front of every name of `\\{x,y}` (default `false`, where the names are separated by **binder-gap**, the macro `\lcSetBinderGap`). **lambda** is the symbol itself and **font** the font of a term.

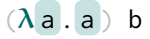
```
\lcSetup{single-lambdas}
\LC{\\{x,y}.\y \x}

\lcSetup{single-lambdas=false, lambda=\Lambda,
↪font=\sffamily}
\LC{\\{x,y}.\y \x}
```

λx.λy.y x
Λx y.y x

strut-line: enlarge lines to fit the ranges, `true`, `false` or `auto` (the default), which enlarges them only while `lambda-ranges` is on. **box**: whether `\LC` and `\RLC` typeset into an unbreakable box (default `true`).

```
\lcSetup{box=false}
\LC{(\\a.\a) \b}
```



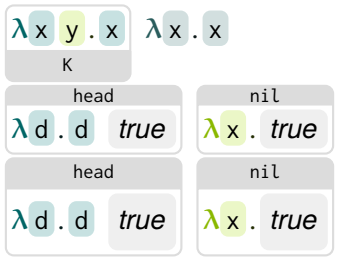
name-strut puts its value into every rule label, and `name-strut` without one means `\strut`, which gives all of them the same height. **inner-strut** is the space around the term in a rule box.

`\lcBelowRewriteRule{`⟨*name*⟩`}{`⟨*term*⟩`}` puts a label below a term.

```
\LC{\lcBelowRewriteRule{K}{\\{x,y}.\x} \\x.\x}

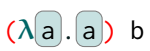
\lcSetup{name-strut={}}
\LC{\head \nil}

\lcSetup{name-strut}
\LC{\head \nil}
```



node-style, **bracket-color**: TikZ options of a variable node and the color of the parentheses.

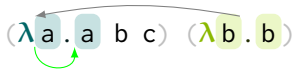
```
\lcSetup{node-style={draw=gray}, bracket-color=red}
\LC{(\\a.\a) \b}
```



3 Marks

`\mark{<name>}{<term>}` and `\\{x}[<name>]` name parts of a term. An arrow between two of them, also within TikZ pictures, is `\reducemarks[`⟨*anchor*⟩`]{`⟨*from*⟩`}{`⟨*to*⟩`}[`⟨*TikZ options*⟩`]`. The arrow follows a flat curve over the terms. The style `lcbend` gives that path back to TikZ, so that `bend right` and friends work.

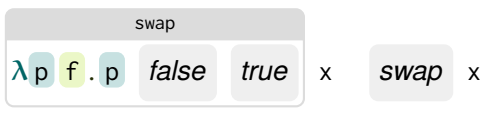
```
\LC{(\\a[a].\a[a2] \b \c) \mark{b}{(\\b.\b)}}
\reducemarks{b}{a}[lcbend, bend right=10]
\reducemarks[south]{a}{a2}[green,-latex]
```



4 Rewriting rules

A rewriting rule is a named term. Rules are defined globally. The name of a rule must not use the rule's own macro. `\lcCreateNewRewritingRule`⟨*cs*⟩`{`⟨*name*⟩`}{`⟨*body*⟩`}` makes ⟨*cs*⟩ available within terms. Single lowercase letters are the variables, so ⟨*cs*⟩ needs a longer name. It is shown as a box with the name and the body. The starred form shows only the name and fits better into running text.

```
\lcCreateNewRewritingRule\swap{swap}
      {\\{p,f}\.\p \false \true}
\LC{\swap \x} \quad \LC{\swap* \x}
```

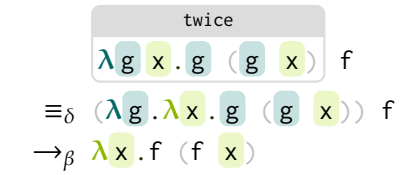


`\lcCreateNewArgumentRewritingRule`⟨*cs*⟩{⟨*name*⟩}{⟨*body*⟩} creates a rule with an argument, which is `\lcArg` in name and body. `\num` is defined as `\lcCreateNewArgumentRewritingRule\num{\lcArg}{...}` with a body that builds the numeral. Rule bodies are written like the argument of `\RLC`. `\lcRawArgumentRule`⟨*cs*⟩ then says that the argument of that rule is no single term but text the body reads itself; `\list` uses it, with `\lcListOf` as the body that turns `a, b, c` into the `\cons` chain. Use it only for encodings whose shape depends on the argument, such as `\num{<n>}`, where every n gives a different term and n is a number, not a λ-term. A term as argument is never needed, as β-reduction inserts it. Write `\twice \f` with an ordinary rule instead, whose reduction also shows that step:

```
\lcCreateNewRewritingRule\twice{twice}
      {\\{g,x}\.\g \(\g \x\)}
\LC{\twice \f} \quad \LC{\num{2}}
```

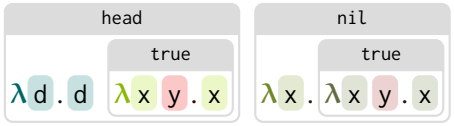



```
\lcReduce{\twice \f}
```



The key `rule-depth` sets how deeply nested rules are expanded (default 1 in `\LC` and `\RLC`, `0` in reductions, where only the rule of the next δ-step is expanded).

```
\lcSetup{rule-depth=2}
\LC{\head \nil}
```



4.1 Predefined rules

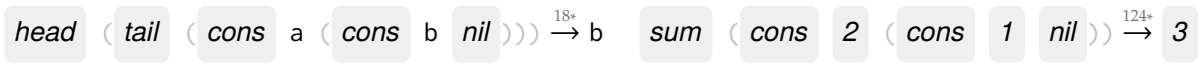
The rules are grouped in modules. All of them are loaded unless you pass the package option `no-default-rewrites`. Then `\lcLoadModule`{⟨*modules*⟩} loads the given modules in the preamble, together with the modules they need. Rules exist only within `\LC`, `\RLC` and the reduction commands, so they do not clash with other packages, such as `\num` of `siunitx`.

```
\usepackage[no-default-rewrites]{lc-visualizer}
\lcLoadModule{bool,list}
```

