Università degli Studi di Torino
Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Formalizations of the Church-Rosser Theorem in Agda

Relatore:
Prof. Ugo de' Liguoro
Correlatore:
Dott. Riccardo Treglia

Candidato:
Andrea Laretto

Anno Accademico 2019/2020

Abstract

In this thesis we present and explore the various concepts in computer-assisted formalizations of the Church-Rosser theorem for β-reduction in the untyped λ-calculus. Using the dependently-typed programming language and proof assistant Agda, we expose four different proof approaches for this theorem along with their complete formalization. The proofs we considered reutilize and exploit the potentiality of an existing infrastructure presented by Wadler et al. [WK19], where they implement λ-calculus and substitutions using de Bruijn indices and the σ-calculus by Abadi et al. [Aba+91]. The above-mentioned authors also present a confluence proof for β-reduction, employing the classic Tait and Martin-Löf method with parallel reduction. After presenting this proof, we firstly expose a small improvement of this development using Takahashi translation to prove the diamond lemma for parallel reduction, as described in Takahashi [Tak95]. We then present the main contribution of the thesis, which is a complete formalization of the confluence of β-reduction with the methods recently developed by Komori et al. [KMY14]. The refinements they present do not employ parallel reduction, and instead focus on an iterated Takahashi translation. This method allows them to obtain an even simpler proof of confluence which also sharpens and quantifies the previous results. Finally, we show that some of the theorems introduced in this last proof can be used to directly formalize another approach presented by Nagele et al. [NOS16] in Isabelle/HOL, where the confluence of β-reduction is derived through the use of the so-called Z-property due to Dehornoy et al. [DO08].

Contents

1 Introduction **1**
1.1 Formalization . . . 3
1.2 Related work . . . 5
1.3 Chapter overview . . . 6
1.4 File structure . . . 7

2 Introduction to Agda **8**
2.1 Comparison with other proof assistants . . . 8
2.2 Interactivity . . . 10
2.3 Constructive type theory . . . 10
2.4 Agda . . . 13
2.4.1 Datatypes . . . 13
2.4.2 Syntactic constructs . . . 14
2.4.3 Functions . . . 16
2.4.4 Equality . . . 18
2.4.5 Postulates . . . 22
2.4.6 Existence . . . 23
2.4.7 Modules . . . 24

3 De Bruijn indices and the σ-calculus **25**
3.1 Perspective . . . 25
3.2 De Bruijn indices . . . 27
3.2.1 Comparison with the original development . . . 30
3.3 Substitution . . . 31
3.3.1 Substitutions as functions . . . 31
3.4 σ-calculus . . . 34
3.4.1 σ-calculus equations . . . 36
3.4.2 Fundamental theorems . . . 37

4 The Church-Rosser Theorem 40
4.1 β-reduction 40
4.2 Substitutivity of β^*-reduction 43
4.2.1 Pointwise β^*-reduction 44
4.2.2 β^*-reduction and renamings 46
4.3 Church-Rosser Theorem 48

5 The Tait/Martin-Löf proof and parallel reduction 50
5.1 Main idea 50
5.2 Proof overview 50
5.3 Parallel reduction 52
5.4 Relations between parallel reduction and β-reduction 55
5.5 Diamond lemma for parallel reduction 57
5.6 Strip lemma 58
5.7 Confluence of parallel reduction 60
5.8 Confluence of β-reduction 61

6 Takahashi translation 62
6.1 Definition 62
6.1.1 Pattern overloading in Agda 63
6.2 Revisiting the confluence of parallel reduction 64
6.2.1 Diamond lemma for parallel reduction 65
6.3 Comparison with the previous proof 65

7 The Komori-Matsuda-Yamakawa proof 66
7.1 Proof overview 66
7.2 Main concepts 68
7.3 Fundamental theorems for confluence 69
7.4 Confluence of β-reduction 71
7.5 Central theorems 73
7.5.1 Lemma 3.3 73
7.5.2 Lemma 3.5 74
7.5.3 Proving Theorem 3.8 74
7.5.4 Proving Lemma 3.5 75
7.6 Lemma 3.4 77
7.7 Takahashi translation for substitutions 79
7.7.1 Generalized Lemma 3.4 79
7.7.2 Lemma 3.4 for applications 82

7.7.3 Takahashi translation and renamings 83
7.7.4 Special case of 0-indexed substitution 84
7.8 Proof remarks 85

8 The Z-property proof 86
8.1 Generic reflexive transitive closure 86
8.2 Semi-confluence 87
8.3 Z-property 89
8.4 Z-property for β-reduction 92
8.5 Comparison with the Komori-Matsuda-Yamakawa proof 93
8.6 Proof overview 93

9 Conclusion 95
9.1 Theoretic perspective 95
9.2 Implementation 96
9.3 Future work 97

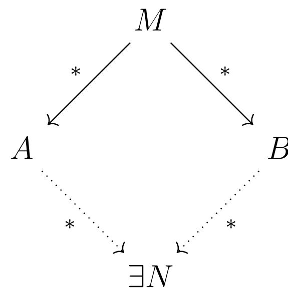
Chapter 1

Introduction

λ-calculus is considered to be one of the greatest formal systems in theoretical computer science. Its simplicity, usefulness, and importance ranges from foundational aspects of mathematics to the most concrete problems in computer science and programming language theory. Since its introduction in the 1930s by the American mathematician Alonzo Church, this elegant formalism still manages to influence and guide the design of modern programming languages, while maintaining to this day its relevance as the topic of novel mathematical results. The Church-Rosser theorem is arguably the most fundamental theorem regarding λ-calculus and its central relation, β-reduction. We define it as follows:

Theorem (Church-Rosser theorem). Given a term M such that $M \to_\beta^* A$ and $M \to_\beta^* B$, there exists a term N such that $A \to_\beta^* N$ and $B \to_\beta^* N$.

