UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Dipartimento di Matematica

MORE CHURCH-ROSSER PROOFS IN BELUGA

Relatore: Prof. Alberto MOMIGLIANO

Tesi di:
Martina SASSELLA
Matricola: 982898

Anno Accademico 2021-2022

dedicato a chi c'è sempre stato,
anche quando non c'ero io, per me.

Preface

This thesis focuses on the formalization of the Church-Rosser theorem using the proof environment Beluga. The Church-Rosser theorem is a classic result in the theory of the lambda calculus and can thus be considered a benchmark for comparing automated reasoning tools. The thesis begins with an overview of the motivations behind the work, the related formalized work, and a description of the system Beluga.

We then introduce the definitions and techniques involved in the proof in a more general setting for abstract reduction systems (ARSs). We discuss different methods to show confluence, such as the TMT method and the Commutative Union Lemma. Later, these concepts are applied to the lambda calculus.

The main part of the thesis is dedicated to the formalization of the Church-Rosser theorem in Beluga. The application of the TMT method and the Commutative Union Lemma to the CR proof for β-reduction and the confluence of the η-reduction and the $\beta\eta$-reduction are discussed. Moreover, we analyze the confluence of System F and we carry out a version of the proof directly in the Beluga's meta-logic.

The thesis concludes with a comparison of Beluga's approach with other encodings and notes pros and cons. The thesis then mentions some future work, such as the formalization of CR for Pure Type Systems, a generalization of the lambda calculus.

Overall, the thesis aims to contribute to the growing interest in using machine-checked methods to study meta-theory of formal systems and demonstrates how to formalize the Church-Rosser theorem using Beluga.

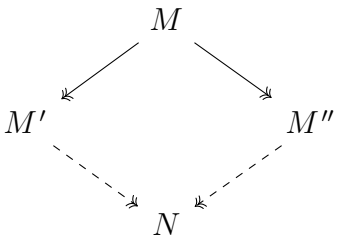
Contents

Chapter 1

Introduction

There are two ways to think about functions. The extensional view is the prevalent one in modern mathematics and it is based on the idea that two functions $f, g : X \to Y$ are equal if they yield on each input the same output, that is, $f(x) = g(x)$ for all $x \in X$ ("functions as graphs"). In computer science, however, it is often convenient to consider "functions as computations": if a computer program is a function that maps inputs to outputs, we are not only interested in its extensional behavior but also about *how* the output is computed. The λ-calculus, introduced by Alonzo Church in the 1930s, is such a theory and it is one of the most important formal systems in theoretical computer science.

The Church-Rosser theorem is one of the most fundamental theorem regarding lambda calculus. It states that if two terms are equivalent (i.e. they are paired by the relation of *conversion*), then there exists another term N and there are two multi-step reductions from both of them to N. This can be proven equivalent to the notion of *confluence*, i.e., if an expression M can be evaluated in two different ways M' and M'', then both will lead to the same result N.



The first efforts in proving this go back to 1936 with Church and Rosser [30]; since then, a lot of work has gone into simplifying the proof, including the improvements introduced by Tait/Martin-Löf (see [6]) and Takahashi [35].

1.1 Motivation

> *"We may think of [the] proof as an iceberg. In the top of it, we find what we usually consider the real proof; underwater, the most of the matter, consisting of all mathematical preliminaries a reader must know in order to understand what is going on."*
>
> - S. Berardi [9]

The PoplMark challenge [4] is a set of *benchmarks* designed to evaluate the state of automated reasoning (or mechanization) in the meta-theory of programming languages, focused on summarizing the best practices for researchers in mechanically checked proof. Moreover, thanks to the PoplMark challenge, mechanization of the meta-theory of programming languages has gained a substantial interest and this resulted in the widespread use of proof assistants to prove properties, for example, of parts of a compiler or of a language design.

The Church-Rosser theorem (CR) is a classic result in the theory of the lambda calculus [30] and can be therefore considered as a benchmark for comparing automated reasoning tools. When it comes to presenting such a result, we need to specify an adequate representation for variables and substitution. If this presentation is done on paper, we can refer to the Barendregt Variable Convention [6], which solves the fundamental issues regarding capture-free substitutions, and α-equivalence.

Translating informal conventions like the one above in a formal setting is a thorny issue, with a by now fifty-years long history. See the PoplMark challenge for a survey (`https://www.seas.upenn.edu/~plclub/poplmark`). One of the solutions which have been formulated is due to Nicolaas G. de Bruijn in 1972, namely, the concept of de Bruijn indices for the first proof-checker AutoMath [11]. However, despite several techniques have been suggested, the problem of handling binders has not found a unanimous solution yet. The choices we make will significantly impact our proof developments, determining how much effort is required, and how feasible, reusable, and extendable they are in practice. Even a seemingly simple mathematical demonstration, such as the Church-Rosser theorem, can expose a multitude of unforeseen complications when considering all the underlying assumptions.

1.2 Related Formalized Work

Given the sophisticated inductive arguments that underlie the proofs of the Church-Rosser theorem for the untyped λ-calculus and its clear mathematical statement, many researchers in theorem proving have been attracted to this challenge. This has led the Church-Rosser theorem to be formalized in pretty much every proof assistant with a

variety of techniques. To name the main approaches, one can choose to use concrete names for variables (named syntax and its modern reconstruction, nominal syntax), a de Bruijn representation, or higher-order abstract syntax (HOAS). The de Bruijn syntax avoids the issue of α-equivalence, however, the named syntax can be considered more corresponding to informal mathematics. Higher-order abstract syntax allows the use of meta-level binders to model object-level binders and this results in freeing the user from dealing with tedious details.

In the beginning, the thrust was to see whether the theorem could be proved at all: the first mechanization of the Church-Rosser proof is due to Shankar [33] in the Boyer-Moore theorem-prover and it was a break-through. He developed the formalization by using a de Bruijn notation, and he made it work by formulating hundreds of small lemmas that could be proved by a fully automatic induction-based theorem prover. Few years later, Nipkow [23; 24] made the proof much more abstract and extended it to $\beta\eta$-reduction, using Isabelle/HOL and the same encoding technique, still with an emphasis on automation. At the same time Huet used Coq [17] formalized the more general *residual theory* of β-reduction, with main results are the Prism Theorem and its corollary Lévy's Cube Lemma which together lead to the confluence theorem and Church-Rosser. This was an interactive proof, again with an overwhelming number of technical lemmas foreign to the mathematics of the problem.

In the following years, the Church-Rosser theorem became a case study to showcase one's clever solution to the variable binding problem. For example, if one is interested in mirroring what is the informal practice of the working mathematicians, see the paper by Vestergaard and Brotherston [36] and the more recent work by Copello, Szasz and Tasistro in Agda [10]. A similar, very concrete approach was pursued by Ford and Mason [15] in PVS, in which they considered the quotient of the named syntax induced by the α-equivalence relation. For the *locally nameless* representation, see McKinna and Pollack in LEGO [20]. One recent example of use of nominal techniques is the proof by Nagele, van Oostrom and Sternagel [21] exploiting the Z-property. For a more recent spin on the de Bruijn syntax, we mention the formalization by Schäfer, Tebbi, Smolka in 2015 [31] in Coq using the library AutoSubst. This is by no means a complete account of all the formalizations available.

Finally, we come to higher-order abstract syntax, which is the technique that we adopt in this thesis. With this approach, the meta-language directly supports common conventions which concern bound variables. In particular, α-conversion of the meta-language models renaming of bound variables and β-reduction in the framework capture-avoiding substitutions, with no need to define and prove them explicitly. The first who exploited HOAS to prove Church-Rosser in a proof-assistant was Pfenning in 1992 in Twelf [25]. We recall also the work done by Accattoli twenty years later, who formalized the residual theory in Abella [2].

1.3 Beluga

The programming and proof environment Beluga ([28], [27]) is a *two-level system*: Beluga employs the logical framework LF ([26]) as its specification layer, i.e., for specifying formal systems via *higher-order-abstract syntax*, in which binders in the object language are represented as binders in the meta-language of LF. Object languages and judgements about them can be encoded as LF types.

Proofs about the encoded object languages are expressed as total programs in *contextual modal type theory* (CMTT) (see [22]), which constitutes the computation level of Beluga. It supports the analysis and the manipulation of LF data via pattern matching, allowing inductive proofs about formal systems in the form of dependently typed recursive functions.

Moreover, Beluga is characterized by its explicit support for *contexts* and *contextual objects*, which are used for hypothetical and parametric derivations. The uniqueness of the Beluga system lies in the fact that it has context variables, allowing abstractions over contexts. A context *schema* can be seen as the contexts counterpart of types: as types classify terms, context schemas classify contexts.

Beluga proper does not provide a tactic language for constructing proofs, thus one need to explicitly specify proof terms, similarly to the Agda system. However, it is possible to mark holes in incomplete proof terms by using the special character **?** and Beluga provides type information in the form of a reasoning context. Finally, Beluga check that the term inhabiting a type is indeed a proof by checking that it is a *total* function.

Note that an interactive prover for Beluga named *Harpoon* ([12]) has been released, which enables the development of proofs using a small set of built-in tactics. We did not use Harpoon in this thesis, since it is still under development.

We will not give further detail on the Beluga system, nor its (complex) metatheory. We refer to its homepage from `http://complogic.cs.mcgill.ca/beluga/`. We will explain by example of Beluga proofs that are written in the next chapters.

Chapter 2

Abstract Reduction Systems

We are interested in a class of mathematical theories, often called *combinatorial*, whose central idea is the definition of an equivalence relation between objects that are obtainable one from the other by applying a certain number of *moves*. Often there is a dichotomy for which the moves can be positive (when they reduce something) or negative (when they add something). Formally speaking, an *abstract reduction system* (ARS) is defined as a pair (A, R), where A is a set and $R \subseteq A \times A$ is a binary relation on A. We will write Rab if $(a, b) \in R$. This relation is called reduction because in many applications two elements are related if the second is obtained from the first one by a positive move, i.e., by "decreasing" something. See [5] for the details.

2.1 Motivation

We can look at an ARS as a directed graph (A, R), where A is the set of nodes and in which two nodes are connected if they are related by R. If $a, b \in A$, Rab does not imply Rba: the order is important. We can then consider the equivalence relation $R^{\equiv}$, derived from R by a reflexive, symmetric, and transitive closure; in particular, $R^{\equiv}ab$ holds if and only if there is a path in the graph between a and b, which can be traversed in both directions. This gives us a notion of equivalence for the elements in A, and models the idea of identity, central to many problems in mathematics and computer science based on equational reasoning. Given the ARS graph, and $a, b \in A$ we would like to decide if the two elements are equivalent. At first, one can think about employing an undirected search algorithm, but it is computationally too expensive. Thus, the idea is to apply unidirectional reduction sequences from both a and b, with subsequent comparison of normal forms; however, this method works only if the reduction system satisfies the so-called Church-Rosser property, named after Alonzo Church and J. Barkley Rosser, who proved it for lambda-calculus (see [30]). In this section, we will introduce several concepts regarding relations in general

that we will then apply to the lambda calculus. In particular, the relations we are going to analyze are the β- and the η-reductions. The reflexive-transitive closure R^* will be the multi-step β-reduction and the equivalence relation $R^{\equiv}$ the *conversion* of lambda-terms. The triangle operator we introduce in Section 2.5 is a general definition for the notion of *complete development* in the context of lambda calculus. Finally, we will apply the results of Section 2.6 about commutation to prove the confluence of $\beta\eta$-reduction.

2.2 Basic Definitions

We will follow the discussion in [34], giving a list of definitions about relations that will be useful for our purposes. Given R a binary relation on a set A we will say:

Definition 2.2.1. *R is* reflexive *if* $\forall x\ Rxx$

Definition 2.2.2. *R is* symmetric *if* $\forall x, y\ Rxy \to Ryx$

Definition 2.2.3. *R is* transitive *if* $\forall x, y, z\ Rxy \wedge Ryz \to Rxz$

Definition 2.2.4. *R is an* equivalence relation *if R is reflexive, symmetric and transitive.*

Furthermore, given R and S relations we want to know when we can consider them extensionally equivalent and thus we need the following:

Definition 2.2.5. *$R \subseteq S$ if R is a subset of S, i.e., $\forall x, y\ Rxy \to Sxy$*

Definition 2.2.6. *$R \cong S$ if $R \subseteq S$ and $S \subseteq R$, i.e., $\forall xy.\ Rxy \leftrightarrow Sxy$*

We then define:

Definition 2.2.7 (Composition of relations)**.** $R \circ S := \{(x, z) \mid \exists y \in B\ (x, y) \in R \wedge (y, z) \in S\}$

Definition 2.2.8 (Union of relations)**.** $R \cup S := \{(x, y) \mid (x, y) \in R \vee (x, y) \in S\}$

Moreover:

Definition 2.2.9 (Inverse relation)**.** $R^- := \{(y, x) \mid (x, y) \in R\}$

Definition 2.2.10 (Symmetric closure)**.** $R^{\leftrightarrow} := R \cup R^-$

Definition 2.2.11 (Reflexive closure)**.** $R^= := R \cup \{(x, x) \mid x \in A\}$

Definition 2.2.12. *The* reflexive transitive closure R^* *of* R *can also be defined inductively by the rules:*

$$\frac{}{R^*yy}\ refl \qquad \frac{Rxy \qquad R^*yz}{R^*xz}\ step$$

In the literature, for example in [25], the reflexive-transitive closure R^* of R as an inductive predicate is sometimes called *multi-step reduction*, especially useful when it comes to mechanizing proofs by induction because the application of the inductive step is easier. It can be proved that this definition agrees with the set-theoretic notion of closure:

Lemma 2.2.1. R^* *is the least reflexive and transitive relation containing* R*.*

Definition 2.2.13 (Equivalence closure)**.** $R^{\equiv} := R^{\leftrightarrow *}$

Again, one can show the equivalence with the closure in the set-theoretic sense:

Lemma 2.2.2. $R^{\equiv}$ *is the least equivalence relation containing* R*.*

2.3 Definition of Confluence and Church-Rosser

To formally define what it means for R to be confluent and to satisfy the Church-Rosser property, we need a few other preliminary notions:

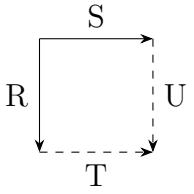
Definition 2.3.1. x *and* y *are* joinable *wrt* R *if* $\exists z, Rxz \wedge Ryz$

Definition 2.3.2. R *satisfies the* diamond property *if* $\forall x, y, z$ *with* $Rxy \wedge Rxz$*, then* y *and* z *are joinable wrt* R

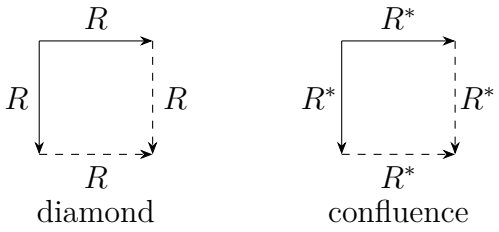
Finally, we have arrived at the core of the dissertation:

Definition 2.3.3. R *is* confluent *if* R^* *satisfies the diamond property*

Every time we formally define confluence-like properties, we can think about it visually through diagrams in the form of squares.



Solid arrows express universal quantification, while dashed arrows represent existence. In particular, we will employ diagrams like the one above to express an implication of the form: $\forall x, y, z,\ Rxy \wedge Sxz \rightarrow \exists w,\ Tyw \wedge Uzw$. Thus, we can visualize the definitions given above as:



To match what we asked for in our initial motivation, we need a slightly different definition, which however turns out to be equivalent to confluence:

Definition 2.3.4. *R satisfies the* Church-Rosser *property if $\forall x, y$ with $R^{\equiv}xy$, then x and y are joinable wrt R^**

If two elements in A are joinable wrt R^*, they are connected by an undirected path and thus equivalent, i.e., being R^* joinable is a sufficient condition for being equivalent; the Church-Rosser property says that this condition is necessary too, and gives us the ability to decide if two elements in A are equivalent by looking for a common successor.

2.4 Basic Results

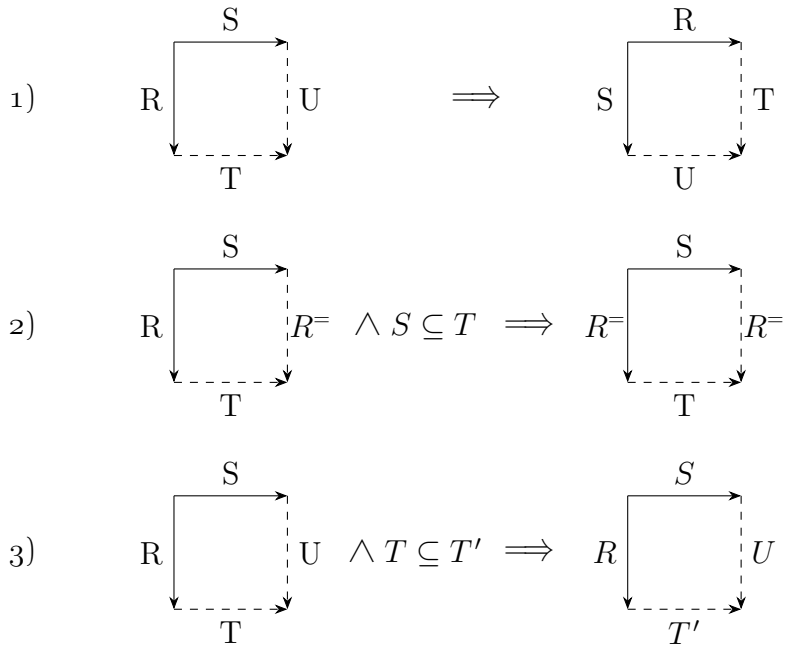
There are a series of interesting results we are going to use in the following; firstly we have a couple of lemmas regarding the reflexive-transitive closure:

Lemma 2.4.1 (monotonicity of $*$)**.** *If $R \subseteq S$ then $R^* \subseteq S^*$.*

Lemma 2.4.2 (idempotence of $*$)**.** *$R^{**} \cong R^*$.*

Moreover, we state and prove a few properties regarding squares as defined above:

Lemma 2.4.3. *The following basic properties hold:*



Proof. 1) : The proof follows by simple logical manipulations. 2) : By definition, $R^{=}xy \leftrightarrow Rxy \vee x = y$. The case $Rxy \wedge Sxz$ is true by assumption. Otherwise, if $x = y$, we have $R^{=}xx \wedge Sxz$. But $S \subseteq T$ means $Sxz \rightarrow Txz$ and thus, taking $w = z$, we obtain the result.

3) : It suffices to observe that $T \subset T'$ means that $Txy \rightarrow T'xy$. □

Looking at the definitions above it is trivial to see that being Church-Rosser implies being confluent, by choosing a specific path for the assumption of equivalence. What is less obvious is that the vice versa holds too. To prove it we need a third notion:

Definition 2.4.1. *R is* semi-confluent *if $\forall x, y, z$ with $R^*xy \wedge Rxz$, then x and y are joinable wrt R^**

At a glance, this property seems weaker than confluence but it turns out to be equivalent; namely:

Theorem 2.4.1. *The following are equivalent:*

1) R has the Church-Rosser property

2) R is confluent

3) R is semi-confluent

Proof. Obviously, it is easy to see 1) $\rightarrow$ 2) $\rightarrow$ 3). To conclude the proof, we show 3) $\rightarrow$ 1). Let R be a semi-confluent relation and let x, y such that $R^{\equiv}xy$, i.e., $R^{\leftrightarrow *}xy$. We proceed by induction on the structure of the deduction $\mathcal{R} :: R^{\leftrightarrow *}xy$. We have two cases:

- $\mathcal{R} = refl$: then $x = y$ and of course x is joinable with itself by $refl$.
- $\mathcal{R} = (step\ \mathcal{R}_1\ \mathcal{R}_2)$ with $\mathcal{R}_1 :: R^{\leftrightarrow}xx'$ and $\mathcal{R}_2 :: R^{\leftrightarrow *}x'y$: by IH we have that exists z such that $\mathcal{R}'_2 :: R^*x'z$ and $\mathcal{R}''_2 :: R^*yz$; recall that $R^{\leftrightarrow}xx' \leftrightarrow Rxx' \vee R^{-}xx'$:
  - if $\mathcal{R}'_1 :: Rxx'$, by considering $\mathcal{S}' = (step\ \mathcal{R}'_1\ \mathcal{R}'_2) :: R^*xz$ together with $\mathcal{R}''_2$ the claim follows;
  - if $\mathcal{R}'_1 :: R^{-}xx' \leftrightarrow Rx'x$, by semi-confluence of R there exists t such that $\mathcal{S}_1 :: R^*xt$ and $\mathcal{S}_2 :: R^*zt$. By transitivity of $*$ applied to $\mathcal{R}''_2$ and $\mathcal{S}_2$ the claim follows.

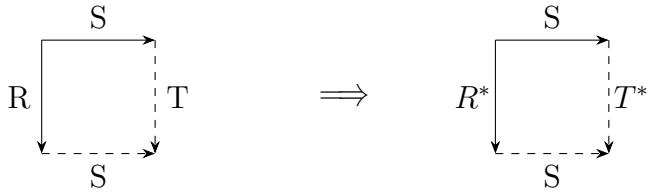
□

It can easily be proved that:

Lemma 2.4.4. *If R satisfies the diamond property, then R is semi-confluent.*

Actually, we can show that the diamond property directly implies confluence, without going through the definition of semi-confluence, but to prove it we will need a preliminary property; we state and prove what is known in the lambda calculus as the *Strip Lemma* but can be proven for general relations.

Lemma 2.4.5. *The generalized Strip Lemma affirms that:*

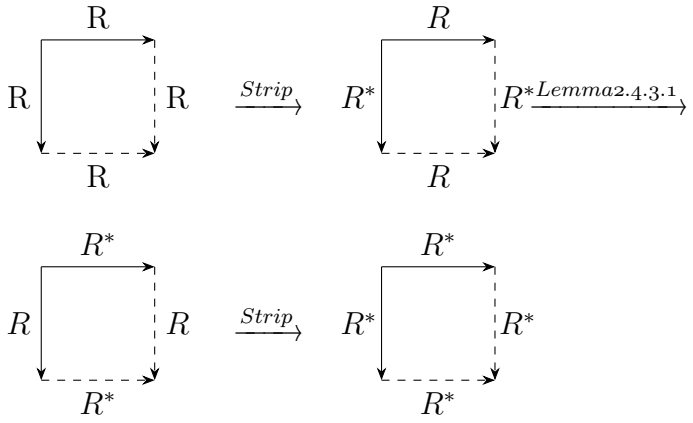


Proof. We proceed by induction on $\mathcal{R} :: R^*xy$: if $\mathcal{R} = refl$, then $x = y$ and taking $w = z$, we obtain the result, with Sxz which holds by assumption, and T^*zz which is true by $refl$. If $\mathcal{R} = (step\ \mathcal{R}_1\ \mathcal{R}_2)$ with $\mathcal{R}_1 :: Rxx'$ and $\mathcal{R}_2 :: R^*x'y$: by assumption, we have w' such that $Sx'w'$ and $\mathcal{T}_1 :: Tzw'$. By IH, there exists w'' with $\mathcal{S} :: Syw''$ and $\mathcal{T}_2 :: Tw'w''$. The thesis is true by taking $\mathcal{S}$ and $\mathcal{T} = (step\ \mathcal{T}_1\ \mathcal{T}_2) :: T^*zw''$. □

We are now able to show that:

Lemma 2.4.6. *If R satisfies the diamond property, then R is confluent.*

Proof. We exploit the diagrammatic proof method:



□

2.5 TMT Method

Among a series of different methods to show confluence, there is a clever one for the lambda calculus due to Tait, Martin-Löf, and Takahashi (see e.g. [35]): the TMT method. The TMT method can be decomposed in two parts:

1. an abstract part with no assumptions about terms; with this part, we obtain confluence and a reduction function;

2. a concrete one where we fix a set of terms and the relevant reductions relations.

We will now focus on the abstract part. The key idea is to show the confluence of a relation R by identifying a suitable auxiliary relation S such that $R \subseteq S \subseteq R^*$, with S satisfying the diamond property.