Module	Needs	Rules
bool	—	\true, \false, \ifelse, \and, \lor, \not
pair	bool	\pair, \fst, \snd
nat	bool	\num{<n>}, \succ, \pred, \add, \sub, \mult, \isZero, \eq
comb	—	\Ycomb, \Tcomb
list	bool, nat, comb	\cons, \nil, \head, \tail, \ListNull, \length, \ListSum, \ListProd, \ListContains, \append, \list{<a, b>}

`\num{<n>}` is the Church numeral of n, and `\Ycomb` and `\Tcomb` are fixed-point combinators for recursion. Lists are pairs ended by `\nil`. The recursive rules `\length`, `\ListSum`, `\ListProd`, `\ListContains` and `\append` use `\Ycomb`. The key `show-term` of the next section puts the term before its normal form, `fold=list` writes a list back as `[a, b]`. Every example below needs the `list` module, which pulls in `bool`, `nat` and `comb`. The second element of a list, and the sum of one:

```
\lcNormalForm[show-term]{\head (\tail [\a, \b])} \quad
\lcNormalForm[show-term, fold=num, max-steps=500]{\ListSum [\num{2}, \num{1}]}
```



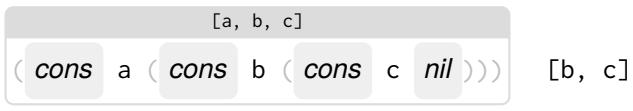
The product of a list, and whether an element is in it:

```
\lcNormalForm[show-term, fold=num, max-steps=500]
   {\ListProd [\num{2}, \num{3}]} \quad
\lcNormalForm[show-term, fold={true, false}, max-steps=500]
   {\ListContains \num{1} [\num{1}]}
```



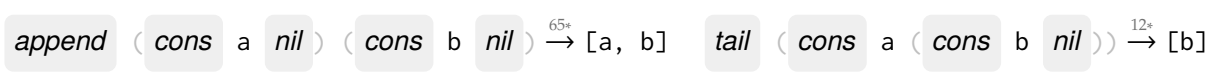
A list can also be *written* as `[a, b, c]`, which reads as `\cons \a (\cons \b (\cons \c \nil))`. It works in every command, in `\RLC` as `\[a, b\]` like its other syntax, and nests. An element made of several terms keeps them together, so `[a b, c]` is `\cons (\a \b) (\cons \c \nil)`. Where the brackets are read before the package sees them, as in a `beamer` frame or a footnote, the `\[a, b\]` form is the one that works. A mark follows its item directly, so `\x[m]` is the variable `x` with the mark `m` while `\lcVar{x} [a, b]` is `x` applied to a list. The rule `\list{a, b, c}` is the same list written as a rule: it shows as *[a, b, c]* and unfolds into the `\cons` chain like every other rule. `\list{}` is the empty list, and a `\list` may stand inside another one.

```
\LC{\list{\a, \b, \c}} \quad
\lcNormalForm[fold=list, max-steps=500]{\tail (\list{\a, \b, \c})}
```



With `fold=list` a list then goes in and comes out in that form:

```
\lcNormalForm[show-term, fold={list, num}, max-steps=500]
   {\append [\a] [\b]} \quad
\lcNormalForm[show-term, fold={list, num}, max-steps=500]
   {\tail [\a, \b]}
```



```
\lcNormalForm[show-term, fold={list, num}, max-steps=800]
   {\ListSum [\num{2}, \num{3}]} \quad
\lcNormalForm[show-term, fold={list, num}, max-steps=800]
   {\length [\a, \b]}
```

sum (cons 2 (cons 3 nil)) $\xrightarrow{124*}$ 5 length (cons a (cons b nil)) $\xrightarrow{92*}$ 2

The length of two lists joined, and the factorial of 3 through `\Ycomb`:

```
\lcNormalForm[show-term, fold=num, max-steps=500]
   {\length (\append [\a] [\b])}

\lcNormalForm[show-term, fold=num, max-steps=3000]
   {\Ycomb (\\{f,n}.\ifelse (\isZero \n) \num{1} (\mult \n (\f (\pred \n)))) \num{3}}
```

length (append (cons a nil) (cons b nil)) $\xrightarrow{216*}$ 2

Y (λf.λn. ifelse (isZero n) 1 (mult n (f (pred n)))) 3 $\xrightarrow{183*}$ 6

Both combinators of `comb` compute fixed points, Curry's **Y** and Turing's Θ. Θ f reduces to f (Θ f) with β-steps alone, so the recursion reappears unchanged, while **Y** f reaches only the unfolded copy of itself.

```
\LC[style=plain]{\Ycomb}\\
\LC[style=plain]{\Tcomb}
```

Y

```
λf.(λx.f (x x)) (λx.f (x x))
```

Θ

```
(λx y.y (x x y)) (λx y.y (x x y))
```

The same factorial through Θ:

```
\lcNormalForm[show-term, fold=num, max-steps=3000]
   {\Tcomb (\\{f,n}.\ifelse (\isZero \n) \num{1} (\mult \n (\f (\pred \n)))) \num{3}}
```

Θ (λf.λn. ifelse (isZero n) 1 (mult n (f (pred n)))) 3 $\xrightarrow{186*}$ 6

5 Reductions

5.1 Steps

A reduction rewrites a term step by step until no step applies. The result is its *normal form*, in which rules that are never applied to anything stay folded. $M[x := N]$ denotes M with the free occurrences of x replaced by N.

