# A Proof of the Church-Rosser Theorem and its Representation in a Logical Framework

Frank Pfenning
September 1992
CMU-CS-92-186

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

We give a detailed, informal proof of the Church-Rosser property for the untyped $\lambda$-calculus and show its representation in LF. The proof is due to Tait and Martin-Löf and is based on the notion of parallel reduction. The representation employs higher-order abstract syntax and the judgments-as-types principle and takes advantage of term reconstruction as it is provided in the Elf implementation of LF. Proofs of meta-theorems are represented as higher-level judgments which relate sequences of reductions and conversions.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

**Keywords:** Lambda calculus, Church-Rosser property, logical framework.

# Contents



# 1 Introduction

The logical framework LF [HHP] has been designed as a formal meta-language for the representation of deductive systems. It is based on a predicative type theory with dependent types in which *judgments* are represented as types and *deductions* are represented as objects. In this report we explore the use of this framework for the formalization of the theory of the untyped $\lambda$-calculus. More specifically, we will develop a proof and representation of the Church-Rosser theorem under $\beta$-reduction. This report will focus on techniques of representation—details of the LF type theory and its implementation in Elf can be obtained from [HHP, Pfe91b]. Elf is a logic programming language based on the LF type theory, although in this report we deemphasize the operational aspects of Elf. All the Elf code in this report has been type-checked and executed in the current implementation [Pfe91a].[1] If the Elf implementation of the proof is ignored, this report can also be read as a detailed, informal proof of the Church-Rosser theorem using the method of parallel reductions due to Tait and Martin-Löf.

The methodology for the representation of meta-theorems (such as the Church-Rosser theorem) can be seen as consisting of three stages. The first stage is the formalization of the abstract syntax of the language under consideration. Here we use the idea of *higher-order abstract syntax* which requires that variables of the object language are represented by variables of the meta-language.

[1]The code in this report and the implementation are available via anonymous ftp. Please send electronic mail to the author at `fp@cs.cmu.edu` for further information.

This allows common conventions in the proofs of meta-theorems which concern bound variables to be supported directly in the meta-language. In particular, we can avoid explicit renaming of bound variables (which is modeled by $\alpha$-conversion in the framework) and have a notation for capture-avoiding substitution (which is modeled by $\beta$-reduction). It may appear that the framework is specifically designed just for the implementation of the $\lambda$-calculus, but in fact bound variables occur in most programming languages and the technique of higher-order abstract syntax has wide applicability in theorem proving and logic programming [Fel89, NM88, Pau86], and the theory of programming languages [Han91, HP92, MP91].

The second stage is the formalization of the semantics of the language which is given via judgments defined by inference rules. The judgments are implemented as types and deductions as objects. Thus the relationship between a deduction and the judgment it establishes is represented as the relationship between an object and its type. In our example, we will represent various reduction and conversion relations in this style. Similar techniques have been used to specify type systems, operational semantics, compilation and other aspects of the semantics of programming languages (see, for example, [Han91, HP92, Har90, MP91]).

The third stage is the formalization of the proofs of meta-theorems in the framework. The construction which is implicit in the proof is represented as a judgment which relates deductions. For example, in the proof of the Church-Rosser theorem we have to show the existence of certain reduction sequences, given other reduction sequences. This is done via an explicit construction which can be represented as a judgment. Verifying that this higher-level judgment indeed represents a proof is left to a process called *schema-checking* (see [PR92, HP92]) which is currently mostly done by hand, since the implementation is still incomplete. This means that there is still the possibility of error in the implementation of the proof.

Thus all three stages, representation of abstract syntax, semantics, and meta-theory, are carried out within the same logical framework. The concrete implementation of framework within the Elf programming language has other features which we will mostly ignore for the purposes of this discussion, but we briefly review Elf here. Its concrete syntax is very simple, since we only have to model the relatively few constructs of LF. While LF is stratified into the levels of kinds, families, and objects, the syntax is overloaded in that, for example, the symbol $\Pi$ constructs dependent function types and dependent kinds. Similarly, juxtaposition is concrete syntax for instantiation of a type family and application of objects. We maintain this overloading in the concrete syntax for Elf and refer to expressions from any of the three levels collectively as *terms*. A signature is given as a sequence of *declarations*.

| | | | | |
|---|---|---|---|---|
| Terms | *term* | ::= | \| *id* | $a$ or $c$ or $x$ |
| | | | \| `{`*id* `:` *term*$_1$`}`*term*$_2$ | $\Pi x{:}A_1.\ A_2$ or $\Pi x{:}A.\ K$ |
| | | | \| `[`*id* `:` *term*$_1$`]` *term*$_2$ | $\lambda x{:}A.\ M$ |
| | | | \| *term*$_1$ *term*$_2$ | $A\ M$ or $M_1\ M_2$ |
| | | | \| `type` | Type |
| | | | \| *term*$_1$ `->` *term*$_2$ | $A_1 \rightarrow A_2$ |
| | | | \| *term*$_1$ `<-` *term*$_2$ | $A_2 \rightarrow A_1$ |
| | | | \| `{`*id*`}`*term* \| `[`*id*`]`*term* \| `_` | omitted terms |
| | | | \| *term*$_1$`:`*term*$_2$ | cast |
| | | | \| `(`*term*`)` | grouping |
| Declarations | *decl* | ::= | *id* `:` *term* `.` | $a{:}K$ or $c{:}A$ |

The terminal *id* stands either for a bound variable, a free variable, or a constant at the level of families or objects. Bound variables and constants in Elf can be arbitrary identifiers, but free variables in a declaration or query must begin with an uppercase letter (an undeclared, unbound lowercase identifier is flagged as an undeclared constant). An uppercase identifier is one which begins with an underscore `_` or a letter in the range `A` through `Z`; all others are considered lowercase, including numerals. Identifiers may contain all characters except `(){}[]:.%` and whitespace. In particular, `A->B` would be a single identifier, while `A -> B` denotes a function type. The left-pointing arrow as in `B <- A` is a syntactic variant and parsed into the same representation as `A -> B`. It improves the readability of some Elf programs. The simple function type `A -> B` is treated as an abbreviation for `{x:A} B` where `x` does not occur in `B`.

The right-pointing arrow `->` is right associative, while the left-pointing arrow `<-` is left associative. Juxtaposition binds tighter than the arrows and is left associative. The scope of quantifications {$x$ : $A$} and abstractions [$x$ : $A$] extends to the next closing parenthesis, bracket, brace or to the end of the term. Term reconstruction fills in the omitted types in quantifications {$x$} and abstractions [$x$] and omitted types or objects indicated by an underscore `_`. In case of essential ambiguity a warning or error message results. Declarations may contain free variables which can be interpreted *schematically*, just as typical inference rules are schematic. This means that a declaration with free variables can intuitively be thought of as representing all its instances. Such declarations are translated into LF by adding (implicit) $\Pi$-quantifiers for all free variables. The corresponding (implicit) arguments are reconstructed by the Elf front end employing a variant of higher-order unification. This and other aspects of Elf are explained in more detail in [Pfe91b], but we hope that the material in the remainder of this report can be understood at a pragmatic level without detailed knowledge about the term reconstruction algorithm.

Single-line comments begin with `%` and extend through the end of the line. A delimited comment begins with `%{` and ends with the matching `}%`, that is, delimited comments may be properly nested. The parser for Elf also supports infix, prefix, and postfix declarations similar to the ones available in Prolog, and we will see some examples of infix declarations later.

## 2 The Untyped $\lambda$-Calculus

We consider the pure untyped $\lambda$-calculus whose syntax is given by

$$\text{Terms} \quad M \quad ::= \quad x \mid M_1\ M_2 \mid \lambda x.\ M.$$

Here $x$ stands for variables. We will use $M$ and $N$ as meta-variables ranging over terms. A term of the form $\lambda x.\ M$ binds the variable $x$ and the rule of $\alpha$-conversion allows the explicit renaming of bound variables. We use the convention that $\alpha$-conversions can be performed implicitly, or, as Barendregt [Bar80] puts it: "*Terms that are $\alpha$-congruent are identified.*" Conventions of this kind are common right from the beginning of the study of the $\lambda$-calculus (see, for example, the original paper with a proof of the Church-Rosser theorem [CR36]). In order to avoid any possible problems which arise from this convention, a common route is to go to combinatory calculi [CF58] or to use de Bruijn indices [dB72]. It is interesting to note that de Bruijn's motivation for his notation for $\lambda$-terms came from a proof of the Church-Rosser theorem, and Shankar's mechanization of the Church-Rosser theorem in the Boyer-Moore theorem prover [Sha88, BM79] uses de Bruijn indices. In LF, the detour via de Bruijn indices is not necessary, since variable naming conventions can be supported directly in the framework.

We use parentheses to disambiguate the concrete syntax of terms. In our presentation, application associates to the left, and the scope of $\lambda$-abstraction extends to the next closing parenthesis or the end of the expression. For example $(\lambda x.\ \lambda y.\ x\ y\ y)\ z$ would be $(\lambda x.\ (\lambda y.\ ((x\ y)\ y)))\ z$ with all explicit parentheses. For further background material on the untyped $\lambda$-calculus, the reader may consult Barendregt's comprehensive book [Bar80].

The representation of the syntax of the untyped $\lambda$-calculus is an archetypical use of *higher-order abstract syntax*. Variables of the object language (the $\lambda$-calculus, in this example) are represented by variables in the meta-language. For such a representation to be correct, variables bound in the object language must also be bound in the meta-language. We define $\ulcorner M \urcorner$, the representation of the term $M$ in Elf, inductively on the structure of $M$. Recall that `[x:A] P` is Elf's concrete syntax for abstraction in the framework and binds a variable `x` of type `A` in the object `P`.

$$\begin{array}{rcl} \ulcorner x \urcorner & = & \mathtt{x} \\ \ulcorner M\ N \urcorner & = & \mathtt{app}\ \ulcorner M \urcorner\ \ulcorner N \urcorner \\ \ulcorner \lambda x.\, M \urcorner & = & \mathtt{lam}\ (\mathtt{[x{:}term]}\ \ulcorner M \urcorner) \end{array}$$

For example,

$$\ulcorner \lambda x.\, \lambda y.\, x \urcorner = \mathtt{lam\ [x{:}term]\ lam\ [y{:}term]\ x}.$$

As far as we know, this representation is due to Wadsworth [Wad76] and used by Meyer [Mey82] in the construction of an environment model of the untyped $\lambda$-calculus. The notation used there is $\Psi$ for `lam` and $\Phi$ for `app`. From the representation above we can read off the type of the constructors, leading to the following signature $T$.

```
term : type.  %name term M

lam : (term -> term) -> term.
app : term -> term -> term.
```

The annotation `%name term M` instructs Elf to use `M`, `M1`, *etc.* as names for new variables of type `term` which may be introduced during search or term reconstruction.

Our notation for the result of substituting $N$ for $x$ in $M$ is $[N/x]M$. We require that no free variable in $N$ is bound in $M$ in order to avoid variable capture. This means that $M$ may have to be renamed into an equivalent form before substitution can be carried out.

The representation function $\ulcorner \cdot \urcorner$ is a bijection between terms in the untyped $\lambda$-calculus and canonical objects in the LF type theory of type `term`. Furthermore, the function is *compositional*, that is, substitution commutes with representation. Formally,

$$\ulcorner [N/x]M \urcorner = [\ulcorner N \urcorner / x] \ulcorner M \urcorner.$$

Note that substitution on the right-hand side is substitution within the LF type theory. We further observe that

$$[\ulcorner N \urcorner / x] \ulcorner M \urcorner \equiv (\mathtt{[x{:}term]}\ \ulcorner M \urcorner)\ \ulcorner N \urcorner$$

which can be paraphrased by saying that substitution at the object-level (the untyped $\lambda$-calculus) is implemented by $\beta$-reduction at the meta-level (the LF type theory). Here, $\equiv$ stands for definitional equality in the framework which includes $\beta$-conversion.

## 3 Reduction and Conversion

The operational semantics of the untyped $\lambda$-calculus is usually given via a reduction relation, where the meaning of a term is its normal form, that is, a term which cannot be reduced further. But is this legitimate? Unless we can show that such a normal form is essentially unique, the semantics would be ambiguous. In this section we will formulate some reduction relations for the untyped $\lambda$-calculus and then investigate their properties in Section 5.

At the heart of the reduction relation lies the rule of *$\beta$-reduction*, whereby a term $(\lambda x.\ M)\ N$ is reduced to $[N/x]M$. Recall that substitution may require renaming of bound variables in $M$ in order to avoid variable capture. This reduction may be applied anywhere inside a term—something which is not true, for example, for evaluation relations for programming languages (both in call-by-name and call-by-value semantics, see [Plo75]). One may consider this as a distinguishing characteristic of general reduction compared to evaluation.