In fact, we can prove that:

Lemma 2.5.1 (Sandwich)**.** *If $R \subseteq S \subseteq R^*$, and S satisfies the diamond property, then R is confluent.*

Proof. By monotonicity of $*$, we have that $R^* \subseteq S^* \subseteq R^{**}$. By idempotence, we know $R^{**} \subseteq R^*$ and thus we obtain $R^* \cong S^*$. This is true if $\forall x, y\ R^*xy \leftrightarrow S^*xy$. Moreover, by Lemma 2.4.6 we obtain that S is confluent. Thus, supposing $R^*xy \wedge R^*xz$, we have that $S^*xy \wedge S^*xz$. But S is confluent, hence there exists w with $S^*yw \wedge S^*zw$. But $R^* \cong S^*$ and hence $R^*yw \wedge R^*zw$, i.e. R confluent. □

Definition 2.5.1. *A* triangle operator for R *(also called* Takahashi function for R*) is a function* $\rho : A \to A$ *such that for all* $x, y \in A$ *if* Rxy *then* $Ry(\rho x)$

It is straightforward to show that:

Lemma 2.5.2. *If* ρ *is a triangle operator for* R*, then* R *satisfies the diamond property.*

Proof. We suppose $Rxy \wedge Rxz$ and want to find an element w such as $Ryw \wedge Rzw$. By definition, ρ is a triangle operator for R, and then $Ry(\rho x) \wedge Rz(\rho x)$, i.e., R satisfies the diamond property with $w = \rho x$. □

We now have all the ingredients to state and prove the following:

Theorem 2.5.1 (Triangle)**.** *If* $R \subseteq S \subseteq R^*$ *and* ρ *is a triangle operator for* S*, then* R *is confluent.*

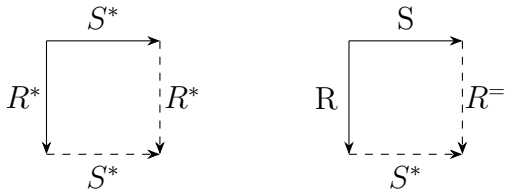
Proof. We start from the ρ triangle operator for S. Then, by Lemma 2.5.2, S satisfies the diamond property. By Lemma 2.4.6, we obtain the confluence of S. Lastly, we prove the result by the Sandwich Lemma (2.5.1). □

2.6 Confluence and Commutation

There are many techniques we can employ to show confluence. Sometimes, it is convenient to consider a reduction R as the union of smaller reductions $R_1, \ldots, R_n$; the idea is trying to prove R confluent by showing their confluence separately. To conclude the proof, however, we need a property of *commutation* among $R_1, \ldots, R_n$. We focus on the case $R = R_1 \cup R_2$ which is however easily generalizable:

Definition 2.6.1. R *and* S commute *if* $\forall x, y, z,\ R^*xy \wedge S^*xz \to \exists w,\ S^*yw \wedge R^*zw$

Definition 2.6.2. R *and* S strongly commute *if* $\forall x, y, z,\ Rxy \wedge Sxz \to \exists w,\ S^*yw \wedge R^=zw$



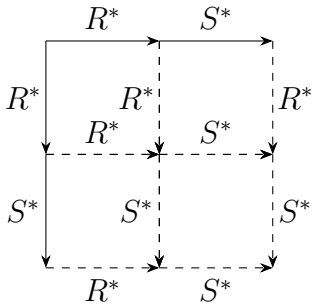
commute strongly comm.

Observe that commutation is a generalization of confluence: in fact, we can say that R is confluent if and only if R commutes with itself.

The main result of this section is the *Commutative Union Lemma* due to Hindley and Rosen (see, e.g., [29]) which basically says that union preserves confluence if the reductions involved commute, that is:

Theorem 2.6.1 (Commutative Union Lemma)**.** *If R and S are confluent and commute, then $R \cup S$ is confluent.*

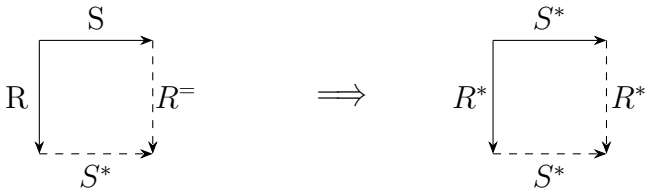
Proof. Note that every $(R \cup S)^*$ reduction can be written as a sequence $R^* \circ S^* \circ R^* \circ \ldots \circ S^*$ of alternating $R*$ and $S*$ reductions. The proof is then based on an easily provable observation: $R \cup S \subseteq R^* \circ S^* \subseteq (R \cup S)^*$[1]. We only need to prove the diamond property for $R^* \circ S^*$ and we do it visually:



where the top-left square is true by the confluence of R, the top-right and the bottom-left squares by the commutativity between R and S, and the bottom-right by the confluence of S. □

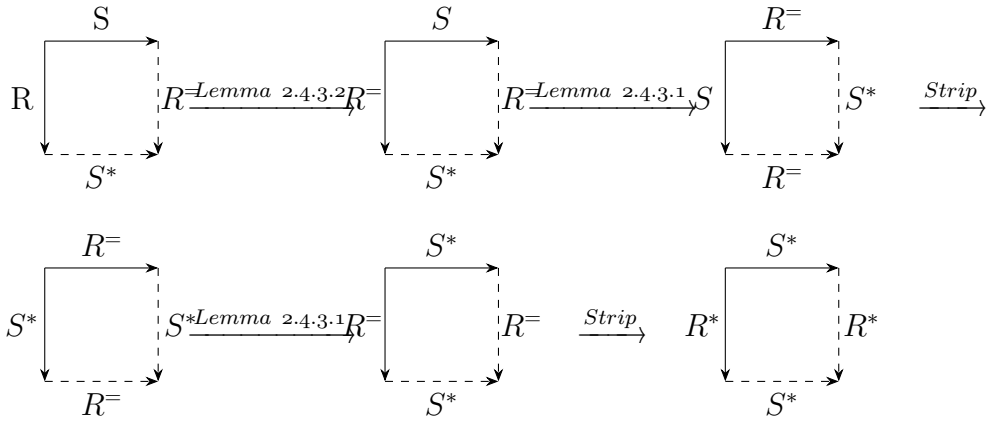
When it comes to proving commutation, there is another very useful lemma due to Hindley and Rosen ([29]) called *Commutation Lemma*:

Lemma 2.6.1. *Two strongly commuting reductions commute, i.e.,*



Proof. We exploit again the diagrammatic proof method:

[1]This reminds us of the assumption in the Sandwich Lemma 2.5.1 which allows us to conclude the confluence of $R \cup S$ by proving $R^* \circ S^*$ confluent. Recall that, by Lemma **??**, if we are able to prove that $R^* \circ S^*$ satisfies the diamond property, we can obtain the confluence of $R^* \circ S^*$.


S
R
$R^=$
S^*
Lemma 2.4.3.2
S
$R^=$
S^*
Lemma 2.4.3.1
$R^=$
S^*
$R^=$
Strip
$R^=$
S^*
$R^=$
Lemma 2.4.3.1
S^*
$R^=$
S^*
Strip
S^*
R^*
S^*
R^*

□

Chapter 3

Mechanization via Beluga

3.1 Confluence for β-reduction

We want to give a detailed explanation of how the concepts we introduced in the previous Chapter 2 can be instantiated in the case of the untyped lambda calculus. Furthermore, we will explore the use of Beluga for the formalization of this theory. In particular, we will develop proofs and representations of the lemmas implying the Church-Rosser theorem for β-reduction.

The approach for the representation of meta-theorems in a proof assistant like Beluga can be divided into three different stages: representation of the abstract syntax of the object language (the untyped lambda calculus), its semantics, and meta-theory. In the first stage, we exploit the idea of higher-order abstract syntax which allows us to avoid explicit renaming of bound variables and have a notation for capture avoiding substitution. The semantics of the object language is then specified using judgments in LF defined by inference rules. These two first stages are carried out in the Logical Framework LF. The third stage, that is, the formalization of the proofs of meta-theorems like Church-Rosser, consists instead of the definition of recursive functions which relate deductions in CMTT.

After a brief introduction regarding the definition and representation of the abstract syntax, we will give a detailed proof based on the TMT method of Section 2.5 applied to the Church-Rosser theorem in the case of β-reduction.

3.1.1 Lambda terms

The logical framework LF([26]), which Beluga uses as a specification language, allows the definition of new type families with corresponding constants. These new type families live in the host universe `type`. To represent lambda-terms in Beluga, we introduce a type `term` together with the constructors `app` and `lam`. To represent this

grammar, however, we need to consider the issue of representing variables. BELUGA supports higher-order abstract syntax which allows us to represent object-level binders via binders in our meta-language.

We can define an LF type `term` for the following grammar of the untyped lambda calculus:

Definition 3.1.1.

$$\boxed{\texttt{term}} \quad M, N ::= x \mid \lambda x.M \mid M\ N$$

```
LF term : type =
| app : term  → term  → term
| lam : (term → term) → term
;
```

Note how the variable case is not needed thanks to the HOAS encoding.

3.1.2 β-reduction

Textbook presentations of the lambda-calculus are rather loose about the issue of bound variables. Consider the congruence rule for abstraction for β-reduction, traditionally known as ξ. When encoding this judgment in a proof assistant that supports HOAS, we have to be more careful. In the aforementioned rule, we introduce a new parameter x and substitute it on both sides, with the proviso that it does not already occur in M or M'. The premise is a judgment *parametric* in x, i.e., we can substitute any term N for x in the deduction of $M \longrightarrow_\beta M'$ to obtain a deduction of $M[N/x] \longrightarrow_\beta M'[N/x]$. To make the relationship between the terms M and M' more explicit, we re-formulate the $lm1$ premise using the judgment $\Gamma, x \vdash M \longrightarrow_\beta M'$, which can be read as "$M \longrightarrow_\beta M'$ in the context Γ, x"; the conclusion in this explicit form will be $\Gamma \vdash \lambda x.M \longrightarrow_\beta \lambda x.M'$. We then extend it to the other rules.[1]

[1] Implicit vs. explicit presentation of judgments is discussed at length in [14]

$\boxed{\Gamma \vdash M \longrightarrow_\beta M'}$: Term M steps to term M' in the context Γ

$$\frac{}{\Gamma \vdash (\lambda x.M)N \longrightarrow_\beta M[N/x]}\,\beta_1$$

$$\frac{\Gamma, x \vdash M \longrightarrow_\beta M'}{\Gamma \vdash \lambda x.M \longrightarrow_\beta \lambda x.M'}\,lm_1$$

$$\frac{\Gamma \vdash M \longrightarrow_\beta M'}{\Gamma \vdash M\ N \longrightarrow_\beta M'\ N}\,apl_1$$

$$\frac{\Gamma \vdash N \longrightarrow_\beta N'}{\Gamma \vdash M\ N \longrightarrow_\beta M\ N'}\,apr_1$$

To represent single-step β-reduction, we define a type family `beta_red` and state that it takes two terms as an argument by declaring its type as `term → term → type`. We then have four constructors, each one corresponding to one of the rules in the operational semantics. The representation of an inference rule can be thought of as a function from deductions of its premises to a deduction of its conclusion. In particular, to represent the rule $beta_1$, which has zero premises, we define a constant. We then exploit the fact that the first argument to the constructor `lam` is an LF function. Thus, we can obtain the substitution $[M_2/x]M_1$ by exploiting the function application `M1 M2`.

```
beta1 : beta_red (app (lam M1) M2) (M1 M2)
```

The premise of the lm_1 rule can be seen as a product type $\Pi x : A.B$, and we represent it by using the notation $\{x : A\}B$, which corresponds to an LF function that accepts an object N of type A and returns an object of type $[N/x]B$.

```
LF beta_red : term → term → type =
| beta1 : beta_red (app (lam M1) M2) (M1 M2)
| lm1   : ({x:term} beta_red (M x) (M' x)) → beta_red (lam M) (lam M')
| apl1  : beta_red M1 M1' → beta_red (app M1 M2) (app M1' M2)
| apr1  : beta_red M2 M2' → beta_red (app M1 M2) (app M1 M2')
;
```

In the following, we will need to introduce several closure operators. Since Beluga does not support polymorphism, those operators will need to be re-defined for each reduction relation, causing some tedious but unproblematic duplication of code.

We start with the reflexive closure of β-reduction represented as the least reflexive relation containing `beta_red`. Note that being the "least" relation is a meta-theoretical property that is not enforced at the LF level.

```
LF beta_red= : term → term → type =
| id_b= : beta_red= M M
```

```
| cl_b= : beta_red M N → beta_red= M N
;
```

Next, the reflexive-transitive closure, which corresponds to multi-step reduction:

```
LF beta_red* : term → term → type =
| id_b* : beta_red* M M
| tr_b* : beta_red* M1 M2 → beta_red* M2 M3
        → beta_red* M1 M3
| cl_b* : beta_red M N → beta_red* M N
;
```

It is convenient to reformulate multi-step reduction $M \longrightarrow_\beta^* M'$ as follows

$$\frac{}{\Gamma \vdash M \longrightarrow_\beta^* M}\, id_1 \qquad \frac{\Gamma \vdash M \longrightarrow_\beta N \quad \Gamma \vdash N \longrightarrow_\beta^* M'}{\Gamma \vdash M \longrightarrow_\beta^* M'}\, step_1$$

This gives us a cleaner induction principle. The rules above are represented in LF as follows:

```
LF mstep : term → term → type =
| id1 : mstep M M
| step1 : beta_red M M' → mstep M' M''
          → mstep M M''
;
```

It is straightforward to show that these two definitions are equivalent, see Appendix A for a complete proof.

```
multi_star : (g:ctx) [g ⊢ mstep M N] → [g ⊢ beta_red* M N]
star_multi : (g:ctx) [g ⊢ beta_red* M N] → [g ⊢ mstep M N]
```

The Church-Rosser theorem needs a notion of equivalence among terms; we thus define *β-conversion* as the smallest equivalence relation on terms that contains the reduction relation. This can be expressed as an inference system with four rules:

$$\frac{}{\Gamma \vdash M \longleftrightarrow M}\, ref$$

$$\frac{\Gamma \vdash M \longleftrightarrow N}{\Gamma \vdash N \longleftrightarrow M}\, sym$$

$$\frac{\Gamma \vdash M \longleftrightarrow M' \qquad \Gamma \vdash M' \longleftrightarrow N}{\Gamma \vdash M \longleftrightarrow N}\, trans$$

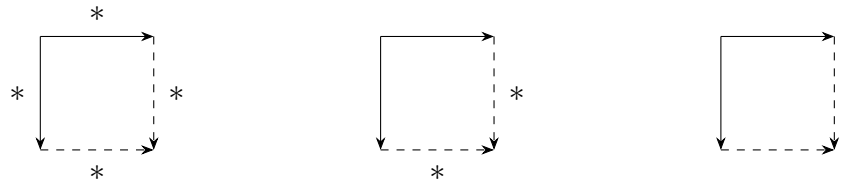
$$\frac{\Gamma \vdash M \longrightarrow^* N}{\Gamma \vdash M \longleftrightarrow N}\, red$$

The conversion relation is represented in BELUGA as follows:

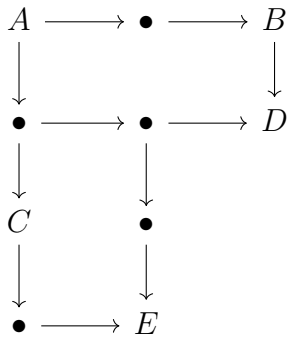
```
LF conv : term → term → type =
| ref : conv M M
| sym  : conv M' M → conv M M'
| trans: conv M' M'' → conv M  M'
         → conv M  M''
| red  : mstep M M' → conv M M'
;
```

3.1.3 The TMT Method

Take any binary relation $\longrightarrow$ on a set, and let $\longrightarrow^*$ be its reflexive transitive closure. Consider the following three properties of such relations:



We recognize (a) as confluence and (c) as the diamond property for $\longrightarrow$ (see Section 2.3). One can naively attempt to prove the Church-Rosser Theorem for β-reduction by firstly proving that $\longrightarrow_\beta$ satisfies (b), which is not difficult to show, and then applying an inductive argument to conclude (a). However, this idea does not work since in general (b) does not imply (a). Consider as an example the following reduction:



This reduction clearly satisfies property (b), but it is not confluent: $A \longrightarrow^* B$ and $A \longrightarrow^* C$, but B and C are not joinable, since the only possible reductions from B and C are respectively $B \longrightarrow D$ and the one that leads to E in two steps.

On the other hand, we have already seen (Lemma 2.4.6) that the diamond property (c) does imply (a); unfortunately, β-reduction does not satisfy (c). One solution to this dilemma has been given by Tait, Martin-Löf, and Takahashi (hence the TMT moniker) that we introduced in section 2.5. The key idea is to show the confluence of a reduction by identifying an auxiliary relation $\Longrightarrow$ which satisfies the diamond property, and whose multi-step reduction $\Longrightarrow^*$ is equivalent to multi-step β-reduction (see Sandwich Lemma 2.5.1). One such auxiliary relation is *parallel* β-reduction.

3.1.4 Parallel β-reduction

The idea behind parallel reduction is that, while contracting a redex, we can also reduce the terms involved in it. Furthermore, the congruence rule for application is generalized so we can reduce both branches in parallel. Finally, we want parallel reduction to be reflexive. In the literature, this is required for variables only, i.e., we want $x \Longrightarrow x$ for every variable x.

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

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

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

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

The rule for variables requires some ingenuity in the HOAS paradigm underlying Beluga. In fact, variables of the object language are represented by meta-variables and thus we would need explicit constructors for them we could match against, which we do not want. We solve this issue by extending the judgment we are defining by hypotheses. Namely, every time we want to derive $M \Longrightarrow M'$, we want to be able to exploit the hypothesis $x \Longrightarrow x$. We can then formulate the *beta* rule in a way that is closer to the Beluga implementation as follows:

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

We represent the derivation $\mathcal{R}$ of the left premise as a function whose arguments are a term x and a deduction u of $x \Longrightarrow x$. By applying this function to a term N and a deduction $\mathcal{S} :: N \Longrightarrow N$ we obtain a derivation of $[N/x]M \Longrightarrow [N/x]M'$; this is accomplished by exploiting the substitution principle. We can repeat the same argument as above for the *lm* rule: whenever a variable x is introduced, we have to assume that $x \Longrightarrow x$ holds.

As we did for the ordinary β-reduction, we define a type family `pred` which takes two terms as an argument `term → term → type`.

```
LF pred : term → term → type =
| beta : ({x:term} pred x x → pred (M1 x) (M1' x)) → pred M2 M2'
         → pred (app (lam M1) M2) (M1' M2')
| lm : ({x:term} pred x x → pred (M x) (M' x))
       → pred (lam M) (lam M')
| ap : pred M1 M1' → pred M2 M2'
       → pred (app M1 M2) (app M1' M2')
;
```

Once again, we introduce multi-step parallel reduction:

```
LF mstep_pred : term → term → type =
| id : mstep_pred M M
| step : pred M M' → mstep_pred M' M''
          → mstep_pred M M''
;
```

3.1.5 The Church-Rosser property for β-reduction

Now that we have defined parallel β-reduction, we have to demonstrate that it is a suitable auxiliary relation to conclude Church-Rosser for ordinary β-reduction. This translates into proving that it meets both the requirements in the Sandwich Lemma 2.5.1; namely, we need to show firstly that $\longrightarrow_\beta \subseteq \Longrightarrow \subseteq \longrightarrow_\beta^*$, and secondly that $\Longrightarrow$ satisfies the diamond property. Once we have proven these two lemmas, we can easily show confluence for ordinary β-reduction and finally Church-Rosser as a corollary.

Preliminary lemmas

The explicit formulation of proof terms and the peculiarities of Beluga 's metatheory can make proofs in Beluga difficult to understand. To help with this, we will start by considering the implementation of a simple lemma as a toy example. This will allow us to better understand what is a proof in Beluga before studying the full Church-Rosser result.

Lemma 3.1.1 (Reflexivity of $\Longrightarrow$)**.** *For any term M, $M \Longrightarrow M$.*

Proof. By induction on the structure of M.

Case $M = x$, x variable

To conclude this case it suffices to apply the variable inference rule obtaining $x \Longrightarrow x$.

Case $M = \lambda x.M'$

Proved by appealing to the induction hypothesis on M'. We then obtain a deduction $\mathcal{R}' :: M' \Longrightarrow M'$. By applying the lm rule we close this case.

Case $M = M_1\ M_2$

We appeal to the induction hypothesis twice using both M_1 and M_2, and combining the two resulting deductions $\mathcal{R}_1 :: M_1 \Longrightarrow M_1$ and $\mathcal{R}_2 :: M_2 \Longrightarrow M_2$ with the ap rule, we conclude the proof. □

When representing this lemma in the language of CMTT, we have to deal with the LF meta-variable `M` at the reasoning level. To do this, we need to wrap statements about it as contextual objects. Moreover, an inductive proof like the one above can be interpreted as a recursive function where case analysis in the proof corresponds to pattern matching in the program and the application of the induction hypothesis corresponds to making a recursive call.

During the induction, we need to consider open terms, viz. the abstraction case above, where the binder is opened and the context is extended accordingly. Thus, we need to quantify over a context and hence define a context schema. Beluga's contexts can be seen as ordered sequences of named records, or *blocks*. The least amount of information that such a record can contain is the existence of a certain LF variable, in this case, `x:term`. It is also possible to package additional assumptions together with the argument by using the keyword **`block`**. Conceptually, every **`block`** amounts to a dependent sum type. *Schema ascription* is a typing mechanism provided by Beluga to deal with contexts, and as types classify terms then schemas classify contexts.

In our representation of the parallel reduction, to include the *var* case in a HOAS system, we formulated the rules in such a way that whenever a variable x is introduced, we assume $x \Longrightarrow x$. To denote that these two assumptions always come in pairs, we define the schema by exploiting the **`block`** construct:

```
schema rctx = block(x:term, t:pred x x);
```

In particular, the schema `rctx` describes a context containing assumptions `x:term`, each associated with a derivation `t:pred x x`. This gives us for free the fact that for every variable there is a unique reduction derivation.

Next, the BELUGA version of this lemma:

Lemma 3.1.2 (Reflexivity of $\Longrightarrow$ in BELUGA)**.** *There exists a total function* `refl_par` *that satisfies the following typing:*

```
refl_par : {g:rctx}{M : [g ⊢ term]}[g ⊢ pred M M]
```

Proof. The process of proving is generally carried out by stating a desired conclusion and working back to known premises. This often needs intermediate sub-goals which may only be partially solved, these are encoded using 'holes'. The theory around this was developed in [19]. Similarly to what happens for other proof checkers like Agda and Idris, in BELUGA it is possible to build proofs interactively by marking holes in incomplete proof terms by using the special character **?** obtaining type information in the form of a reasoning context. Our basic setup looks like this:

```
rec refl_par : {g:rctx}{M : [g ⊢ term]}[g ⊢ pred M M] =
mlam g ⇒ mlam M ⇒ ?
```

The type is universally quantified over a context `g` and the contextual object `M : [g ⊢ term]`. In the program, we use **`mlam`**-abstractions as the corresponding proof term for introducing universal quantifiers; this is necessary since we quantify over `M` and `g`.

We now consider all possible forms of `[g ⊢ M]`.

Variable case:

Parameter variables `#p` allow us to consider a generic case that matches a declaration **`block`**`(x:term, t: pred x x)`. Since our pattern match proceeds on the structure of terms, we want the first component of the parameter `#p`, written as `#p.1` or `#p.x`. We then have to exhibit a witness for `[g ⊢ pred #p.1 #p.1]`, but this is the unique reduction tied to `#p.1`, i.e., `[g ⊢ #p.2]`.

Application case:

In this case we have `M = [g ⊢ app M1 M2]`, with `M1, M2 : [g ⊢ term]`. The application case is easier since we can directly appeal to the induction hypothesis to both `[g ⊢ M1]` and `[g ⊢ M2]`, without changes in the context, obtaining `[g ⊢ IH1] : [g ⊢ pred z z]` and `[g ⊢ IH2] : [g ⊢ pred z1 z1]` respectively. It suffices to exhibit `[g ⊢ ap IH1 IH2] : [g ⊢ pred (app z z1) (app z z1)]` to conclude the proof.

Abstraction case:

We eventually consider the abstraction case, that is, the case in which `M = [g ⊢ lam \x.M']`; here, the binder is opened and thus the context is extended with `M' : [g, x:term ⊢ term]`. We would like to appeal to the induction hypothesis using `M'`; BELUGA supports declaration weakening and this allows us to use `M'` in the extended context `g,b:block(x:term,u:pred x x)`. To retrieve `x` and `u` we take the first and the second projections `b.1` and `b.2` respectively. To move from context `g, x:term` to context `g,b:block(x:term,u:pred x x)` we construct a weakening substitution `[…,b.1]` that simply renames `x` to `b.1` in `M'`; Thus, we appeal to the induction hypothesis using `M'[…,b.1]`. We name the resulting derivation `[g, b:block(x:term,u:pred x x) ⊢ IH[…,b.1,b.2]]`. This means that `IH : [g,x:term,u:pred x x ⊢ pred M'[…,x] M'[…,x]]`. Now, exploiting the substitution principle underlying the definition of the reduction rule `lm`, we have that `[g ⊢ lm \x.\u.IH[…,x,u]]` is a derivation of `[g ⊢ pred (lam (\x. M')) (lam (\x. M'))]` and we conclude this case.

We now put everything together to obtain the following recursive function

```
rec refl_par : {g:rctx}{M : [g ⊢ term]}[g ⊢ pred M M] =
/ total m (refl_par _ m) /
mlam g ⇒ mlam M ⇒
        case [g ⊢ M] of
        | [g ⊢ #p.1] ⇒ [g ⊢ #p.2]
        | [g ⊢ lam \x.M'[…,x]] ⇒
            let [g, b:block(x:term,v:pred x x) ⊢ IH[…,b.1,b.2]] =
            refl_par [g, b:block(x:term,v:pred x x)] [g,b ⊢ M'[…,b.1]] in
            [g ⊢ lm \x.\v.IH[…,x,v]]
        | [g ⊢ app M1 M2] ⇒
            let [g ⊢ IH1] = refl_par [g] [g ⊢ M1] in
            let [g ⊢ IH2] = refl_par [g] [g ⊢ M2] in
            [g ⊢ ap IH1 IH2]
;
```

□

While this term type does inhabit the theorem, we need a final ingredient to make sure that it is indeed a proof of the desired statement. A (recursive) term is a proof only if it is a total function, i.e., it must be defined on all inputs and it must be terminating. The totality annotation `/ total M (refl_par _ M) /` instructs BELUGA 's type checker to verify that all recursive calls are structurally decreasing in the second argument `[g ⊢ M]` and that the case analysis is complete.