Step	Rule	Meaning
β-reduction	`(λx.M) N` $\to_\beta M[x := N]$	applies a function to its argument. In the row before the step, an arrow points from the argument to the binder
α-conversion	`λx.M` $=_\alpha$ `λx'.M`, with x' for x in M	renames a bound variable, only when a β-reduction would otherwise capture a free variable of the argument
η-conversion	`λx.M x` $\to_\eta$ `M`, x not free in M	a function that only passes its argument on is that function
δ-step	`c` $\equiv_\delta$ `body`	replaces a rewriting rule by its definition once it is applied to an argument. The row before shows the definition

The steps are taken in normal order (leftmost-outermost), which finds the normal form whenever there is one. A rule behaves exactly like its body, so free variables in a body are never captured either. In $(\lambda x.\lambda y.x)\,y$, the argument y would end up under λy, so λy becomes $\lambda y'$ first:

```
\lcReduce{(\\x.\\y.\x) \y}
```

$$\begin{aligned} & (\lambda x.\lambda y.x)\ y \\ =_\alpha\ & (\lambda x.\lambda y'.x)\ y \\ \to_\beta\ & \lambda y'.y \end{aligned}$$

```
\lcReduce{\true \a \b}
```

$$\begin{aligned} & \underset{\lambda x\,y.x}{\texttt{true}}\ a\ b \\ \equiv_\delta\ & (\lambda x.\lambda y.x)\ a\ b \\ \to_\beta\ & (\lambda y.a)\ b \\ \to_\beta\ & a \end{aligned}$$

5.2 Semantics

Terms are

$$M, N \ ::= \ x \ \mid \ \lambda x.M \ \mid \ M\,N \ \mid \ (M) \ \mid \ c$$

with variables x, parentheses that only group, and constants c. A constant is a rewriting rule or any other macro. Its unfolding $\rho(c)$ is the body of the rule, for `\num{<n>}` the Church numeral, for a rule with a term argument A its body with A substituted, and $\rho(c) = \bot$ for a macro that is no rule.

The free variables $\mathcal{F}(M)$ of a term are those that no binder of M encloses. A constant stands for its unfolding, so it contributes the free variables of that:

$$\mathcal{F}(x) = \{x\} \qquad \mathcal{F}(\lambda x.M) = \mathcal{F}(M) \setminus \{x\}$$

$$\mathcal{F}(M\,N) = \mathcal{F}(M) \cup \mathcal{F}(N) \qquad \mathcal{F}(c) = \begin{cases} \mathcal{F}(\rho(c)) & \rho(c) \neq \bot \\ \varnothing & \text{otherwise} \end{cases}$$

Substitution $M[x := N]$ is the usual one, with two details: it stops at a binder λx, and it enters a constant only for $x \in \mathcal{F}(c)$, where $c[x := N] = \rho(c)[x := N]$. Before it starts, $\mathrm{ren}_{x,N}(M)$ renames every binder λy with $y \in \mathcal{F}(N)$ that has a free x in its body to the first of $y', y'', \ldots$ that does not occur in the term. That renaming is the α-step, after which the substitution captures nothing.

A single step $\mathrm{step}(M) = (k, M')$ with $k \in \{\alpha, \beta, \eta, \delta\}$ takes the first case that applies, which is normal order. It is undefined if no case applies:

$$
\begin{aligned}
\mathrm{step}(\lambda x.M\ x) &= (\eta, M) && \text{if } x \notin \mathcal{F}(M)\\
\mathrm{step}(\lambda x.M) &= (k, \lambda x.M') && \text{if } \mathrm{step}(M) = (k, M')\\
\mathrm{step}((\lambda x.M)\ N) &= \begin{cases} (\alpha, (\lambda x.\mathrm{ren}_{x,N}(M))\ N) & \mathrm{ren}_{x,N}(M) \neq M \\ (\beta, M[x := N]) & \text{otherwise} \end{cases}\\
\mathrm{step}(c\ N) &= (\delta, \rho(c)\ N) && \text{if } \rho(c) \neq \bot\\
\mathrm{step}(M\ N) &= (k, M'\ N) && \text{if } \mathrm{step}(M) = (k, M')\\
\mathrm{step}(M\ N) &= (k, M\ N') && \text{if } \mathrm{step}(N) = (k, N')\\
\mathrm{step}(c) &= (\delta, \rho(c)) && \text{if } c \text{ takes an argument, } \mathrm{step}(\rho(c)) \text{ defined}
\end{aligned}
$$

The last case unfolds a rule with an argument when its body can still be reduced. Other rules stay folded until they are applied. A rule whose unfolding contains a substituted variable is replaced by its unfolding without a step of its own. With `eta=false` or `delta=false` the respective cases are left out.

Concrete semantics. A trace $\tau = M_0 \xrightarrow{k_1} M_1 \cdots \xrightarrow{k_j} M_j$ records the steps, with $\mathrm{last}(\tau) = M_j$ and $|\tau| = j$. For $n =$ `max-steps`, the trace semantics of a term is the least fixpoint

$$[\![M]\!] = \mathrm{lfp}\, F_M, \qquad F_M(X) = \{M\} \cup \{\, \tau \xrightarrow{k} M' \mid \tau \in X,\ |\tau| < n,\ \mathrm{step}(\mathrm{last}(\tau)) = (k, M') \,\}$$

on sets of traces ordered by inclusion. As step is a function, $[\![M]\!]$ is the chain of prefixes of one trace. Its longest element τ_M is what the package computes, step by step, as the Kleene iteration $F_M^i(\varnothing)$.

Abstractions. Every view is an abstraction $[\![M]\!]^\sharp = \alpha_\sharp([\![M]\!])$ of that trace:

- $[\![M]\!]^{\mathrm{nf}} = \alpha_{\mathrm{nf}}(\tau_M)$ with $\alpha_{\mathrm{nf}}(\tau) = \mathrm{last}(\tau)$ if $\mathrm{step}(\mathrm{last}(\tau))$ is undefined and $\bot$ otherwise, typeset as $\mathrm{last}(\tau)$ followed by $\cdots$. `\lcNormalForm` shows it, `\iflcNormalFormReached` tells whether it is not $\bot$.
- $[\![M]\!]^{\mathrm{rows}} = \alpha_{\mathrm{rows}}(\tau_M)$ keeps M_0, M_j and every M_i that `hide`, `skip`, `compact`, `from` and `to` do not remove, and merges the steps in between. `\lcReduce` shows it.
- The relation of a merged segment is the join of its step kinds in the lattice $k \sqsubseteq k^* \sqsubseteq *$ for each kind k, with $k \sqcup k = k^*$ and $k \sqcup k' = *$ for $k \neq k'$. Hidden kinds are left out, and the join of nothing but hidden kinds is taken over all of them. These are the symbols $\to_\beta$, $\to_\beta^*$ and $\to^*$.
- $[\![M]\!]^{\mathrm{fold}}$ abstracts the subterms of $[\![M]\!]^{\mathrm{nf}}$ by the rules of `fold`. A rule c stands for $\gamma(c) = \{M \mid M =_\alpha \rho(c)\}$, and α_{fold} replaces a subterm M by the first listed c with $M \in \gamma(c)$, for `num` by `\num{<n>}` if M is α-equivalent to the Church numeral of n.
- With `tooltips`, a merged segment also carries its concretization, the skipped terms $M_{i+1}, \ldots, M_{l-1}$, as hover text.