Thus the first judgment we would like to define is $M \longrightarrow M'$ (read: $M$ reduces to $M'$). This judgment is defined by a set of inference rules. These rules are subscripted by "1" in order to indicate that this is the first formulation we are considering. In the course of the proof of the Church-Rosser theorem we will need to consider other reduction relations.

$$\frac{}{(\lambda x.\ M_1)\ M_2 \longrightarrow [M_2/x]M_1}\ \mathsf{beta}_1$$

$$\frac{M \longrightarrow M'}{\lambda x.\ M \longrightarrow \lambda x.\ M'}\ \mathsf{lm}_1$$

$$\frac{M_1 \longrightarrow M_1'}{M_1\ M_2 \longrightarrow M_1'\ M_2}\ \mathsf{apl}_1$$

$$\frac{M_2 \longrightarrow M_2'}{M_1\ M_2 \longrightarrow M_1\ M_2'}\ \mathsf{apr}_1$$

The first rule $\mathsf{beta}_1$ is the $\beta$-reduction rule proper. The other three allow us to perform the $\beta$-reduction anywhere inside a term. These rules are frequently refered to as *congruence rules.* Note that the rule $\mathsf{lm}_1$ is somewhat peculiar, since we require that the bound variable on both sides be named $x$, even though we made the general assumption that the names of bound variables should be irrelevant. Here is a simple example of a deduction.

$$\frac{\dfrac{}{(\lambda x.\ \lambda y.\ x)\ z \longrightarrow \lambda y.\ z}\ \mathsf{beta}_1}{(\lambda x.\ \lambda y.\ x)\ z\ z \longrightarrow (\lambda y.\ z)\ z}\ \mathsf{apl}_1$$

Using the judgments-as-types principle, a deduction is now represented as an object whose type describes the judgment. Thus a type of the form `red` $\ulcorner M \urcorner\ \ulcorner M' \urcorner$ represents the type of all deductions of the judgment $M \longrightarrow M'$. Since $\ulcorner M \urcorner$ and $\ulcorner M' \urcorner$ are of type `term`, the so-called *type family* `red` has *kind* `term -> term -> type`. Actually, instead of using `red` in prefix notation, we use `-->` in infix notation. The `%infix` annotation below has this effect.[2] The `%name` annotation indicates that Elf should use `R`, `R1`, *etc.* as meta-variables ranging over deductions.

```
--> : term -> term -> type.  %infix none 10 -->
                             %name --> R
```

In the first approximation, the representation of an inference rule is a function from deductions of its premisses to a deduction of its conclusion. For example, $\mathsf{beta}_1$, which has no premisses, is represented as a constant `beta1`.

[2]The keyword `none` declares that the operator `-->` is not associative and `10` is its precedence, with higher precedence binding tighter. Keywords `left` and `right` instead of `none` declare left and right associative operators, respectively.

```
beta1 :   (app (lam M1) M2) --> M1 M2.
```

Here, `M1` has type `term -> term` and represents the scope of a $\lambda$-abstraction. Applying this function to `M2`, the representation of the argument, is definitionally equal to the representation of $[M_2/x]M_1$, where $x$ is the variable bound by $\lambda$. We are thus taking advantage of the compositionality of the representation as expressed by

$$\ulcorner[M_2/x]M_1\urcorner = [\ulcorner M_2\urcorner/x]\ulcorner M_1\urcorner \equiv (\texttt{[x:term]}\ \ulcorner M_1\urcorner)\ \ulcorner M_2\urcorner.$$

The declaration above can be understood schematically, just as the inference rule itself: any valid instance of `beta1` is a valid object of the appropriate type. In a more explicit version, `M1` and `M2` could be made explicit arguments to `beta1`, as in the declaration `beta1'` below.

```
beta1' : {M1:term -> term} {M2:term}
            (app (lam M1) M2) --> M1 M2.
```

To continue in the representation, the rule $\mathsf{lm}_1$ introduces an additional complication: the explicit mention of the bound variable $x$. The solution is to introduce a new parameter `x` and substitute it on both sides. A formulation along these lines as an inference rule might be

$$\frac{[x/y]M \longrightarrow [x/y']M'}{\lambda y.\ M \longrightarrow \lambda y'.\ M'}\,\mathsf{lm}_1$$

with the proviso that the parameter $x$ does not already occur in $M$ or $M'$. This can now readily be implemented in Elf, using the same idea as above to represent substitution. This still leaves us to deal with the proviso, which is common in deductive systems. We consider the premiss a judgment *parametric* in $x$, that is, we should be able to substitute any term $N$ for $x$ in the deduction of the premiss to obtain a deduction of $\ulcorner[N/x]M\urcorner \longrightarrow \ulcorner[N/x]M'\urcorner$. Recall that `{x:A} B` (usually written in $\Pi x{:}A.\ B$) is the Elf notation for the type of an LF function which accepts an object $P$ of type $A$ and returns an object of type $[P/x]B$.

```
lm1 :    ({x:term} M x --> M' x)
    ->          (lam M) --> (lam M').
```

The remaing two rules are simpler since they do not involve variable binding.

```
apl1  :               M1 --> M1'
       ->    (app M1 M2) --> (app M1' M2).

apr1  :               M2 --> M2'
       ->    (app M1 M2) --> (app M1 M2').
```

The example deduction above is represented by

```
apl1 beta1 : app (lam [x] lam [y] x) z --> lam [y] z.
```

where `z : term`. A slightly more complicated example:

```
lm1 [x:term] beta1 : (lam [x] (app (lam [y] y) x)) --> lam [x] x.
```

This list of declarations can also be used as a logic program to reduce a given term. A *goal*, usually an atomic formula in Prolog, is given by a type in Elf. Instead of attempting to find a proof of a formula as in Prolog, Elf searches for a closed object of the given type. This search proceeds in a depth-first fashion as in Prolog, considering each inference rule in turn and constructing an appropriate object incrementally. When the signature above is used as a program it will find the leftmost-outermost redex first and reduce it. Upon backtracking, other possible reductions will be enumerated. For example, consider enumerating the (single-step) reductions of $(\lambda x.\, x\; x)\,((\lambda y.\, y)\,(\lambda z.\, z))$.

```
?- R : (app (lam [x] (app x x)) (app (lam [y] y) (lam [z] z))) --> M'.
Solving...

M' =
   app (app (lam ([y:term] y)) (lam ([z:term] z)))
       (app (lam ([y:term] y)) (lam ([z:term] z))).

R = beta1.
;

M' = app (lam ([x:term] app x x)) (lam ([z:term] z)).

R = apr1 beta1.
;
no more solutions
```

Here, `M'` is a free variable (a *logic variable* in the Prolog terminology) which is instantiated by unification during search. The variable `R` will be bound to the resulting deduction. In this example there are two possible single-step reductions, one which reduces the top-level redex, another which reduces the redex in the right-hand side. The corresponding deductions consist of only one or two inferences. The semi-colon in the transcript indicates that the user asked for further solutions.

The next task is to encode multi-step reductions. One usually defines $M \longrightarrow^* M'$ iff there exists a sequence of reductions

$$M = M_0 \longrightarrow M_1 \longrightarrow \cdots \longrightarrow M_n = M'$$

for some $n \geq 0$. While the logical framework does not have an immediate notation for this sort of definition, we can also define it via a very simple deductive system.

$$\frac{}{M \longrightarrow^* M}\,\mathsf{id}_1 \qquad \frac{M \longrightarrow M' \qquad M' \longrightarrow^* M''}{M \longrightarrow^* M''}\,\mathsf{step}_1$$

Reconsider the example above.

$$\frac{\dfrac{\dfrac{}{(\lambda y.\, y)\,(\lambda z.\, z) \longrightarrow \lambda z.\, z}\,\mathsf{beta}_1}{(\lambda x.\, x\; x)\,((\lambda y.\, y)\,(\lambda z.\, z)) \longrightarrow (\lambda x.\, x\; x)\,(\lambda z.\, z)}\,\mathsf{apr}_1 \qquad \begin{array}{c} R^* \\ (\lambda x.\, x\; x)\,(\lambda z.\, z) \longrightarrow^* (\lambda z.\, z)\,(\lambda z.\, z) \end{array}}{(\lambda x.\, x\; x)\,((\lambda y.\, y)\,(\lambda z.\, z)) \longrightarrow^* (\lambda z.\, z)\,(\lambda z.\, z)}\,\mathsf{step}_1$$

where

$$R^* = \frac{\dfrac{}{(\lambda x.\, x\; x)\,(\lambda z.\, z) \longrightarrow (\lambda z.\, z)\,(\lambda z.\, z)}\,\mathsf{beta}_1 \qquad \dfrac{}{(\lambda z.\, z)\,(\lambda z.\, z) \longrightarrow^* (\lambda z.\, z)\,(\lambda z.\, z)}\,\mathsf{id}_1}{(\lambda x.\, x\; x)\,(\lambda z.\, z) \longrightarrow^* (\lambda z.\, z)\,(\lambda z.\, z)}\,\mathsf{step}_1$$

The implementation of the inference rules in Elf is simple, since it does not involve any side-conditions or bound variables.

```
-->* : term -> term -> type.  %infix none 10 -->*
                              %name -->* R*

id1   :    M -->* M.

step1 :    M --> M'
     ->  M' -->* M''
     ->    M -->* M''.
```

The interpretation of this declaration as a program is now much less useful, since execution can easily lead to infinite regression even though solutions may exist. This is because the operational semantics of Elf will solve the subgoals which arise after an application of the `step1` rule in an order which is inconvenient in this example. This illustrates a general phenomenon: in many cases, a straightforward specification of an inference system will not be useful as a program. In order to obtain a program we have to design an *algorithm* and then implement it separately from the specification. A complete strategy for multi-step reduction is a left-most outermost strategy. This reduction strategy can also be implemented and its completeness can be proved in Elf, but we leave this to a future report. Briefly, Elf searches through a signature in a depth-first fashion, trying inference rules from the top to the bottom, solving the innermost subgoal first. For more information on the operational semantics of Elf the reader is referred to [Pfe91b] or [MP91] for a more tutorial presentation. Through sheer luck, however, we can generate the deduction above even with this operationally inadequate signature. It is given as the third and final answer before the program diverges.

```
?- R* : (app (lam [x] (app x x)) (app (lam [y] y) (lam [z] z))) -->* M'.
Solving...

M' = app (lam ([x:term] app x x))
         (app (lam ([y:term] y)) (lam ([z:term] z))).

R* = id1.
;

M' =
   app (app (lam ([y:term] y)) (lam ([z:term] z)))
       (app (lam ([y:term] y)) (lam ([z:term] z))).

R* = step1 beta1 id1.
;

M' = app (lam ([x:term] app x x)) (lam ([z:term] z)).

R* = step1 (apr1 beta1) id1.
;

M' = app (lam ([z:term] z)) (lam ([z:term] z)).
```

```
R* = step1 (apr1 beta1) (step1 beta1 id1).
;

interrupt
```

Finally we come to conversion, a notion of equality generated from (multi-step) reduction. It is the smallest equivalence relation on terms which contains the reduction relation. This can be expressed as an inference system with four rules: the first three for reflexivity, symmetry, and transitivity express that conversion, written as $\longleftrightarrow$, is an equivalence relation. The fourth rule expresses that if one term can be reduced to another, the two should be convertible.

$$\frac{}{M \longleftrightarrow M}\,\mathsf{refl} \qquad\qquad \frac{M \longleftrightarrow M'}{M' \longleftrightarrow M}\,\mathsf{sym}$$

$$\frac{M \longleftrightarrow M' \qquad M' \longleftrightarrow M''}{M \longleftrightarrow M''}\,\mathsf{trans} \qquad\qquad \frac{M \longrightarrow^* M'}{M \longleftrightarrow M'}\,\mathsf{red}$$

The representation in Elf is a direct transcription.

```
<-> : term -> term -> type.  %infix none 10 <->
                             %name <-> C

refl :    M <-> M.

sym  :    M <-> M'
     ->  M' <-> M.

trans:    M <-> M'
     ->  M' <-> M''
     ->   M <-> M''.

red  :    M -->* M'
     ->   M <-> M'.
```

The Church-Rosser theorem [CR36] now states that if $M \longleftrightarrow M'$ then there exists some $N$ such that $M \longrightarrow^* N$ and $M' \longrightarrow^* N$. We are taking the liberty of simply using a judgment $J$ to stand for the meta-language proposition "*J is derivable*" or "*J is evident*". We hope that this will not lead to any confusion on the part of the reader. The Church-Rosser theorem is also described by the following diagram.

$$\begin{tikzcd}
M \arrow[rr, leftrightarrow] \arrow[dr, dotted] & & M' \arrow[dl, dotted] \\
& N &
\end{tikzcd}$$

The solid lines indicate that a certain relation is assumed, the dotted line means that the existence of the relation is asserted. Instead of $*$ we use a double-headed arrow to indicate multi-step reductions. We will usually label the lines with a variable for deductions of the corresponding judgment. The Church-Rosser theorem is then more explicitly described by the following diagram.

$$\begin{tikzcd} M \arrow[rr, leftrightarrow, "C"] \arrow[dr, dotted, "R^*"'] & & M' \arrow[dl, dotted, "R^{*\prime}"] \\ & N & \end{tikzcd}$$

As a warm-up exercise we prove a few lemmas about the multi-step reduction relation and give the representation of these proofs in Elf. First we would like to show that multi-step reduction is transitive. In general we use the notation $D :: J$ to express that $D$ is a deduction of the judgment $J$. In this particular example, $R :: M \longrightarrow M'$ can be read as *$R$ is a reduction from $M$ to $M'$*, and similarly for $R^* :: M \longrightarrow^* M'$. Note that the existence of an explicit notation for deductions gives us an explicit notation for reductions, sequences of reductions, and conversions. We generally use $R$ and $S$ to range over (single-step) reductions, $R^*$ and $S^*$ to range over multi-step reduction, and $C$ to range over conversion. Each of these thus ranges over deductions of particular judgments.

**Lemma 1** (Transitivity of $\longrightarrow^*$) *If $M \longrightarrow^* M'$ and $M' \longrightarrow^* M''$ then $M \longrightarrow^* M''$.*

**Proof:** The proof is by induction over the structure of $R^* :: M \longrightarrow^* M'$. We will provide an explicit description of a method for constructing $S^{*\prime} :: M \longrightarrow^* M''$ given $R^*$ and $S^* :: M' \longrightarrow M''$.