For another detailed example, we show that multi-step reduction is transitive with the relative representation in BELUGA.

Lemma 3.1.3 (Transitivity of $\longrightarrow_\beta^*$)**.** *If* $M \longrightarrow_\beta^* M'$ *and* $M' \longrightarrow_\beta^* M''$*, then* $M \longrightarrow_\beta^* M''$.

In this case, the implementation could be carried out with an empty context. However, we want to generalize the lemma and talk about open terms, thus we define the following:

```
schema ctx = term;
```

In this way, we declare well-formed contexts as the ones containing only declarations of type `term`.

The representation in BELUGA looks as follows:

```
trans_mstep : (g:ctx)[g ⊢ mstep M M'] → [g ⊢ mstep M' M'']
                                      → [g ⊢ mstep M M'']
```

Proof. Our basic setup looks like this:

```
rec trans_mstep : (g:ctx)[g ⊢ mstep M M'] → [g ⊢ mstep M' M'']
                                      → [g ⊢ mstep M M''] =
/ total d (trans_mstep _ _ _ _ d _) /
fn d ⇒ fn s ⇒ ?
```

The type is implicitly universally quantified over a context `g` and three contextual objects `M`, `M'`, `M''` : `[g ⊢ term]`. To specify its schema, we need to give the context explicitly but it will be treated as an implicit argument like the other three when we apply `trans_mstep`. Both the derivations `d : [g ⊢ mstep M M']` and `s : [g ⊢ mstep M' M'']` are introduced into the local context with an **fn**-abstraction that is the proof term for → introduction. We now consider all possible forms of `d`. Besides generating all the cases, pattern matching in `d` refines what `M` and `M'` stand for. In the first case, we have `d = [g ⊢ id1] : [g ⊢ mstep x x]` and `M = M' = x`; what we need to provide is a contextual object of type `[g ⊢ mstep x M'']`. However, we already have a contextual object of this type and it is simply `s`.

In the second case, we have `d = [g ⊢ step1 D1 D2]` with `[g ⊢ D1] : [g ⊢ step y M '1]` and `[g ⊢ D2] : [g ⊢ mstep M'1 z]`.; We then appeal to the induction hypothesis using `D2` and `s`. This corresponds to making a recursive call `trans_mstep [g ⊢ D2] s` and we name the resulting derivation `[g ⊢ IH]`. Finally, we construct our derivation `[g ⊢ step1 D1 IH]` for `[g ⊢ mstep y M'']`. The complete implementation reads as follows:

```
rec trans_mstep : (g:ctx) [g ⊢ mstep M M'] → [g ⊢ mstep M' M'']
                          → [g ⊢ mstep M M''] =
/ total d (trans_mstep _ _ _ _ d _) /
fn d ⇒ fn s ⇒ case d of
| [g ⊢ id1] ⇒ s
| [g ⊢ step1 D1 D2] ⇒ let [g ⊢ IH] = trans_mstep [g ⊢ D2] s in
                       [g ⊢ step1 D1 IH]
;
```

□

To carry out the proof of the equivalence between multi-step parallel and ordinary β-reduction, we need to show that $\longrightarrow_\beta^*$ is a congruence. We say an inference rule to be *admissible* if any instance of the rule is derivable.

Lemma 3.1.4 (Congruence of $\longrightarrow^*$)**.** *The inference rules given below are admissible:*

$$\frac{\Gamma \vdash M_1 \longrightarrow^* M_1'}{\Gamma \vdash M_1\ M_2 \longrightarrow^* M_1'\ M_2}\ apl1* \qquad \frac{\Gamma \vdash M_2 \longrightarrow^* M_2'}{\Gamma \vdash M_1\ M_2 \longrightarrow^* M_1\ M_2'}\ apr1*$$

$$\frac{\Gamma, x \vdash M \longrightarrow^* M'}{\Gamma \vdash \lambda x.M \longrightarrow^* \lambda x.M'}\ lm1*$$

In Beluga we will then have the following implementation of the three rules:

```
apl1* :  (g:ctx) {M2:[g ⊢ term]}[g ⊢ mstep M1 M1']
                 → [g ⊢ mstep (app M1 M2) (app M1' M2)]

apr1* : (g:ctx){M1:[g ⊢ term]} [g ⊢ mstep M2 M2']
                → [g ⊢ mstep (app M1 M2) (app M1 M2')]

lm1* : (g:ctx) [g,x:term ⊢  mstep M M']
                →  [g ⊢ mstep (lam \x.M) (lam \x.M')]
```

Proof. We give a full proof for the `lm1*` case, the other two are very similar. As before, we introduce the derivation `d : [g,x:term ⊢  mstep M M']` into the local context with an explicit abstraction. We now proceed by analyzing the two possible cases of `d`. If `d = [g,x:term ⊢ id1] : [g,x:term ⊢  mstep y y]`, with `y` being a contextual object of type `[g,x:term ⊢ term]`, by unification of the pattern matching `M = M' = y`. Thus, to conclude this case, we provide as a witness the derivation `[g ⊢ id1] : [g ⊢ mstep (lam (\x. y)) (lam (\x. y))]`. In the second and last case, we have `d = [g,x:term ⊢ step1 D1 D2]` with `[g,x ⊢ D1] : [g,x ⊢ step1 y M'1]` and `[g,x ⊢ D2] : [g,x ⊢ mstep M'1 z]`. By appealing to the induction hypothesis using `[g, x ⊢ D2]`, we obtain `[g ⊢ IH] : [g ⊢ mstep (lam \x.M'1) (lam \x.z)]`. Finally, by exploiting the `step` constructor `lm1` we construct the derivation `[g ⊢ step1 (lm1 \x.D1) IH]` of `[g ⊢ mstep (lam \x.y) (lam \x.z)]`. We provide now for the complete formalization:

```
rec lm1* : (g:ctx) [g,x:term ⊢  mstep M M']
      →  [g ⊢ mstep (lam \x.M) (lam \x.M')] =
/ total d (lm1* _ _ _ d) /
fn d ⇒ case d of
| [g,x:term ⊢  id1] ⇒ [g ⊢ id1]
| [g,x:term ⊢  step1 D1 D2] ⇒  let [g ⊢ IH] = lm1* [g, x ⊢ D2] in
                                [g ⊢ step1 (lm1 \x.D1) IH]
;
```

```
rec apl1* :  (g:ctx) {M2:[g ⊢ term]}[g ⊢ mstep M1 M1'] → [g ⊢ mstep (app M1
    M2) (app M1' M2)] =
/ total s (apl1* _ _ _ s) /
mlam M2 ⇒ fn s ⇒ case s of
| [g ⊢ id1] ⇒ [g ⊢ id1]
| [g ⊢ step1 S1 S2] ⇒ let [g ⊢ IH] = apl1* [g ⊢ M2] [g ⊢ S2] in
                      [g ⊢ step1 (apl1 S1) IH]
;

rec apr1* : (g:ctx){M1:[g ⊢ term]} [g ⊢ mstep M2 M2'] → [g ⊢ mstep (app M1 M2
    ) (app M1 M2')] =
/ total s (apr1* _ _ _ s) /
mlam M1 ⇒ fn s ⇒ case s of
| [g ⊢ id1] ⇒ [g ⊢ id1]
| [g ⊢ step1 S1 S2] ⇒ let [g ⊢ IH] = apr1* [g ⊢ M1] [g ⊢ S2] in
                      [g ⊢ step1 (apr1 S1) IH]
;
```

□

Equivalence of Ordinary and Parallel Reductions

We now address the equivalence between $\Longrightarrow^*$ and $\longrightarrow_\beta^*$. To do this, we start by proving that $\longrightarrow_\beta \subseteq \Longrightarrow \subseteq \longrightarrow_\beta^*$ holds.

Firstly, we begin by showing the first inclusion:

Lemma 3.1.5. *If* $M \longrightarrow_\beta M'$, *then* $M \Longrightarrow M'$.

The relative representation in Beluga is the following:

```
beta_to_par: (g:rctx) [g ⊢ beta_red M N] → [g ⊢ pred M N]
```

Proof. The proof is by induction on `d : [g ⊢ beta_red M N]`, by exploiting the reflexivity of the parallel reduction `refl_par`:

```
rec beta_to_par: (g:rctx) [g ⊢ beta_red M N] → [g ⊢ pred M N] =
/ total d (beta_to_par _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ beta1] : [g ⊢ beta_red (app (lam \x. M1) M2) M1[…,M2]] ⇒
          let [g,b:block(x:term,t:pred x x) ⊢  I1[…,b.1,b.2]] =
          refl_par [g,b:block (x:term, t:pred x x)]  [g,b  ⊢  M1[…,b.1] ] in
          let [g ⊢ I2] = refl_par [g] [g ⊢ M2 ] in
          [g ⊢ beta (\x.\u. I1[…,x,u]) I2]
| [g ⊢ lm1 \y.D[…,y]] ⇒ let [g,b:block(x:term,u:pred x x) ⊢ IH[…,b.1,b.2]] =
                          beta_to_par [g,b:block(x:term,u:pred x x) ⊢ D[…,b
    .1]] in
                          [g ⊢ lm \x.\u.IH[…,x,u]]
| [g ⊢ apl1 D] : [g ⊢ (beta_red (app M1 M2) (app M1' M2))] ⇒
```

```
                    let [g ⊢ IH] = beta_to_par [g ⊢ D] in
                    let [g ⊢ I2] = refl_par [g] [g ⊢ M2] in
                    [g ⊢ ap IH I2]
| [g ⊢ apr1 D] : [g ⊢ (beta_red (app M1 M2) (app M1 M2'))] ⇒
                    let [g ⊢ IH] = beta_to_par [g ⊢ D] in
                    let [g ⊢ I1] = refl_par [g] [g ⊢ M1] in
                    [g ⊢ ap I1 IH]
;
```

□

For the second inclusion:

Lemma 3.1.6. *If* $M \Longrightarrow M'$, *then* $M \longrightarrow_\beta^* M'$.

In Beluga we will have:

```
par_to_beta : (g:rctx) [g ⊢ pred M N] → [g ⊢ mstep M N]
```

Proof. The proof proceeds by induction on the structure of `d : [g ⊢ pred M N]` by using the fact that `pred` is a congruence, as we have shown in Lemma 3.1.4, and the transitivity of multi-step ordinary reduction `trans_be`.

```
rec par_to_beta : (g:rctx) [g ⊢ pred M N] → [g ⊢ mstep M N] =
/ total d (par_to_beta _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ #p.2] ⇒ [g ⊢ id1]
| {D1: [g ⊢ pred M1 N1]}{D2: [g ⊢ pred M2 N2]}[g ⊢ ap D1 D2] ⇒
                    let [g ⊢ IH1] = par_to_beta [g ⊢ D1] in
                    let [g ⊢ S1]  = apl1* [g ⊢ M2] [g ⊢ IH1] in
                    let [g ⊢ IH2] = par_to_beta [g ⊢ D2] in
                    let [g ⊢ S2] = apr1* [g ⊢ N1 ] [g ⊢ IH2] in
                    trans_mstep [g ⊢ S1] [g ⊢ S2]
| [g ⊢ lm \y.\v.D[…,y,v]] ⇒
          let [g, b:block(x:term, u:pred x x) ⊢ IH[…,b.1]] = par_to_beta [g, b:
    block(x:term, u:pred x x)⊢ D[…,b.1,b.2]] in
          lm1* [g,x:term ⊢ IH[…,x]]
| [g ⊢ beta (\y.\v.D1[…,y,v]) D2] ⇒
  let {M1':[g,x:term ⊢  term]}{IH1:[g,x:term ⊢  mstep M1 M1']}
  [g, b:block(x:term, u:pred x x) ⊢ IH1[…,b.1]] = par_to_beta [g, b:block(x:
    term, u:pred x x)⊢ D1[…,b.1,b.2]] in
  let {M2:[g ⊢ term]}{IH2:[g ⊢ mstep M2 M2']}[g ⊢ IH2] = par_to_beta [g ⊢ D2]
     in
  let [g ⊢ S'] = lm1* [g,x ⊢ IH1[…,x]] in
  let [g ⊢ S''] = apl1*  [g ⊢ M2 ] [g ⊢ S'] in
  let [g ⊢ S'''] = apr1* [g ⊢ lam \x. M1' ] [g ⊢ IH2 ] in
  let [g ⊢ S1']  = trans_mstep [g ⊢ S''] [g ⊢ S'''] in
    trans_mstep [g ⊢ S1'] [g ⊢ step1 beta1 id1 ]
;
```

□

We are now able to conclude the equivalence between $\longrightarrow_\beta^*$ and $\Longrightarrow^*$, with the following lemmas:

Lemma 3.1.7. *If* $M \Longrightarrow^* N$, *then* $M \longrightarrow_\beta^* N$.

Lemma 3.1.8. *If* $M \longrightarrow_\beta^* N$, *then* $M \Longrightarrow^* N$.

Both of these lemmas have a trivial proof that exploits the inclusions we have proven above. For the sake of completeness, we give the full formalization in BELUGA:

```
rec mpar_to_mbeta: (g:rctx) [g ⊢ mstep_pred M N] → [g ⊢ mstep M N] =
/ total d (mpar_to_mbeta _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ id] ⇒ [g ⊢ id1]
| [g ⊢ step D1 D2] ⇒ let [g ⊢ IH1] = par_to_beta [g ⊢ D1] in
                      let [g ⊢ IH2] = mpar_to_mbeta [g ⊢ D2] in
                      trans_mstep [g ⊢ IH1] [g ⊢ IH2]
;

rec mbeta_to_mpar: (g:rctx) [g ⊢ mstep M N] → [g ⊢ mstep_pred M N] =
/ total d (mbeta_to_mpar _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ id1] ⇒ [g ⊢ id]
| [g ⊢ step1 D1 D2] ⇒ let [g ⊢ IH1] = mbeta_to_mpar [g ⊢ D2] in
                       let [g ⊢ IH2] = beta_to_par [g ⊢ D1] in
                       [g ⊢ step IH2 IH1]
;
```

Establishing the Diamond property via the Triangle Operator

As anticipated, we will now develop the proof of the diamond property for parallel β-reduction, by exploiting the idea of Takahashi (see [35]) and defining a triangle operator for parallel β-reduction. Recall from Section 2.5 that a triangle operator for a relation R is a function ρ such that for all x, y if Rxy then $Ry(\rho x)$. One such operator is known as *complete development* or *maximal parallel one-step reduct* M^* of a term M:

- $x^* = x$, for any variable x
- $(M\ N)^* = M^*\ N^*$, if $(M\ N)$ is not a β-redex
- $((\lambda x.M)\ N)^* = M^*[N^*/x]$
- $(\lambda x.M)^* = \lambda x.M^*$

Note that the difference between the complete development relation and the parallel β-reduction is the side condition on the complete development application rule, which stops it from overlapping with β-reduction.

While it would be possible to encode complete development as a function, that is at the computational level, we would not be able to use it in proofs: in fact, BELUGA is a two-level system that strictly separates the specification from the reasoning level. However, it is immediate to define at the LF level as a relation, a.k.a a type-family `cd : term → term → type`, with the additional obligation to show that it is indeed a functional relation.

Luckily, looking at the definition of the triangle operator and how it is used we can notice that we do not need complete development to be a function. In fact, it suffices to have existence without uniqueness to be able to conclude the diamond property (see Lemma 2.5.2).

Moreover, we have to encode the side condition on the complete development application rule, which is the fact that a term $(M\ N)$ is not a β-redex, i.e., M is not a λ-abstraction. Since LF does not have negation, we do this by defining the judgment:

```
LF notlam : term → type =
| nl_app : notlam (app _ _)
;
```

Note that a variable x is clearly not a λ-abstraction, but for the usual reasons, we can not refer directly to variables since they are represented by those in the meta-language. As we did previously, we will include the rule for variables by extending the complete development judgment by hypotheses; this will translate into assuming, every time that we introduce a new variable x, that x is not a lambda term and that `cd x x` holds. Thus, every lemma regarding complete development will need to be stated in a context with the following schema:

```
schema cctx = block(x:term, u: notlam x, v: cd x x);
```

The representation of complete development in BELUGA reads as follows:

```
LF cd : term → term → type =
| cd_beta : ({x:term} notlam x → cd x x → cd (M x) (M' x)) → cd N N'
            → cd (app (lam M) N) (M' N')
| cd_ap : notlam M1 → cd M1 M1' → cd M2 M2'
          → cd (app M1 M2) (app M1' M2')
| cd_lm : ({x:term} notlam x → cd x x → cd (M x) (M' x))
          → cd (lam M) (lam M')
;
```

As a lemma, we need to prove:

Lemma 3.1.9. *For every term M, either M is or is not a lambda abstraction.*

Since, again, LF does not have disjunction we encode the latter with the judgment `lam_or_not` as follows:

```
LF lam_or_not : term → type =
| lon_lam : lam_or_not (lam _)
| lon_not : notlam E → lam_or_not E
;
```

This lemma talks about terms, so we definitely need `term` assumptions, and the judgment `notlam` only makes sense with its assumptions. The hypotheses of `pred` and `cd` are unnecessary.

```
schema nctx = block(x:term, u:notlam x);
```

Now it is easy to formulate the lemma in Beluga:

```
rec decide : {g:nctx}{E : [g ⊢ term]} [g ⊢ lam_or_not E] =
/ total e (decide e) /
mlam g ⇒ mlam E ⇒ case [g ⊢ E] of
| [g ⊢ #p.1] ⇒ [g ⊢ lon_not #p.2]
| [g ⊢ lam \x.E'] ⇒ [g ⊢ lon_lam]
| [g ⊢ app E1 E2] ⇒ [g ⊢ lon_not (nl_app)]
;
```

What we need to show now is that the relation is *total*. This means that for every term M there exists another term N coupled with M through complete development. When representing this statement in Beluga, we have to encode existential quantification in the proposition "there exists a term N complete development of M", but there is no existential quantification in Beluga's theory; to solve this issue, we will define a separate judgment `reduces M`, where the type family `reduces : term → type` has a unique constructor `red_res : cd T1 T2 → reduces T1`. This trick allows us to turn an existential into a universal quantifier, as the rule can be read as: For all `M`, if `cd M M'` then `reduces M` halts.

First, we prove a lemma showing that an application reduces if its components do:

Lemma 3.1.10. *For every M, N terms, if M and N reduces, then $(M\ N)$ reduces.*

```
rec tot_app : (g:cctx) [g ⊢ lam_or_not E1]
              → [g ⊢ cd E1 E1']
              → [g ⊢ cd E2 E2']
              → [g ⊢ reduces (app E1 E2)] =
/ total d (tot_app _ _ _ _ _ d _ _) /
fn d ⇒ fn s ⇒ fn r ⇒ case d of
| [g ⊢ lon_lam] ⇒
            let [g ⊢ cd_lm \x.\u.\v.S] = s in
            let [g ⊢ R] = r in
            [g ⊢ red_res (cd_beta (\x.\u.\v.S) R)]
| [g ⊢ lon_not D] ⇒
            let [g ⊢ S] = s in
            let [g ⊢ R] = r in
            [g ⊢ red_res (cd_ap D S R)]
;
```

We are finally able to prove totality for complete development:

Lemma 3.1.11. *For every term M there exists a term N such that N is the complete development of M.*

In BELUGA:

```
rec cd_tot : {g : cctx}{E : [g ⊢ term]} [g ⊢ reduces E] =
/ total e (cd_tot _ e) /
mlam g ⇒ mlam E ⇒  case [g ⊢ E] of
| [g ⊢ #p.1] ⇒ [g ⊢ red_res #p.3]
| [g ⊢ lam \x.E'[…,x]] ⇒
    let [g, b:block(x:term, u: notlam x, v: cd x x) ⊢ red_res IH[…,b.1,b.2,b
    .3]] =
    cd_tot [g,b:block(x:term, u: notlam x, v: cd x x)] [g,b ⊢ E'[…,b.1]] in
    [g ⊢ red_res (cd_lm \x.\u.\v.IH[…,x,u,v])]
| [g ⊢ app E1 E2] ⇒ let [g ⊢ red_res D1] = cd_tot [g] [g ⊢ E1] in
                     let [g ⊢ red_res D2] = cd_tot [g] [g ⊢ E2] in
                     let [g ⊢ H] = decide [g] [g ⊢ E1] in
                     tot_app [g ⊢ H] [g ⊢ D1] [g ⊢ D2]
;
```

We also will need that the substitution of a parallel reduction is the reduction of the substitution; in other words:

Lemma 3.1.12 (Substitution Lemma for $\Longrightarrow$)**.** *If $\mathcal{R} :: \Gamma, x \vdash M \Longrightarrow M'$ and $\mathcal{S} :: \Gamma \vdash N \Longrightarrow N'$, then $\mathcal{R}' :: \Gamma \vdash [N/x]M \Longrightarrow [N'/x]M'$.*

The substitution lemma can be reformulated in a more conceptual way; if we consider any derivation of $M \Longrightarrow M'$ and in it, we replace any axiom $x \Longrightarrow x$ by $N \Longrightarrow N'$, the result is a derivation of $M[N/x] \Longrightarrow M'[N'/x]$.

```
subst : (g:rctx)[g, x:term, u:pred x x ⊢ pred M[…,x] M'[…,x]]
                → [g ⊢ pred N N']
                → [g ⊢ pred M[…,N] M'[…,N']]
```

Proof. We proceed by induction on the structure of $\mathcal{R}$, i.e., we pattern match on `d : [g, x:term, u:pred x x ⊢ pred M[…,x] M'[…,x]]`.

Case $\mathcal{R} = \dfrac{}{x \Longrightarrow x}\, var$

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

In BELUGA, we have `d = [g, x:term, u:pred x x ⊢ u]` and we simply exhibit `s : [g ⊢ pred N N']` as a witness.

Case $\mathcal{R} = \dfrac{}{y \implies y}\, var$

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

In BELUGA this case is represented using the parameter variable `#p` which stands for a generic `block(y:term, v:pred y y)` in the context `rctx`; to retrieve `v`, we simply take the projection `#p.2[…]` with the identity substitution highlighting the non-dependency of `pred y y` from `x`, which simply means $x \neq y$. What we need to give is a witness for `[g ⊢ pred y y]` but we already have it and this is `[g ⊢ #p.2]`.

Case $\mathcal{R} = \dfrac{\begin{array}{c}\mathcal{R}_1 \\ M_1 \implies M_1'\end{array}}{(\lambda x.M_1) \implies (\lambda x.M_1')}\, lm$

We apply the induction hypothesis to $\mathcal{R}_1$ to obtain a derivation $\mathcal{R}_1' :: [N/x]M_1 \implies [N'/x]M_1'$. By applying the lm rule to $\mathcal{R}_1'$ we obtain the desired deduction $\mathcal{R}'$.

In BELUGA we have `d : [g, x:term, u:pred x x ⊢ lm \y.\v.D[…,y,v,x,u]]`. In order to apply the induction hypothesis to `D` we open the binders and extend the context accordingly, obtaining `[g, b:block(y:term, v: pred y y) ⊢ IH[…,b.1,b.2]]`. Recall that the totality annotation `/ total d (subst _ _ _ _ _ d _) /` instructs the type checker to verify that all recursive calls are structurally decreasing in the sixth argument `d` and that the case analysis is complete. In this particular case, BELUGA does not recognize this recursive call as applied to a smaller argument, and thus totality fails. However, it is easy to see that this is not true mathematically speaking. Thus, we do not worry about it and we simply comment by using `%` the totality checker call. Since we still want to be sure that the case analysis is complete, we add an annotation `--coverage` that takes the place of the totality checker for checking coverage. The desired derivation is then obtained by applying the `lm` rule to `IH`.

Case $\mathcal{R} = \dfrac{\begin{array}{cc}\mathcal{R}_1 & \mathcal{R}_2 \\ M_1 \implies M_1' & M_2 \implies M_2'\end{array}}{(\lambda x.M_1)\ M_2 \implies M_1'[M_2'/x]}\, \beta$

This case is very similar to the one above. We apply the induction hypothesis to $\mathcal{R}_1$ to obtain a derivation $\mathcal{R}_1' :: [N/x]M_1 \implies [N'/x]M_1'$ and to $\mathcal{R}_2$ to obtain a deduction $\mathcal{R}_2' :: [N/x]M_2 \implies [N'/x]M_2'$. Combining these with the β rule we obtain a deduction $\mathcal{R}'$ of the required judgment.