5.3 Commands

Command	Typesets
`\lcReduce[`⟨*keys*⟩`]{`⟨*term*⟩`}`	every step
`\lcNormalForm[`⟨*keys*⟩`]{`⟨*term*⟩`}`	only the result
`\lcComputeReduction[`⟨*keys*⟩`]{`⟨*term*⟩`}`	nothing
`\lcReduceSetup{`⟨*keys*⟩`}`	nothing, changes the defaults

LaTeX needs two runs to place the arrows and ranges (three with a table of contents). Within a run, identical reductions (e.g. on the slides of a beamer frame) are computed only once; `\lcForgetMemo` throws that memory away, which only a benchmark needs. With `layout=lines`, a reduction starts a new paragraph and breaks across pages. Inside a box or table cell it falls back to `layout=inline`. For long reductions combine `compact` with `hide=delta`, for wide terms use `format=\small` or `from` and `to`. After each reduction:

Macro	Content
`\lcLastNormalForm`	the result as a string (λ written as `\`, e.g. `\y'.y`)
`\lcLastReductionSequence`	the kinds of the steps, e.g. `alpha,beta,eta`
`\lcLastStepCount`	their number
`\lcLastShownSteps`	for `\lcReduce`: each shown term as its step number, after the kind of the relation leading to it (`beta`, `beta*` for several, `*` for mixed), e.g. `0,beta* 2`
`\iflcNormalFormReached`	whether a normal form was reached

```
\lcComputeReduction{(\\x.\\y.\x) \y}
\texttt{\lcLastNormalForm}:
   \lcLastReductionSequence{}
   (\lcLastStepCount{} steps)
```

\y'.y: alpha,beta (2 steps)

`\lcIfAlphaEquivalent{`⟨*term*⟩`}{`⟨*term*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}` compares two terms up to α-conversion, where rules count as their bodies. `\lcCanonicalForm`⟨*cs*⟩`{`⟨*term*⟩`}` stores a term in the string form of `\lcLastNormalForm`.

```
\lcIfAlphaEquivalent{\\x.\x}{\\y.\y}
↪{equal}{different}
```

equal

`\lcDeBruijn[`⟨*keys*⟩`]{`⟨*term*⟩`}` typesets a term with de Bruijn indices, where a binder loses its name and a variable becomes the number of binders between it and its own, counted from 1. A free name stays as it is, a constant is replaced by its unfolding, and `\lcDeBruijnForm`⟨*cs*⟩`{`⟨*term*⟩`}` stores the same as text. Two terms are α-equivalent exactly if these forms are equal, which is how `\lcIfAlphaEquivalent` compares them.

```
\lcDeBruijn{(\\x.\\y.\x \y) \z}

\lcDeBruijn[style=plain-math]{\true}
```

(λλ(2 1) z)
$\lambda\lambda 2$

5.4 Options

The examples below use `\true` and `\false` of the `bool` module and `\num`, `\add` and `\succ` of `nat`, both loaded by default.

eta, **delta**: enable η-conversions and δ-steps (default `true`).

```
\lcNormalForm{\\x.\f \x}\\
\lcNormalForm[eta=false]{\\x.\f \x}
```

f
λx.f x

```
\lcNormalForm{\true \a \b}\\
\lcNormalForm[delta=false]{\true \a \b}
```

a
true a b

max-steps: stop after this many steps (default `50`). An unfinished reduction ends with $\cdots$. Recursive functions like `\length` need several hundred steps.

```
\lcReduce[max-steps=2]
   {(\\x.\x \x) (\\x.\x \x)}
```

(λx.x x) (λx.x x)
$\to_\beta$ (λx.x x) (λx.x x)
$\to_\beta$ (λx.x x) (λx.x x)
$\cdots$

fold: rules (by the name of their macro, with or without backslash, `num` for numerals, or `list` for the list notation of the `list` module, which also reads `[a, b]` as input) to write the result with. Every subterm that is α-equivalent to one of them is replaced, and the first match wins, so `fold={false,num}` keeps $\lambda x.\lambda y.y$ as `\false` instead of `0`.

```
\lcNormalForm{\add \num{2} \num{3}} \quad
\lcNormalForm[fold=num]{\add \num{2} \num{3}}
```

λf.λx.f (f (f (f (f x)))) *5*

```
\lcNormalForm[fold={list, num}, max-steps=300]
   {\cons \num{1} (\cons \num{2} \nil)} \quad
\lcNormalForm[fold=list, max-steps=300]
   {\append (\cons \a \nil) (\cons \b \nil)}
```

[*1* , *2*] [a, b]

layout: `lines` (a table, default) or `inline`.

```
\lcReduce[layout=inline]{(\\x.\\y.\x) \a \b}
```

(λx.λy.x) a b $\to_\beta$ (λy.a) b $\to_\beta$ a

hide: kinds of steps whose resulting terms are not shown.

```
\lcReduce[hide=alpha]{(\\x.\\y.\x) \y \a}
```

(λx.λy.x) y a
$\to_\beta$ (λy'.y) a
$\to_\beta$ y

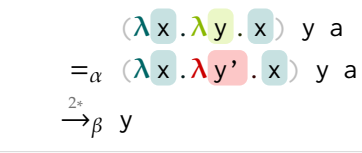
skip: numbers or ranges (e.g. `skip={1,4-6}`) of the steps whose resulting terms are not shown.

```
\lcReduce[skip={1, 2}]{(\\x.\\y.\x) \y \a}
```