**Case:**

$$R^* = \frac{}{M \longrightarrow^* M}\,\mathsf{id}_1$$

By assumption we have a deduction $S^* :: M' \longrightarrow^* M''$ and $M = M'$. Thus $S^{*\prime} = S^* :: M \longrightarrow^* M''$ is sufficient to prove the lemma in this case.

**Case:**

$$R^* = \frac{\begin{array}{c} R_1 \\ M \longrightarrow M_1 \end{array} \qquad \begin{array}{c} R_2^* \\ M_1 \longrightarrow^* M' \end{array}}{M \longrightarrow^* M'}\,\mathsf{step}_1$$

By the induction hypothesis on $R_2^*$ and $S^*$ there exists a deduction $S_2^{*\prime} :: M_1 \longrightarrow^* M''$. Applying the rule $\mathsf{step}_1$ to $R_1$ and $S_2^{*\prime}$ then yields the desired deduction of $M \longrightarrow^* M''$.

□

We represent the algorithmic content of this proof as a judgment which relates the three deductions involved, $R^* :: M \longrightarrow^* M'$, $S^* :: M' \longrightarrow M''$, and $S^{*\prime} :: M \longrightarrow M''$. This judgment is then encoded in Elf as a type family

```
appd :  M -->* M'  ->  M' -->* M''  ->  M -->* M''  ->  type.
```

such that whenever there exists a closed object of type `appd R* S* S*'` then `S*'` represents the reduction sequence generated by applying the algorithm which is implicit in the proof above to `R*` and `S*`. A moment's reflection reveals that this algorithm does nothing but append the reduction sequences $R^*$ and $S^*$. Note that we use the a left-pointing arrow in notation inspired by logic programming in order to emphasize the computational nature of the rules. Semantically, there is no difference between `A -> B` and `B <- A`.

```
appd_id   : appd id1 S* S*.
appd_step : appd (step1 R1 R2*) S* (step1 R1 S2*')
         <- appd R2* S* S2*'.
```

Term reconstruction (which includes type-checking) of these declarations guarantees that reduction sequences are composed only when this is sensible, that is, the result of one reduction sequence is the starting point of another. However, type-checking does not guarantee is that appd is total in its first two arguments. This is the responsibility of *schema-checking* which, in essence, checks that the judgment is primitive recursive in some argument and must therefore be total. The implementation of schema-checking is currently still incomplete and must be carried out by hand. Automation, that is, the mechanical construction of representations of proofs such as the one above is subject of current research—for now we concentrate merely on the representation of deductions found first by informal reasoning.

We summarize the basic principles. A proof by induction over the structure of a deduction is represented as a higher-level judgment which relates deductions. Each case in the proof by induction corresponds to an inference rule defining the higher-level judgment. An appeal to the induction hypothesis manifests itself in the premiss of such an inference rule. The judgment and inference rules are then translated into Elf using the familiar judgments-as-types principle. The resulting signature can be executed as a logic program to exhibit the computational content of the original, informal proof.

The next lemma shows that multi-step reduction is a congruence. An inference rule is *admissible* if any (ground) instance of the rule is derivable.

**Lemma 2** (Congruence of $\longrightarrow^*$) *The rules* $\mathsf{lm}_1^*$, $\mathsf{apl}_1^*$, *and* $\mathsf{apr}_1^*$ *below are admissible rules of inference.*

$$\frac{M \longrightarrow^* M'}{\lambda x.\, M \longrightarrow^* \lambda x.\, M'}\,\mathsf{lm}_1^* \qquad \frac{M_1 \longrightarrow^* M_1'}{M_1\ M_2 \longrightarrow^* M_1'\ M_2}\,\mathsf{apl}_1^* \qquad \frac{M_2 \longrightarrow^* M_2'}{M_1\ M_2 \longrightarrow^* M_1\ M_2'}\,\mathsf{apr}_1^*$$

**Proof:** The proof in each case is by induction over the structure of the derivation $R^*$ of the premiss. We explicitly construct a deduction $S^*$ of the conclusion. The basic idea is to distribute the uses of the the congruence to all the single-step reductions which make up the multi-step reduction. We show the proof only for the rule $\mathsf{lm}_1^*$. The others are very similar and we directly give the representation of the argument in Elf.

**Case:**

$$R^* = \frac{}{M \longrightarrow^* M}\,\mathsf{id}_1$$

Then $\lambda x.\, M \longrightarrow^* \lambda x.\, M$ also by the identity rule $\mathsf{id}_1$

**Case:**

$$R^* = \frac{\begin{array}{c}R_1\\ M \longrightarrow M_1\end{array} \qquad \begin{array}{c}R_2^*\\ M_1 \longrightarrow^* M'\end{array}}{M \longrightarrow^* M'}\,\mathsf{step}_1$$

From the induction hypothesis on $R_2^*$ we know there exists a deduction $S_2^* :: \lambda x.\ M_1 \longrightarrow^* \lambda x.\ M'$. We thus construct:

$$S^* = \dfrac{\dfrac{\begin{array}{c}R_1\\ M \longrightarrow M_1\end{array}}{\lambda x.\ M \longrightarrow \lambda x.\ M_1}\,\mathsf{lm}_1 \qquad \begin{array}{c}S_2^*\\ \lambda x.\ M_1 \longrightarrow^* \lambda x.\ M'\end{array}}{\lambda x.\ M \longrightarrow \lambda x.\ M'}\,\mathsf{step}_1$$

□

The main difficulty in the representation of these lemmas is the bound variable in the case of the $\lambda$-congruence. As before, we represent the premiss as a function from a term $N$ to a deduction which shows that $[N/x]M \longrightarrow^* [N/x]M'$. This reflects that the premiss is a parametric judgment.

```
lm1* : ({x:term} M x -->* M' x)
     ->      (lam M) -->* (lam M')
     -> type.

lm1*_id   : lm1* ([x:term] id1) id1.
lm1*_step : lm1* ([x:term] step1 (R1 x) (R2* x)) (step1 (lm1 R1) S2*)
              <- lm1* R2* S2*.

apl1* :            M1 -->* M1'
      ->  (app M1 M2) -->* (app M1' M2)
      -> type.

apl1*_id   : apl1* id1 id1.
apl1*_step : apl1* (step1 R1 R2*) (step1 (apl1 R1) S2*)
               <- apl1* R2* S2*.

apr1* :            M2 -->* M2'
      ->  (app M1 M2) -->* (app M1 M2')
      -> type.

apr1*_id   : apr1* id1 id1.
apr1*_step : apr1* (step1 R1 R2*) (step1 (apr1 R1) S2*)
           <- apr1* R2* S2*.
```

## 4 Parallel Reduction and Conversion

The main tool in this proof of the Church-Rosser theorem is the notion of parallel reduction, usually referred to as the Tait/Martin-Löf method (see [Bar80]). We write $M \Longrightarrow M'$ for $M$ *reduces in parallel to* $M'$. Parallel reduction is useful, since it will satisfy the so-called *diamond property* which is depicted in the following diagram.

$$
\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
M' \arrow[dr, dotted] & & M'' \arrow[dl, dotted] \\
 & N &
\end{tikzcd}
$$

A similar diagram holds for ordinary[3] *multi-step* reduction $\longrightarrow^*$, but not for the ordinary single-step reduction $\longrightarrow$. The idea behind parallel reduction is that, besides contracting a redex, we can also reduce the terms involved in the redex at the same time. Furthermore, the congruence rule for application is generalized so we can perform reduction in both branches in parallel. A possible reduction may or may not be peformed, which means that in the extreme we should allow $M \Longrightarrow M$. In a slight departure from previously published proofs we assume this for variables only. This simplifies some of the case analyses later on, but does not have a deep impact on the structure of the proofs.

$$\frac{M_1 \Longrightarrow M_1' \qquad M_2 \Longrightarrow M_2'}{(\lambda x.\, M_1)\, M_2 \Longrightarrow [M_2'/x]M_1'}\,\mathsf{beta}$$

$$\frac{M_1 \Longrightarrow M_1' \qquad M_2 \Longrightarrow M_2'}{M_1\, M_2 \Longrightarrow M_1'\, M_2'}\,\mathsf{ap}$$

$$\frac{M \Longrightarrow M'}{\lambda x.\, M \Longrightarrow \lambda x.\, M'}\,\mathsf{lm}$$

$$\frac{}{x \Longrightarrow x}\,\mathsf{var}$$

Thus parallel reduction can take bigger steps than ordinary reduction. One has to keep in mind, however, that the ordinary definition of a normal form (a term $M$ such that there does not exist an $M'$ such that $M \longrightarrow M'$) must be modified for parallel reduction, since every term reduces to itself. Under parallel reduction a term is in normal form if it only reduces to itself. As an example for parallel reduction, we reconsider an earlier term.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x \Longrightarrow x}\,\mathsf{var} \qquad \dfrac{}{x \Longrightarrow x}\,\mathsf{var}}{x\; x \Longrightarrow x\; x}\,\mathsf{ap}}{(\lambda x.\, x\; x) \Longrightarrow (\lambda x.\, x\; x)}\,\mathsf{lm} \qquad \dfrac{\dfrac{\dfrac{}{y \Longrightarrow y}\,\mathsf{var}}{\lambda y.\, y \Longrightarrow \lambda y.\, y}\,\mathsf{lm} \qquad \dfrac{\dfrac{}{z \Longrightarrow z}\,\mathsf{var}}{\lambda z.\, z \Longrightarrow \lambda z.\, z}\,\mathsf{lm}}{((\lambda y.\, y)\, (\lambda z.\, z)) \Longrightarrow (\lambda z.\, z)}\,\mathsf{beta}}{(\lambda x.\, x\; x)\; ((\lambda y.\, y)\, (\lambda z.\, z)) \Longrightarrow (\lambda z.\, z)\, (\lambda z.\, z)}\,\mathsf{beta}$$

The representation of parallel reduction is again as a type family, indexed by two objects.

[3] In order to distinguish reduction as introduced in the previous section we will often refer to it as *ordinary* reduction. In the diagrams we will not explicitly distinguish between parallel and ordinary reduction, but it should be clear from the context which form of reduction is depicted.

```
=> : term -> term -> type.  %infix none 10 =>
                            %name => R
```

The first problem one encounters when considering the representation of the inference rules is the rule for variables. Recall that variables of the untyped $\lambda$-calculus are represented by meta-variables and that we thus do not have explicit constructors for them we could match against. This is a frequent problem when dealing with higher-order abstract syntax. The solution is generally to extend the judgment we are defining by hypotheses. That is, while deriving $M_1 \Longrightarrow M_1'$ we are allowed to use the hypothesis $x \Longrightarrow x$. The following formulation of the rule comes closer to the Elf implementation.

$$
\dfrac{\begin{array}{c}\overline{x \Longrightarrow x}^{\,u} \\ R \\ M_1 \Longrightarrow M_1'\end{array} \qquad M_2 \Longrightarrow M_2'}{(\lambda x.\, M_1)\, M_2 \Longrightarrow [M_2'/x]M_1'}\,\mathsf{beta}^u
$$

The label $u$ on the inference rule $\mathsf{beta}^u$ indicates that the assumptions labelled $u$ are discharged at this inference and not available elsewhere in the deduction. This is the essence of the notion of hypothetical judgment (see, for example, [ML80]). We represent the deduction $R$ of the (hypothetical) judgment in the left premiss as a function whose first argument is a term $x$ and whose second argument is a deduction $u$ of $x \Longrightarrow x$. Applying this function to a term $N$ and a deduction $S :: N \Longrightarrow N$ yields a deduction of $[N/x]M \Longrightarrow [N/x]M'$. This deduction is obtained by substituting $N$ for $x$ in $R$ and then substituting the deduction $S :: N \Longrightarrow N$ at each place the hypothesis $x \Longrightarrow x$ labelled $u$ is used in $R$.

```
beta : ({x:term} x => x -> M1 x => M1' x)
     ->                        M2 => M2'
     ->        (app (lam M1) M2) => M1' M2'.
```

We use the same technique in the $\mathsf{lm}$ rule: we need to assume the appropriate $\mathsf{var}$ reduction wherever a variable is introduced.

```
lm   : ({x:term} x => x -> M x => M' x)
     ->                   lam M => lam M'.
```

The rule for application does not require a hypothetical judgment.

```
ap   :             M1 => M1'
     ->            M2 => M2'
     ->  (app M1 M2) => (app M1' M2').
```

These three rules complete the signature for parallel reduction. The deduction above can be generated by the Elf interpreter, which is complete for (single-step) parallel reduction.

```
?- R : (app (lam [x] (app x x)) (app (lam [y] y) (lam [z] z))) => M'.
Solving...
M' = app (lam ([x:term] x)) (lam ([x:term] x)).
R = beta ([x:term] [R:x => x] ap R R)
      (beta ([x:term] [R:x => x] R) (lm ([x:term] [R:x => x] R))).
```

As this example demonstrates, parallel reduction is not yet sufficient to reduce terms to normal form, since a parallel reduction can introduce new redices. We generate sequences of parallel reductions as before.

$$\frac{}{M \Longrightarrow^* M}\,\mathsf{id} \qquad \frac{M \Longrightarrow M' \qquad M' \Longrightarrow^* M''}{M \Longrightarrow^* M''}\,\mathsf{step}$$

We represent the step rule by an infix semi-colon to simplify writing down and reading sequences of parallel reductions.

```
=>* : term -> term -> type.  %infix none 10 =>*
                             %name =>* R*

id   :           M =>* M.

;    :           M => M'
     ->         M' =>* M''
     ->          M =>* M''.  %infix right 10 ;
```

Once again, this is insufficient as a program to enumerate parallel reduction sequences in a complete fashion. Using these declaration, we can check that

```
(beta ([x:term] [R:x => x] ap R R)
      (beta ([x:term] [R:x => x] R) (lm ([x:term] [R:x => x] R)))
 ; beta ([x] [R] R) (lm [x] [R] R)
 ; id) : M =>* M'
```

and obtain the answer

```
M = app (lam ([x:term] app x x))
        (app (lam ([x:term] x)) (lam ([x:term] x))),
M' = lam ([x:term] x).
```

That is, we can reach a normal form in two parallel reduction steps.