The respective representation is `d : g, x:term, u:pred x x ⊢ beta (\y.\v.D1[…,y, v,x,u]) D2]` and again it is necessary to extend the context before making a recursive

call on D1.

$$\textbf{Case} \quad \mathcal{R} = \dfrac{\begin{array}{cc}\mathcal{R}_1 & \mathcal{R}_2 \\ M_1 \Longrightarrow M_1' & M_2 \Longrightarrow M_2'\end{array}}{M_1\ M_2 \Longrightarrow M_1'\ M_2'}\ ap$$

In this case, we simply apply the induction hypothesis to $\mathcal{R}_1$ and $\mathcal{R}_2$ and combine the resulting deductions with the *ap* rule. We can carry out the exact same reasoning when representing it in Beluga: `d : [g, x:term, u:pred x x ⊢ ap D1 D2]` and we appeal to the induction hypothesis using resp. `D1` and `D2`.

The full formalization in Beluga of the substitution lemma is:

```
rec subst : (g:rctx)[g, x:term, u:pred x x ⊢ pred M[…,x] M'[…,x]]
            → [g ⊢ pred N N'] → [g ⊢ pred M[…,N] M'[…,N']] =
/ total d (subst _ _ _ _ _ d _) /
fn d ⇒ fn s ⇒ case d of
| [g, x:term, u:pred x x ⊢ u] ⇒ s
| [g, x:term, u:pred x x ⊢ #p.2[…]] ⇒ [g ⊢ #p.2]
| [g, x:term, u:pred x x ⊢ lm \y.\v.D[…,y,v,x,u]] ⇒
              let [g ⊢ S] = s in
              let [g, b:block(y:term, v: pred y y) ⊢ IH[…,b.1,b.2]] =
              subst [g, b:block(y:term, v: pred y y), x:term, u:pred x x ⊢ D[…,
   b.1,b.2,x,u]] [g,b ⊢ S[…]] in
              [g ⊢ lm \x.\u.IH[…,x,u]]
| [g, x:term, u:pred x x ⊢ beta (\y.\v.D1[…,y,v,x,u]) D2] ⇒
              let [g ⊢ S] = s in
              let [g ⊢ D2'] = subst [g,x,u ⊢ D2] s in
              let [g, b:block(y:term, v: pred y y) ⊢ IH[…,b.1,b.2]] =
              subst [g, b:block(y:term, v: pred y y), x:term, u:pred x x ⊢ D1[…
   ,b.1,b.2,x,u]] [g,b ⊢ S[…]] in
              [g ⊢ beta (\y.\v.IH[…,y,v]) D2']
| [g, x:term, u:pred x x ⊢ ap D1 D2] ⇒
                     let [g ⊢ IH1] = subst [g,x,u ⊢ D1] s in
                     let [g ⊢ IH2] = subst [g,x,u ⊢ D2] s in
                     [g ⊢ ap IH1 IH2]
;
```

□

Now we have all the necessary ingredients to prove that complete development is actually a triangle operator for parallel β-reduction. The property we have to prove is that:

Lemma 3.1.13 (Triangle property)**.** *For every choice of M and M'' terms if $M \Longrightarrow M''$, then $M'' \Longrightarrow M^*$, where M^* is the complete development of M.*

In other words, given a split of parallel reduction on one side and complete development on the other, there is a parallel reduction on the other side of the triangle. Furthermore, since the lemma we are considering takes into consideration both complete development and parallel reduction, we will need:

```
schema pcctx = block(x:term, w: pred x x, u: notlam x, v: cd x x);
```

The relative representation in Beluga will be:

```
tri : (g:pcctx) [g ⊢ cd M M'] → [g ⊢ pred M M'']
      → [g ⊢ pred M'' M']
```

Proof. The proof proceeds by induction on `d : [g ⊢ cd M M']`.

The simplest case is the variable case which we obtain by taking the fourth projection of a generic declaration `#p = block(x:term, w: pred x x, u: notlam x, v: cd x x)`, i.e., `d = [g ⊢ #p.4] : [g ⊢ cd x x]`. We then have as a goal `[g ⊢ pred x M'']`, but by inversion we obtain `M'' = x` and thus it suffices to exhibit as a witness `[g ⊢ #p.2]`.

The trickiest case is `d = [g ⊢ cd_beta (\x.\u.\v.D1[…,x,u,v]) D2] : [g ⊢ cd (app (lam (\x. M1)) N) (M'1[…, N'])]`, since we then have two possible options for `s : [g ⊢ pred (app (lam (\x. M1)) N) M'']`; in fact, we can have `s = [g ⊢ beta (\y.\w.S1[…,y,w]) S2]` or `[g ⊢ ap (lm \y.\w.S1[…,y,w]) S2]`. In the first case, we then proceed by appealing to the induction hypothesis twice using `[g ⊢ D2] : [g ⊢ cd M2 N']` and `[g ⊢ S2] : [g ⊢ pred M2 M2']` to obtain `[g ⊢ IH1] : [g ⊢ pred x z]`, and then using `[g, x : term, u : notlam x, v : cd x x ⊢ D1] : [g, x : term, u : notlam x, v : cd x x ⊢ cd (M3[…, x]) (M'1[…, x])]` and `[g, x:term, u: pred x x ⊢ S1] : [g, x : term, u : pred x x ⊢ pred (M3[…, x]) (M1'[…, x])]` to obtain `[g, x : term, w : pred x x ⊢ IH2] : [g, x : term, w : pred x x ⊢ pred (M1'[…, x]) (M'1[…, x])]`. To conclude the case it suffices to exhibit a witness for `[g ⊢ pred (M1'[…, x]) (M'1[…, z])]` and this is done by applying the substitution lemma `subst` to `[g,x:term,u:pred x x ⊢ IH2[…,x,u]]` and `[g ⊢ IH1]`. In the second sub-case `s = [g ⊢ ap (lm \y.\w.S1[…,y,w]) S2]` we again appeal to the induction hypothesis twice, obtaining `[g ⊢ IH1] : [g ⊢ pred x z]` and `[g, x:term, w: pred x x⊢ IH2] : [g, x : term, w : pred x x ⊢ pred (M'2[…, x]) (M'1[…, x])]`. To conclude the case we need a witness for `[g ⊢ pred (app (lam (\x. M'2)) x) (M'1[…, z])]` and this is simply `[g ⊢ beta (\x.\v.IH2[…,x,v]) IH1]`.

The third possibility for `d` is to be of the form `[g ⊢ cd_lm \x.\u.\v.D]`. This means that `d : [g ⊢ cd (lam (\x. M1)) (lam (\x. M'1))]` and `s : [g ⊢ pred (lam (\x. M1)) M'']`. By inversion, `s = [g ⊢ lm \x.\v.S[…,x,v]]`. We appeal to the induction hypothesis using `[g, x:term, v:pred x x ⊢ S]` and obtaining `[g, x:term, v:pred x x ⊢ IH]`; we then exhibit as a witness `[g ⊢ lm \x.\v.IH[…,x,v]]`.

The last case is the one in which `d = [g ⊢ cd_ap D' D1 D2]`. By inversion, `s = [g ⊢ ap H1 H2]`. Here, it suffices to appeal to the IH twice using respectively `[g ⊢ D1]`

and [g ⊢ H1], and [g ⊢ D2] and [g ⊢ H2] and combining the results with ap.

```
rec tri : (g:pcctx) [g ⊢ cd M M'] → [g ⊢ pred M M''']
            → [g ⊢ pred M'' M'] =
 / total d (tri _ _ _ _ d _) /
 fn d ⇒ fn s ⇒ case d of
 | [g ⊢ #p.4[…]] ⇒
     let [g ⊢ #p.2] = s in [g ⊢ #p.2]
 | [g ⊢ cd_beta (\x.\u.\v.D1[…,x,u,v]) D2] ⇒
     (case s of
         | [g ⊢ beta (\y.\w.S1[…,y,w]) S2] ⇒
             let [g ⊢ IH1] = tri [g ⊢ D2] [g ⊢ S2] in
             let [g, b:block(x:term, w: pred x x, u: notlam x, v: cd x x) ⊢
IH2[…,b.1,b.2]] =
             tri [g, b:block(x:term, w: pred x x, u: notlam x, v: cd x x)⊢
D1[…,b.1,b.3,b.4]]
             [g,b ⊢ S1[…,b.1,b.2]] in
             subst [g,x:term,u:pred x x ⊢ IH2[…,x,u]] [g ⊢ IH1]
         | [g ⊢ ap (lm \y.\w.S1[…,y,w]) S2] ⇒
             let [g ⊢ IH1] = tri [g ⊢ D2] [g ⊢ S2] in
             let [g, b:block(x:term, w: pred x x, u: notlam x, v: cd x x) ⊢
IH2[…,b.1,b.2]] =
             tri [g, b:block(x:term, w: pred x x, u: notlam x, v: cd x x)⊢
D1[…,b.1,b.3,b.4]]
             [g,b ⊢ S1[…,b.1,b.2]]
             in [g ⊢ beta (\x.\v.IH2[…,x,v]) IH1]
         )
 | [g ⊢ cd_lm \x.\u.\v.D] ⇒
     let [g ⊢ lm \x.\v.S[…,x,v]] = s in
     let [g,b:block(x:term, w: pred x x, u: notlam x, v: cd x x) ⊢ IH[…,b.1,
b.2]] =
     tri [g,b:block(x:term, w: pred x x, u: notlam x, v: cd x x) ⊢ D[…,b.1,b
.3,b.4]]
     [g,b ⊢ S[…,b.1,b.2]] in [g ⊢ lm \x.\v.IH[…,x,v]]
 | [g ⊢ cd_ap D' D1 D2] ⇒
     let [g ⊢ ap H1 H2] = s in
     let [g ⊢ IH1] = tri [g ⊢ D1] [g ⊢ H1] in let [g ⊢ IH2] =
     tri [g ⊢ D2] [g ⊢ H2] in [g ⊢ ap IH1 IH2]
 ;
```

□

We have finally arrived at the core of this Section: the diamond property for parallel β-reduction. Recall that it is one of the two assumptions in the Sandwich Lemma which allows us to conclude confluence for ordinary β-reduction.

Lemma 3.1.14 (Diamond property for $\Longrightarrow$)**.** *The parallel β-reduction $\Longrightarrow$ satisfies the diamond property.*

When representing this statement in BELUGA, we have to again encode existential quantification; we do this by defining the auxiliary judgment:

```
LF joinable : term → term → type =
| dia_res : pred M' N → pred M'' N → joinable M' M''
;
```

We prove the diamond property by first generating a parallel reduction for M and then making two triangles, which complete the square.

```
rec dia : (g:pcctx)[g ⊢ pred M M'] → [g ⊢ pred M M'']
        → [g ⊢ joinable M' M''] =
/ total d (dia _ _ _ _ d _) /
fn d ⇒ fn s ⇒ let [g ⊢ S] : [g ⊢ pred M M''] = s in
               let [g ⊢ red_res H] = cd_tot [g] [g ⊢ M] in
               let [g ⊢ H1] = tri [g ⊢ H] d in
               let [g ⊢ H2] = tri [g ⊢ H] s in
               [g ⊢ dia_res H1 H2]
;
```

Confluence of parallel β-reduction

Following the abstract reasoning done in Section 2, we state and prove the following lemma, called Strip Lemma, which is the last ingredient we need to prove confluence of $\Longrightarrow$.

Lemma 3.1.15 (Strip Lemma)**.** *If $M \Longrightarrow M'$ and $M \Longrightarrow^* M''$, then there exists N such that $M' \Longrightarrow^* N$ and $M'' \Longrightarrow N$.*

Proof. □

Again, we exploit the trick above to encode existential quantification by defining:

```
LF strip_prop: term → term → type =
| strip_result: mstep_pred M' N → pred M'' N
               → strip_prop M' M'';
```

The BELUGA representation of the Strip lemma is then the following:

```
strip : (g:rctx)[g ⊢ pred M M'] → [g ⊢ mstep_pred M M'']
               → [g ⊢ strip_prop M' M'']
```

Proof. Once again, we proceed by induction on `s : [g ⊢ mstep_pred M M'']`. The case `s = [g ⊢ id]` is straightforward since `M'' = N` and thus we can exhibit `[g ⊢ strip_prop id R]` as a witness, by wrapping `r : [g ⊢ pred M M']` as a contextual object `[g ⊢ R]`. The second and last case `s : [g ⊢ step R1 R2]` exploits the diamond property and thus the application of the function `dia` to `r` and `[g ⊢ R1] : [g ⊢ pred M x]` obtaining the contextual objects `[g ⊢ S1] : [g ⊢ pred M' z]` and `[g ⊢ S1'] : [g ⊢ pred x z]`. Now

we can appeal to the induction hypothesis by using `[g ⊢ S1']` and `[g ⊢ R2] : [g ⊢ mstep_pred x M'']` obtaining `[g ⊢ S2] : [g ⊢ mstep_pred z y]` and `[g ⊢ pred M'' y]`. We conclude this case by exhibiting `[g ⊢ strip_result (step S1 S2) S2']`.

```
rec strip : (g:rctx)[g ⊢ pred M M'] → [g ⊢ mstep_pred M M'']
                    → [g ⊢ strip_prop M' M''] =
/ total s (strip g m m' m'' r s) /
fn r ⇒ fn s ⇒ case s of
| [g ⊢ id] ⇒
        let [g ⊢ R] = r in
        [g ⊢ strip_result id R]
| [g ⊢ step R1 R2] ⇒
        let [g ⊢ dia_result _ S1 S1'] = dia r [g ⊢ R1] in
        let [g ⊢ strip_result S2 S2'] = strip [g ⊢ S1'] [g ⊢ R2] in
        [g ⊢ strip_result (step S1 S2) S2']
;
```

□

Finally, we can state and prove confluence for parallel β-reduction.

Lemma 3.1.16 (Confluence of $\Longrightarrow$). *If $M \Longrightarrow^* M'$ and $M \Longrightarrow^* M''$, then there exists N such that $M' \Longrightarrow^* N$ and $M'' \Longrightarrow^* N$.*

Now it is trivial to represent the confluence of $\Longrightarrow^*$ in Beluga:

```
conf : (g:rctx) [g ⊢ mstep_pred M M'] → [g ⊢ mstep_pred M M'']
                → [g ⊢ conf_prop M' M'']
```

Proof. By induction on `d : [g ⊢ mstep_pred M M']`, exploiting the Strip Lemma 3.1.15 and appealing to the induction hypothesis:

```
rec conf : (g:rctx) [g ⊢ mstep_pred M M'] → [g ⊢ mstep_pred M M'']
                    → [g ⊢ conf_prop M' M''] =
/ total d (conf _ _ _ _ d _) /
fn d ⇒ fn s ⇒ case d of
| [g ⊢ id] ⇒
    let [g ⊢ S] = s in [g ⊢ conf_result S id]
| [g ⊢ step D1 D2] ⇒
    let [g ⊢ strip_result D1' D2'] = strip [g ⊢ D1] s in
    let [g ⊢ conf_result S1 S2] = conf [g ⊢ D2] [g ⊢ D1'] in
    [g ⊢ conf_result S1 (step D2' S2)]
;
```

□

Church-Rosser for ordinary β-reduction

Let us take a step back: what we are trying to do is to prove the Church-Rosser property for ordinary β-reduction. In order to achieve this result, we have followed the TMT method, and in particular the Sandwich Lemma 2.5.1. In fact, we have identified as the auxiliary relation required by the lemma parallel β-reduction and we have stated and proved that it satisfies the diamond property. From there, we have proceeded by showing the confluence of $\Longrightarrow$, using the Strip Lemma. We are finally able to prove confluence and Church-Rosser properties for ordinary β-reduction.

Lemma 3.1.17 (Confluence of $\longrightarrow_\beta$)**.** *If $M \longrightarrow_\beta^* M'$ and $M \longrightarrow_\beta^* M''$, then there exists N such that $M' \longrightarrow_\beta^* N$ and $M'' \longrightarrow_\beta^* N$.*

We will encode the existential quantification with the following auxiliary judgment:

```
LF conf_ord_prop : term → term → type =
| conf_ord_result : mstep M' N → mstep M'' N
                  → conf_ord_prop M' M'';
```

The statement in Beluga reads as follows:

```
confluence : (g:rctx) [g ⊢ mstep M M']
           → [g ⊢ mstep M M'']
                      → [g ⊢ conf_ord_prop M' M'']
```

Proof. The proof consists of the application of the equivalence between `mstep_pred` and `mstep` and the confluence of `pred`.

```
rec confluence : (g:rctx) [g ⊢ mstep M M']
                  → [g ⊢ mstep M M'']
                                → [g ⊢ conf_ord_prop M' M''] =
/ total d (confluence _ _ _ _ d _) /
fn d ⇒ fn s ⇒ let [g ⊢ D] = mbeta_to_mpar d in
        let [g ⊢ S] = mbeta_to_mpar s in
        let [g ⊢ conf_result H1 H2] = conf [g ⊢ D] [g ⊢ S] in
        let [g ⊢ H1'] = mpar_to_mbeta [g ⊢ H1] in
        let [g ⊢ H2'] = mpar_to_mbeta [g ⊢ H2] in
        [g ⊢ conf_ord_result H1' H2']
;
```

□

We have arrived at the core of our dissertation: the Church-Rosser theorem (for β-reduction). Now that we have all the lemmas we need, the proof of the theorem is relatively easy:

Theorem 3.1.1 (Church-Rosser)**.** *If $M \longleftrightarrow M'$, then there exists N such that $M \longrightarrow_\beta^* N$ and $M' \longrightarrow_\beta^* N$.*

We finally represent the Church-Rosser theorem for β-reduction and the respective proof in BELUGA.

```
churchrosser : (g:rctx) [g ⊢ conv M M'] → [g ⊢ conf_ord_prop M M']
```

Proof. The proof proceeds by pattern matching on `d : [g ⊢ conv M M']`.

```
rec churchrosser : (g:rctx) [g ⊢ conv M M'] → [g ⊢ conf_ord_prop M M'] =
/ total d (churchrosser _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ ref] ⇒ [g ⊢ conf_ord_result id1 id1]
| [g ⊢ sym R] ⇒
       let [g ⊢ conf_ord_result H1 H2] = churchrosser [g ⊢ R] in
       [g ⊢ conf_ord_result H2 H1]
| [g ⊢ trans R1 R2] ⇒
       let [g ⊢ conf_ord_result IH1 IH2] = churchrosser [g ⊢ R1] in
       let [g ⊢ conf_ord_result IH1' IH2'] = churchrosser [g ⊢ R2] in
       let [g ⊢ conf_ord_result H1 H2] = confluence [g ⊢ IH2'] [g ⊢ IH1] in
       let [g ⊢ H1'] = trans_mstep [g ⊢ IH1'] [g ⊢ H1] in
       let [g ⊢ H2'] = trans_mstep [g ⊢ IH2] [g ⊢ H2] in
       [g ⊢ conf_ord_result H1' H2']
| [g ⊢ red R] ⇒ [g ⊢ conf_ord_result R id_be]
;
```

□

In the following, we will use the formulation of confluence with `beta_red*`, which is easily provable by exploiting the equivalence with `mstep`.

```
LF confl_prop_beta : term → term → type =
| confl_result_beta : beta_red* M1 N → beta_red* M2 N → confl_prop_beta M1 M2
;

rec beta_confluence : (g:ctx) [g ⊢ beta_red* M M1] → [g ⊢ beta_red* M M2]
                    → [g ⊢ confl_prop_beta M1 M2]
```

A Direct Proof of the Diamond Property

Instead of going through the notion of triangle operator, the diamond property can be proved directly (see [6]). The major drawback is that one has to consider a laborious case distinction on the relative positions of the two redexes contracted at the root of the diamond and each case leads to a different completion of it.

Proof.
```
dia : (g:rctx) [g ⊢ pred M M1] → [g ⊢ pred M M2]
            → [g ⊢ joinable M1 M2]
```

The proof is by induction on `d : [g ⊢ pred M M1]`. We will analyze in depth the case in which `d = [g ⊢ ap D1 D2] : [g ⊢ pred (app M4 M3) (app M1' M2')]` with `[g ⊢ D1] : [g ⊢ pred M4 M1']` and `[g ⊢ D2] : [g ⊢ pred M3 M2']`, the others are left to the reader. Since `M = (app M4 M3)`, then `s : [g ⊢ pred (app M4 M3) M2]` and we have thus two possible cases for `s`. In the first case, we have `s = [g ⊢ beta (\y.\v. D1') (D2')]`, with `[g, x:term, u:pred x x ⊢ D1'] : [g, x : term, u : pred x x ⊢ pred (M7[…, x]) (M1'1[…, x])]` and `[g ⊢ D2'] : [g ⊢ pred M5 M2'1]`. In this case, `M` is specialized to `(app (lam (\x. M6)) M5)` and `[g ⊢ D1] : [g ⊢ pred (lam (\x. M6)) M1']`. The only possible rule to obtain `[g ⊢ D1]` is `lm`, thus by inversion we obtain `[g, x:term, u: pred x x ⊢ R1] : [g, x : term, u : pred x x ⊢ pred (M7[…, x]) (M'[…, x])]` such that `[g ⊢ lm \y.\v.R1[…,y,v]] = [g ⊢ D1]`. We can now appeal to the induction hypothesis twice: first by using `[g ⊢ D2]` and `[g ⊢ D2']` obtaining `[g ⊢ S2] : [g ⊢ pred y H2]` and `[g ⊢ S2'] : [g ⊢ pred x H2]`. Then by using `[g, x:term, u:pred x x ⊢ R1]` and `[g, x:term, u:pred x x ⊢ D1']` obtaining `[g, x:term, u:pred x x ⊢ S1] : [g, x : term, u : pred x x ⊢ pred (M'[…, x]) (H1[…, x])]` and `[g, x:term, u:pred x x ⊢ S1'] : [g, x : term, u : pred x x ⊢ pred (M1'1[…, x]) (H1[…, x])]`. The situation by far is the following:

$$\begin{array}{ccc} M_3 & \longrightarrow & y \\ \downarrow & & \vdots \\ x & \dashrightarrow & H_2 \end{array} \qquad\qquad \begin{array}{ccc} M_7[..,x] & \longrightarrow & M'[..,x] \\ \downarrow & & \vdots \\ M'_{11}[..,x] & \dashrightarrow & H_1[..,x] \end{array}$$

Now, the application of the `subst` lemma to `[g, y:term, v:pred y y ⊢ S1']` and `[g ⊢ S2']` yields `S'' : [g ⊢ pred (M1'1[…, x]) (H1[…, H2])]`, and we can then conclude this case by combining `[g, x:term, u:pred x x ⊢ S1]` and `[g ⊢ S2]` with the rule *beta* and exhibiting as a witness `[g ⊢ (dia_result H1[…,H2] (beta (\y.\v.S1[…,y,v]) S2) (S''))]`.

$$\begin{array}{ccc} (\lambda x.M_7)\ M_5 & \longrightarrow & M'_{11}[..,x] \\ \downarrow & & \vdots \\ (\lambda x.M')\ y & \dashrightarrow & H_1[..,H_2] \end{array}$$

```
rec dia : (g:rctx) [g ⊢ pred M M1] → [g ⊢ pred M M2]
          → [g ⊢ joinable M1 M2] =
/ total d (dia _ _ _ _ d _) /
fn d ⇒ fn s ⇒ case d of
| [g ⊢ #p.2] ⇒ let [g ⊢ #q.2] = s in [g ⊢ dia_res _ (#q.2) (#q.2)]
| [g ⊢ ap D1 D2] ⇒
    (case s of
    | [g ⊢ beta (\y.\v. D1') (D2')]  ⇒
```

```
        let [g ⊢ lm \y.\v.R1[…,y,v]] = [g ⊢ D1] in
        let [g ⊢ dia_res H2 S2 (S2')] = dia [g ⊢ D2] [g ⊢ D2'] in
        let [g, b:block(x:term,u:pred x x) ⊢ dia_res H1[…,b.1] S1[…,b.1,b.2] (
    S1'[…,b.1,b.2])] =
        dia [g,b:block(x:term,u:pred x x) ⊢ R1[…,b.1,b.2]] [g,b ⊢ D1'[…,b.1,b
    .2]] in
        let [g ⊢ S''] = subst [g, y:term, v:pred y y ⊢  S1'[…,y,v]] [g ⊢ S2']
    in
        [g ⊢ (dia_res H1[…,H2] (beta (\y.\v.S1[…,y,v]) S2) (S''))]
    | [g ⊢ ap (D1') (D2')] ⇒
        let [g ⊢ dia_res H1 S1 S1'] = dia [g ⊢ D1] [g ⊢ D1'] in
        let [g ⊢ dia_res H2 S2 S2'] = dia [g ⊢ D2] [g ⊢ D2'] in
        [g ⊢ dia_res _ (ap S1 S2) (ap S1' S2')])
| [g ⊢ lm \y.\v.D[…,y,v]] ⇒
        let [g ⊢ lm \x.\u.D'[…,x,u]] = s in
        let [g,b:block(x:term,u:pred x x) ⊢ dia_res _ IH1[…,b.1,b.2] IH2[…,b.1,
    b.2]] =
        dia [g,b:block(x:term,u:pred x x) ⊢ D[…,b.1,b.2]] [g,b ⊢ D'[…,b.1,b
    .2]] in
        [g ⊢ dia_res _ (lm \x.\u.IH1[…,x,u]) (lm \x.\u.IH2[…,x,u])]
| [g ⊢ beta (\y.\v.D1[…,y,v]) D2] ⇒
    (case s of
    | [g ⊢ ap D1' D2'] ⇒
        let [g ⊢ lm \y.\v.R1[…,y,v]] = [g ⊢ D1'] in
        let [g ⊢ dia_res H2 S2 (S2')] = dia [g ⊢ D2] [g ⊢ D2'] in
        let [g, b:block(y:term,u:pred y y) ⊢ dia_res H1[…,b.1] S1[…,b.1,b.2]
    S1'[…,b.1,b.2]] =
        dia [g, b:block(y:term,u:pred y y) ⊢ D1[…,b.1,b.2]] [g,b ⊢ R1[…,b.1,b
    .2]] in
        let [g ⊢ S] = subst [g,y,v ⊢  S1[…,y,v]] [g ⊢ S2] in
        [g ⊢ dia_res _ S (beta (\y.\v.S1'[…,y,v]) S2')]
    | [g ⊢ beta (\x.\u.D1'[…,x,u]) D2'] ⇒
        let [g ⊢ dia_res H2 S2 (S2')] = dia [g ⊢ D2] [g ⊢ D2'] in
        let [g, b:block(y:term,u:pred y y) ⊢ dia_res H1[…,b.1] S1[…,b.1,b.2]
    S1'[…,b.1,b.2]] =
        dia [g, b:block(y:term,u:pred y y) ⊢ D1[…,b.1,b.2]] [g,b ⊢ D1'[…,b.1,b
    .2]] in
        let [g ⊢ S'] = subst [g,y,v ⊢  S1[…,y,v]] [g ⊢ S2] in
        let [g ⊢ S''] = subst [g,y,v ⊢  S1'[…,y,v]] [g ⊢ S2'] in
        [g ⊢ dia_res _ S' S''])
;
```

□

Chapter 4

More Church-Rosser Mechanizations

This Chapter extends the formalization carried out in the previous Chapter, this time by exploiting the results we have seen in Section 2.6. After a brief overview concerning syntax and the semantics, we will carry out a proof of confluence for η-reduction by using the observation that a relation is confluent if and only if it commutes with itself; moreover, we will again refer to the commutation lemmas of Section 2.6 and particularly the Commutative Union Theorem (2.6.1) to conclude confluence for $\beta\eta$-reduction, from the already proven results of confluence for both β and η, and the commutation between them.