(λx.λy.x) y a
$\overset{3*}{\to}$ y

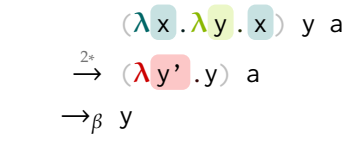
compact: show only the last of consecutive steps of the same kind. Hidden steps do not interrupt such a run. The relation of a row only names the kinds of steps that are not hidden.

```
\lcReduce[compact]{(\\x.\\y.\x) \y \a}
```

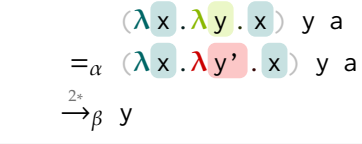


from, **to**: do not show the terms before/after that step. The start term and the result are always shown.

```
\lcReduce[from=2]{(\\x.\\y.\x) \y \a}
```



```
\lcReduce[to=1]{(\\x.\\y.\x) \y \a}
```



alpha-symbol, **beta-symbol**, **eta-symbol**, **delta-symbol**: the arrow of each kind, as math. Several steps of one kind get a star.

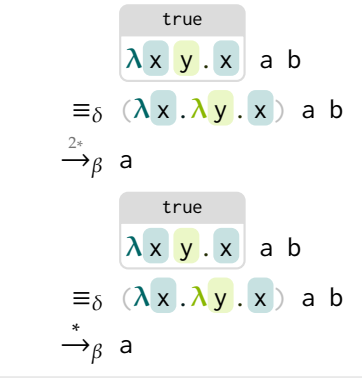
```
\lcReduce[beta-symbol=\rightarrow]
   {(\\x.\x) \y}
```



step-count: write the number of merged steps next to the arrow instead of a star (default `true`). It counts only steps of that arrow's kind, so a kind `hide` removes does not count. **step-mark** (`\lcSetStepMark`) sets how that number looks, with `#1` and `\lcSteps` for the number, which **multi-symbol** can use as well; `\lcStepMark` is the mark itself, the star when **step-count** is off.

```
\lcReduce[compact]{\true \a \b}

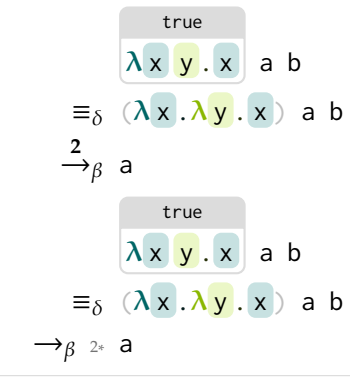
\lcReduce[compact, step-count=false]{\true \a \b}
```



step-mark-place: above the arrow (default) or behind it, which keeps the row as low as a row without a mark.

```
\lcReduce[compact,
   step-mark={\textbf{#1}}]{\true \a \b}

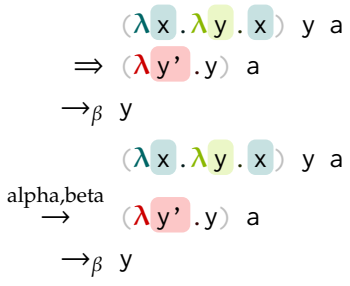
\lcReduce[compact,
   step-mark-place=behind]{\true \a \b}
```



multi-symbol: relation for several steps of different kinds (default $\rightarrow^*$). The kinds of the merged steps are `#1`, so they can be named.

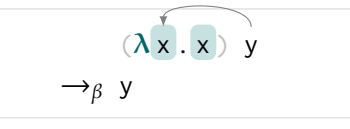
```
\lcReduce[multi-symbol=\Rightarrow,
↪skip=1]{(\\x.\\y.\x) \y \a}

\lcReduce[multi-symbol={\overset{\text{#1}}{\to}},
↪skip=1]{(\\x.\\y.\x) \y \a}
```

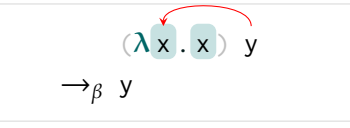


arrows, **arrow-style**: draw the β-arrows (default `false`) and their TikZ options. An arrow points from the argument to the binder it is substituted for. Every variable needs a pair of marker nodes for that, which `arrows` switches on, so a reduction without them is the faster one.

```
\lcReduce[arrows]{(\\x.\x) \y}
```




```
\lcReduce[arrows, arrow-style={red, -stealth}]
   {(\\x.\x) \y}
```



marks: give every variable those marker nodes without drawing arrows, so `\reducemarks` can point at them.

underline: underline what the next step reduces. It needs no nodes and no colors, so it also works with `style=plain` and in an `export`. `\lcSetRedexLine{`⟨*code*⟩`}` sets how that line looks, `\lcRedexLine{`⟨*term*⟩`}` draws one by hand.

```
\lcReduce[underline]{(\\x.\\y.\x) \y \a}
```

$$\begin{aligned} & (\lambda x.\lambda y.x)\ y\ a \\ =_\alpha\ & \underline{(\lambda x.\lambda y'.x)\ y\ a} \\ \to_\beta\ & \underline{(\lambda y'.y)\ a} \\ \to_\beta\ & y \end{aligned}$$

```
\lcReduce[underline, style=plain]{\true \a \b}
```

$$\begin{aligned} & \overbrace{\lambda x\ y.x}^{\texttt{true}}\ a\ b \\ \equiv_\delta\ & \underline{(\lambda x.\lambda y.x)\ a\ b} \\ \to_\beta\ & \underline{(\lambda y.a)\ b} \\ \to_\beta\ & a \end{aligned}$$

rule-depth: how deeply rules are expanded when typeset (default `0`, only the rule of the next δ-step is shown expanded).

```
\lcNormalForm{\snd}\\
\lcNormalForm[rule-depth=1]{\snd}
```

$$\begin{aligned} & \mathit{snd} \\ & \overbrace{\lambda p.p\ \mathit{false}}^{\texttt{snd}} \end{aligned}$$

color-start: start of the color cycle.

```
\lcReduce[color-start=3]{(\\x.\x) \y}
```