Finally, we define a notion of parallel conversion. This can be defined as the congruence closure of parallel reduction, but we will define it in a slightly different way to illustrate alternatives. The judgment is written as $M \Longleftrightarrow M'$.

$$\frac{M \Longrightarrow^* M'}{M \Longleftrightarrow M'}\,\mathsf{reduce} \qquad \frac{M \Longrightarrow^* M'}{M' \Longleftrightarrow M}\,\mathsf{expand}$$

$$\frac{M \Longleftrightarrow M' \qquad M' \Longleftrightarrow M''}{M \Longleftrightarrow M''}\,\mathsf{comp}$$

In the Elf implementation we use `;;` as an infix notation for composition.

```
<=> : term -> term -> type.  %infix none 10 <=>
                             %name <=> C

reduce :  M  =>* M'
```

```
       -> M  <=> M'.

expand :  M  =>* M'
       -> M' <=> M.

;;     :  M  <=> M'
       -> M' <=> M''
       -> M  <=> M''.  %infix none 8 ;;
```

Again, as a simple lemma we prove an earlier remark, namely that every term reduces to itself under parallel reduction.

**Lemma 3** (Reflexivity of $\Longrightarrow$) *For any term $M$, $M \Longrightarrow M$.*

**Proof:** The proof is by induction on the structure of $M$.

**Case:** $M = x$. In this case we apply the var rule.

**Case:** $M = \lambda x.\ M_1$. By induction hypothesis there exists an $R_1 :: M_1 \Longrightarrow M_1$. Applying the lm rule to $R_1$ yields a deduction of $\lambda x.\ M_1 \Longrightarrow \lambda x.\ M_1$.

**Case:** $M = M_1\ M_2$. By induction hypothesis on $M_1$ and $M_2$ there are deductions $R_1 :: M_1 \Longrightarrow M_1$ and $R_2 :: M_2 \Longrightarrow M_2$. Application of the ap rule yields the desired conclusion.

□

In the implementation, as in a previous example, there will be no uniform case for variables. Instead, the appropriate reduction rule is assumed whenever a parameter is introduced. For stylistic reasons, we make $M$ explicit as an argument, since it is the induction variable.

```
identity : {M:term} M => M  ->  type.

id_lam : identity (lam M1) (lm R1)
       <- {x:term} {eqx: x => x}
             identity x eqx -> identity (M1 x) (R1 x eqx).

id_app : identity (app M1 M2) (ap R1 R2)
       <- identity M1 R1
       <- identity M2 R2.
```

A second lemma shows that multi-step parallel reduction is transitive.

**Lemma 4** (Transitivity of $\Longrightarrow^*$) *The following is an admissible rule of inference.*

$$\frac{M \Longrightarrow^* M' \qquad M' \Longrightarrow^* M''}{M \Longrightarrow^* M''}\ \mathsf{append}$$

**Proof:** By induction on the structure of the reduction $R^* :: M \Longrightarrow^* M'$. In each case we assume a deduction $S^* :: M' \Longrightarrow^* M''$ and construct a deduction $S^{*\prime} :: M \Longrightarrow M''$. The proof is implemented as a family

```
append : M =>* M'  ->  M' =>* M''  ->  M =>* M''  -> type.
```

**Case:** $R^*$ is the identity. Then $M' = M$ and we can let $S^{*\prime} = S^*$.

```
append_id   : append id S* S*.
```

**Case:** $R^*$ ends in a reduction step, that is,

$$R^* = \cfrac{\begin{array}{c}R_1\\ M \Longrightarrow M_1\end{array} \qquad \begin{array}{c}R_2^*\\ M_1 \Longrightarrow^* M'\end{array}}{M \Longrightarrow^* M'}\,\textsf{step}$$

Then we apply the induction hypothesis to $R_2^*$ and $S^*$ to obtain a deduction $S_2^{*\prime} :: M_1 \Longrightarrow^* M''$. We add the step $R_1$ to the beginning of $S_2^{*\prime}$ to obtain $S^{*\prime}$. In Elf:

```
append_step : append (R1 ; R2*) S* (R1 ; S2*')
               <- append R2* S* S2*'.
```

Recall that the infix semi-colon is our notation for the rule `step`.

□

## 5 The Proof of the Church-Rosser Theorem

The proof of the Church-Rosser Theorem proceeds via a sequence of lemmas. The first important property is the substitution lemma, which is crucial in the later proof of the diamond property. In fact, it is the substitution lemma which motivates the notion of parallel reduction. We make the reductions explicit in the formulation of the lemma to simplify the correspondence to the implementation. Another mechanical verification of the Church-Rosser theorem was carried out by Shankar [Sha88] using the Boyer-Moore theorem prover [BM79]. Shankar's proof used de Bruijn's representation for term of the $\lambda$-calculus [dB72]; here we try a perhaps more direct route using the idea of higher-order abstract syntax. We hope that this provides a good basis for comparison of representation and proof techniques in different systems.

**Lemma 5** (Substitution Lemma) *If $R :: M \Longrightarrow M'$ and $S :: N \Longrightarrow N'$ then there exists an $R' :: [N/x]M \Longrightarrow [N'/x]M'$.*

We will intersperse the implementation of the proof with the proof itself. First note that $R$ above is (implicitly) a parametric and hypothetical judgment: it contains the free variable $x$ and may appeal to the hypothesis that $x \Longrightarrow x$. Putting this together with the idea that substitution is representated via $\beta$-reduction at the meta-level (recall compositionality: $\ulcorner[N/x]M\urcorner = [\ulcorner N\urcorner/x]\ulcorner M\urcorner$) yields the declaration

```
subst : ({x:term} x => x -> M x => M' x)
       -> N => N'
       -> M N => M' N'
       -> type.
```

**Proof:** (of the Substitution Lemma) The proof is by induction on the structure of $R$.

**Case:**

$$R = \frac{}{x \Longrightarrow x}\,\textsf{var}.$$

In this case, where $M = x$, we have to show that there exists a derivation $R'$ of $[N/x]x \Longrightarrow [N'/x]x$. But $[N/x]x = N$ and $[N'/x]x = N'$ so we can let $R'$ be $S$.

In Elf, this case manifests itself as an appeal to the hypothesis `idx : x => x` which is an explicit parameter in the first argument to `subst`.

```
subst_idx : subst ([x:term] [idx: x => x] idx) S S.
```

**Case:**

$$R = \frac{}{y \Longrightarrow y}\,\textsf{var}$$

and $y \neq x$. In this case $[N/x]y = y = [N'/x]y$ and we can let $R' = R$.

This case is represented as an assumption about the behavior of `subst` on the hypothesis that `y => y`, wherever such a hypothesis is introduced. This is necessary in the case of a $\beta$-reduction and a $\lambda$-congruence, that is, for the rules **beta** and **lm**.

**Case:** The last inference is a $\beta$-reduction, that is,

$$R = \frac{\begin{array}{c}R_1\\ M_1 \Longrightarrow M_1'\end{array} \qquad \begin{array}{c}R_2\\ M_2 \Longrightarrow M_2'\end{array}}{(\lambda x.\, M_1)\; M_2 \Longrightarrow [M_2'/x]M_1'}\,\textsf{beta}.$$

In this case we apply the induction hypothesis to $R_1$ to obtain a deduction $R_1' :: [N/x]M_1 \Longrightarrow [N'/x]M_1'$ and to $R_2$ to obtain a deduction $R_2' :: [N/x]M_2 \Longrightarrow [N'/x]M_2'$. Combining these with the **beta** rule yields a deduction

$$R' :: ([N/x]M_1)\;([N/x]M_2) \Longrightarrow ([N'/x]M_1')\;([N'/x]M_2').$$

Using the equation $([N/x]M_1)\;([N/x]M_2) = [N/x](M_1\; M_2)$ from the definition of substitution and a similar equation for the right-hand side reveals that $R'$ is a deduction of the required judgment.

Note how in the realization of this case in Elf, we make the appropriate assumption about the behavior of `subst` on the hypothesis that `y` reduces to `y`.

```
subst_beta : subst ([x:term] [idx: x => x] beta (R1 x idx) (R2 x idx))
                   S (beta R1' R2')
           <- ({y:term} {idy: y => y}
                 subst ([x:term] [idx: x => x] idy) S idy
                 -> subst ([x:term] [idx: x => x] R1 x idx y idy)
                       S (R1' y idy))
           <- subst R2 S R2'.
```

Also note that both premisses are again hypothetical judgments, that is, they also may contain `x` free and may use the rule `x => x`.

**Case:** $R$ ends in the congruence for application, that is,

$$R = \frac{\begin{array}{c} R_1 \\ M_1 \Longrightarrow M_1' \end{array} \qquad \begin{array}{c} R_2 \\ M_2 \Longrightarrow M_2' \end{array}}{M_1\ M_2 \Longrightarrow M_1'\ M_2'}\ \mathsf{ap}$$

In this case we simply apply the induction hypothesis to $R_1$ and $R_2$ and combine the resulting deductions $R_1'$ and $R_2'$ with the `ap` rule.

```
subst_ap  : subst ([x:term] [idx: x => x] ap (R1 x idx) (R2 x idx))
                  S (ap R1' R2')
           <- subst R1 S R1'
           <- subst R2 S R2'.
```

**Case:** $R$ ends in the congruence for $\lambda$.

$$R = \frac{\begin{array}{c} R_1 \\ M_1 \Longrightarrow M_1' \end{array}}{\lambda x.\ M \Longrightarrow \lambda x.\ M'}\ \mathsf{lm}$$

This case is similar to the case for `beta`: we apply the induction hypothesis to $R_1$ to obtain an $R_1'$ and then use the `lm` rule to obtain the desired conclusion. In this case we need to know that $[N/x](\lambda y.\ M) = \lambda y.\ [N/x]M$ which is valid by the implicit assumption that $y$ is distinct from $x$ and different from all variables free in $N$.

Just as in the `subst_beta` rule, we need to make an assumption about the behavior of `subst` on the hypothesis that `y => y` according to the case for variables $y \neq x$ given above.

```
subst_lm  : subst ([x:term] [idx: x => x] lm (R1 x idx))
                  S (lm R1')
           <- ({y:term} {idy: y => y}
                 subst ([x:term] [idx: x => x] idy) S idy
                 -> subst ([x:term] [idx: x => x] R1 x idx y idy)
                            S (R1' y idy)).
```

□

This completes the proof of the substitution lemma. The next important property is the so-called *diamond lemma* which, in this case, concerns single-step parallel reduction.

**Theorem 6** *If $R' :: M \Longrightarrow M'$ and $R'' :: M \Longrightarrow M''$ then there exists an $N$ and reductions $S' :: M' \Longrightarrow N$ and $S'' :: M'' \Longrightarrow N$. In the form of a picture:*

$$\begin{tikzcd} & M \arrow[dl, "R'"'] \arrow[dr, "R''"] & \\ M' \arrow[dr, dotted, "S'"'] & & M'' \arrow[dl, dotted, "S''"] \\ & N & \end{tikzcd}$$

**Proof:** The proof is by simultaneous induction on the structure of $R'$ and $R''$. It is implemented as a type family

```
dia : M => M'  ->  M => M''  ->  M' => N  ->  M'' => N  -> type.
```

such that there will be an object of type `dia R' R'' S' S''` whenever the construction in the proof yields $S'$ and $S''$ from $R'$ and $R''$. In this proof we will informally apply inference rules to deductions of the premisses to indicate the shape of a given reduction. We also heavily use *inversion* in this proof. Inversion in this context means that, given the form of a conclusion, we examine all available inference rules and eliminate those from consideration which could not produce a conclusion of the given form. For example, if the conclusion has the form $M_1\ M_2 \Longrightarrow N$ for some $M_1$, $M_2$, and $N$, we know that the last inference must either be beta or ap, but it could not be lm or var. Using inversion it is easy to see that the cases we consider below are exhaustive.

**Case:**

$$R' = \frac{}{x \Longrightarrow x}\,\mathsf{var}.$$

Since $M' = M = x$, we know by inversion that also $M'' = x$ and $R'' = R' = \mathsf{var}$. Hence we can let $N = x$ and complete the diagram.

$$\begin{tikzcd}
 & x \arrow[dl, "\mathsf{var} = R'"'] \arrow[dr, "R'' = \mathsf{var}"] & \\
x \arrow[dr, dotted, "\mathsf{var} = S'"'] & & x \arrow[dl, dotted, "S'' = \mathsf{var}"] \\
 & x &
\end{tikzcd}$$

As usual, this case will not be represented explicitly in the Elf program, but folded into the cases where parameters are introduced.

**Case:**

$$R'' = \frac{}{x \Longrightarrow x}\,\mathsf{var}.$$

This is same as the previous case, since by inversion, $R' = R''$ in this case.