4.1 Confluence for η-reduction

If we consider a term M not containing x and the term $M' = \lambda x.M\ x$, then for every A we obtain that $MA = BA$. If we accept the principle of *extensionality*, we want M and M' to be considered equal as terms. However, with the rules defined so far for β-reduction M and M' are not β-equivalent. In fact, the latter is a consequence of the Church-Rosser theorem for β: x and $\lambda y.\ x\ y$ are differentβ-normal forms. Thus we need an additional rule:

$$\frac{}{\Gamma \vdash (\lambda x.Mx) \longrightarrow_\eta M}\ \eta$$

with the proviso that x is not a free variable in M. Adding to this rule the usual rules of congruence we obtain a reduction relation called η-reduction.

$$\frac{\Gamma \vdash M \longrightarrow_\eta M'}{\Gamma \vdash M\ N \longrightarrow_\eta M'\ N}\, cong - app1$$

$$\frac{\Gamma \vdash N \longrightarrow_\eta N'}{\Gamma \vdash M\ N \longrightarrow_\eta M\ N'}\, cong - app2$$

$$\frac{\Gamma, x \vdash M \longrightarrow_\eta M'}{\Gamma \vdash \lambda x.M \longrightarrow_\eta \lambda x.M'}\, cong - lam$$

We then represent η-reduction $\longrightarrow_\eta$ in Beluga with the following type family:

```
LF eta_red : term → term → type =
| eta : eta_red (lam \x.(app M x)) M
| cong_app1 : eta_red M1 M1' → eta_red (app M1 M2) (app M1' M2)
| cong_app2 : eta_red M2 M2' → eta_red (app M1 M2) (app M1 M2')
| cong_lam : ({x:term} eta_red (M x) (M' x))
              → eta_red (lam M) (lam M')
;
```

Note how the proviso is realized within HOAS: in the function `\x.(app M x)`, the meta-variable `M` does *not* depend on `x`.

As usual, we define the reflexive closure of η-reduction and we represent it as the least reflexive relation containing `eta_red`:

```
LF eta_red= : term → term → type =
| id_e= : eta_red= M M
| cl_e= : eta_red M N → eta_red= M N
;
```

Furthermore, we can consider the reflexive-transitive closure:

```
LF eta_red* : term → term → type =
| id_e* : eta_red* M M
| tr_e* : eta_red* M1 M2 → eta_red* M2 M3
        → eta_red* M1 M3
| cl_e* : eta_red M N → eta_red* M N
;
```

We want to achieve confluence for η-reduction by applying the Commutation Lemma 2.6.1. This translates into proving that η strongly commutes with itself, and then, by applying the preliminary lemmas used to prove the Commutation Lemma 2.6.1, concluding that η is confluent.

Before being able to prove strong commutation between η and itself (also called *strong confluence*) we need to prove that both the reflexive and the reflexive-transitive closure are in fact congruences.

Lemma 4.1.1. *The inference rules given below are admissible:*

$$\frac{\Gamma \vdash M_1 \longrightarrow_\eta^= M_1'}{\Gamma \vdash M_1\ M_2 \longrightarrow_\eta^= M_1'\ M_2}\ congapp1 = \qquad \frac{\Gamma \vdash M_2 \longrightarrow_\eta^= M_2'}{\Gamma \vdash M_1\ M_2 \longrightarrow_\eta^= M_1\ M_2'}\ congapp2 =$$

$$\frac{\Gamma, x \vdash M \longrightarrow_\eta^= M'}{\Gamma \vdash \lambda x.M \longrightarrow_\eta^= \lambda x.M'}\ conglam =$$

$$\frac{\Gamma \vdash M_1 \longrightarrow_\eta^* M_1'}{\Gamma \vdash M_1\ M_2 \longrightarrow_\eta^* M_1'\ M_2}\ congapp1* \qquad \frac{\Gamma \vdash M_2 \longrightarrow_\eta^* M_2'}{\Gamma \vdash M_1\ M_2 \longrightarrow_\eta^* M_1\ M_2'}\ congapp2*$$

$$\frac{\Gamma, x \vdash M \longrightarrow_\eta^* M'}{\Gamma \vdash \lambda x.M \longrightarrow_\eta^* \lambda x.M'}\ conglam*$$

We can easily represent and prove these rules are admissible in Beluga as we did in Section 3.1.5 for $\longrightarrow_\beta^*$. See Appendix B for the complete formalization.

To prove strong confluence we will need also a technical "strengthening" lemma[1], which allows us to delete a spurious dependency while performing a case analysis on η reduction:

Lemma 4.1.2. *If $\Gamma, x \vdash M \longrightarrow_\eta N$, and M does not contain x as a free variable, then N does not contain x as a free variable and $\Gamma \vdash M \longrightarrow_\eta N$.*

Since this lemma concerns a contextual existential property it cannot be stated at the LF level. Luckily Beluga's theory [27] accounts for a notion of *inductive* type, thereby giving the user the possibility of stating such property at the computational level (`ctype`):

```
inductive exStrEtaRed : (g:ctx) [g,x:term ⊢ eta_red M[…] N] → ctype =
| exStrEta : {D : [g,x:term ⊢ eta_red M[…] N]}
             [g ⊢ eta_red M N'] → [g,x:term ⊢ eq N'[…] N]
             → exStrEtaRed [g,x:term ⊢ D]
;

--coverage
rec strengthen : {g : ctx} {D : [g, x:term ⊢ eta_red M[…] N]}
                 → exStrEtaRed [g, x:term ⊢ D] =
 mlam g ⇒ mlam D ⇒ case [g, x:term ⊢ D] of
 | [g,x:term ⊢ eta] ⇒
           exStrEta [g, x:term ⊢ D[…]] [g ⊢ eta] [g,x:term ⊢ refl]
 | [g,x:term ⊢ cong_lam \y.D'[…,x,y]] ⇒
           let exStrEta [g,x:term,y:term ⊢ D'[…,x,y]] [g,x:term ⊢ H1] e =
```

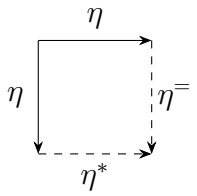
[1] We thank Brigitte Pientka for suggesting its formulation (and proof).

```
            strengthen [g,x:term] [g,x:term,y:term ⊢ D'[…,y,x]] in
            let [g,x:term,y:term ⊢ refl] = e in
            exStrEta [g, x:term ⊢ D] [g ⊢ cong_lam \x.H1[…,x]]
            [g,x:term ⊢ refl]
  | [g, x:term ⊢ cong_app1 D1] ⇒
            let exStrEta [g,x:term ⊢ D1[…,x]] [g ⊢ H1] e =
            strengthen [g] [g,x:term ⊢ D1[…,x]] in
            let [g,x:term ⊢ refl] = e in
            exStrEta [g, x:term ⊢ D] [g ⊢ cong_app1 H1] [g,x:term ⊢ refl]
  | [g, x:term ⊢ cong_app2 D2] ⇒
            let exStrEta [g,x:term ⊢ D2[…,x]] [g ⊢ H2] e =
            strengthen [g] [g,x:term ⊢ D2[…,x]] in
            let [g,x:term ⊢ refl] = e in
            exStrEta [g, x:term ⊢ D] [g ⊢ cong_app2 H2] [g,x:term ⊢ refl]
  ;
```

In the interest of full disclosure, while this function type-checks and respects coverage, it cannot be shown as total due to a current limitation of Beluga's termination checker.

The square lemma we want to prove is the following:

Lemma 4.1.3. *η-reduction strongly commutes with itself:*



As usual, we encode the existential quantifier with a suitable type family:

```
LF eta*_eta=_joinable : term → term → type =
| eta*_eta=_result : eta_red* M1 N → eta_red= M2 N → eta*_eta=_joinable M1 M2
;
```

We are now able to prove it by induction on `[g ⊢ eta_red M M1]` and inversion on `[g ⊢ eta_red M M2]` :

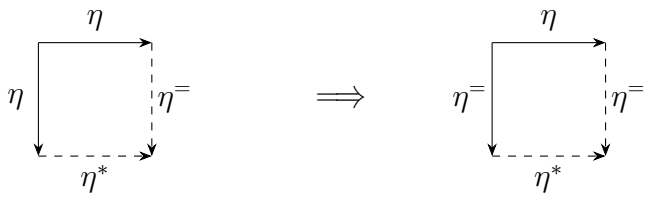
```
rec square : (g:ctx) {M : [g ⊢ term]}{M1 : [g ⊢ term]}{M2 : [g ⊢ term]}
          [g ⊢ eta_red M M1] → [g ⊢ eta_red M M2]
          → [g ⊢ eta*_eta=_joinable M1 M2] =
/ total d (square _ _ _ _ d _) /
mlam M ⇒ mlam M1 ⇒ mlam M2 ⇒ fn d ⇒ fn s ⇒
case d of
| [g ⊢ eta] ⇒
    (case s of
    | [g ⊢ eta] ⇒ [g ⊢ eta*_eta=_result id* id=]
```

```
    | [g ⊢ cong_lam \y.(cong_app1 S'[…,y])] ⇒
        let exStrEta [g,x:term ⊢ S'] [g ⊢ H1] e =
        strengthen [g] [g,x:term ⊢ S'] in
        let [g,x:term ⊢ refl] = e in
        [g ⊢ eta*_eta=_result (cl* H1) (cl= eta)]
    )
| [g ⊢ cong_app1 D1] : [g ⊢ eta_red (app M1' M') (app M2' M')] ⇒
    (case s of
    | [g ⊢ cong_app1 S1] : [g ⊢ eta_red (app N1' N') (app N2' N')] ⇒
        let [g ⊢ eta*_eta=_result IH1 IH2] =
        square [g ⊢ N1'] _ [g ⊢ N2'] [g ⊢ D1] [g ⊢ S1] in
        let [g ⊢ IH'] = cong_app1* [g ⊢ N'] [g ⊢ IH1] in
        let [g ⊢ IH''] = cong_app1= [g ⊢ N'] [g ⊢ IH2] in
        [g ⊢ eta*_eta=_result IH' IH'']
    | [g ⊢ cong_app2 S2] : [g ⊢ eta_red (app M' N1') (app M' N2')] ⇒
        let [g ⊢ H1] = cong_app2* [g ⊢ M2'] [g ⊢ (cl* S2)] in
        let [g ⊢ H2] = cong_app1= [g ⊢ N2'] [g ⊢ (cl= D1)] in
        [g ⊢ eta*_eta=_result H1 H2]
    )
| [g ⊢ cong_app2 D2] : [g ⊢ eta_red (app M' M1') (app M' M2')] ⇒
    (case s of
    | [g ⊢ cong_app1 S1] : [g ⊢ eta_red (app N1' N') (app N2' N')] ⇒
        let [g ⊢ H1] = cong_app1* [g ⊢ M2'] [g ⊢ (cl* S1)] in
        let [g ⊢ H2] = cong_app2= [g ⊢ N2'] [g ⊢ (cl= D2)] in
        [g ⊢ eta*_eta=_result H1 H2]
    | [g ⊢ cong_app2 S2] : [g ⊢ eta_red (app N' N1') (app N' N2')] ⇒
        let [g ⊢ eta*_eta=_result IH1 IH2] =
        square [g⊢ N1'] [g ⊢ M2'] [g ⊢ N2'] [g ⊢ D2] [g ⊢ S2] in
        let [g ⊢ IH'] = cong_app2* [g ⊢ N'] [g ⊢ IH1] in
        let [g ⊢ IH''] = cong_app2= [g ⊢ N'] [g ⊢ IH2] in
        [g ⊢ eta*_eta=_result IH' IH'']
    )
| [g ⊢ cong_lam \x.D[…,x]] : [g ⊢ eta_red (lam (\x. M')) (lam (\x. M1'))] ⇒
    (case s of
    | [g ⊢ cong_lam \y.D'[…,y]] : [g ⊢ eta_red (lam (\x. M')) (lam (\x. M2'))]
     ⇒
        let [g, y:term ⊢ eta*_eta=_result IH1[…,y] IH2[…,y]] =
        square [g, y:term ⊢ M'[…,y]][g, y:term ⊢ M1'[…,y]][g, y:term ⊢ M2'[…,
    y]]
            [g, y:term ⊢ D[…,y]] [g, y:term ⊢ D'[…,y]] in
        let [g ⊢ H1] = cong_lam* [g, x:term ⊢ IH1[…,x]] in
        let [g ⊢ H2] = cong_lam= [g, x:term ⊢ IH2[…,x]] in
        [g ⊢ eta*_eta=_result H1 H2]
    | [g ⊢ eta] ⇒
        let [g, x:term ⊢ cong_app1 D1[…,x]] = [g, x:term ⊢ D[…,x]] in
        let exStrEta [g,x:term ⊢ D1] [g ⊢ H1] e =
        strengthen [g] [g,x:term ⊢ D1] in
        let [g,x:term ⊢ refl] = e in
```

```
        [g ⊢ eta*_eta=_result (cl* eta) (cl= H1)]
    )
;
```

Now that we have shown the hypothesis of the Commutation Lemma, we just have to apply all the steps in its proof to obtain confluence for η.

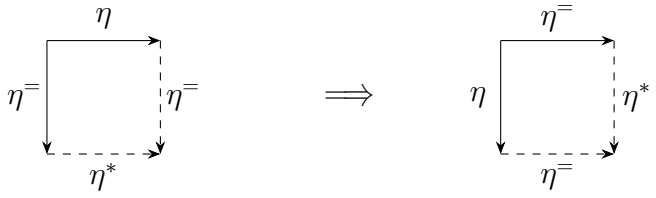
Lemma 4.1.4.



Note that this is an application of Lemma 2.4.3.2) we have seen in general for Abstract Reduction Systems.

```
rec lemma_three : (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
                      [g ⊢ eta_red M M1] → [g ⊢ eta_red M M2]
                      → [g ⊢ eta*_eta=_joinable M1 M2])
                      → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
                      [g ⊢ eta_red= M' M1'] → [g ⊢ eta_red M' M2'] → [g ⊢
    eta*_eta=_joinable M1' M2']) =
/ total s (lemma_three _ _ _ _ _ s _) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
case s of
      | [g ⊢ id=] ⇒ let [g ⊢ T] = t in
                      [g ⊢ eta*_eta=_result (cl* T) id=]
      | [g ⊢ cl= S] ⇒ let [g ⊢ eta*_eta=_result H1 H2] = d _ _ _ [g ⊢ S] t in
                        [g ⊢ eta*_eta=_result H1 H2]
;
```

Lemma 4.1.5.



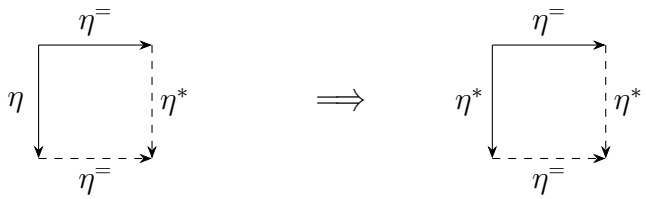
For the proof, we do not even need to analyze two different cases, but we just swap the two reductions we obtain by hypothesis. This is a case of application of Lemma 2.4.3.1).

```
LF eta=_eta*_joinable : term → term → type =
| eta=_eta*_result : eta_red= M1 N → eta_red* M2 N → eta=_eta*_joinable M1 M2
;

rec lemma_one_first : (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
                      [g ⊢ eta_red= M M1] → [g ⊢ eta_red M M2]
                      → [g ⊢ eta*_eta=_joinable M1 M2])
                       → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
                       [g ⊢ eta_red M' M1'] → [g ⊢ eta_red= M' M2'] → [g ⊢
    eta=_eta*_joinable M1' M2']) =
/ total t (lemma_one_first _ _ _ _ _ _ t) /

fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
    let [g ⊢ eta*_eta=_result H1 H2] = d _ _ _ t s in
    [g ⊢ eta=_eta*_result H2 H1]
;
```

Lemma 4.1.6.

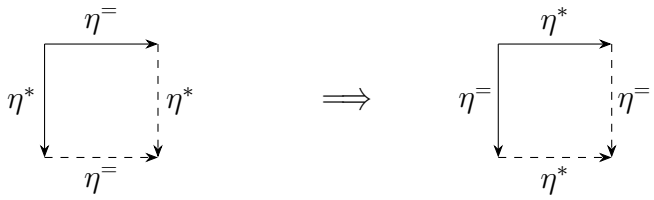


Note that this is the generalized Strip Lemma 2.4.5 applied to a particular case.

```
rec lemma_four_first:(g:ctx)({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
    [g ⊢ eta_red M M1] → [g ⊢ eta_red= M M2]
    → [g ⊢ eta=_eta*_joinable M1 M2])
            → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
            [g ⊢ eta_red* M' M1'] → [g ⊢ eta_red= M' M2']
            → [g ⊢ eta=_eta*_joinable M1' M2']) =
/ total s (lemma_four_first _ _ _ _ _ s _) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
case t of
| [g ⊢ id=] ⇒
    let [g ⊢ S] = s in
    [g ⊢ eta=_eta*_result id= S]
| [g ⊢ cl= T] ⇒
    (case s of
    | [g ⊢ id*] ⇒ [g ⊢ eta=_eta*_result (cl= T) id*]
    | [g ⊢ cl* S'] ⇒
        let [g ⊢ eta=_eta*_result H1 H2] = d _ _ _ [g ⊢ S'] t in
        [g ⊢ eta=_eta*_result H1 H2]
    | [g ⊢ tr* S1 S2] ⇒
```

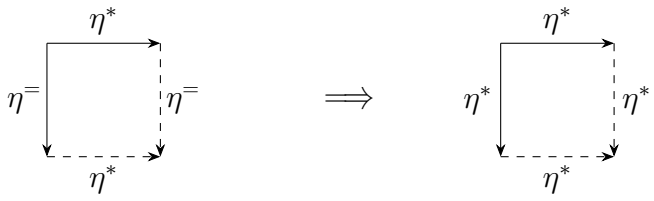
```
        let [g ⊢ eta=_eta*_result H1 H2] = lemma_four_first d _ _ _ [g ⊢ S1] t
     in
        let [g ⊢ eta=_eta*_result H1' H2'] = lemma_four_first d _ _ _ [g ⊢ S2]
     [g ⊢ H1] in
        [g ⊢ eta=_eta*_result H1' (tr* H2 H2')]
    )
;
```

Lemma 4.1.7.




```
rec lemma_one_second:(g:ctx)({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
    [g ⊢ eta_red* M M1] → [g ⊢ eta_red= M M2]
    → [g ⊢ eta=_eta*_joinable M1 M2])
            → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
            [g ⊢ eta_red= M' M1'] → [g ⊢ eta_red* M' M2']
            → [g ⊢ eta*_eta=_joinable M1' M2']) =
/ total s (lemma_one_second _ _ _ _ _ s _) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
    let [g ⊢ eta=_eta*_result IH1 IH2] = d _ _ _ t s in
    [g ⊢ eta*_eta=_result IH2 IH1]
;
```

Lemma 4.1.8.




```
LF confl_prop : term → term → type =
| confl_result : eta_red* M1 N → eta_red* M2 N → confl_prop M1 M2
;
```

To prove this lemma we will use the fact that the reflexive closure of η is a subset of the reflexive-transitive closure, i.e.:

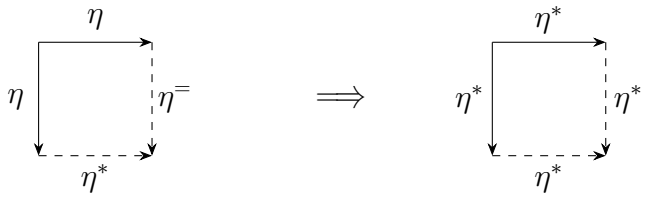
```
rec etaeq_etastar : (g:ctx) [g ⊢ eta_red= M N] → [g ⊢ eta_red* M N] =
/ total d (etaeq_etastar _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ id_e=] ⇒ [g ⊢ id_e*]
| [g ⊢ cl_e= D] ⇒ [g ⊢ cl_e* D]
;
```

This is again the Generalized Strip Lemma 2.4.5, applied to $R = T = \longrightarrow_\eta^=$ and $S = \longrightarrow_\eta^*$.

```
rec lemma_four_second:(g:ctx)({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
        [g ⊢ eta_red= M M1] → [g ⊢ eta_red* M M2]
        → [g ⊢ eta*_eta=_joinable M1 M2])
            → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
            [g ⊢ eta_red* M' M1'] → [g ⊢ eta_red* M' M2']
            → [g ⊢ confl_prop M1' M2']) =
/ total s (lemma_four_second _ _ _ _ _ s _) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒ case s of
| [g ⊢ id*] ⇒ let [g ⊢ T] = t in [g ⊢ confl_result T id*]
| [g ⊢ cl* S] ⇒
    let [g ⊢ eta*_eta=_result H1 H2] = d _ _ _ [g ⊢ (cl= S)] t in
    let [g ⊢ H2'] = etaeq_etastar [g ⊢ H2] in
    [g ⊢ confl_result H1 H2']
| [g ⊢ tr* S1 S2] ⇒
    let [g ⊢ confl_result IH1' IH2'] = lemma_four_second d _ _ _ [g ⊢ S1] t in
    let [g ⊢ confl_result IH1'' IH2''] = lemma_four_second d _ _ _ [g ⊢ S2] [g
     ⊢ IH1'] in
    [g ⊢ confl_result IH1'' (tr* IH2' IH2'')]
;
```

The lemmas we have listed above are simply the steps in the Commutation Lemma's proof 2.6.1 to the case of $R = S = \eta$. In fact:

Lemma 4.1.9 (Commutation Lemma)**.**




```
rec commutation_lemma:(g:ctx)({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
        [g ⊢ eta_red M M1] → [g ⊢ eta_red M M2]
        → [g ⊢ eta*_eta=_joinable M1 M2])
            → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
            [g ⊢ eta_red* M' M1'] → [g ⊢ eta_red* M' M2']
```

```
            → [g ⊢ confl_prop M1' M2']) =
/ total s (commutation_lemma _ _ _ _ _ s _) /
fn d ⇒ lemma_four_second (lemma_one_second (lemma_four_first (lemma_one_first
    (lemma_three d))))
;
```

Now it suffices to apply Commutation Lemma to the fact that η-reduction is strongly confluent, as we proved in lemma 4.1.3.