$$\begin{aligned} & (\lambda x.x)\ y \\ \to_\beta\ & y \end{aligned}$$

plain: typeset without colors

```
\lcReduce[plain]{(\\x.\\y.\x) \y \a}
```

$$\begin{aligned} & (\lambda x.\lambda y.x)\ y\ a \\ =_\alpha\ & (\lambda x.\lambda y'.x)\ y\ a \\ \to_\beta\ & (\lambda y'.y)\ a \\ \to_\beta\ & y \end{aligned}$$

show-term: `\lcNormalForm` puts the term and $\to^*$ before the result.

```
\lcNormalForm[show-term]{(\\x.\\y.\x) \y \a}
```

$$(\lambda x.\lambda y.x)\ y\ a \xrightarrow{3*} y$$

format: code to run before typesetting.

```
\lcReduce[format=\small, lambda-ranges=false]
  {(\\x.\\y.\x) \a \b}
```

$$\begin{aligned} & (\lambda x.\lambda y.x)\ a\ b \\ \to_\beta\ & (\lambda y.a)\ b \\ \to_\beta\ & a \end{aligned}$$

tooltips: hover over a merged arrow to read the terms it skips, as a PDF annotation (pdfTEX or LuaTEX). `\lcNormalForm` puts all steps into one tooltip on the result.

```
\lcNormalForm[tooltips, fold=num]
   {\add \num{1} \num{1}}
```

2

```
\lcReduce[tooltips, compact, hide=delta]
   {\true \a \b}
```

true
λx y . x a b
$\xrightarrow{2*}_{\beta}$ a

alpha-index, **beta-index**, **eta-index**, **delta-index**: the subscript of an arrow, by default the Greek letter of its kind, as in $\rightarrow_{\beta}$.

```
\lcReduce[beta-symbol=\Rightarrow,
↪beta-index=b]{(\\x.\x) \y}
```

(λx . x) y
$\Rightarrow_{b}$ y

tooltip-author: the author a tooltip annotation carries (default `lc-visualizer`).

alpha-text, **beta-text**, **eta-text**, **delta-text**, **lambda-text**: the relations and the λ in tooltips (PDF strings, e.g. `\lcUnichar{955}` for λ).

```
\lcNormalForm[tooltips, beta-text=beta,
↪lambda-text=\textbackslash]
   {(\\x.\\y.\x \y) \f \a}
```

f a

symbol-align: where the relation sits in its column, `right` (default), `left` or `center`.

```
\lcReduce[symbol-align=left, skip=1]
   {(\\x.\\y.\x) \y \a}
```

(λx . λy . x) y a
$\xrightarrow{2*}$ (λy' . y) a
$\rightarrow_{\beta}$ y

symbol-width, **row-sep**: width of the relation column (default `2.5em`) and space between rows and around the reduction (default: 70 % of the line distance) with `layout=lines`.

```
\lcReduce[symbol-width=1.5em, row-sep=5pt]
   {(\\x.\\y.\x) \a \b}
```

(λx . λy . x) a b
$\rightarrow_{\beta}$ (λy . a) b
$\rightarrow_{\beta}$ a

beamer, **beamer-start**, **row-hook**: see subsection 5.5. **cache**: see subsection 5.6.

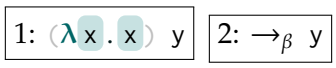
5.5 Animations

Every shown term is a row, numbered from 1. In beamer, with `beamer=uncover`, row n appears on slide n. With `beamer=only`, it is shown only on slide n (with `layout=inline` the terms stay in place). `beamer-start=<k>` moves the first row to slide k.

```
\begin{frame}
  \lcReduce[beamer=uncover]{(\\f.\\x.\f \x) \x}
\end{frame}
```

`row-hook` is code with the row number as `#1` and the row itself as `#2`. With `layout=lines` it is called for the relation and for the term. It replaces a `beamer` setting and vice versa. Set it with `\lcReduceSetup` outside of a beamer frame, as frames do not allow `#` in their body.

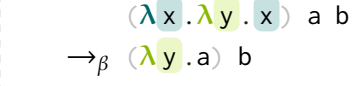
```
\lcReduceSetup{row-hook=\fbox{#1: #2}}
\lcReduce[layout=inline]{(\\x.\x) \y}
```



5.6 Caching and generated code

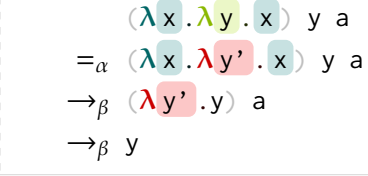
With `cache=<name>`, `\lcReduce` and `\lcNormalForm` write the code they typeset to `<name>.tex` (`cache` alone names it after the document and the hash, `\jobname-lc-<hash>.tex`). Its first line holds a hash of the term, the options and the rules. While it matches, later compilations input the file instead of reducing again. `\lcReduceSetup{cache}` caches every reduction. The file also restores the `\lcLast...` macros. It is LaTeX code built from `\RLC` and internal macros of the package. A file that does not start with the hash line is never overwritten (this raises an error), reductions with errors are not cached, and the directory of `<name>` must exist. The cache saves the time of the reduction, not of typesetting it.

```
\lcReduce[cache=reduce-k]{(\\x.\\y.\x) \a \b}
```



`export`=⟨name⟩ writes the same reduction to ⟨*name*⟩`.tex` as an `array` of `\RLC` rows, without the internals of the cache. That file is meant to be read and changed: copy it into the document and edit the terms, the relations or the arrows. It uses only macros of the package: `\lcStepState{`⟨*steps*⟩`}{`⟨*mark*⟩`}` hands the next arrow its number and its mark, and the arrow itself is `\lcStepSymbol{`⟨*relation*⟩`}{`⟨*kind*⟩`}{`⟨*mark*⟩`}`, or `\lcMultiSymbol{`⟨*kinds*⟩`}` when it merges several kinds. It is written on every run and never read back, and it carries no arrows, as those are pictures that a math `array` cannot hold.

```
\lcReduce[export=\jobname-export]
   {(\\x.\\y.\x) \y \a}
```