**Case:** Both $R'$ and $R''$ end in an application of (parallel) $\beta$-reduction.

$$R' = \frac{\begin{array}{cc} R'_1 & R'_2 \\ M_1 \Longrightarrow M'_1 & M_2 \Longrightarrow M'_2 \end{array}}{(\lambda x.\ M_1)\ M_2 \Longrightarrow [M'_2/x]M'_1}\,\mathsf{beta}$$

and

$$R'' = \frac{\begin{array}{cc} R''_1 & R''_2 \\ M_1 \Longrightarrow M''_1 & M_2 \Longrightarrow M''_2 \end{array}}{(\lambda x.\ M_1)\ M_2 \Longrightarrow [M''_2/x]M''_1}\,\mathsf{beta}.$$

Note that this case is not trivial, since $M' = [M_2'/x]M_1'$ may be different from $M'' = [M_2''/x]M_1''$. By two applications of the induction hypothesis we obtain the following diagrams.

$$
\begin{tikzcd}
& M_1 \arrow[dl, "R_1'"'] \arrow[dr, "R_1''"] & \\
M_1' \arrow[dr, dotted, "S_1'"'] & & M_1'' \arrow[dl, dotted, "S_1''"] \\
& N_1 &
\end{tikzcd}
\qquad
\begin{tikzcd}
& M_2 \arrow[dl, "R_2'"'] \arrow[dr, "R_2''"] & \\
M_2' \arrow[dr, dotted, "S_2'"'] & & M_2'' \arrow[dl, dotted, "S_2''"] \\
& N_2 &
\end{tikzcd}
$$

Now the substitution lemma on $S_1'$ and $S_2'$ yields an $S' :: [M_2'/x]M_1' \Longrightarrow [N_2/x]N_1$. Similarly, the substitution lemma on $S_1''$ and $S_2''$ yields an $S'' :: [M_2''/x]M_1'' \Longrightarrow [N_2/x]N_1$ and we can fill in the diagram:

$$
\begin{tikzcd}
& (\lambda x.\, M_1)\; M_2 \arrow[dl, "\mathsf{beta}(R_1', R_2') = R'"'] \arrow[dr, "R'' = \mathsf{beta}(R_1'', R_2'')"] & \\
{[M_2'/x]M_1'} \arrow[dr, dotted, "S'"'] & & {[M_2''/x]M_1''} \arrow[dl, dotted, "S''"] \\
& {[N_2/x]N_1} &
\end{tikzcd}
$$

The implementation of this case is complicated, since we need to make the assumption that `x` reduces to itself, and how `dia` behaves on this assumed reduction. This assumption incorporates the case for variables above.

```
dia_bb : dia (beta R1' R2') (beta R1'' R2'') S' S''
      <- ({x:term} {idx: x => x}
            dia idx idx idx idx
            -> dia (R1' x idx) (R1'' x idx)
                   (S1' x idx) (S1'' x idx))
      <- dia R2' R2'' S2' S2''
      <- subst S1' S2' S'
      <- subst S1'' S2'' S''.
```

Note that one would get a type-checking error if the various reductions did not share a source or target as required by the diagrams, including the check on the substitution conditions.

**Case:** The reduction $R'$ is a $\beta$-reduction and $R''$ is an application of the congruence rule `ap`. Then

$$
R' = \quad \frac{\begin{matrix} R_1' \\ M_1 \Longrightarrow M_1' \end{matrix} \qquad\qquad \begin{matrix} R_2' \\ M_2 \Longrightarrow M_2' \end{matrix}}{(\lambda x.\, M_1)\; M_2 \Longrightarrow [M_2'/x]M_1'}\,\mathsf{beta}
$$

and

$$R'' = \dfrac{\dfrac{\hat{R}_1}{(\lambda x.\, M_1) \Longrightarrow \hat{M}_1} \qquad \dfrac{R''_2}{M_2 \Longrightarrow M''_2}}{(\lambda x.\, M_1)\; M_2 \Longrightarrow \hat{M}_1\; M''_2}\,\mathsf{ap}.$$

By inversion, we see that $\hat{R}_1$ must end in an application of the $\lambda$-congruence rule $\mathsf{lm}$, since this is the only rule which reduces a term of the form $\lambda x.\, M_1$. Thus $\hat{M}_1 = (\lambda x.\, M''_1)$ and

$$R'' = \dfrac{\dfrac{\dfrac{R''_1}{M_1 \Longrightarrow M''_1}}{(\lambda x.\, M_1) \Longrightarrow \lambda x.\, M''_1}\,\mathsf{lm} \qquad \dfrac{R''_2}{M_2 \Longrightarrow M''_2}}{(\lambda x.\, M_1)\; M_2 \Longrightarrow (\lambda x.\, M''_1)\; M''_2}\,\mathsf{ap}.$$

Now we can apply the induction hypothesis twice to obtain:

$$\begin{tikzcd} & M_1 \arrow[dl, "R'_1"'] \arrow[dr, "R''_1"] & \\ M'_1 \arrow[dr, dotted, "S'_1"'] & & M''_1 \arrow[dl, dotted, "S''_1"] \\ & N_1 & \end{tikzcd} \qquad \begin{tikzcd} & M_2 \arrow[dl, "R'_2"'] \arrow[dr, "R''_2"] & \\ M'_2 \arrow[dr, dotted, "S'_2"'] & & M''_2 \arrow[dl, dotted, "S''_2"] \\ & N_2 & \end{tikzcd}$$

By the substitution lemma there is an $S' :: [M'_2/x]M'_1 \Longrightarrow [N_2/x]N_1$. Furthermore, we can apply the $\beta$ rule to $S''_1$ and $S''_2$ to obtain a $S'' :: (\lambda x.\, M''_1)\; M''_2 \Longrightarrow [N_2/x]N_1$ to complete the diagram.

$$\begin{tikzcd} & (\lambda x.\, M_1)\; M_2 \arrow[dl, "\mathsf{beta}(R'_1, R'_2) = R'"'] \arrow[dr, "R'' = \mathsf{ap}(\mathsf{lm}(R''_1), R''_2)"] & \\ {[M'_2/x]M'_1} \arrow[dr, dotted, "S'"'] & & (\lambda x.\, M''_1)\; M''_2 \arrow[dl, dotted, "S'' = \mathsf{beta}(S''_1, S''_2)"] \\ & {[N_2/x]N_1} & \end{tikzcd}$$

Again, in the implementation we have to assume a rule about the variable `x`.

```
dia_bal : dia (beta R1' R2') (ap (lm R1'') R2'')
              S' (beta S1'' S2'')
        <- ({x:term} {idx: x => x}
              dia idx idx idx idx
              -> dia (R1' x idx) (R1'' x idx)
                     (S1' x idx) (S1'' x idx))
        <- dia R2' R2'' S2' S2''
        <- subst S1' S2' S'.
```

**Case:** The reduction $R'$ is an application congruence and $R''$ is a $\beta$ reduction. This is dual to the previous case.

```
dia_alb : dia (ap (lm R1') R2') (beta R1'' R2'')
              (beta S1' S2') S''
       <- ({x:term} {idx: x => x}
              dia idx idx idx idx
              -> dia (R1' x idx) (R1'' x idx)
                     (S1' x idx) (S1'' x idx))
       <- dia R2' R2'' S2' S2''
       <- subst S1'' S2'' S''.
```

**Case:** Both sides end in an application of the ap rule. Then

$$R' = \dfrac{\begin{array}{cc} R_1' & R_2' \\ M_1 \Longrightarrow M_1' & M_2 \Longrightarrow M_2' \end{array}}{M_1\ M_2 \Longrightarrow M_1'\ M_2'}\ \mathsf{ap}$$

and

$$R'' = \dfrac{\begin{array}{cc} R_1'' & R_2'' \\ M_1 \Longrightarrow M_1'' & M_2 \Longrightarrow M_2'' \end{array}}{M_1\ M_2 \Longrightarrow M_1''\ M_2''}\ \mathsf{ap}.$$

We apply the induction hypothesis twice to complete the following diagrams.

$$\begin{tikzcd}
& M_1 \arrow[dl, "R_1'"'] \arrow[dr, "R_1''"] & \\
M_1' \arrow[dr, dotted, "S_1'"'] & & M_1'' \arrow[dl, dotted, "S_1''"] \\
& N_1 &
\end{tikzcd}
\qquad
\begin{tikzcd}
& M_2 \arrow[dl, "R_2'"'] \arrow[dr, "R_2''"] & \\
M_2' \arrow[dr, dotted, "S_2'"'] & & M_2'' \arrow[dl, dotted, "S_2''"] \\
& N_2 &
\end{tikzcd}$$

We combine the result by using the ap congruence on each side of the split.

$$\begin{tikzcd}
& M_1\ M_2 \arrow[dl, "\mathsf{ap}(R_1', R_2') = R'"'] \arrow[dr, "R'' = \mathsf{ap}(R_1'', R_2'')"] & \\
M_1'\ M_2' \arrow[dr, dotted, "\mathsf{ap}(S_1', S_2') = S'"'] & & M_1''\ M_2'' \arrow[dl, dotted, "S'' = \mathsf{ap}(S_1'', S_2'')"] \\
& N_1\ N_2 &
\end{tikzcd}$$

The implementation is straightforward.

```
dia_aa  : dia (ap R1' R2') (ap R1'' R2'') (ap S1' S2') (ap S1'' S2'')
       <- dia R1' R1'' S1' S1''
       <- dia R2' R2'' S2' S2''.
```

**Case:** In the final case both sides end in an application of the $\mathsf{lm}$-congruence.

$$R' = \quad \frac{\begin{array}{c} R'_1 \\ M_1 \Longrightarrow M'_1 \end{array}}{\lambda x.\, M_1 \Longrightarrow \lambda x.\, M'_1}\,\mathsf{lm}$$

and

$$R'' = \quad \frac{\begin{array}{c} R''_1 \\ M_1 \Longrightarrow M''_1 \end{array}}{\lambda x.\, M_1 \Longrightarrow \lambda x.\, M''_1}\,\mathsf{lm}.$$

We apply the induction hypothesis to fill in the following diagram.

$$\begin{tikzcd}
 & M_1 \arrow[dl, "R'_1"'] \arrow[dr, "R''_1"] & \\
M'_1 \arrow[dr, dotted, "S'_1"'] & & M''_1 \arrow[dl, dotted, "S''_1"] \\
 & N_1 &
\end{tikzcd}$$

Now, applying the congruence to the resulting reductions $S'_1$ and $S''_1$ we complete the diagram.

$$\begin{tikzcd}
 & \lambda x.\, M_1 \arrow[dl, "\mathsf{lm}(R'_1) = R'"'] \arrow[dr, "R'' = \mathsf{lm}(R''_1)"] & \\
\lambda x.\, M'_1 \arrow[dr, dotted, "\mathsf{lm}(S'_1) = S'"'] & & \lambda x.\, M''_1 \arrow[dl, dotted, "S'' = \mathsf{lm}(S''_1)"] \\
 & \lambda x.\, N_1 &
\end{tikzcd}$$

Once again, assumptions for variables need to be made here.

```
dia_ll  : dia (lm R1') (lm R1'') (lm S1') (lm S1'')
       <- ({x:term} {idx: x => x}
             dia idx idx idx idx
             -> dia (R1' x idx) (R1'' x idx) (S1' x idx) (S1'' x idx)).
```

□

The Elf rules given in the proof above are a complete implementation of the proof: whenever we have reduction $R' :: M \Longrightarrow M'$ and $R'' :: M \Longrightarrow M''$ then the Elf program will find an $N$ and reductions $S' :: M' \Longrightarrow N$ and $S'' :: M'' \Longrightarrow N$ which complete the diagram according to the algorithm which is implicit in the proof. Type-checking the signature above guarantees weak form of correctness: whenever we apply `dia` to concrete derivations $R'$ and $R''$ and `dia` terminates, then we can read off a valid diagram. The process of schema-checking guarantees that that `dia` is total in its first two arguments. These observations together verify the diamond lemma. Schema-checking is sketched in [PR92], but the implementation is incomplete and most of it still has to be done by hand. Other non-trivial examples have been carried out using the methodology, such as a verification of type soundness of Mini-ML [MP91] and a compiler from Mini-ML to a variant of the Categorial Abstract Machine (CAM) [HP92].

As an example for the execution of the Elf program above, reconsider the term

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

which can be reduced in four different ways: the outer redex, the inner redex, both, or neither. Thus, the following query will enumerate 16 different diagrams (we show two). Here we use the special, top-level form `sigma [x:A] B` to *stage* queries, that is, solving `sigma [x:A] B` first solves `A`, binds `x` to the result and then solves `B` under this binding. This operational behavior can be simulated in Elf without this special form of query, but only in a relatively cumbersome way.

```
?- sigma [R' : (app (lam [x] (app x x)) (app (lam [y] y) (lam [z] z))) => M']
   sigma [R'' : (app (lam [x] (app x x)) (app (lam [y] y) (lam [z] z))) => M'']
   dia R' R'' (S' : M' => N) (S'' : M'' => N).
Solving...

M' = app (lam ([x:term] x)) (lam ([x:term] x)),
M'' = app (lam ([x:term] x)) (lam ([x:term] x)),
R' = beta ([x:term] [R:x => x] ap R R)
          (beta ([x:term] [R:x => x] R) (lm ([x:term] [R:x => x] R))),
R'' = beta ([x:term] [R:x => x] ap R R)
           (beta ([x:term] [R:x => x] R) (lm ([x:term] [R:x => x] R))),
N = app (lam ([x:term] x)) (lam ([x:term] x)),
S' = ap (lm ([x:term] [idx:x => x] idx)) (lm ([x:term] [idx:x => x] idx)),
S'' = ap (lm ([x:term] [idx:x => x] idx)) (lm ([x:term] [idx:x => x] idx)).
;

M' = app (lam ([x:term] x)) (lam ([x:term] x)),
M'' = app (app (lam ([x:term] x)) (lam ([x:term] x)))
          (app (lam ([x:term] x)) (lam ([x:term] x))),
R' = beta ([x:term] [R:x => x] ap R R)
          (beta ([x:term] [R:x => x] R) (lm ([x:term] [R:x => x] R))),
R'' = beta ([x:term] [R:x => x] ap R R)
           (ap (lm ([x:term] [R:x => x] R)) (lm ([x:term] [R:x => x] R))),
N = app (lam ([x:term] x)) (lam ([x:term] x)),
S' = ap (lm ([x:term] [idx:x => x] idx)) (lm ([x:term] [idx:x => x] idx)),
S'' =
   ap (beta ([x:term] [idx:x => x] idx) (lm ([x:term] [idx:x => x] idx)))
      (beta ([x:term] [idx:x => x] idx) (lm ([x:term] [idx:x => x] idx))).
```

The next step in the proof of the Church-Rosser theorem is the *strip lemma* which is depicted in the following diagram.

$$\begin{tikzcd}
& M \arrow[dl, "R'"'] \arrow[dr, "R^{*\prime\prime}"] & \\
M' \arrow[dr, dotted, "S^{*\prime}"'] & & M'' \arrow[dl, dotted, "S''"] \\
& N &
\end{tikzcd}$$

Here, $R^{*\prime\prime}$ and $S^{*\prime}$ stand for multi-step parallel reductions.