Theorem 4.1.1 (Confluence for η-reduction)**.** *If $M \longrightarrow_\eta^* M'$ and $M \longrightarrow_\eta^* M''$, then there exists N such that $M' \longrightarrow_\eta^* N$ and $M'' \longrightarrow_\eta^* N$.*

```
rec eta_confluence : (g:ctx) [g ⊢ eta_red* M M1] → [g ⊢ eta_red* M M2]
                     → [g ⊢ confl_prop M1 M2] =
/ total d (eta_confluence _ _ _ _ d _ ) /
fn d ⇒ fn s ⇒ (commutation_lemma square _ _ _ d s)
;
```

4.2 Confluence for $\beta\eta$-reduction

The relation resulting from the union between β and η-reductions is called $\beta\eta$-reduction. We could prove confluence for $\beta\eta$ by defining parallel $\beta\eta$-reduction in a straightforward way and then applying the TMT method as we did for the β case. However, since we have already shown confluence for both β and η, it makes more sense to exploit the Commutative Union Lemma 2.6.1. Its assumption is that β and η commute, i.e., for all M, M', M'', if $M \longrightarrow_\beta^* M'$ and $M \longrightarrow_\eta^* M''$, then there exists N such that $M' \longrightarrow_\eta^* N$ and $M'' \longrightarrow_\beta^* N$. We can prove that this assumption holds by first proving the strong commutation between $\longrightarrow_\beta$ and $\longrightarrow_\eta$ and then by applying the Commutation Lemma 2.6.1 as we did for η-confluence. Moreover, we will define an auxiliary relation $\beta^* \cup \eta^*$ whose reflexive-transitive closure is equivalent to the one of $\beta\eta$. We aim to apply the Sandwich Lemma 2.5.1 to have a proof of the Commutative Union Lemma 2.6.1, thus we will need $\beta^* \cup \eta^*$ to satisfy the diamond property, but this is straightforward from the assumptions.

We formally encode $\beta\eta$-reduction as follows:

```
LF beta_eta : term → term → type =
| beta_be : beta_red M M' → beta_eta M M'
| eta_be : eta_red M M' → beta_eta M M'
;
```

We want to show confluence, thus we will need the usual notion of reflexive-transitive closure:

```
LF beta_eta* : term → term → type =
| id_be* : beta_eta* M M
| tr_be* : beta_eta* M1 M2 → beta_eta* M2 M3
        → beta_eta* M1 M3
| cl_be* : beta_eta M N → beta_eta* M N
;
```

A property is local if it is quantified only over one-step reductions; it is global if it is quantified over all rewrite sequences. Commutation is a global condition, and thus difficult to test. What turns the Hindley-Rosen lemma into an effective method is the presence of local sufficient conditions. One of the easiest is strong commutation, which we tackle next.

We have already shown that $\longrightarrow_\eta^*$ is a congruence, but to prove this lemma we will need a similar result for $\longrightarrow_{\overline{\beta}}$, and we will encode these rules as `lm=`, `apl=` and `apr=`. See Appendix A for the complete representation.

Moreover, the proof requires two different results regarding η substitutions; namely:

Lemma 4.2.1. *If $M \longrightarrow_\eta M'$, with M and M' terms depending on x, then for every N term, $\Gamma \vdash M[N/x] \longrightarrow_\eta M'[N/x]$.*

In Beluga we will have:

```
rec subst_eta_closed : (g:ctx) {N : [g ⊢ term]}
                       [g, x:term ⊢ eta_red M[…,x] M'[…,x]]
                       → [g ⊢ eta_red M[…,N] M'[…,N]] =
/ total d (subst_eta_closed _ _ _ _ d) /
mlam N ⇒ fn d ⇒ case d of
| [g, x:term ⊢ eta] ⇒ [g ⊢ eta]
| [g, x:term ⊢ cong_lam \y.D'[…,y,x]] ⇒
                  let [g, x:term⊢ IH[…,x]] =
                  subst_eta_closed [g, x:term ⊢ N[…]]
                  [g, x:term, y:term ⊢ D'[…,x,y]] in
                  [g ⊢ cong_lam \x.IH[…,x]]
| [g, x:term ⊢ cong_app1 D1[…,x]] ⇒
        let [g ⊢ IH] =
        subst_eta_closed [g ⊢ N] [g, x:term ⊢ D1[…,x]] in
        [g ⊢ cong_app1 IH]
| [g, x:term ⊢ cong_app2 D2[…,x]] ⇒
        let [g ⊢ IH] =
        subst_eta_closed [g ⊢ N] [g, x:term ⊢ D2[…,x]] in
        [g ⊢ cong_app2 IH]
;
```

Lemma 4.2.2. *If M is a term depending on x and $M_1 \longrightarrow_\eta M_2$, then $M[M_1/x] \longrightarrow_\eta^* M[M_2/x]$.*

To prove this lemma we will exploit an additional trivial property:

```
rec lemma_eta_app : (g:ctx)[g ⊢ eta_red* M M'] → [g ⊢ eta_red* N N'] → [g ⊢
    eta_red* (app M N) (app M' N')] =
/ total d (lemma_eta_app _ _ _ _ _ d _) /
fn d ⇒ fn s ⇒ let [g ⊢ D] : [g ⊢ eta_red* M M'] = d in
               let [g ⊢ S] : [g ⊢ eta_red* N N'] = s in
               let [g ⊢ D'] = cong_app1* [g ⊢ N] [g ⊢ D] in
               let [g ⊢ S'] = cong_app2* [g ⊢ M'] [g ⊢ S] in
               [g ⊢ tr_e* D' S']
;
```

We are now able to represent Lemma 4.2.2 in Beluga:

```
rec subst_eta : {g:ctx} {M : [g,x:term ⊢ term]} [g ⊢ eta_red M1 M2] → [g ⊢
    eta_red* M[…,M1] M[…,M2]] =
/ total m (subst_eta _ m _ _ _) /
mlam g ⇒ mlam M ⇒ fn d ⇒
    case [g,x:term ⊢ M] of
    | [g,x:term ⊢ x] ⇒ let [g ⊢ D] = d in [g ⊢ cl_e* D]
    | [g, x:term ⊢ #p[…]] ⇒ [g ⊢ id_e*]
    | [g,x:term ⊢ lam \y.M'[…,x,y]] ⇒
        let [g ⊢ D] = d in
        let [g,x:term ⊢ H[…,x]] =
        subst_eta [g,x:term] [g,x:term,y:term ⊢ M'[…,y,x]] [g,x:term ⊢ D[…]]
    in
        cong_lam* [g,x:term ⊢ H[…,x]]
    | [g,x:term ⊢ app M1' M2'] ⇒
          let [g ⊢ H1] = subst_eta [g] [g, x:term ⊢ M1'] d in
          let [g ⊢ H2] = subst_eta [g] [g, x:term ⊢ M2'] d in
          lemma_eta_app [g ⊢ H1] [g ⊢ H2]
;
```

Lastly, we will need a strengthening lemma for β, which extends the one we proved in the η case (see 4.1.2).

```
strengthen_beta : {g : ctx} {D : [g, x:term ⊢ beta_red M[…] N]}
                  → exStrBetaRed [g, x:term ⊢ D]
```

For the complete representation in Beluga see Appendix A.

We are finally able to prove:

Lemma 4.2.3. *β and η-reductions strongly commute, i.e., for all M, M', M'', if $M \longrightarrow_\beta M'$ and $M \longrightarrow_\eta M''$, then there exists N such that $M' \longrightarrow_\eta^* N$ and $M'' \longrightarrow_\beta^= N$.*

```
be_square : (g:ctx) {M : [g ⊢ term]}{M1 : [g ⊢ term]}{M2 : [g ⊢ term]}
          [g ⊢ beta_red M M1] → [g ⊢ eta_red M M2]
          → [g ⊢ eta*_beta=_joinable M1 M2]
```

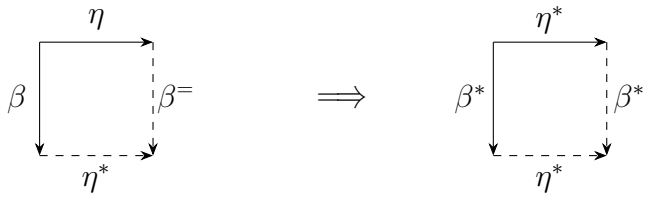
Here we assume to have defined a type family:

```
LF eta*_beta=_joinable : term → term → type =
| eta*_beta=_result : eta_red* M1 N → beta_red= M2 N
                      → eta*_beta=_joinable M1 M2
;
```

See Appendix C for the complete proof.

As we did in the previous Section for proving η confluence, we can now apply the following lemma to obtain commutation between β and η; the full representation of the steps composing its proof is in Appendix C since it is very similar to the η case.

Lemma 4.2.4 (Commutation Lemma)**.**




```
LF commute : term → term → type =
| commute_result : eta_red* M1 N → beta_red* M2 N → commute M1 M2
;

rec commutation_lemma :
  (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
  [g ⊢ beta_red M M1] → [g ⊢ eta_red M M2]
  → [g ⊢ eta*_beta=_joinable M1 M2])
    → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
        [g ⊢ beta_red* M' M1'] → [g ⊢ eta_red* M' M2']
        → [g ⊢ commute M1' M2']) =
/ total s (commutation_lemma _ _ _ _ _ s _) /
fn d ⇒ lemma_four_second_be (lemma_one_second_be (lemma_four_first_be (
    lemma_one_first_be (lemma_three_be d))))
;
```

Now, we would like to apply the Commutative Union Theorem 2.6.1 to conclude confluence for $\beta\eta$ since we have all the assumptions needed: confluence for β, confluence for η and commutation between β and η. However, to be able to prove this theorem, we need a few additional lemmas. Then, we could give a direct abstract proof of this theorem and then apply it to the commutation and confluence lemmas (see Appendix C for a complete formalization); however, for the sake of clarity, we will prove confluence for $\beta\eta$ step-by-step, following what is all in all the entire proof for the Commutative Union Theorem, but takes into consideration directly the commutation and confluence lemmas we have already proved. The idea is to define again an auxiliary relation $\beta^* \cup \eta^*$, show the equivalence between the two reflexive-transitive closures and then

prove the diamond property for $\beta^* \cup \eta^*$, in order to satisfy all the assumptions in the Sandwich Lemma 2.5.1. We start by proving the equivalence between $(\beta \cup \eta)^*$ and $(\beta^* \cup \eta^*)^*$.

Lemma 4.2.5. $(\beta^* \cup \eta^*)^* \cong (\beta \cup \eta)^*$

To encode this lemma in BELUGA we first represent the relation $\beta^* \cup \eta^*$ and then its reflexive-transitive closure $(\beta^* \cup \eta^*)^*$:

```
LF union_b*e* : term → term → type =
| eta* : eta_red* M1 M2 → union_b*e* M1 M2
| beta* : beta_red* M1 M2 → union_b*e* M1 M2
;

LF beta*_eta* : term → term → type =
| tr_b*e* : beta*_eta* M1 M2 → beta*_eta* M2 M3
            → beta*_eta* M1 M3
| cl_b*e* : union_b*e* M M' → beta*_eta* M M'
;
```

The proof of one of the two inclusion is straightforward:

Lemma 4.2.6. $(\beta \cup \eta)^* \subseteq (\beta^* \cup \eta^*)^*$

```
rec be1_to_be2 : (g:ctx) [g ⊢ beta_eta* M M'] → [g ⊢ beta*_eta* M M'] =
/ total d (be1_to_be2 _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ id_be*] ⇒ [g ⊢ cl_b*e* (eta* id_e*)]
| [g ⊢ tr_be* D1 D2] ⇒
    let [g ⊢ D1'] = be1_to_be2 [g ⊢ D1] in
    let [g ⊢ D2'] = be1_to_be2 [g ⊢ D2] in
    [g ⊢ tr_b*e* D1' D2']
| [g ⊢ cl_be* D] ⇒
    (case [g ⊢ D] of
    | [g ⊢ beta_be D1] ⇒ [g ⊢ cl_b*e* (beta* (cl_b* D1))]
    | [g ⊢ eta_be D1] ⇒ [g ⊢ cl_b*e* (eta* (cl_e* D1))]
    )
;
```

We then prove two lemmas `lemma_eta` and `lemma_beta`, that respectively state the inclusions $\eta^* \subseteq (\beta \cup \eta)^*$ and $\beta^* \subseteq (\beta \cup \eta)^*$.

```
rec lemma_eta : (g:ctx) [g ⊢ eta_red* M M'] → [g ⊢ beta_eta* M M'] =
/ total d (lemma_eta _ _ _ d) /
fn d ⇒ case d of
    | [g ⊢ id_e*] ⇒ [g ⊢ id_be*]
    | [g ⊢ tr_e* D1 D2] ⇒
        let [g ⊢ IH1] = lemma_eta [g ⊢ D1] in
        let [g ⊢ IH2] = lemma_eta [g ⊢ D2] in
```

```
        [g ⊢ tr_be* IH1 IH2]
    | [g ⊢ cl_e* D] ⇒ [g ⊢ cl_be* (eta_be D)]
;

rec lemma_beta : (g:ctx) [g ⊢ beta_red* M M'] → [g ⊢ beta_eta* M M'] =
/ total d (lemma_beta _ _ _ d) /
fn d ⇒ case d of
    | [g ⊢ id_b*] ⇒ [g ⊢ id_be*]
    | [g ⊢ tr_b* D1 D2] ⇒
        let [g ⊢ IH1] = lemma_beta [g ⊢ D1] in
        let [g ⊢ IH2] = lemma_beta [g ⊢ D2] in
        [g ⊢ tr_be* IH1 IH2]
    | [g ⊢ cl_b* D] ⇒ [g ⊢ cl_be* (beta_be D)]
;
```

We are now ready to prove the second inclusion:

Lemma 4.2.7. $(\beta^* \cup \eta^*)^* \subseteq (\beta \cup \eta)^*$

```
rec be2_to_be1 : (g:ctx) [g ⊢ beta*_eta* M M'] → [g ⊢ beta_eta* M M'] =
/ total d (be2_to_be1 _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ cl_b*e* D] ⇒
    (case [g ⊢ D] of
    | [g ⊢ eta* D'] ⇒
        (case [g ⊢ D'] of
        | [g ⊢ id_e*] ⇒ [g ⊢ id_be*]
        | [g ⊢ tr_e* D1 D2] ⇒
            let [g ⊢ H1] = lemma_eta [g ⊢ D1] in
            let [g ⊢ H2] = lemma_eta [g ⊢ D2] in
            [g ⊢ tr_be* H1 H2]
        | [g ⊢ cl_e* D''] ⇒ [g ⊢ cl_be* (eta_be D'')]
        )
    | [g ⊢ beta* D'] ⇒
        (case [g ⊢ D'] of
        | [g ⊢ id_b*] ⇒ [g ⊢ id_be*]
        | [g ⊢ tr_b* D1 D2] ⇒
            let [g ⊢ H1] = lemma_beta [g ⊢ D1] in
            let [g ⊢ H2] = lemma_beta [g ⊢ D2] in
            [g ⊢ tr_be* H1 H2]
        | [g ⊢ cl_b* D''] ⇒ [g ⊢ cl_be* (beta_be D'')]
        )
    )
| [g ⊢ tr_b*e* D1 D2] ⇒
        let [g ⊢ H1] = be2_to_be1 [g ⊢ D1] in
        let [g ⊢ H2] = be2_to_be1 [g ⊢ D2] in
        [g ⊢ tr_be* H1 H2]
;
```

The other assumption in the Sandwich Lemma is that the auxiliary relation satisfies the diamond property:

Lemma 4.2.8 (Diamond property for $\beta^* \cup \eta^*$)**.** *$\beta^* \cup \eta^*$ satisfies the diamond property.*

```
LF dia_un_prop : term → term → type =
| dia_un_res : union_b*e* M' N → union_b*e* M'' N → dia_un_prop M' M''
;

rec dia_un : (g:ctx) [g ⊢ union_b*e* M M'] → [g ⊢ union_b*e* M M'']
                        → [g ⊢ dia_un_prop M' M''] =
/ total d (dia_un _ _ _ _ d s) /
fn d ⇒ fn s ⇒ case d of
| [g ⊢ eta* D] ⇒
    (case s of
        | [g ⊢ eta* S] ⇒
                let [g ⊢ confl_result_eta H1 H2] = eta_confluence [g ⊢ D] [g ⊢
     S] in
                [g ⊢ dia_un_res (eta* H1) (eta* H2)]
        | [g ⊢ beta* S] ⇒
                let [g ⊢ commute_result H1 H2] =
        commutation_lemma be_square _ _ _ [g ⊢ S] [g ⊢ D] in
                [g ⊢ dia_un_res (beta* H2) (eta* H1)]
        )
| [g ⊢ beta* D] ⇒
        (case s of
        | [g ⊢ eta* S] ⇒
                let [g ⊢ commute_result H1 H2] =
        commutation_lemma be_square _ _ _ [g ⊢ D] [g ⊢ S] in
                [g ⊢ dia_un_res (eta* H1) (beta* H2)]
        | [g ⊢ beta* S] ⇒
                let [g ⊢ confl_result_beta H1 H2] = beta_confluence [g ⊢ D] [g
     ⊢ S] in
                [g ⊢ dia_un_res (beta* H1) (beta* H2)]
    )
;
```

Similarly to what we have done for the parallel β-reduction in Section 3.1.5, we can now state and prove a strip property for $\beta^* \cup \eta*$ and then conclude its confluence.

```
LF strip_un_prop: term → term → type =
| strip_un_result: beta*_eta* M' N → union_b*e* M'' N
                → strip_un_prop M' M'';

strip_un : (g:rctx)[g ⊢ union_b*e* M M'] → [g ⊢ beta*_eta* M M'']
                → [g ⊢ strip_un_prop M' M'']

LF conf_un_prop : term → term → type =
| conf_un_res : beta*_eta* M' N → beta*_eta* M'' N
```

```
                → conf_un_prop M' M''
;

conf_un : (g:ctx) [g ⊢ beta*_eta* M M'] → [g ⊢ beta*_eta* M M'']
                  → [g ⊢ conf_un_prop M' M'']
```

Again, the complete formalization of these proofs is in Appendix C.

Now it suffices to exploit the equivalence between $(\beta^* \cup \eta^*)^*$ and $(\beta \cup \eta)^* = \beta\eta^*$ to conclude confluence for $\beta\eta$.

Lemma 4.2.9 (Confluence for $\beta\eta$-reduction)**.** *If $M \longrightarrow^*_{\beta\eta} M'$ and $M \longrightarrow^*_{\beta\eta} M''$, then there exists N such that $M' \longrightarrow^*_{\beta\eta} N$ and $M'' \longrightarrow^*_{\beta\eta} N$.*

```
LF conf_beta_eta : term → term → type =
| conf_be_result : beta_eta* M1 N → beta_eta* M2 N → conf_beta_eta M1 M2
;

rec eta_beta_confluence : (g:ctx) [g ⊢ beta_eta* M M1] → [g ⊢ beta_eta* M M2]
                        → [g ⊢ conf_beta_eta M1 M2] =
/ total d (eta_beta_confluence _ _ _ _ d _ ) /
fn d ⇒ fn s ⇒
        let [g ⊢ H1] = be1_to_be2 d in
        let [g ⊢ H2] = be1_to_be2 s in
        let [g ⊢ conf_un_res H1' H2'] = conf_un [g ⊢ H1] [g ⊢ H2] in
        let [g ⊢ H1''] = be2_to_be1 [g ⊢ H1'] in
        let [g ⊢ H2''] = be2_to_be1 [g ⊢ H2'] in
        [g ⊢ conf_be_result H1'' H2'']
;
```

4.3 System F

We now switch gears and address confluence properties in *typed* lambda-calculi. While the main definitions still apply, we must be mindful to apply reductions only to well-typed terms [32]. In a dependently typed proof environment such as Beluga, this can be very elegantly accomplished using *intrinsic terms* [8; 1], that is by allowing only well-typed terms and indexing the judgments under study by the former. What is remarkable is that literally the same proof of the Church-Rosser theorem carries over to the simply-typed lambda calculus. Of course, minor conservative extensions are needed when adding different types or richer type systems.

The technique of intrinsic types is known to apply to a wide range of type systems, but not everywhere. This is why we concentrate on the polymorphic lambda calculus, also called "System F" or "L2", although confluence could be in principle established once and for all over the lambda cube [6], for which intrinsic typing is more problematic.

Let us recall the basic idea behind System F: If we consider the identity function $\lambda x^A.x$, it has type $A \to A$, for a given A. However, we know that it should hold for every A. We can achieve this by introducing a new type abstractor and type former: $\Lambda\alpha.\lambda x^\alpha.x$ has type $\forall\alpha.\alpha \to \alpha$. Another way to think about such a function is as a family of terms indexed by a type.

Let α vary over type-variables. The types of System F are given by the following grammar:

Definition 4.3.1.

$$A, B ::= \alpha \mid A \to B \mid \forall\alpha.A$$

The LF-encoding will be:

```
LF ty : type =
   | arr  : ty → ty → ty
   | all : (ty → ty) → ty
;
```

We can now define the grammar for the terms of System F:

Definition 4.3.2.

$$M, N ::= x \mid \alpha \mid M\ N \mid \lambda x^A.M \mid M\ A \mid \Lambda\alpha.M$$

The changes introduced by System F are type application $M\ A$, which is the application of a function M to a type A, and type abstraction $\Lambda\alpha.M$, which denotes the function that takes as an argument a type α and yields a term M.

In Beluga we encode terms with the following type-family:

```
LF tm : ty → type =
| abs : (tm A → tm B) → tm (arr A B)
| app : tm (arr A B) → tm A → tm B
| tlam : ({a:ty} tm (A a)) → tm (all A)
| tapp : tm (all A) → {B:ty} tm (A B)
;
```

In System F, there are two rules for β-reduction: `rbeta` is for the application of a function to a term, `rbetall` for the application of a function to a type. Furthermore, the congruence rules are as expected.

```
LF step : tm A → tm A → type =
| rbeta : step (app (abs M) N) (M N)
| rbetall : step (tapp (tlam M) A) (M A)
| rabs  : ({x : tm A} step (M x) (M' x))
            → step (abs M) (abs M')
| rtlam  : ({x : ty} step (M x) (M' x))
            → step (tlam M) (tlam M')
| rappl : step M M' → step (app M N) (app M' N)
```

```
| rappr : step N N' → step (app M N) (app M N')
| tappl : step M M' → step (tapp M A) (tapp M' A)
;
```

Note how the signature of `step` enforces the invariant that reduction preserves typing. Since we now have two kinds of variables, the judgment is parametric both on terms and on types. This corresponds to the fact that we have to encode two different kinds of assumptions when talking about open terms.

Luckily, Beluga contexts may consist of records of varying shapes. A schema definition for such heterogeneous contexts looks as follows:

$$S := S_1 + S_2 + \cdots + S_n$$

Here each S_i corresponds to a separate block definition, where implicitly existentially quantified parameters are local to each block, and each $p \in \gamma : S$ has to match one of the S_i.

When proving statements involving `step`, we will then need to quantify over contexts satisfying the following schema:

```
schema cxt = tm A + ty;
```

The statement of the Church-Rosser theorem holds for polymorphic lambda calculus is the same:

Theorem 4.3.1 (Church-Rosser for System F)**.** *System F satisfies the Church-Rosser property, both for β-reduction and for $\beta\eta$-reduction.*

The technique to prove this theorem is practically identical to the one we exploited to show Church-Rosser for the untyped lambda calculus. In particular, to prove confluence for β-reduction, one can use the TMT method, by defining a suitable parallel reduction:

```
LF pred : tm A → tm A → type =
| lm : ({x:tm A} pred x x → pred (M x) (M' x))
       → pred (abs M) (abs M')
| ap : pred M1 M1' → pred M2 M2'
       → pred (app M1 M2) (app M1' M2')
| tlm : ({X:ty} pred (M X) (M' X))
       → pred (tlam M) (tlam M')
| tap : pred M M'
                → pred (tapp M A) (tapp M' A)
| beta : ({x:tm A} pred x x → pred (M1 x) (M1' x)) → pred M2 M2'
         → pred (app (abs M1) M2) (M1' M2')
| tbeta : ({X:ty} pred (M1 X) (M1' X))
                 → pred (tapp (tlam M1) A) (M1' A)
;
```

The proof is then carried out in the exact same way, except for the definition of suitable heterogeneous context schemas which add the possibility of having `ty` assumptions.

```
schema pctx = some [a:ty] block(x:tm a, v:pred x x) + ty;
schema cctx = some [a:ty] block(x:tm a, u: notabs x, v: cd x x) + ty;
schema pcctx = some [a:ty] block(x:tm a, w: pred x x, u: notabs x, v: cd x x) +
    ty;
```

We define complete development as before with a judgment `cd` and we can then prove that the triangle property holds for `cd`:

```
tri : (g:pcctx) {M : [g ⊢ tm A]} [g ⊢ cd M M'] → [g ⊢ pred M M'']
                → [g ⊢ pred M'' M']
```

We then exploit `tri` to prove the diamond property for `pred`:

```
dia : (g:pcctx){M: [g ⊢ tm A]}[g ⊢ pred M M'] → [g ⊢ pred M M'']
                → [g ⊢ dia_prop M' M'']
```

The proofs of the equivalence between the multi-step β-reduction and its parallel counterpart and the strip property are a simple extension of the untyped lambda calculus case. Finally, we can state confluence for ordinary β-reduction in the polymorphic lambda calculus:

```
conf : (g:pcctx){M: [g ⊢ tm A]}[g ⊢ mstep_pred M M'] → [g ⊢ mstep_pred M M'']
        → [g ⊢ conf_prop M' M'']
```

For all the details we refer to the online repository.