That reduction wrote this file, which typesets the same rows:

```
% lc-visualizer 3 steps
\lcSetup{color-by-name}
\[\begin{array}{@{}r@{~}l@{}}
  & \LC{(\\x.\\y.\x) \y \a}\\
{\lcStepState{1}{\textcolor{black!55}{\scriptscriptstyle 1{*}}}\lcStepSymbol{=}{\alpha}{}} & \LC{(\\x.\\{y'}.\x) \y \a}\\
{\lcStepState{1}{\textcolor{black!55}{\scriptscriptstyle 1{*}}}\lcStepSymbol{\to}{\beta}{}} & \LC{(\\{y'}.\y) \a}\\
{\lcStepState{1}{\textcolor{black!55}{\scriptscriptstyle 1{*}}}\lcStepSymbol{\to}{\beta}{}} & \LC{\y}\\
```

```
\end{array}\]
```

A The reduction in the source

Every listing in this appendix comes out of `lc-visualizer.sty` itself. The steps carry the names of Section 5.2.

A parsed term is a triple: a letter for the kind and two groups whose meaning follows from it.

Triple	Letter	Term
`{v}{<name>}{}`	`v`, variable	x
`{l}{<name>}{<body>}`	`l`, lambda	$\lambda x.M$
`{a}{<fun>}{<arg>}`	`a`, application	$M\,N$
`{c}{<tokens>}{<unfolding>}`	`c`, constant	c, unfolding $\rho(c)$
`{B}{<abstraction>}{<arg>}`	`B`, marked β-redex	$(\lambda x.M)\,N$, the next step
`{D}{<tokens>}{<unfolding>}`	`D`, marked δ-redex	c, the next step

The last two letters live only between a step and its row and carry the part that the arrow and `underline` mark. Every walk over a term branches on the letter, so a term printer is four macros, one per letter; `\__lc_tp_l:nnn` prints an abstraction.

One step. `\__lc_step_aux:nnn` is step. It clears `\g__lc_kind_str`, and a case that finds nothing leaves it empty, which is how the next case learns that it has to try.

```
\str_case:nn {#1}
  {
    {l} { \__lc_step_l:nn {#2} {#3} }
    {a} { \__lc_step_a:nn {#2} {#3} }
    {c} { \bool_if:NT \l__lc_delta_bool { \__lc_try_unfold:nn {#2} {#3} } }
  }
```

Under a binder, η comes before a step in the body.

```
\bool_if:NT \l__lc_eta_bool { \__lc_try_eta:nnnn {#1} #2 }
\str_if_empty:NT \g__lc_kind_str { \__lc_step_in:nnn {#2} { {l} {#1} } { } }
```

η needs the body to be an application of a variable that the body does not use otherwise, which is $x \notin \mathcal{F}(M)$.

```
\str_if_eq:nnT {#2} {a}
  {
    \__lc_if_var:nnT {#1} {#4}
      {
        \__lc_if_free:nnTF {#1} {#3} { }
          {
            \str_gset:Nn \g__lc_kind_str { eta }
            \tl_gset:Nn \g__lc_new_tl {#3}
            \tl_gset:Nn \g__lc_marked_tl { {l} {#1} { {a} {#3} {#4} } }
          }
      }
  }
```

An application first tries itself, as a β- or a δ-redex, then the function, then the argument. That order is what makes the whole thing normal order.

```
    \str_case:en { \__lc_kind:n {#1} }
      {
        {l} { \__lc_beta:nnnn #1 {#2} }
        {c} { \bool_if:NT \l__lc_delta_bool { \__lc_delta:nnnn #1 {#2} } }
      }
    \str_if_empty:NT \g__lc_kind_str { \__lc_step_in:nnn {#1} { {a} } { {#2} } }
    \str_if_empty:NT \g__lc_kind_str { \__lc_step_in:nnn {#2} { {a} {#1} } { } }
```

β renames first. `\__lc_rename:nn` is $\mathrm{ren}_{x,N}$ and reports through `\g__lc_renamed_bool` whether it had to touch anything: if it did, that renaming is the step, and the substitution waits for the next one.

```
    \__lc_prepare_rename:n {#4}
    \__lc_rename:nn {#2} {#3}
    \bool_if:NTF \g__lc_renamed_bool
      {
        \str_gset:Nn \g__lc_kind_str { alpha }
        \tl_gset:Nx \g__lc_new_tl
          { {a} { {l} { \exp_not:n {#2} } { \exp_not:V \g__lc_ret_tl } } { \exp_not:n {#4} } }
        \tl_gset:Nn \g__lc_marked_tl { {a} { {l} {#2} {#3} } {#4} }
      }
      {
        \str_gset:Nn \g__lc_kind_str { beta }
        \tl_gset:Nx \g__lc_new_tl { \__lc_subst:nnn {#2} {#4} {#3} }
        \tl_gset:Nn \g__lc_marked_tl { {B} { {l} {#2} {#3} } {#4} }
      }
```

The substitution itself is one macro per letter again. A binder of the same name ends it, a constant is entered only when the name occurs free in its unfolding.

```
\cs_new:Npn \__lc_subst_v:nnnn #1#2#3#4 % a variable is replaced if the names match
  { \str_if_eq:nnTF {#1} {#3} { \exp_not:n {#2} } { \exp_not:n { {v} {#3} {} } } }
\cs_new:Npn \__lc_subst_c:nnnn #1#2#3#4 % substitute into the unfolding of a constant
  {
    \tl_if_empty:nTF {#4} { \use_ii:nn } { \__lc_if_free:nnTF {#1} {#4} }
      { \__lc_subst:nnn {#1} {#2} {#4} } { \exp_not:n { {c} {#3} {#4} } }
  }
\cs_new:Npn \__lc_subst_a:nnnn #1#2#3#4 % substitute in both parts of an application
  { {a} { \__lc_subst:nnn {#1} {#2} {#3} } { \__lc_subst:nnn {#1} {#2} {#4} } }
\cs_new:Npn \__lc_subst_l:nnnn #1#2#3#4 % a binder of the same name stops the substitution
  {
    \str_if_eq:nnTF {#1} {#3}
      { \exp_not:n { {l} {#3} {#4} } }
      { {l} { \exp_not:n {#3} } { \__lc_subst:nnn {#1} {#2} {#4} } }
  }
```