**Lemma 7** (Strip Lemma) *If* $R' :: M \Longrightarrow M'$ *and* $R^{*\prime\prime} :: M \Longrightarrow^* M''$ *then there exists an* $N$ *and reductions* $S^{*\prime} :: M' \Longrightarrow^* N$ *and* $S'' :: M'' \Longrightarrow N$*.*

**Proof:** By induction over the structure of $R^{*\prime\prime}$. The proof is implemented as type family `strip`.

```
strip : M => M'  ->  M =>* M''  ->  M' =>* N  ->  M'' => N -> type.
```

**Case:** $R^{*\prime\prime}$ is the identity reduction. Then $M'' = M$ and we can let $N$ be $M'$.

$$\begin{tikzcd}
& M \arrow[dl, "R'"'] \arrow[dr, "\mathsf{id}"] & \\
M' \arrow[dr, dotted, "\mathsf{id}"'] & & M \arrow[dl, dotted, "R'"] \\
& M' &
\end{tikzcd}$$

```
strip_id   : strip R' id id R'.
```

**Case:** $R^{*\prime\prime}$ ends in a reduction step.

$$R^{*\prime\prime} = \frac{\overset{R_1''}{M \Longrightarrow M_1''} \qquad \overset{R_2^{*\prime\prime}}{M_1'' \Longrightarrow^* M''}}{M \Longrightarrow^* M''}\,\mathsf{step}$$

Now we can appeal to the diamond lemma on $R'$ and $R_1''$ to obtain an $S_1'$ and $S_1''$. Next the induction hypothesis on $S_1''$ and $R_2^{*\prime\prime}$ completes the diagram.

$$\begin{tikzcd}
& M \arrow[dl, "R'"'] \arrow[dr, "R_1''"] & & \\
M' \arrow[dr, dotted, "S_1'"'] & & M_1'' \arrow[dl, dotted, "S_1''"] \arrow[dr, "R_2^{*\prime\prime}"] & \\
& N_1 \arrow[dr, dotted, "S_2^{*\prime}"'] & & M'' \arrow[dl, dotted, "S''"] \\
& & N &
\end{tikzcd}$$

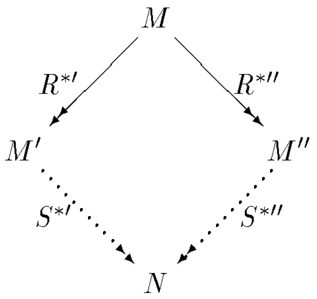
Recall that the rule `step` was written as an infix semi-colon.

```
strip_step : strip R' (R1'' ; R2*'') (S1' ; S2*') S''
          <- dia R' R1'' S1' S1''
          <- strip S1'' R2*'' S2*' S''.
```

□

Now we can prove the diamond property for multi-step reduction which we call *confluence*. In the literature this property is often refered to as the Church-Rosser theorem, since in most situations it is equivalent to the property of conversion actually proved in [CR36] (here: Theorem 16).

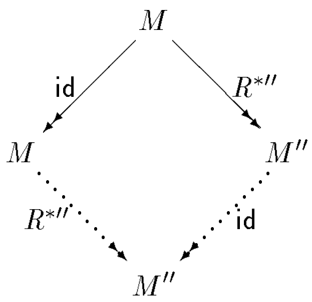
**Lemma 8** (Confluence) *If $R^{*\prime} :: M \Longrightarrow^* M'$ and $R^{*\prime\prime} :: M \Longrightarrow^* M''$ then there exists an $N$ and reductions $S^{*\prime} :: M' \Longrightarrow^* N$ and $S^{*\prime\prime} :: M'' \Longrightarrow^* N$.*




**Proof:** By induction on the structure of $R^{*\prime}$. The implementation is as a type family `conf`.

```
conf :  M =>* M'  ->  M =>* M''  ->  M' =>* N  ->  M'' =>* N -> type.
```

**Case:** $R^{*\prime}$ ends in the identity. Then $M' = M$ and we can let $N$ be $M''$ to fill the diagram.




```
conf_id   : conf id R*'' R*'' id.
```

**Case:** $R^{*\prime}$ ends in a reduction step $R_1'$ followed by $R_2^{*\prime}$. Then we apply the strip lemma and then the induction hypothesis on $R_2^{*\prime}$ to fill in the diagram.

$$\begin{tikzcd}
 & & M \arrow[dl, "R_1'"'] \arrow[dr, "R^{*\prime\prime}"] & \\
 & M_1' \arrow[dl, "R_2^{*\prime}"'] \arrow[dr, dotted, "S_1^{*\prime}"] & & M'' \arrow[dl, dotted, "S_1''"] \\
M' \arrow[dr, dotted, "S^{*\prime}"'] & & N_1 \arrow[dl, dotted, "S_2^{*\prime\prime}"] & \\
 & N & &
\end{tikzcd}$$

```
conf_step : conf (R1' ; R2*') R*'' S*' (S1'' ; S2*'')
          <- strip R1' R*'' S1*' S1''
          <- conf R2*' S1*' S*' S2*''.
```

□

Finally we are ready to prove the Church-Rosser theorem for parallel conversion and reduction.

**Theorem 9** (Church-Rosser) *If $M \Longleftrightarrow M'$ then there exists a term $N$ and reductions $S^* :: M \Longrightarrow^* N$ and $S^{*\prime} :: M' \Longrightarrow^* N$*

$$\begin{tikzcd}
M \arrow[rr, leftrightarrow, "C"] \arrow[dr, dotted, "S^*"'] & & M' \arrow[dl, dotted, "S^{*\prime}"] \\
 & N &
\end{tikzcd}$$

**Proof:** By induction over the structure of $C :: M \Longleftrightarrow M'$. The proof is implemented as a family

```
cr :  M <=> M'  ->  M =>* N  ->  M' =>* N  -> type.
```

**Case:** $C$ is a reduction $R^* :: M \Longrightarrow^* M'$. Then we let $N$ be $M'$.

$$\begin{tikzcd}
M \arrow[rr, leftrightarrow, "\mathsf{reduce}(R^*)"] \arrow[dr, dotted, "R^*"'] & & M' \arrow[dl, dotted, "\mathsf{id}"] \\
 & M' &
\end{tikzcd}$$

```
cr_reduce  : cr (reduce R*) R* id.
```

**Case:** $C$ is a reduction $R^* :: M' \Longrightarrow^* M$. Then we let $N$ be $M$.

$$\begin{tikzcd}
M \arrow[rr, leftrightarrow, "\mathsf{expand}(R^*)"] \arrow[dr, dotted, "\mathsf{id}"'] & & M' \arrow[dl, dotted, "R^*"] \\
 & M &
\end{tikzcd}$$

```
cr_expand  : cr (expand R*) id R*.
```

**Case:** $C$ is a composition of conversions. This is the interesting case.

$$C = \dfrac{\begin{array}{c} C_1 \\ M \Longleftrightarrow M'' \end{array} \qquad \begin{array}{c} C_2 \\ M'' \Longleftrightarrow M' \end{array}}{M \Longleftrightarrow M'}\,\textsf{comp}$$

Then we apply the induction hypothesis to $C_1$ and $C_2$, followed by an appeal to confluence and the transitivity of parallel multi-step reduction.

$$\begin{tikzcd}
M \arrow[rr, leftrightarrow, "C_1"] \arrow[dr, dotted, "S_1^*"'] & & M'' \arrow[dl, dotted, "R_1^*"] \arrow[dr, dotted, "R_2^*"'] \arrow[rr, leftrightarrow, "C_2"] & & M' \arrow[dl, dotted, "S_2^*"] \\
& N_1 \arrow[dr, dotted, "T_1^*"'] & & N_2 \arrow[dl, dotted, "T_2^*"] & \\
& & N & &
\end{tikzcd}$$

The Elf code makes the call to the transitivity lemma explicit which is only implicit in the diagram (we need to append the reduction sequence $S_1^*$ and $T_1^*$ on the left, and $S_2^*$ and $T_2^*$ on the right).

```
cr_compose : cr (C1 ;; C2) S* S*'
          <- cr C1 S1* R1*
          <- cr C2 R2* S2*
          <- conf R1* R2* T1* T2*
          <- append S1* T1* S*
          <- append S2* T2* S*'.
```

□

## 6 Equivalence of Ordinary and Parallel Reduction

In this section we will prove that multi-step ordinary reduction and multi-step parallel reduction define the same relation between terms. As a direct corollary we obtain the Church-Rosser theorem for ordinary reduction. The first lemma states that parallel reduction can be simulated by multi-step ordinary reduction.

**Lemma 10** *If $M \Longrightarrow N$ then $M \longrightarrow^* N$.*

**Proof:** By induction on the structure of $R :: M \Longrightarrow N$. In each case we explicitly construct a reduction $S^* :: M \longrightarrow^* N$. We heavily use Lemmas 2 and 1 which state that multi-step reduction is congruent and transitive. The proof is implemented in Elf by a type family `eq1`.

```
eq1 : M => N  ->  M -->* N  ->  type.
```

**Case:**

$$R = \dfrac{}{x \Longrightarrow x}\,\mathsf{var}$$

Then $\mathsf{id}_1$ is a multi-step reduction from $x$ to $x$. As usual, this case is not directly represented as a separate declaration in the Elf implementation, but folded into the cases where parameters are introduced.

**Case:**

$$R = \dfrac{\begin{array}{c} R_1 \\ M_1 \Longrightarrow M_1' \end{array} \qquad \begin{array}{c} R_2 \\ M_2 \Longrightarrow M_2' \end{array}}{(\lambda x.\ M_1)\ M_2 \Longrightarrow [M_2'/x]M_1'}\,\mathsf{beta}$$

$S_1^* :: M_1 \longrightarrow^* M_1'$ By ind. hyp. on $R_1$
$S_1^{*\prime} :: \lambda x.\ M_1 \longrightarrow^* \lambda x.\ M_1'$ By congruence
$S_1^{*\prime\prime} :: (\lambda x.\ M_1)\ M_2 \longrightarrow^* (\lambda x.\ M_1')\ M_2$ By congruence
$S_2^* :: M_2 \longrightarrow^* M_2'$ By ind. hyp. on $R_2$
$S_2^{*\prime} :: (\lambda x.\ M_1')\ M_2 \longrightarrow^* (\lambda x.\ M_1')\ M_2'$ By congruence
$S_3 :: (\lambda x.\ M_1')\ M_2' \longrightarrow [M_2'/x]M_1'$ By $\mathsf{beta}_1$
$S^* :: (\lambda x.\ M_1)\ M_2 \longrightarrow^* [M_2'/x]M_1'$ By transitivity from $S_1^{*\prime\prime}$, $S_2^{*\prime}$, and $S_3$.

The implementation of this case is a fairly direct translation of the above algorithm. Since $M_1$ is in the scope of $x$ we need to make an appropriate assumption about reductions from $x$ to $x$, namely that $x \Longrightarrow x$ is translated to $\mathsf{id}_1$ as indicated in the previous case. Appeals to congruence use the admissible rules from Lemma 2, depending on which congruence is required.

```
eq1_beta : eq1 (beta R1 R2) S*
         <- ({x:term} {eqx : x => x}
               eq1 eqx id1 -> eq1 (R1 x eqx) (S1* x))
         <- lm1* S1* S1*'
         <- apl1* S1*' S1*''
         <- eq1 R2 S2*
         <- apr1* S2* S2*'
         <- appd S2*' (step1 beta1 id1) S*'
         <- appd S1*'' S*' S*.
```

**Case:**

$$R = \dfrac{\begin{array}{c} R_1 \\ M_1 \Longrightarrow M_1' \end{array} \qquad \begin{array}{c} R_2 \\ M_2 \Longrightarrow M_2' \end{array}}{M_1\ M_2 \Longrightarrow M_1'\ M_2'}\,\mathsf{ap}$$

$S_1^* :: M_1 \longrightarrow^* M_1'$ By ind. hyp.
$S^{*\prime} :: M_1\ M_2 \longrightarrow^* M_1'\ M_2$ By congruence
$S_2^* :: M_2 \longrightarrow^* M_2'$ By ind. hyp.
$S^{*\prime\prime} :: M_1'\ M_2 \longrightarrow^* M_1'\ M_2'$ By congruence
$S^* :: M_1\ M_2 \longrightarrow^* M_1'\ M_2'$ By transitivity from $S^{*\prime}$ and $S^{*\prime\prime}$

```
eq1_ap : eq1 (ap R1 R2) S*
       <- eq1 R1 S1*
       <- apl1* S1* S*'
       <- eq1 R2 S2*
       <- apr1* S2* S*''
       <- appd S*' S*'' S*.
```

**Case:**

$$R = \cfrac{\begin{array}{c} R_1 \\ M_1 \Longrightarrow M_1' \end{array}}{\lambda x.\ M_1 \Longrightarrow \lambda x.\ M_1'}\ \mathsf{lm}$$

$S_1^* :: M_1 \longrightarrow^* M_1'$ By ind. hyp.
$S^* :: \lambda x.\ M_1 \longrightarrow^* \lambda x.\ M_1'$ By congruence

In the implementation, we once again have to make the proper assumption for the variable $x$, which may be reduced to itself.

```
eq1_lm : eq1 (lm R1) S*
       <- ({x:term} {eqx : x => x}
            eq1 eqx id1 -> eq1 (R1 x eqx) (S1* x))
       <- lm1* S1* S*.
```

□

The next lemma goes in the opposite direction, but this time we directly replace ordinary single-step reduction by parallel single-step reduction.