4.4 Computational Level

In the two-level approach endorsed by Beluga, the syntax and the semantics of a formal system are specified at the LF level, whereas reasoning is carried out at the computational level in form of recursive functions. LF features hypothetical judgments and those give for free momentous properties of contexts, such as exchange, weakening and substitution, which need not to be stated and proved on a case by case need. This *is* the spirit of LF as shown in many remarkable applications (see `http://twelf.org/wiki/Main_Page`) and [25], for its application to our case study.

On the flip side, communication of context information to the reasoning level requires possibly complex schema and relations among them that may look too "noisy". This is a point made by Accattoli, in his work [2] in Abella, where he showed how encoding judgments directly in Abella's meta-logic significantly streamlined the proof of the Church-Rosser theorem. While Beluga and Abella's meta-theory have significant differences, it warrants the investigation of the pros and cons w.r.t. our current experiment.

In fact, BELUGA, since [27], allows one to define inductive (and) stratified relations at the meta-level. The idea is therefore to keep at the LF level only the syntax and move all the other judgments at the computation level: to wit

```
inductive pred : (g:ctx) [g ⊢ term] → [g ⊢ term] → ctype = ….
```

However, we have to accept that `pred` cannot be hypothetical anymore and that context properties such as weakening have to be proven explicitly. The question is what are the trade-offs. It turns out that the context exchange property is more delicate than expected and requires significant changes to the setup, which are not required in Abella. Basically, we have to "explain" to BELUGA's meta-logic what a variable is and this entails, in some sense, redeveloping a bit of the de Bruijn infrastructure.

In fact, we need to slightly change the definition of parallel reduction for variables by exploiting another judgment on terms:

```
inductive isVar : (g : ctx) {M: [g ⊢ term]} ctype =
  | vp: {#q: [g ⊢ term]} isVar [g ⊢ #q]
;
```

We can now reformulate the parallel reduction definition as follows, where the variable case is reified by the `isVar` judgment and no hypothetical assumptions are allowed. On the other hand, no blocks are needed.

```
inductive pred : (g:ctx) [g ⊢ term] → [g ⊢ term] → ctype =
| var : isVar [g ⊢ M]
          → pred [g ⊢ M] [g ⊢ M]
| beta : pred [g, x:term ⊢ M1[…,x]] [g, x:term ⊢ M1'[…,x]] → pred [g ⊢ M2] [
    g ⊢ M2']
          → pred [g ⊢ app (lam \x.M1[…,x]) M2] [g ⊢ M1'[…,M2']]
| lm : pred [g, x:term ⊢ M[…,x]] [g, x:term ⊢ M'[…,x]]
       → pred [g ⊢ lam \x.M[…,x]] [g ⊢ lam \x.M'[…,x]]
| ap : pred [g ⊢ M1] [g ⊢ M1'] → pred [g ⊢ M2] [g ⊢ M2']
       → pred [g ⊢ app M1 M2] [g ⊢ app M1' M2']
;
```

We then proceed in the same way as we did for the proof at the specification level, this time without worrying about the contexts in which every use of the predicate `pred` makes sense, i.e., without giving the set of current assumptions under which the relation holds.

As an example, we present the proof of reflexivity of parallel reduction at the computational level:

```
rec refl_par : {g:ctx}{M : [g ⊢ term]} pred [g ⊢ M] [g ⊢ M] =
/ total m (refl_par g m) /
mlam g ⇒ mlam M ⇒ case [g ⊢ M] of
| [g ⊢ #p] ⇒ var (vp _)
| [g ⊢ app M1 M2] ⇒ let h1 = refl_par [g] [g ⊢ M1] in
                     let h2 = refl_par [g] [g ⊢ M2] in
```

```
                        ap h1 h2
| [g ⊢ lam \x.M'[…,x]] ⇒ let h = refl_par [g,x:term] [g,x:term ⊢ M'] in
                          lm h
;
```

As well-known, things turn ugly as soon as we tackle substitution, which is essential, as before, to establish the diamond property for parallel reduction:

```
subst: (g:ctx)
    pred [g, y:term ⊢ M[…,y]] [g, y:term ⊢ M'[…,y]]
                → pred [g ⊢ N] [g ⊢ N']
                → pred [g ⊢ M[…,N]] [g ⊢ M'[…,N']] =
```

It is not the case that in the LF version this property came for free, because it did not. However, it was straightforward. When contexts are explicit instead, in the lambda case we will need to establish weakening. This in turn requires a form of *exchange* in the same case.

One way to prove exchange lemma, is to find an appropriate notion. One such is context *swapping*, where we only relate two variables at a time. The way to implement it in Beluga is via a ternary relation, where the swapping is actually realized by a substitution $\$S$ that witnesses the swap from a context `g` to a context `h`. Note that this is the first time we use Beluga's feature of *first-class* substitutions, where we can quantify over those.

```
inductive swap : {g:ctx} {h:ctx}{$S: [h ⊢ g]} ctype =
| sw_nil : swap [g] [g] [ g  ⊢ … ]
| sw_last : swap [g,x:term,y:term] [g,x:term,y:term] [g, x:term, y:term ⊢ …, y,
    x]
| sw_ind : swap [g] [h] [h ⊢ $S] → swap [g,x:term] [h,x:term] [h, x ⊢ $S[…],
   x]
;
```

The plan would be to show a parallel reduction that is stable under swapping. Unfortunately, this requires for Beluga to realize that `#p[$S]` is a variable; this is clearly true, but `#p[$S]` is a first-class *parameter* substitution, currently not supported by Beluga.

This requires a technical roundabout[2], which we will not discuss in detail. The idea behind our definition of parallel reduction is that if `isVar [g ⊢ M[$S]]` exists, then we can easily conclude that there is a parallel reduction using the rule `var`; thus, we need to prove the following lemma:

```
rec swap_var :  isVar [g ⊢ M] →
                 swap [g] [h] [h ⊢ $S] →
                        isVar [h ⊢ M[$S]]
```

[2]Again, kindly suggested by Brigitte Pientka

The proof of this lemma requires to show other two properties regarding the predicate `isVar`, namely `shift` and `is_var`, but we refer the reader to the Appendix A.1 for the details.

With all of this said, we are finally able to prove that parallel reduction is preserved by swapping:

```
rec exc0 : (g:ctx) (h:ctx) pred [g ⊢ M] [g ⊢ M']
                           → swap [g] [h] [h ⊢ $S]
                           → pred [h ⊢ M[$S]] [h ⊢ M'[$S]] =
```

We then specialize it at the case in which the substitution involved in the swap between contexts simply exchanges the last two variables in the context, that is what we need in particular to prove weakening.

```
rec exc1 : (g:ctx) pred [g,w:term,y:term ⊢ M[…,w,y]]
                        [g,w:term,y:term ⊢ M'[…,w,y]]
                  → pred [g,y:term,x:term ⊢ M[…,x,y]]
                         [g,y:term,x:term ⊢ M'[…,x,y]] =
/ total d (exc1 _ _ _ d) /
fn d ⇒  exc0 d sw_last
;
```

The weakening property is then easily proved by induction on the reduction relation:

```
rec weak : (g:ctx) pred [g ⊢ N] [g ⊢ N']
           → pred [g, y:term ⊢ N[…]] [g, y: term ⊢ N'[…]] =
/ total d (weak _ _ _ d) /
fn d ⇒ case d of
  | var (vp [_ ⊢ #q] ) ⇒ var (vp _)
  | beta d1 d2 ⇒ let ih1 = weak d1 in
                 let ih2 = weak d2 in
                 let d1' = exc1 ih1 in
                  beta d1' ih2
| lm d' ⇒ let ih = weak d' in
           let d'' = exc1 ih in
           lm d''
| ap d1 d2 ⇒ let ih1 = weak d1 in
              let ih2 = weak d2 in
              ap ih1 ih2
;
```

The substitution lemma has now all the ingredients needed to be proved:

```
rec subst : (g:ctx) pred [g, y:term ⊢ M[…,y]] [g, y:term ⊢ M'[…,y]] →
                    pred [g ⊢ N] [g ⊢ N']
                    → pred [g ⊢ M[…,N]] [g ⊢ M'[…,N']] =
fn d ⇒ fn s ⇒ case d of
  | var (vp [_, y:term ⊢ #q] )
        ⇒ (case [_, y:term ⊢ #q] of
```

```
        | [_, y:term ⊢ y] ⇒ s
        | [_, y:term ⊢ #q[…]] ⇒ var (vp _))
| beta d1 d2 ⇒ let ih2 = subst d2 s in
               let s2 = weak s in
               let ih1 = subst (exc1 d1) s2 in
               beta ih1 ih2
| lm d' ⇒  let s2 = weak s in
           let ih = subst (exc1 d') s2 in lm ih
| ap d1 d2 ⇒ let ih1 = subst d1 s in
             let ih2 = subst d2 s in
             ap ih1 ih2
;
```

The proof of the diamond property has now a much cleaner presentation, which we invite the reader to read in the Appendix A.1:

```
inductive dia_prop: (g:ctx) [g ⊢ term] → [g ⊢ term] → ctype =
| dia_result: pred [g ⊢ M'] [g ⊢ N] → pred [g ⊢ M''] [g ⊢ N] → dia_prop [g
   ⊢ M'] [g ⊢ M'']
;

dia : (g:ctx) pred [g ⊢ M] [g ⊢ M1] → pred [g ⊢ M] [g ⊢ M2]
         → dia_prop [g ⊢ M1] [g ⊢ M2]
```

We evaluate this solution further in the next Chapter.

Chapter 5

Conclusion

5.1 Evaluation

The Church-Rosser theorem's representation in Beluga displays several implementation techniques: higher-order abstract syntax (HOAS), judgments-as-types, and proofs as recursive functions. As expected, and as already pointed out by Pfenning in [25] for β-reduction, with these approaches the user can focus on the mathematical content of the proof, ignoring tedious details regarding variable bindings, capture-avoiding substitutions and context-related properties. Since this has been the object of several papers in the literature, viz. [13], we will be concise.

If we compare our formalization in Beluga with, for example, the de Bruijn indices notation used in [24] in Isabelle/HOL we notice that while the basic mathematical ideas are practically the same (and in fact we have followed his lead to prove confluence for $\eta\beta$), Nipkow puts much effort to deploy the de Bruijn representation and proving it correct, see in particular the discussion of the possible encoding of η-reduction, and related technical lemmas; in Beluga, instead, everything is taken care of by the meta-language: η-reduction does not warrant any particular care, the proviso on its application being handled by a vacuous meta-function. We do need a simple strengthening lemma, which is ubiquitous in other HOAS implementations such as the ones in the Abella library (`https://abella-prover.org/`), where they are known as *pruning* lemma.

Another obvious difference with [24] is that many of the proofs in Nipkow's formalization are found automatically once the right series of lemmas have been developed. In contrast, in Beluga, almost all of the details we find in the informal proof are specified in the mechanization. This is just in the nature of essentially a proof checker such as Beluga compared to a proof assistant such as Isabelle/HOL, which sports the best automation in the field. Of course, there are other encodings of

our theorem that sit in the middle, for example the one in Isabelle Nominal[1]

A disadvantage in using Beluga that looms large in this case study is the impossibility to quantify over relations. This is because Beluga's reasoning logic is first-order and this results in an annoying repetition of definitions and lemmas for every single reduction involved in our proof. Whereas, it would be handy to prove the main properties over *abstract* reduction system and then instantiate them, as Nipkow very elegantly does. However, HOAS and higher order quantification are notoriously problematic. A middle ground would be to rely on rich logic such as Coq and use a library to obtain a form of HOAS. A case in point is AutoSubst [31], see a start in this direction in [18].

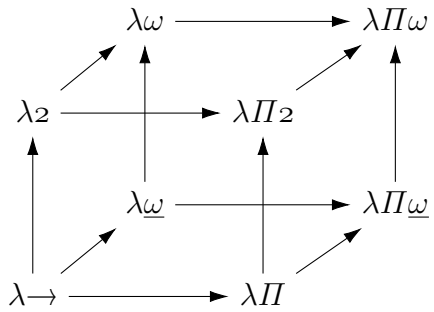
Beluga and *Abella* share the same two-level approach and in fact, the TMT proof in the latter has the same structure. A main difference, though, is that Beluga features a dependently typed specification logic (LF). When extending the Church-Rosser proof from the untyped lambda calculus to a typed setting such as System F, in Beluga we can do this effortlessly by using intrinsic terms. In Abella, while proving confluence for System F is definitely doable, it has to be done maintaining the type invariants explicitly, and this makes proof tedious and fragile.

Lastly, Accattoli, in his remarkable work [2] in Abella, makes a strong case of carrying out formalizations of the meta-theory of the untyped lambda calculus directly at the computational level, since variable contexts are automatically managed by the meta-logic. Since the latter incorporates a notion of equivariance, issue that has troubled us, namely formalizing and proving exchange over contextual types disappear. However, this approach works (beautifully) until it does not, namely if our object logic is typed. That's because the type system of Abella's meta-logic does not support dependent types. A proof of confluence of System F in Abella carried out in the meta-logic would be, in our estimate, a nightmare.

5.2 Future Work

Pure type systems (PTSs) constitute a framework introduced for the first time by Barendregt in [7] that captures several typed lambda calculi. In particular, he proves that systems like STLC, L2, and LF, can be considered sub-systems of the Calculus of Constructions (CC) by defining which forms of abstraction and (dependent) function type formation are permitted via a system of sorts. PTSs are a generalization of the so-called λ-cube, which represents the inclusion hierarchy of the various systems.

[1] `https://isabelle.in.tum.de/library/HOL/HOL-Nominal-Examples/CR.html`



Seen as pre-terms, that is without the sorting system, PTS are but untyped lambda calculus with abstraction and dependent products. As such, it is not difficult to establish the Church-Rosser for β, see for example [20]. However, to then claim to have established the property over the whole cube, we need to impose the sorting discipline. This is well-known to be problematic, although more or less elegant solutions have been presented [20; 3] and [18]. HOAS solutions have not been forthcoming and it would be interesting to tackle that. This entails addressing two problems:

1. It is not obvious how to give an intrinsically-typed encoding of PTS,

2. more vitally, the Church-Rosser property for $\beta\eta$ does *not* hold for arbitrary PTS [16], although some well-behaved fragments have been located.

Ringraziamenti

Grazie al mio relatore Alberto, che con pazienza e competenza mi ha accompagnato passo dopo passo in questi mesi; grazie per la professionalità e l'umanità, è stato un onore lavorare con lei.

Grazie a Brigitte Pientka per l'estrema disponibilità e gentilezza, aver avuto l'opportunità di conoscerla è un privilegio per me.

Grazie alla mia famiglia e ai miei amici vecchi e nuovi, vi voglio bene.

Grazie a Mattia per essere stato la mia roccia in tutti questi anni, per aver creduto in me più di quanto io potessi fare e per avermi spinta fino al traguardo.

Appendix A

Complete formalizations for β

```
rec multi_star : (g:ctx) [g ⊢ mstep M N] → [g ⊢ beta_red* M N] =
/ total d (multi_star _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ id1] ⇒ [g ⊢ id_b*]
| [g ⊢ step1 D1 D2] ⇒ let [g ⊢ IH2] = multi_star [g ⊢ D2] in
                        [g ⊢ tr_b* (cl_b* D1) IH2]

;

rec star_multi : (g:ctx) [g ⊢ beta_red* M N] → [g ⊢ mstep M N] =
/ total d (star_multi _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ id_b*] ⇒ [g ⊢ id1]
| [g ⊢ tr_b* D1 D2] ⇒ let [g ⊢ IH1] = star_multi [g ⊢ D1] in
                      let [g ⊢ IH2] = star_multi [g ⊢ D2] in
                      trans_mstep [g ⊢ IH1] [g ⊢ IH2]
| [g ⊢ cl_b* D] ⇒ [g ⊢ step1 D id1]
;
```

□

Equivalence between `beta_red*` *and* `mstep`. $\longrightarrow_\beta^=$ *is a congruence.*

```
rec lm= : (g:ctx) [g,x:term ⊢  beta_red= M M']
      →  [g ⊢ beta_red= (lam \x.M) (lam \x.M')] =
/ total d (lm= _ _ _ d) /
fn d ⇒ case d of
| [g,x:term ⊢  id_b=] ⇒ [g ⊢ id_b=]
| [g,x:term ⊢  cl_b= D] ⇒ [g ⊢ cl_b= (lm1 \x.D[…,x])]
;

rec apl= :  (g:ctx) {M2:[g ⊢ term]}[g ⊢ beta_red= M1 M1']
```

```
                                        → [g ⊢ beta_red= (app M1 M2) (app M1' M2)] =
/ total s (apl= _ _ _ s) /
mlam M2 ⇒ fn s ⇒ case s of
| [g ⊢ id_b=] ⇒ [g ⊢ id_b=]
| [g ⊢ cl_b= S] ⇒ [g ⊢ cl_b= (apl1 S)]
;

rec apr= : (g:ctx){M1:[g ⊢ term]} [g ⊢ beta_red= M2 M2'] → [g ⊢ beta_red= (
    app M1 M2) (app M1 M2')] =
/ total s (apr= _ _ _ s) /
mlam M1 ⇒ fn s ⇒ case s of
| [g ⊢ id_b=] ⇒ [g ⊢ id_b=]
| [g ⊢ cl_b= S] ⇒ [g ⊢ cl_b= (apr1 S)]
;
```

□

Strengthening Lemma for β.

```
inductive exStrBetaRed : (g:ctx) [g,x:term ⊢ beta_red M[…] N] → ctype =
| exStrBeta : {D : [g,x:term ⊢ beta_red M[…] N]}
              [g ⊢ beta_red M N'] → [g,x:term ⊢ eq N'[…] N]
              → exStrBetaRed [g,x:term ⊢ D]
;

rec strengthen_beta : {g : ctx} {D : [g, x:term ⊢ beta_red M[…] N]} →
    exStrBetaRed [g, x:term ⊢ D] =
 %/ total d (strengthen_beta _ _ _ d) /
 mlam g ⇒ mlam D ⇒ case [g, x:term ⊢ D] of
 | [g,x:term ⊢ beta1] ⇒ exStrBeta [g, x:term ⊢ D[…]] [g ⊢ beta1] [g,x:term ⊢
     refl]
 | [g,x:term ⊢ lm1 \y.D'[…,x,y]] ⇒
            let exStrBeta [g,x:term,y:term ⊢ D'[…,x,y]] [g,x:term ⊢ H1] e =
            strengthen_beta [g,x:term] [g,x:term,y:term ⊢ D'[…,y,x]] in
            let [g,x:term,y:term ⊢ refl] = e in
            exStrBeta [g, x:term ⊢ D] [g ⊢ lm1 \x.H1[…,x]] [g,x:term ⊢ refl]
 | [g, x:term ⊢ apl1 D1] ⇒
            let exStrBeta [g,x:term ⊢ D1[…,x]] [g ⊢ H1] e =
            strengthen_beta [g] [g,x:term ⊢ D1[…,x]] in
            let [g,x:term ⊢ refl] = e in
            exStrBeta [g, x:term ⊢ D] [g ⊢ apl1 H1] [g,x:term ⊢ refl]
 | [g, x:term ⊢ apr1 D2] ⇒
            let exStrBeta [g,x:term ⊢ D2[…,x]] [g ⊢ H2] e =
            strengthen_beta [g] [g,x:term ⊢ D2[…,x]] in
            let [g,x:term ⊢ refl] = e in
            exStrBeta [g, x:term ⊢ D] [g ⊢ apr1 H2] [g,x:term ⊢ refl]
 ;
```

□

A.1 Computational Level

Exchange Lemma.

```
rec shift :  isVar [g, x:term  #p]
                isVar [g, y:term, x:term  #p[…,x]] =
/ total v (shift _ _ v) /
fn v    let vp [g, x:term ⊢ #p] = v in
       vp [g, y:term, x:term ⊢ #p[…,x]]
;

rec swap_var :  isVar [g  M]
              swap [g] [h] [h  S]isVar[hM[S]] =
/ total s (swap_var _ _ _ _ v s) /
 fn v    fn s    let vp [g, x : term  ⊢ #q] = v in
      (case s of
        | sw_nil   vp _
        | sw_ind sw
          (case [g, x:term ⊢ #q] of
         | [g, x:term ⊢ x] ⇒ vp _
         | [g, x:term ⊢ #q[…]] ⇒
                let ih = swap_var (vp [g ⊢ #q]) sw in
                vn ih
                )
        | sw_last   vp _)
;


rec exc0 : (g:ctx) (h : ctx) pred [g ⊢ M] [g ⊢ M'] 
                           → swap [g] [h] [h  S]− > pred[h| − M[S]] [h ⊢ M'[
```

□

Diamond property at the computational level.

```
inductive dia_prop: (g:ctx) [g ⊢ term] → [g ⊢ term] → ctype =
| dia_result: pred [g ⊢ M'] [g ⊢ N] → pred [g ⊢ M''] [g ⊢ N] → dia_prop [g
    ⊢ M'] [g ⊢ M'']
;
rec dia : (g:ctx) pred [g ⊢ M] [g ⊢ M1] → pred [g ⊢ M] [g ⊢ M2]
          → dia_prop [g ⊢ M1] [g ⊢ M2] =
/ total {d s} (dia _ _ _ _ d s) /
fn d ⇒ fn s ⇒ case d of
| var d' ⇒ let exV [g ⊢ #p] refl = is_var d' in
            let var s' = s in
            dia_result (var d') (var s')
| ap d1 d2 ⇒
    (case s of
    | var s' ⇒ let exV [g ⊢ #p] e = is_var s' in
                impossible e
```

```
    | ap s1 s2 ⇒
           let (dia_result h1 h2) = dia d1 s1 in
           let (dia_result h1' h2') = dia d2 s2 in
           dia_result (ap h1 h1') (ap h2 h2')
    | beta s1 s2 ⇒
           (case d1 of
           | lm r1 ⇒
               let (dia_result t2 t2') = dia d2 s2 in
                   let (dia_result t1 t1') = dia r1 s1 in
                   let s'' = subst t1' t2' in
                   dia_result (beta t1 t2) s''
           | var r1 ⇒ let exV [g ⊢ #p] e = is_var r1 in
                      impossible e
           )
      )
| lm d' ⇒
    (case s of
    | var s' ⇒ let exV [g ⊢ #p] e = is_var s' in
               impossible e
    | lm d'' ⇒
        let (dia_result ih1 ih1') = dia d' d'' in
        dia_result (lm ih1) (lm ih1')
    )
| beta d1 d2 ⇒
    (case s of
    | var s' ⇒ let exV [g ⊢ #p] e = is_var s' in
               impossible e
    | ap s1 s2 ⇒ (case s1 of
           | var s1' ⇒ let exV [g ⊢ #p] e = is_var s1' in
               impossible e
           | lm s1' ⇒
               let (dia_result ih1 ih1') = dia d1 s1' in
               let (dia_result ih2 ih2') = dia d2 s2 in
               let s = subst ih1 ih2 in
               dia_result s (beta ih1' ih2')
           )
    | beta s1 s2 ⇒ let (dia_result ih2 ih2') = dia d2 s2 in
               let (dia_result ih1 ih1') = dia d1 s1 in
               let s = subst ih1 ih2 in
               let s' = subst ih1' ih2' in
               dia_result s s'
    )
;
```

□

Appendix B

Complete formalizations for η

$\longrightarrow_\eta^=$ *is a congruence.*

```
rec cong_lam= : (g:ctx) [g,x:term ⊢  eta_red= M M']
      →  [g ⊢ eta_red= (lam \x.M) (lam \x.M')] =
/ total d (cong_lam= _ _ _ d) /
fn d ⇒ case d of
| [g,x:term ⊢  id_e=] ⇒ [g ⊢ id_e=]
| [g,x:term ⊢  cl_e= D] ⇒ [g ⊢ cl_e= (cong_lam \x.D[…,x])]
;

rec cong_app1= :  (g:ctx) {M2:[g ⊢ term]}[g ⊢ eta_red= M1 M1']
                                  → [g ⊢ eta_red= (app M1 M2) (app M1' M2)] =
/ total s (cong_app1= _ _ _ s) /
mlam M2 ⇒ fn s ⇒ case s of
| [g ⊢ id_e=] ⇒ [g ⊢ id_e=]
| [g ⊢ cl_e= S] ⇒ [g ⊢ cl_e= (cong_app1 S)]
;

rec cong_app2= : (g:ctx){M1:[g ⊢ term]} [g ⊢ eta_red= M2 M2'] → [g ⊢ eta_red=
     (app M1 M2) (app M1 M2')] =
/ total s (cong_app2= _ _ _ s) /
mlam M1 ⇒ fn s ⇒ case s of
| [g ⊢ id_e=] ⇒ [g ⊢ id_e=]
| [g ⊢ cl_e= S] ⇒ [g ⊢ cl_e= (cong_app2 S)]
;
```