δ replaces a constant that stands in function position by its unfolding, and keeps the constant in the `{D}` triple so the row can still show it.

```
    \tl_if_empty:nF {#3}
      {
        \str_gset:Nn \g__lc_kind_str { delta }
        \tl_gset:Nn \g__lc_new_tl { {a} {#3} {#4} }
        \tl_gset:Nn \g__lc_marked_tl { {a} { {D} {#2} {#3} } {#4} }
      }
```

The whole reduction. `\__lc_reduce:n` parses the term and applies step until it is undefined or `max-steps` is reached. Every term it passes is stored, which is the trace τ_M.

```
\__lc_parse:nN {#1} \l__lc_term_tl
\tl_gset_eq:NN \g__lc_start_tl \l__lc_term_tl
\bool_gset_false:N \g__lc_nf_bool
\int_zero:N \l__lc_count_int
\bool_set_true:N \l__lc_loop_bool
\bool_while_do:Nn \l__lc_loop_bool
  {
    \bool_gset_false:N \g__lc_avoid_bool
    \__lc_step:V \l__lc_term_tl
```

The views read that trace: `\__lc_typeset_reduction:` keeps the rows that `hide`, `skip`, `compact`, `from` and `to` leave and merges the steps in between, `\__lc_typeset_symbol:n` turns the kinds of one merged segment into its relation, `\__lc_typeset_normal_form:` shows the last term alone.

B Benchmarking

Reducing is cheap, typesetting is what costs, and every TikZ picture adds to it. `bench/run.sh` writes the table from the cases in `bench/cases.txt`. `bench/measure.tex` takes the times inside the document with the timer of pdfTEX, so the start of LATEX and the shipout of the pages are no part of a number. Every case is measured five times, in a pdfTEX run of its own, and `\lcForgetMemo` drops what the last of those runs remembered, so that none of them is answered from the memory of the run before:

- **Reduce** is `\lcComputeReduction[max-steps=<n>]`, which reduces the term and prints nothing.
- **Typeset** is `\lcReduce` with the same `max-steps` and the default keys, so it reduces *and* shows every row. **Min** and **Max** are its fastest and its slowest of the five runs.
- **Cached** is the same `\lcReduce` with `cache`, the file already written, so the rows are read instead of computed.

Term	Rows	Reduce	Typeset	Min	Max	Cached
no redex	1	0.3 ms	0.7 ms	0.7 ms	0.8 ms	0.6 ms
four steps	5	0.9 ms	11.7 ms	10.6 ms	15.1 ms	9.8 ms
factorial of 2	60	246.4 ms	1 070.5 ms	1 017.4 ms	1 129.0 ms	651.4 ms
factorial of 2, arrows	60	241.0 ms	13 437.4 ms	12 294.4 ms	13 938.4 ms	14 265.6 ms
factorial of 2, compact	60	240.8 ms	579.5 ms	523.8 ms	634.4 ms	252.7 ms
factorial of 2, no nodes	60	241.3 ms	1 047.3 ms	955.6 ms	1 126.1 ms	644.4 ms
factorial of 3, cut off	100	336.5 ms	1 285.8 ms	1 187.9 ms	1 401.6 ms	735.5 ms

Index of commands and keys

alpha-index (key), 16
alpha-symbol (key), 13
alpha-text (key), 16
arrow-from (key), 3
arrow-style (key), 14
arrows (key), 14

beamer (key), 16
beamer-start (key), 16
beta-index (key), 16
beta-symbol (key), 13
beta-text (key), 16
binder-gap (key), 4
box (key), 3, 5
bracket-color (key), 3, 5

cache (key), 16
color-by-name (key), 3
color-mixin (key), 3, 4
color-start (key), 3, 15
colors (key), 3
compact (key), 13

delta (key), 12
delta-index (key), 16
delta-symbol (key), 13
delta-text (key), 16
duplicate-mixin (key), 3, 4

eta (key), 12
eta-index (key), 16
eta-symbol (key), 13
eta-text (key), 16
export (key), 17

fold (key), 12
font (key), 3, 4
format (key), 15
from (key), 13

glow (key), 3, 4

hide (key), 12

inner-strut (key), 3, 5

lambda (key), 3, 4
lambda-ranges (key), 3
lambda-text (key), 16
layout (key), 12
\lcBelowRewriteRule, 5
\lcCanonicalForm, 11
\lcComputeReduction, 10
\lcCreateNewArgumentRewritingRule, 6
\lcCreateNewRewritingRule, 5
\lcDeBruijn, 11
\lcDeBruijnForm, 11
\lcEnableChar, 2
\lcForgetMemo, 11, 20
\lcIfAlphaEquivalent, 11
\lcLoadModule, 6
\lcMultiSymbol, 17
\lcNewStyle, 3
\lcNormalForm, 10
\lcRawArgumentRule, 6
\lcRedexLine, 15
\lcReduce, 10
\lcReduceSetup, 3, 10, 17
\lcSetBinderGap, 4
\lcSetRedexLine, 15
\lcSetStepMark, 13
\lcSetup, 2, 3
\lcStepMark, 13
\lcSteps, 13
\lcStepState, 17
\lcStepSymbol, 17

marks (key), 14
max-steps (key), 12
multi-symbol (key), 13, 14

name-strut (key), 3, 5
node-background (key), 3, 4
node-style (key), 3, 5

paren-depth (key), 3
plain (key), 3, 15

\reducemarks, 5, 14
row-hook (key), 16
row-sep (key), 16
rule-depth (key), 3, 15

shade-same (key), 3, 4
show-term (key), 15
single-lambdas (key), 3, 4
skip (key), 12
step-count (key), 13
step-mark (key), 13
step-mark-place (key), 14
strut-line (key), 3, 5

style (key), 3
symbol-align (key), 16
symbol-width (key), 16

text-mixin (key), 3, 4
textcolor (key), 3, 4
to (key), 13
tooltip-author (key), 16
tooltips (key), 16

underline (key), 15