**Lemma 11** *If $M \longrightarrow N$ then $M \Longrightarrow N$.*

**Proof:** The proof is by induction on $R :: M \longrightarrow N$. In each case we explicitly construct an $S :: M \Longrightarrow N$. In an ordinary reduction fewer subterms are reduced, so we need to "pad" the reductions with identities to obtain the parallel reductions. For this, we employ Lemma 3 which states the reflexivity of parallel reduction.

```
eq2 : M --> N  ->  M => N  ->  type.
```

**Case:**

$$R = \cfrac{}{(\lambda x.\ M_1)\ M_2 \longrightarrow [M_2/x]M_1}\ \mathsf{beta}_1$$

Then

$$S = \cfrac{\begin{array}{c} I_1 \\ M_1 \Longrightarrow M_1 \end{array} \qquad \begin{array}{c} I_2 \\ M_2 \Longrightarrow M_2 \end{array}}{(\lambda x.\ M_1)\ M_2 \longrightarrow [M_2/x]M_1}\ \mathsf{beta}$$

where $I_1$ and $I_2$ exist by reflexivity of parallel reduction.

Recall the type of the implementation of Lemma 3:

```
identity : {M:term} M => M  ->  type.
```

Since we have chosen to make the argument `M` explicit we now need to supply appropriate terms wherever we appeal to reflexivity.

```
eq2_beta1 : eq2 (beta1) (beta I1 I2)
          <- ({x:term} {eqx : x => x}
                identity x eqx -> identity (M1 x) (I1 x eqx))
          <- identity M2 I2.
```

**Case:**

$$R = \frac{\begin{array}{c} R_1 \\ M_1 \longrightarrow M_1' \end{array}}{\lambda x.\, M_1 \longrightarrow \lambda x.\, M_1'}\,\mathsf{lm}_1$$

By the induction hypothesis on $R_1$ we know there exists an $S_1 :: M_1 \Longrightarrow M_1'$. By an application of the $\mathsf{lm}$ rule we conclude that $\lambda x.\, M_1 \Longrightarrow \lambda x.\, M_1'$.

In the Elf implementation we need to introduce a new parameter for the bound variable $x$. Note that this variable does not reduce to itself, since ordinary reduction has no case $x \longrightarrow x$.

```
eq2_lm1   : eq2 (lm1 R1) (lm ([x:term] [eqx : x => x] S1 x))
          <- {x:term} eq2 (R1 x) (S1 x).
```

**Case:**

$$R = \frac{\begin{array}{c} R_1 \\ M_1 \longrightarrow M_1' \end{array}}{M_1\ M_2 \longrightarrow M_1'\ M_2}\,\mathsf{apl}_1$$

By induction hypothesis there is an $S_1 :: M_1 \Longrightarrow M_1'$ and from the reflexivity of parallel reduction we know there is an $I_2 :: M_2 \Longrightarrow M_2$. Thus we can let

$$S = \frac{\begin{array}{c} S_1 \\ M_1 \Longrightarrow M_1' \end{array} \qquad \begin{array}{c} I_2 \\ M_2 \Longrightarrow M_2 \end{array}}{M_1\ M_2 \Longrightarrow M_1'\ M_2}\,\mathsf{ap}$$

```
eq2_apl1  : eq2 (apl1 R1) (ap S1 I2)
          <- eq2 R1 S1
          <- identity M2 I2.
```

**Case:**

$$R = \frac{\begin{array}{c} R_1 \\ M_2 \longrightarrow M_2' \end{array}}{M_1\ M_2 \longrightarrow M_1\ M_2'}\,\mathsf{apr}_1$$

This is symmetric to the previous case.

```
    eq2_apr1  : eq2 (apr1 R2) (ap I1 S2)
              <- eq2 R2 S2
              <- identity M1 I1.
```

□

From Lemmas 10 and 11 the equivalence of the generated multi-step reduction relations can be proved easily.

**Theorem 12** *$M \longrightarrow^* N$ iff $M \Longrightarrow^* N$.*

**Proof:** In both directions by simple inductions over reduction sequences. We will leave the informal proof to the reader and give only the implementation in Elf. Recall the type families

```
eq1 : M => N  ->  M -->* N  ->  type.
eq2 : M --> N  ->  M => N  ->  type.
```

which implement Lemmas 10 and 11, respectively. The families `eq3` and `eq4` implement the two claimed implications.

```
eq3 : M -->* N  ->  M =>* N  ->  type.

eq3_id : eq3 id1 id.
eq3_step : eq3 (step1 R1 R2*) (S1 ; S2*)
         <- eq2 R1 S1
         <- eq3 R2* S2*.

eq4 : M =>* N  ->  M -->* N  ->  type.

eq4_id : eq4 id id1.
eq4_step : eq4 (R1 ; R2*) S*
         <- eq1 R1 S1*
         <- eq4 R2* S2*
         <- appd S1* S2* S*.
```

□

From the equivalence of the reduction relations, the equivalence of conversion also follows almost immediately.

**Lemma 13** *If $M \Longleftrightarrow N$ then $M \longleftrightarrow N$.*

**Proof:** By induction on the structure of $C :: M \Longleftrightarrow N$. In each case, we explicitly construct a $C' :: M \longleftrightarrow N$, taking advantage of Theorem 12. Since the proof is trivial, we only give its implementation in Elf. Recall that $\longleftrightarrow$ is defined as the equivalence closure of $\longrightarrow$, while $\Longleftrightarrow$ is defined as a reduction, expansion (inverse of reduction) or composition of two conversions.

```
eq5 : M <=> N  ->  M <-> N  ->  type.

eq5_red   : eq5 (reduce R*) (red S*)
```

```
           <- eq4 R* S*.
eq5_exp    : eq5 (expand R*) (sym (red S*))
           <- eq4 R* S*.
eq5_trans : eq5 (C1 ;; C2) (trans C1' C2')
           <- eq5 C1 C1'
           <- eq5 C2 C2'.
```

□

Because of the definition of parallel conversion via reduction and expansion instead of symmetry and transitivity, we need to explicitly show the symmetry of parallel conversion as a simple lemma.

**Lemma 14** *If $M \Longleftrightarrow N$ then $N \Longleftrightarrow M$.*

**Proof:** The proof is a simple induction on the structure of $C :: M \Longleftrightarrow N$. We only show the implementation of this proof in Elf.

```
sym_pconv : M <=> N  ->  N <=> M  ->  type.

spc_red   : sym_pconv (reduce R*) (expand R*).
spc_exp   : sym_pconv (expand R*) (reduce R*).
spc_trans : sym_pconv (C1 ;; C2) (C2' ;; C1')
           <- sym_pconv C1 C1'
           <- sym_pconv C2 C2'.
```

□

**Lemma 15** *If $M \longleftrightarrow N$ then $M \Longleftrightarrow N$.*

**Proof:** By induction on the structure of $C :: M \longleftrightarrow N$. In each case we explicitly construct a $C' :: M \Longleftrightarrow N$. The implementation is as a type family

```
eq6 : M <-> N  ->  M <=> N  ->  type.
```

**Case:**

$$C = \frac{}{M \longleftrightarrow M}\,\textsf{refl}$$

Then we let

$$C' = \frac{\dfrac{}{M \Longrightarrow^* M}\,\textsf{id}}{M \Longleftrightarrow M}\,\textsf{reduce}$$

```
eq6_refl  : eq6 refl (reduce id).
```

**Case:**

$$C = \frac{\begin{array}{c} C_1 \\ N \longleftrightarrow M \end{array}}{M \longleftrightarrow N}\,\textsf{sym}$$

By induction hypothesis there exists a $C_1' :: N \Longleftrightarrow M$. By symmetry of parallel conversion (Lemma 14) we obtain a $C' :: M \Longleftrightarrow N$.

```
eq6_sym    : eq6 (sym C1) C'
           <- eq6 C1 C1'
           <- sym_pconv C1' C'.
```

**Case:**

$$C = \cfrac{\begin{array}{cc} C_1 & C_2 \\ M \longleftrightarrow M' & M' \longleftrightarrow N \end{array}}{M \longleftrightarrow N}\,\mathsf{trans}$$

Then $C'$ follows from the induction hypothesis on $C_1$ and $C_2$ and the transitivity rule for parallel conversion.

```
eq6_trans : eq6 (trans C1 C2) (C1' ;; C2')
           <- eq6 C1 C1'
           <- eq6 C2 C2'.
```

**Case:**

$$C = \cfrac{\begin{array}{c} R^* \\ M \longrightarrow^* N \end{array}}{M \longleftrightarrow N}\,\mathsf{red}$$

By Theorem 12 there exists an $S^* :: M \Longrightarrow^* N$ and we let

$$C' = \cfrac{\begin{array}{c} S^* \\ M \Longrightarrow^* N \end{array}}{M \Longleftrightarrow N}\,\mathsf{reduce}$$

```
eq6_red    : eq6 (red R*) (reduce S*)
           <- eq3 R* S*.
```

□

Now we can prove the Church-Rosser theorem for ordinary conversion by translating to parallel reduction. Not all of the lemmas above are actually necessary to prove this theorem.

**Theorem 16** (Church-Rosser) *If $M \longleftrightarrow M'$ then there exists an $N$ such that $M \longrightarrow^* N$ and $M' \longrightarrow^* N$.*

**Proof:** By Lemma 15, there exists a $C' :: M \Longleftrightarrow M'$. By the Church-Rosser theorem for parallel conversion (Theorem 9) we obtain an $N$ and parallel multi-step reduction $R^* :: M \Longrightarrow^* N$ and $R^{*\prime} :: M' \Longrightarrow^* N$. By Theorem 12 there exist $S^* :: M \longrightarrow^* N$ and $S^{*\prime} :: M' \longrightarrow^* N$.

```
cr_ord : M <-> M'  ->  M -->* N  ->  M' -->* N  ->  type.

cr_all : cr_single C S* S*'
       <- eq6 C C'
       <- cr C' R* R*'
       <- eq4 R* S*
       <- eq4 R*' S*'.
```

□

# 7 Conclusion

We have demonstrated the use of the logical framework LF and its realization in the Elf programming language for the implementation of abstract syntax, semantics, and meta-theory of an object language, the untyped $\lambda$-calculus. The main meta-theorem, the Church-Rosser property under $\beta$-reduction, is non-trivial and its implementation in Elf illustrates various representation techniques such as higher-order abstract syntax, judgments-as-types, and proofs of meta-theorems as higher-level judgments. These techniques permit the user to concentrate on the mathematical content of a proof and largely ignore details of variable naming and capture-avoiding substitution as is usually done in informal proofs. This and the power of term reconstruction in Elf lead to a remarkably close correspondence between informal and formal proof. Starting from an understanding of the basic idea of parallel reduction and the substitution lemma, the formalization of the core of this proof was done by the author in one afternoon, cleanup work and the relation to ordinary reduction took up another day. We hope to have convinced the reader that with some practice, representation of non-trivial languages and their properties is possible with a resonable amount of effort.

It is interesting to compare this representation with the proof by Shankar [Sha88] in the Boyer-Moore theorem prover [BM79]. While the basic mathematical ideas are very similar, Shankar expends much effort to develop an appropriate representation (using de Bruijn numbers [dB72]) and proving it correct. Many of the actual *proofs* are not even explicitly represented, since they are found automatically once the right series of lemmas has been developed. In contrast, in our representation almost all the details of the informal proof are present in the formalization (with the exception of the details inferred by type reconstruction). Thus the representations are of comparable length in the two implementations, but the content of what is actually written down is very different. In future work we hope to consider the question how much of the construction of the meta-level judgments which implement induction proofs can be automated. Intuitively, they often are straightforward from the stringent constraints imposed by type dependencies. This indicates that there is a great potential for the *automation* of meta-theory which has yet to be explored.

## Acknowledgments

I would like to thank John Reynolds who wrote the LaTeX macros I used for drawing the diagrams, and Ekkehard Rohwedder for proof-reading a draft of this report.