□

$\longrightarrow_\eta^*$ *is a congruence.*

```
rec cong_lam* : (g:ctx) [g,x:term ⊢  eta_red* M M']
      →  [g ⊢ eta_red* (lam \x.M) (lam \x.M')] =
/ total d (cong_lam* _ _ _ d) /
fn d ⇒ case d of
```

```
| [g, x:term ⊢ id_e*] ⇒ [g ⊢ id_e*]
| [g, x:term ⊢ tr_e* D1 D2] ⇒
    let [g ⊢ H1] = cong_lam* [g,x:term ⊢ D1] in
    let [g ⊢ H2] = cong_lam* [g,x:term ⊢ D2] in
    [g ⊢ tr_e* H1 H2]
| [g, x:term ⊢ cl_e* D] ⇒
       [g ⊢ cl_e* (cong_lam \x.D)]
;

rec cong_app1* :  (g:ctx) {M2:[g ⊢ term]}[g ⊢ eta_red* M1 M1']
                                    → [g ⊢ eta_red* (app M1 M2) (app M1' M2)] =
/ total s (cong_app1* _ _ _ s) /
mlam M2 ⇒ fn d ⇒ case d of
| [g ⊢ id_e*] ⇒ [g ⊢ id_e*]
| [g ⊢ tr_e* D1 D2] ⇒
    let [g ⊢ H1] = cong_app1* [g ⊢ M2] [g ⊢ D1] in
    let [g ⊢ H2] = cong_app1*[g ⊢ M2] [g ⊢ D2] in
    [g ⊢ tr_e* H1 H2]
| [g ⊢ cl_e* D] ⇒
    [g ⊢ cl_e* (cong_app1 D)]
;

rec cong_app2* :  (g:ctx) {M1:[g ⊢ term]}[g ⊢ eta_red* M2 M2']
                                    → [g ⊢ eta_red* (app M1 M2) (app M1 M2')] =
/ total s (cong_app2* _ _ _ s) /
mlam M1 ⇒ fn d ⇒ case d of
| [g ⊢ id_e*] ⇒ [g ⊢ id_e*]
| [g ⊢ tr_e* D1 D2] ⇒
    let [g ⊢ H1] = cong_app2* [g ⊢ M1] [g ⊢ D1] in
    let [g ⊢ H2] = cong_app2*[g ⊢ M1] [g ⊢ D2] in
    [g ⊢ tr_e* H1 H2]
| [g ⊢ cl_e* D] ⇒
    [g ⊢ cl_e* (cong_app2 D)]
;
```

□

Appendix C

Complete formalizations for $\beta\eta$

Strong commutation between β and η.

```
    LF beta_eta_square : term → term → type =
| beta_eta_square_result : eta_red* M1 N → beta_red= M2 N
                          → beta_eta_square M1 M2
;

rec be_square : (g:ctx) {M : [g ⊢ term]}{M1 : [g ⊢ term]}{M2 : [g ⊢ term]}
          [g ⊢ beta_red M M1] → [g ⊢ eta_red M M2]
          → [g ⊢ beta_eta_square M1 M2] =
/ total d (be_square _ _ _ _ d _) /
mlam M ⇒ mlam M1 ⇒ mlam M2 ⇒ fn d ⇒ fn s ⇒
    case d of
    | [g ⊢ beta1] : [g ⊢ beta_red (app (lam (\x. M')) M1') (M'[…, M1'])] ⇒
        (case s of
        | [g ⊢ cong_app1 S] ⇒
            (case [g ⊢ S] of
            | [g ⊢ eta] ⇒ [g ⊢ beta_eta_square_result id_e* id_b=]
            | [g ⊢ cong_lam \x.S'[…,x]] ⇒
                  let [g ⊢ S''] = subst_eta_closed [g ⊢ M1'] [g,x:term ⊢ S'[…,
    x]] in
                  [g ⊢ beta_eta_square_result (cl_e* S'') (cl_b= beta1)]
            )
        | [g ⊢ cong_app2 S] : [g ⊢ eta_red (app (lam (\x. M')) N) (app (lam (\
    x. M')) M2')] ⇒
              let [g ⊢ S'] = subst_eta [g] [g, x:term ⊢ M'] [g ⊢ S] in
              [g ⊢ beta_eta_square_result S' (cl_b= beta1)]
        )
    | [g ⊢ lm1 \x.D[…,x]] ⇒
        (case s of
        | [g ⊢ eta] ⇒ (case [g, x:term ⊢ D] of
                       | [g, x:term ⊢ apl1 D'] ⇒
                           let exStrBeta [g,x:term ⊢ D'] [g ⊢ H1] e =
                           strengthen_beta [g] [g,x:term ⊢ D'] in
```

```
                        let [g,x:term ⊢ refl] = e in
                        [g ⊢ beta_eta_square_result (cl_e* eta) (cl_b= H1)]
                    | [g, x:term ⊢ beta1] ⇒
                        [g ⊢ beta_eta_square_result (id_e*) (id_b=)]
                    )

        | [g ⊢ cong_lam \y.S[…,y]] ⇒
            let [g,x:term ⊢ beta_eta_square_result H1 H2] = be_square _ _ _ [g,
    x:term ⊢ D] [g,x ⊢ S] in
            let [g ⊢ H1'] = cong_lam* [g,x:term ⊢ H1[…,x]] in
            let [g ⊢ H2'] = lm= [g,x:term ⊢ H2[…,x]] in
            [g ⊢ beta_eta_square_result H1' H2']
        )
    | [g ⊢ apl1 D] : [g ⊢ beta_red (app M1' M') (app M2' M')] ⇒
        (case s of
        | [g ⊢ cong_app1 S] : [g ⊢ eta_red (app N1' N') (app N2' N')] ⇒
            let [g ⊢ beta_eta_square_result H1 H2] =
            be_square _ _ _ [g ⊢ D] [g ⊢ S] in
            let [g ⊢ H1'] = cong_app1* [g ⊢ N'] [g ⊢ H1] in
            let [g ⊢ H2'] = apl= [g ⊢ N'] [g ⊢ H2] in
            [g ⊢ beta_eta_square_result H1' H2']
        | [g ⊢ cong_app2 S] : [g ⊢ eta_red (app N' N1') (app N' N2')] ⇒
            let [g ⊢ H1] = cong_app2* [g ⊢ M2'] [g ⊢ (cl_e* S)] in
            let [g ⊢ H2] = apl= [g ⊢ N2'] [g ⊢ (cl_b= D)] in
            [g ⊢ beta_eta_square_result H1 H2]
        )
    | [g ⊢ apr1 D] : [g ⊢ beta_red (app M' M1') (app M' M2')] ⇒
        (case s of
        | [g ⊢ cong_app1 S] : [g ⊢ eta_red (app N1' N') (app N2' N')] ⇒
            let [g ⊢ H1] = cong_app1* [g ⊢ M2'] [g ⊢ (cl_e* S)] in
            let [g ⊢ H2] = apr= [g ⊢ N2'] [g ⊢ (cl_b= D)] in
            [g ⊢ beta_eta_square_result H1 H2]
        | [g ⊢ cong_app2 S] : [g ⊢ eta_red (app M' M1') (app M' M2')] ⇒
            let [g ⊢ beta_eta_square_result H1 H2] =
            be_square _ _ _ [g ⊢ D] [g ⊢ S] in
            let [g ⊢ H1'] = cong_app2* [g ⊢ M'] [g ⊢ H1] in
            let [g ⊢ H2'] = apr= [g ⊢ M'] [g ⊢ H2] in
            [g ⊢ beta_eta_square_result H1' H2']
        )
;
```

□

Steps in Commutation Lemma for β and η.

```
%Preliminary lemma
rec betaeq_betastar : (g:ctx) [g ⊢ beta_red= M N] → [g ⊢ beta_red* M N] =
/ total d (betaeq_betastar _ _ _ d) /
fn d ⇒ case d of
| [g ⊢ id_b=] ⇒ [g ⊢ id_b*]
```

```
| [g ⊢ cl_b= D] ⇒ [g ⊢ cl_b* D]
;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Lemma 3) (square R S T R=) & (S c T) → square R= S T R=
%In particular, we need: 3) square beta eta eta* beta= → square beta= eta eta*
     beta=


rec lemma_three_be : (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
                      [g ⊢ beta_red M M1] → [g ⊢ eta_red M M2]
                      → [g ⊢ eta*_beta=_joinable M1 M2])
                      → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
                      [g ⊢ beta_red= M' M1'] → [g ⊢ eta_red M' M2'] → [g ⊢
    eta*_beta=_joinable M1' M2']) =
/ total s (lemma_three_be _ _ _ _ _ s _) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
    case s of
      | [g ⊢ id_b=] ⇒ let [g ⊢ T] = t in
                      [g ⊢ eta*_beta=_result (cl_e* T) id_b=]
      | [g ⊢ cl_b= S] : [g ⊢ beta_red= M' M1'] ⇒ let [g ⊢ T] : [g ⊢ eta_red M
    ' M2'] = t in
                      let [g ⊢ D] = d [g ⊢ M'] [g ⊢ M1'] [g ⊢ M2'] [g ⊢ S] [g
     ⊢ T] in
                      let [g ⊢ eta*_beta=_result H1 H2] = [g ⊢ D] in
                      [g ⊢ eta*_beta=_result H1 H2]
;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Lemma 1) : square R S T U → square S R U T
%We use it twice in the Commutation Lemma's proof:

%1.1) square beta= eta eta* beta= → square eta beta= beta= eta*

LF beta=_eta*_joinable : term → term → type =
| beta=_eta*_result : beta_red= M1 N → eta_red* M2 N → beta=_eta*_joinable M1
    M2
;

rec lemma_one_first_be : (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢
    term]}
                      [g ⊢ beta_red= M M1] → [g ⊢ eta_red M M2]
                      → [g ⊢ eta*_beta=_joinable M1 M2])
                       → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
                       [g ⊢ eta_red M' M1'] → [g ⊢ beta_red= M' M2'] → [g ⊢
    beta=_eta*_joinable M1' M2']) =
```

```
/ total t (lemma_one_first_be _ _ _ _ _ _ t) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
    let [g ⊢ eta*_beta=_result H1 H2] = d _ _ _ t s in
    [g ⊢ beta=_eta*_result H2 H1]
;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Lemma 4) : square R S S T → square R* S S T*
%We use it twice in the Commutation Lemma's proof:

%4.1) square eta beta= beta= eta* → square eta* beta= beta= eta*


rec lemma_four_first_be : (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢
    term]}
                    [g ⊢ eta_red M M1] → [g ⊢ beta_red= M M2]
                    → [g ⊢ beta=_eta*_joinable M1 M2])
                    → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
                    [g ⊢ eta_red* M' M1'] → [g ⊢ beta_red= M' M2'] → [g ⊢
    beta=_eta*_joinable M1' M2']) =
/ total s (lemma_four_first_be _ _ _ _ _ s _) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
    case t of
        | [g ⊢ id_b=] ⇒ let [g ⊢ S] = s in
                [g ⊢ beta=_eta*_result id_b= S]
        | [g ⊢ cl_b= T] : [g ⊢ beta_red= M' M2'] ⇒
                (case s of
                | [g ⊢ id_e*] ⇒ [g ⊢ beta=_eta*_result (cl_b= T) id_e*]
                | [g ⊢ cl_e* S'] : [g ⊢ eta_red* M' M1'] ⇒
                    let [g ⊢ D] = d [g ⊢ M'] [g ⊢ M1'] [g ⊢ M2'] [g ⊢ S'] [
    g ⊢ (cl_b= T)] in
                    let [g ⊢ beta=_eta*_result H1 H2] = [g ⊢ D] in
                    [g ⊢ beta=_eta*_result H1 H2]
                | [g ⊢ tr_e* S1 S2] : [g ⊢ eta_red* M' M1'] ⇒
                    let [g ⊢ S'] : [g ⊢ eta_red* M' M] = [g ⊢ S1] in
                    let [g ⊢ IH1] = lemma_four_first_be d [g ⊢ M'] [g ⊢ M] [
    g ⊢ M2'] [g ⊢ S'] t in
                    let [g ⊢ beta=_eta*_result H1 H2] = [g ⊢ IH1] in
                    let [g ⊢ IH2] = lemma_four_first_be d _ [g ⊢ M1'] _ [g ⊢
     S2] [g ⊢ H1] in
                    let [g ⊢ beta=_eta*_result H1' H2'] = [g ⊢ IH2] in
                    [g ⊢ beta=_eta*_result H1' (tr_e* H2 H2')]
                )
;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%1.2) square eta* beta= beta= eta* → square beta= eta* eta* beta=


rec lemma_one_second_be : (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢
    term]}
                      [g ⊢ eta_red* M M1] → [g ⊢ beta_red= M M2]
                      → [g ⊢ beta=_eta*_joinable M1 M2])
                      → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
                      [g ⊢ beta_red= M' M1'] → [g ⊢ eta_red* M' M2'] → [g ⊢
    eta*_beta=_joinable M1' M2']) =
/ total s (lemma_one_second_be _ _ _ _ _ s _) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
    let [g ⊢ beta=_eta*_result IH1 IH2] = d _ _ _ t s in
    [g ⊢ eta*_beta=_result IH2 IH1]
;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%4.2) square beta= eta* eta* beta= → square beta* eta* eta* beta*

LF commute : term → term → type =
| commute_result : eta_red* M1 N → beta_red* M2 N → commute M1 M2
;

rec lemma_four_second_be : (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢
    term]}
                      [g ⊢ beta_red= M M1] → [g ⊢ eta_red* M M2]
                      → [g ⊢ eta*_beta=_joinable M1 M2])
                      → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
                      [g ⊢ beta_red* M' M1'] → [g ⊢ eta_red* M' M2'] → [g ⊢
    commute M1' M2']) =
/ total s (lemma_four_second_be _ _ _ _ _ s _) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
    case s of
      | [g ⊢ id_b*] ⇒ let [g ⊢ T] = t in [g ⊢ commute_result T id_b*]
      | [g ⊢ cl_b* S] ⇒ let [g ⊢ eta*_beta=_result H1 H2] = d _ _ _ [g ⊢ (
    cl_b= S)] t in
                          let [g ⊢ H2'] = betaeq_betastar [g ⊢ H2] in
                          [g ⊢ commute_result H1 H2']
      | [g ⊢ tr_b* S1 S2] ⇒ let [g ⊢ commute_result IH1' IH2'] =
                              lemma_four_second_be d _ _ _ [g ⊢ S1] t in
                              let [g ⊢ commute_result IH1'' IH2''] =
                              lemma_four_second_be d _ _ _ [g ⊢ S2] [g ⊢ IH1']
     in
                              [g ⊢ commute_result IH1'' (tr_b* IH2' IH2'')]
;
```

□

Strip property for $\beta^* \cup \eta^*$.

```
LF strip_un_prop: term → term → type =
| strip_un_result: beta*_eta* M' N → union_b*e* M'' N
                → strip_un_prop M' M'';
rec strip_un : (g:rctx)[g ⊢ union_b*e* M M'] → [g ⊢ beta*_eta* M M'']
                → [g ⊢ strip_un_prop M' M''] =
/ total s (strip_un g m m' m'' r s) /
fn r ⇒ fn s ⇒ case s of
| [g ⊢ cl_b*e* S] ⇒
                let [g ⊢ dia_un_res H1 H2] = dia_un r [g ⊢ S] in
                [g ⊢ strip_un_result (cl_b*e* H1) H2]
| [g ⊢ tr_b*e* S1 S2] ⇒
                let [g ⊢ strip_un_result H1 H2] = strip_un r [g ⊢ S1] in
                let [g ⊢ strip_un_result H1' H2'] = strip_un [g ⊢ H2] [g ⊢ S2]
    in
                [g ⊢ strip_un_result (tr_b*e* H1 H1') H2']
;
```

□

Confluence for $\beta^* \cup \eta^*$.

```
LF conf_un_prop : term → term → type =
| conf_un_res : beta*_eta* M' N → beta*_eta* M'' N → conf_un_prop M' M''
;

rec conf_un : (g:ctx) [g ⊢ beta*_eta* M M'] → [g ⊢ beta*_eta* M M'']
                      → [g ⊢ conf_un_prop M' M''] =
/ total {d s} (conf_un _ _ _ _ d s) /
fn d ⇒ fn s ⇒ case d of
| [g ⊢ cl_b*e* D] ⇒
                let [g ⊢ strip_un_result H1 H2] = strip_un [g ⊢ D] s in
                [g ⊢ conf_un_res H1 (cl_b*e* H2)]
| [g ⊢ tr_b*e* D1 D2] ⇒
                let [g ⊢ conf_un_res H1 H2] = conf_un [g ⊢ D1] s in
                let [g ⊢ conf_un_res H1' H2'] = conf_un [g ⊢ D2] [g ⊢ H1] in
                [g ⊢ conf_un_res H1' (tr_b*e* H2 H2')]
;
```

□

Commutative Union Theorem.

```
LF dia_beta_eta : term → term → type =
| dia_result : beta_eta* M1 N → beta_eta* M2 N → dia_beta_eta M1 M2
;

rec hindley_rosen:
```

```
    (g:ctx) ({M: [g ⊢ term]}{M1: [g ⊢ term]}{M2: [g ⊢ term]}
            [g ⊢ beta_red* M M1] → [g ⊢ eta_red* M M2]
            → [g ⊢ commute M1 M2])
                → ({M': [g ⊢ term]}{M1': [g ⊢ term]}{M2': [g ⊢ term]}
                [g ⊢ beta_eta* M' M1'] → [g ⊢ beta_eta* M' M2']
                → [g ⊢ dia_beta_eta M1' M2']) =
/ total {s t} (hindley_rosen _ _ _ _ _ s t) /
fn d ⇒ mlam M' ⇒ mlam M1' ⇒ mlam M2' ⇒ fn s ⇒ fn t ⇒
    case s of
    | [g ⊢ id_be*] ⇒ let [g ⊢ T] = t in [g ⊢ dia_result T id_be*]
    | [g ⊢ tr_be* S1 S2] ⇒
       let [g ⊢ dia_result H1 H2] = hindley_rosen d _ _ _ [g ⊢ S1] t in
       let [g ⊢ dia_result H1' H2'] = hindley_rosen d _ _ _ [g ⊢ S2] [g ⊢ H1]
     in
       [g ⊢ dia_result H1' (tr_be* H2 H2')]
    | [g ⊢ cl_be* S'] ⇒
       (case t of
       | [g ⊢ id_be*] ⇒ [g ⊢ dia_result id_be* (cl_be* S')]
       | [g ⊢ tr_be* T1 T2] ⇒
          let [g ⊢ dia_result H1 H2] = hindley_rosen d _ _ _ s [g ⊢ T1] in
          let [g ⊢ dia_result H1' H2'] = hindley_rosen d _ _ _ [g ⊢ H2] [g ⊢
     T2] in
          [g ⊢ dia_result (tr_be* H1 H1') H2']
    | [g ⊢ cl_be* T'] ⇒
       (case [g ⊢ S'] of
       | [g ⊢ beta_be S''] ⇒
          (case [g ⊢ T'] of
          | [g ⊢ beta_be T''] ⇒
             let [g ⊢ confl_result_beta H1 H2] =
             beta_confluence [g ⊢ cl_b* S''] [g ⊢ cl_b* T''] in
             let [g ⊢ H1'] = be2_to_be1 [g ⊢ cl_b*e* (beta* H1)] in
             let [g ⊢ H2'] = be2_to_be1 [g ⊢ cl_b*e* (beta* H2)] in
             [g ⊢ dia_result H1' H2']
          | [g ⊢ eta_be T''] ⇒
             let [g ⊢ commute_result H1 H2] =
             d _ _ _ [g ⊢ cl_b* S''] [g ⊢ cl_e* T''] in
             let [g ⊢ H1'] = be2_to_be1 [g ⊢ cl_b*e* (eta* H1)] in
             let [g ⊢ H2'] = be2_to_be1 [g ⊢ cl_b*e* (beta* H2)] in
             [g ⊢ dia_result H1' H2']
          )
       | [g ⊢ eta_be S''] ⇒
          (case [g ⊢ T'] of
          | [g ⊢ beta_be T''] ⇒
             let [g ⊢ commute_result H1 H2] =
             d _ _ _ [g ⊢ cl_b* T''] [g ⊢ cl_e* S''] in
             let [g ⊢ H1'] = be2_to_be1 [g ⊢ cl_b*e* (eta* H1)] in
             let [g ⊢ H2'] = be2_to_be1 [g ⊢ cl_b*e* (beta* H2)] in
             [g ⊢ dia_result H2' H1']
```

```
            | [g ⊢ eta_be T''] ⇒
                let [g ⊢ confl_result H1 H2] =
                eta_confluence [g ⊢ cl_e* S''] [g ⊢ cl_e* T''] in
                let [g ⊢ H1'] = be2_to_be1 [g ⊢ cl_b*e* (eta* H1)] in
                let [g ⊢ H2'] = be2_to_be1 [g ⊢ cl_b*e* (eta* H2)] in
                [g ⊢ dia_result H1' H2']
            )
        )
    )
;

rec eta_beta_confluence : (g:ctx) [g ⊢ beta_eta* M M1] → [g ⊢ beta_eta* M M2]
                    → [g ⊢ dia_beta_eta M1 M2] =
/ total d (eta_beta_confluence _ _ _ _ d _ ) /
fn d ⇒ fn s ⇒ hindley_rosen (commutation_lemma be_square) _ _ _ d s
;
```

□

Bibliography

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. Poplmark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29:e19, 2019.

[2] Beniamino Accattoli. Proof Pearl: Abella Formalization of λ-Calculus Cube Property. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2012.

[3] Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004.

[4] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[5] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[6] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics.* North-Holland, 1985.

[7] Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[8] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. Autom. Reason.*, 49(2):141–159, 2012.

[9] Stefano Berardi. Girard normalization proof in LEGO. In *Proceedings of the First Workshop on Logical Frameworks*, pages 67–78, 1990.

[10] Ernesto Copello, Nora Szasz, and Álvaro Tasistro. Machine-checked Proof of the Church-Rosser Theorem for the Lambda Calculus Using the Barendregt Variable Convention in Constructive Type Theory. In Sandra Alves and Renata Wasserman, editors, *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017*, volume 338 of *Electronic Notes in Theoretical Computer Science*, pages 79–95. Elsevier, 2017.

[11] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[12] Jacob Errington, Junyoung Jang, and Brigitte Pientka. Harpoon: Mechanizing metatheory interactively - (system description). In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 636–648. Springer, 2021.

[13] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations - part 2 - A survey. *J. Autom. Reason.*, 55(4):307–372, 2015.

[14] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *Math. Struct. Comput. Sci.*, 28(9):1507–1540, 2018.

[15] Jonathan M. Ford and Ian A. Mason. Operational techniques in PVS - A preliminary evaluation. In Colin J. Fidge, editor, *Computing: The Australasian Theory Symposium, CATS 2001, Gold Coast, Australia, January 29-30, 2001*, volume 42 of *Electronic Notes in Theoretical Computer Science*, pages 124–142. Elsevier, 2001.

[16] Herman Geuvers and Benjamin Werner. On the church-rosser property for expressive type systems and its consequences for their metatheoretic study. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 320–329. IEEE Computer Society, 1994.

[17] Gérard Huet. Residual theory in lambda-calculus: A formal development. *J. Funct. Program.*, 4:371–394, 01 1994.

[18] Jonas Kaiser, Brigitte Pientka, and Gert Smolka. Relating system F and lambda2: A case study in coq, abella and beluga. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 21:1–21:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[19] Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, UK, 2000.

[20] James McKinna and Robert Pollack. Pure Type Systems Formalized. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 1993.

[21] Julian Nagele, Vincent van Oostrom, and Christian Sternagel. A Short Mechanized Proof of the Church-Rosser Theorem by the Z-property for the $\lambda\beta$-calculus in Nominal Isabelle. *CoRR*, abs/1609.03139, 2016.

[22] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3):23:1–23:49, 2008.

[23] Tobias Nipkow. More Church-Rosser Proofs (in Isabelle/HOL). In Michael A. McRobbie and John K. Slaney, editors, *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, volume 1104 of *Lecture Notes in Computer Science*, pages 733–747. Springer, 1996.

[24] Tobias Nipkow. More church-rosser proofs. *J. Autom. Reason.*, 26(1):51–66, 2001.

[25] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. Technical report, Carnegie Mellon University, 1992. Tech. Rep. CMU-CS-92-186.

[26] Frank Pfenning. Logical frameworks. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1063–1147. Elsevier and MIT Press, 2001.

[27] Brigitte Pientka and Andrew Cave. Inductive beluga: Programming proofs. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 2015.

[28] Brigitte Pientka and Jana Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2010.

[29] Barry K. Rosen. Tree-manipulating systems and church-rosser theorems. *J. ACM*, 20(1):160–187, 1973.

[30] J. Barkley Rosser. Extensions of some theorems of gödel and church. *J. Symb. Log.*, 1(3):87–91, 1936.

[31] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015.

[32] Peter Selinger. Lecture notes on the lambda calculus, 2013.

[33] Natarajan Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, 1988.

[34] Gert Smolka. *Confluence and Normalization in Reduction Systems Lecture Notes*. Saarland University, 2015.

[35] Masako Takahashi. Parallel reductions in lambda-calculus. *Inf. Comput.*, 118(1):120–127, 1995.

[36] René Vestergaard and James Brotherston. A Formalised First-Order Confluence Proof for the lambda-Calculus Using One-Sorted Variable Names. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 306–321. Springer, 2001.