# A Summary of the Representation in Elf

In this appendix we summarize the Elf code shown in various places throughout the report for easy reference. The source is also labeled with the name of the file in which it appears in the implementation which is available via anonymous ftp.[4]

## A.1 The untyped $\lambda$-calculus

```
%%% File: lam.elf
%%% Untyped lambda-calculus

term : type.  %name term M

lam : (term -> term) -> term.
app : term -> term -> term.
```

## A.2 Ordinary reduction

```
%%% File: ord-red.elf
%%% Ordinary reduction for the untyped lambda-calculus

--> : term -> term -> type.  %infix none 10 -->
                             %name --> R

beta1 :   (app (lam M1) M2) --> M1 M2.

lm1   :       ({x:term} M x --> M' x)
      ->            (lam M) --> (lam M').

apl1  :              M1 --> M1'
      ->    (app M1 M2) --> (app M1' M2).

apr1  :              M2 --> M2'
      ->    (app M1 M2) --> (app M1 M2').

% Multi-step reduction

-->* : term -> term -> type.  %infix none 10 -->*
                              %name -->* R*

id1   :    M -->* M.

step1 :    M --> M'
      ->  M' -->* M''
      ->   M -->* M''.

% Conversion

<-> : term -> term -> type.  %infix none 10 <->
                             %name <-> C
```

[4] Please send electronic mail to the author at fp@cs.cmu.edu for further information.

```
refl :    M <-> M.

sym  :    M <-> M'
     ->  M' <-> M.

trans:    M <-> M'
     ->  M' <-> M''
     ->   M <-> M''.

red  :    M -->* M'
     ->   M <-> M'.
```

### A.3 Parallel reduction

```
%%% File: par-red.elf
%%% Parallel reduction in the untyped lambda calculus

=> : term -> term -> type.  %infix none 10 =>
                            %name => R

beta : ({x:term} x => x -> M1 x => M1' x)
     ->                       M2 => M2'
     ->       (app (lam M1) M2) => M1' M2'.

ap   :            M1 => M1'
     ->           M2 => M2'
     ->  (app M1 M2) => (app M1' M2').

lm   : ({x:term} x => x -> M x => M' x)
     ->                  lam M => lam M'.

% Parallel, multi-step reduction

=>* : term -> term -> type.  %infix none 10 =>*
                             %name =>* R*

id   :           M =>* M.

;    :           M => M'
     ->         M' =>* M''
     ->          M =>* M''.  %infix right 10 ;

% Parallel conversion

<=> : term -> term -> type.  %infix none 10 <=>
                             %name <=> C

reduce :  M  =>* M'
       -> M  <=> M'.

expand :  M  =>* M'
       -> M' <=> M.
```

```
;;     :  M  <=> M'
       -> M' <=> M''
       -> M  <=> M''.  %infix none 8 ;;
```

### A.4 Lemmas about parallel reduction

```
%%% File: par-lemmas.elf
%%% Basic lemmas concerning parallel reductions

% Every term reduces to itself (in parallel)

identity : {M:term} M => M  ->  type.

id_lam : identity (lam M1) (lm R1)
       <- {x:term} {eqx: x => x} identity x eqx -> identity (M1 x) (R1 x eqx).

id_app : identity (app M1 M2) (ap R1 R2)
       <- identity M1 R1
       <- identity M2 R2.

%  Parallel multi-step reduction is transitive.

append : M =>* M'  ->  M' =>* M''  ->  M =>* M''  -> type.

append_id   : append id S* S*.
append_step : append (R1 ; R2*) S* (R1 ; S2*')
           <- append R2* S* S2*'.
```

### A.5 The Church-Rosser theorem for parallel reduction

```
%%% File: par-cr.elf
%%% The Church-Rosser theorem for parallel reduction

% Substitution lemma for parallel reduction

subst : ({x:term} x => x -> M x => M' x)
      -> N => N'
      -> M N => M' N'
      -> type.

subst_idx : subst ([x:term] [idx: x => x] idx) S S.

subst_beta : subst ([x:term] [idx: x => x] beta (R1 x idx) (R2 x idx))
                   S (beta R1' R2')
           <- ({y:term} {idy: y => y}
                 subst ([x:term] [idx: x => x] idy) S idy
                 -> subst ([x:term] [idx: x => x] R1 x idx y idy)
                        S (R1' y idy))
           <- subst R2 S R2'.
```

```
subst_ap  : subst ([x:term] [idx: x => x] ap (R1 x idx) (R2 x idx))
                  S (ap R1' R2')
          <- subst R1 S R1'
          <- subst R2 S R2'.

subst_lm  : subst ([x:term] [idx: x => x] lm (R1 x idx))
                  S (lm R1')
          <- ({y:term} {idy: y => y}
                subst ([x:term] [idx: x => x] idy) S idy
                -> subst ([x:term] [idx: x => x] R1 x idx y idy)
                         S (R1' y idy)).

% Diamond property for parallel reduction

dia : M => M'  ->  M => M''  ->  M' => N  ->  M'' => N  -> type.

% Proof by induction on the structure of the first two derivations.
% We consider the various possible cases.
% b = beta, a = ap, l = lm,

dia_bb : dia (beta R1' R2') (beta R1'' R2'') S' S''
      <- ({x:term} {idx: x => x}
            dia idx idx idx idx
            -> dia (R1' x idx) (R1'' x idx)
                   (S1' x idx) (S1'' x idx))
      <- dia R2' R2'' S2' S2''
      <- subst S1' S2' S'
      <- subst S1'' S2'' S''.

dia_bal : dia (beta R1' R2') (ap (lm R1'') R2'')
             S' (beta S1'' S2'')
       <- ({x:term} {idx: x => x}
             dia idx idx idx idx
             -> dia (R1' x idx) (R1'' x idx)
                    (S1' x idx) (S1'' x idx))
       <- dia R2' R2'' S2' S2''
       <- subst S1' S2' S'.

dia_alb : dia (ap (lm R1') R2') (beta R1'' R2'')
             (beta S1' S2') S''
       <- ({x:term} {idx: x => x}
             dia idx idx idx idx
             -> dia (R1' x idx) (R1'' x idx)
                    (S1' x idx) (S1'' x idx))
       <- dia R2' R2'' S2' S2''
       <- subst S1'' S2'' S''.

dia_aa  : dia (ap R1' R2') (ap R1'' R2'') (ap S1' S2') (ap S1'' S2'')
       <- dia R1' R1'' S1' S1''
       <- dia R2' R2'' S2' S2''.
```

```
dia_ll  : dia (lm R1') (lm R1'') (lm S1') (lm S1'')
       <- ({x:term} {idx: x => x}
             dia idx idx idx idx
             -> dia (R1' x idx) (R1'' x idx) (S1' x idx) (S1'' x idx)).

% The strip lemma for parallel reduction.

strip : M => M'  ->  M =>* M''  ->  M' =>* N  ->  M'' => N -> type.

strip_id   : strip R' id id R'.
strip_step : strip R' (R1'' ; R2*'') (S1' ; S2*') S''
          <- dia R' R1'' S1' S1''
          <- strip S1'' R2*'' S2*' S''.

% Confluence for parallel multi-step reduction.

conf :  M =>* M'  ->  M =>* M''  ->  M' =>* N  ->  M'' =>* N -> type.

conf_id   : conf id R*'' R*'' id.
conf_step : conf (R1' ; R2*') R*'' S*' (S1'' ; S2*'')
         <- strip R1' R*'' S1*' S1''
         <- conf R2*' S1*' S*' S2*''.

% Church-Rosser Theorem for parallel reduction

cr :  M <=> M'  ->  M =>* N  ->  M' =>* N  -> type.

cr_reduce  : cr (reduce R*) R* id.
cr_expand  : cr (expand R*) id R*.
cr_compose : cr (C1 ;; C2) S* S*'
          <- cr C1 S1* R1*
          <- cr C2 R2* S2*
          <- conf R1* R2* T1* T2*
          <- append S1* T1* S*
          <- append S2* T2* S*'.
```

## A.6 Lemmas about ordinary reduction

```
%%% File: ord-lemmas.elf
%%% Lemmas concerning ordinary multi-step reduction

% Transitivity of multi-step reduction

appd :  M -->* M'  ->  M' -->* M''  ->  M -->* M''  ->  type.
appd_id   : appd id1 S* S*.
appd_step : appd (step1 R1 R2*) S* (step1 R1 S2*')
         <- appd R2* S* S2*'.

% Multi-step reduction is a congruence

lm1* : ({x:term} M x -->* M' x)
     ->      (lam M) -->* (lam M')
     -> type.
```

```
lm1*_id   : lm1* ([x:term] id1) id1.
lm1*_step : lm1* ([x:term] step1 (R1 x) (R2* x)) (step1 (lm1 R1) S2*)
              <- lm1* R2* S2*.

apl1* :            M1 -->* M1'
      ->  (app M1 M2) -->* (app M1' M2)
      -> type.

apl1*_id   : apl1* id1 id1.
apl1*_step : apl1* (step1 R1 R2*) (step1 (apl1 R1) S2*)
               <- apl1* R2* S2*.

apr1* :            M2 -->* M2'
      ->  (app M1 M2) -->* (app M1 M2')
      -> type.

apr1*_id : apr1* id1 id1.
apr1*_step : apr1* (step1 R1 R2*) (step1 (apr1 R1) S2*)
           <- apr1* R2* S2*.
```

## A.7 Equivalence of ordinary and parallel reduction

```
%%% File: equiv.elf
%%% Equivalence of ordinary and parallel reduction.

% If M => N then M -->* N.

eq1 : M => N  ->  M -->* N  ->  type.

eq1_beta : eq1 (beta R1 R2) S*
         <- ({x:term} {eqx : x => x}
               eq1 eqx id1 -> eq1 (R1 x eqx) (S1* x))
         <- lm1* S1* S1*'
         <- apl1* S1*' S1*''
         <- eq1 R2 S2*
         <- apr1* S2* S2*'
         <- appd S2*' (step1 beta1 id1) S*'
         <- appd S1*'' S*' S*.

eq1_ap : eq1 (ap R1 R2) S*
       <- eq1 R1 S1*
       <- apl1* S1* S*'
       <- eq1 R2 S2*
       <- apr1* S2* S*''
       <- appd S*' S*'' S*.

eq1_lm : eq1 (lm R1) S*
       <- ({x:term} {eqx : x => x}
             eq1 eqx id1 -> eq1 (R1 x eqx) (S1* x))
       <- lm1* S1* S*.
```

```
% If M --> N then M => N.

eq2 : M --> N  ->  M => N  ->  type.

eq2_beta1 : eq2 (beta1) (beta I1 I2)
          <- ({x:term} {eqx : x => x}
                identity x eqx -> identity (M1 x) (I1 x eqx))
          <- identity M2 I2.

eq2_lm1   : eq2 (lm1 R1) (lm ([x:term] [eqx : x => x] S1 x))
          <- {x:term} eq2 (R1 x) (S1 x).

eq2_apl1  : eq2 (apl1 R1) (ap S1 I2)
          <- eq2 R1 S1
          <- identity M2 I2.

eq2_apr1  : eq2 (apr1 R2) (ap I1 S2)
          <- eq2 R2 S2
          <- identity M1 I1.

% If M -->* N then M =>* N.

eq3 : M -->* N  ->  M =>* N  ->  type.

eq3_id : eq3 id1 id.
eq3_step : eq3 (step1 R1 R2*) (S1 ; S2*)
         <- eq2 R1 S1
         <- eq3 R2* S2*.

% If M =>* N then M -->* N.

eq4 : M =>* N  ->  M -->* N  ->  type.

eq4_id : eq4 id id1.
eq4_step : eq4 (R1 ; R2*) S*
         <- eq1 R1 S1*
         <- eq4 R2* S2*
         <- appd S1* S2* S*.

% If M <=> N then M <-> N.

eq5 : M <=> N  ->  M <-> N  ->  type.

eq5_red   : eq5 (reduce R*) (red S*)
          <- eq4 R* S*.
eq5_exp   : eq5 (expand R*) (sym (red S*))
          <- eq4 R* S*.
eq5_trans : eq5 (C1 ;; C2) (trans C1' C2')
          <- eq5 C1 C1'
          <- eq5 C2 C2'.
```

```
% If M <=> N then N <=> M.

sym_pconv : M <=> N  ->  N <=> M  ->  type.

spc_red   : sym_pconv (reduce R*) (expand R*).
spc_exp   : sym_pconv (expand R*) (reduce R*).
spc_trans : sym_pconv (C1 ;; C2) (C2' ;; C1')
          <- sym_pconv C1 C1'
          <- sym_pconv C2 C2'.

% If M <-> N then M <=> N.

eq6 : M <-> N  ->  M <=> N  ->  type.

eq6_refl  : eq6 refl (reduce id).
eq6_sym   : eq6 (sym C1) C'
          <- eq6 C1 C1'
          <- sym_pconv C1' C'.
eq6_trans : eq6 (trans C1 C2) (C1' ;; C2')
          <- eq6 C1 C1'
          <- eq6 C2 C2'.
eq6_red   : eq6 (red R*) (reduce S*)
          <- eq3 R* S*.
```

## A.8 The Church-Rosser theorem for ordinary reduction

```
%%% File: ord-cr.elf
%%% The Church-Rosser theorem for ordinary reduction

cr_ord : M <-> M'  ->  M -->* N  ->  M' -->* N  ->  type.

cr_all : cr_ord C S* S*'
       <- eq6 C C'
       <- cr C' R* R*'
       <- eq4 R* S*
       <- eq4 R*' S*'.
```

# References


[Bar80] H. P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. North-Holland, 1980.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. ACM monograph series. Academic Press, New York, 1979.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 36(3):472–482, May 1936.

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989.

[Han91] John Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, January 1991. Available as MS-CIS-91-09.

[Har90] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[HHP] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 199? To appear. Available as Technical Report CMU-CS-89-173, Carnegie Mellon University. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Mey82] Albert R. Meyer. What is a model of the lambda calculus. *Information and Control*, 52:87–122, 1982.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[NM88] Gopalan Nadathur and Dale Miller. An overview of $\lambda$Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[Pau86] Lawrence Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pfe91a] Frank Pfenning. An implementation of the Elf core language in Standard ML. Available via ftp over the Internet, September 1991. Send mail to elf-request@cs.cmu.edu for further information.

[Pfe91b] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PR92] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[Sha88] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery*, 35(3):475–522, July 1988.

[Wad76] Christopher P. Wadsworth. The relation between computational and denotational properties for Scott's $D_\infty$-models of the lambda-calculus. *SIAM Journal of Computing*, 5(3):488–521, September 1976.