This theorem states the *confluence* of β-reduction, and expresses the fact that given any two different reduction orders there always exists a common term reuniting them. We can also denote this property with a commutative diagram:



Since the first proof in 1936 by Church and J. Barkley Rosser [CR36], many other approaches for this theorem have been established, and one of the first major improvements was presented by William Tait and Per Martin-Löf around 1972. Their proof can be found in [HS86] and [Bar85], and it is now considered to be one of the most well-known proofs of this fundamental property. The idea is to define the concept of a parallel reduction, which can contract multiple redexes at the same time and for which confluence is easier to prove. This development has since been improved by Masako Takahashi in 1989 [Tak95], with the introduction of Takahashi translation to explicitly provide a confluent term for parallel reduction. The term provided is independent on the form of the reductions, and this results in an even more direct confluence proof. A more detailed account of the history of λ-calculus and the Church-Rosser theorem can be found in [CH09].

The Komori-Matsuda-Yamakawa proof

Some of the most recent results include the methods presented in 2014 by Yuichi Komori, Naosuke Matsuda, and Fumika Yamakawa in [KMY14]. Their work solely focuses on iterated Takahashi translation, where it is quantitatively related to β-reduction and $\beta\eta$-reduction. As stated by the authors, this can be useful to prove confluence of those systems where parallel reduction is harder to treat. Their approach also allows them to obtain even more precise confluence results, which we summarize in the following two theorems. In terms of notation, we use $\to_\beta^n$ to indicate n steps of β-reduction and M^{*n} for Takahashi translation iteratively applied n times:

Theorem 3.8. Given two λ-terms M and N:

$$M \to_\beta^n N \Longrightarrow N \to_\beta^* M^{*n}$$

Theorem 3.9 (Confluence). Given any two β-reductions $M \to_\beta^n A$ and $M \to_\beta^m B$, the term $M^{*\max\{n,m\}}$ is a confluent term for the two reductions. That is, we have that $A \to_\beta^* M^{*\max\{n,m\}}$ and $B \to_\beta^* M^{*\max\{n,m\}}$.

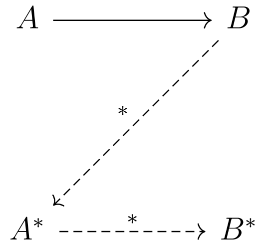
These results even more precisely quantify the advancements presented by Takahashi, and they can be obtained without the use of parallel reduction. The two final reductions still do not rely on the specific form of the reduction steps applied, and jointly specify the confluent term.

The Z-property proof

Another important approach which directly relates β-reduction to confluence is that presented in 2016 by Patrick Dehornoy and Vincent van Oostrom [DO08], where they define the so-called Z-property. We can state its definition as follows:

Definition (Z-property). A relation $\rightarrow$ on T is said to have the *Z-property* if there exists a map $* : T \rightarrow T$ such that $A \rightarrow B$ implies $B \rightarrow^* A^*$ and $A^* \rightarrow^* B^*$, for any A and B.

The name of this property comes from the fact that its statement, expressed in a schematic form, resembles the letter Z:



It can be shown that the Z-property directly implies semi-confluence, and therefore confluence. The Church-Rosser theorem is then obtained by showing that Takahashi translation is a map that respects the Z-property for β^*-reduction.
In this thesis we present a complete formalization of the main advancements introduced by Komori et al. [KMY14], and then relate the two proofs by also showing how some of these results can be used to directly satisfy the Z-property.

1.1 Formalization

With the birth of modern computers and computer-assisted mathematics in the 1970s, the importance of the Church-Rosser theorem has been further emphasized by the surprising number of machine-checked proofs aimed at its formalization. Many theorem provers have been employed, and we here quote some of the most important formal developments that can be found in the literature:

- Boyer/Moore theorem prover (Shankar, 1988 [Sha88])
- Elf (Pfenning, 1992 [Pfe92])
- LEGO (McKinna and Pollack, 1999 [MP99])

- Coq (Huet, 1994 [Hue94], Schäfer, Tebbi, Smolka, 2015 [STS15])
- Isabelle/ZF (Rasmussen, 1995 [Ras95])
- Isabelle/HOL (Nipkow, 1996 [Nip96], Vestergaard and Brotherston, 2001 [VB01])
- HOL (Homeier, 2001 [Hom01])
- Nominal Isabelle (Nagele, van Oostrom, Sternagel, 2016 [NOS16])
- Abella (Accattoli, 2012 [Acc12])
- Agda (Copello, Szasz, Tasistro, 2017 [CST17], Wadler, Kokke, et al. 2019 [WK19])

The formalizations described in this thesis have been constructed using the dependently-typed programming language and interactive theorem prover Agda, initially developed by Ulf Norell and Catarina Coquand in 1999 [BDN09].
In order to formally present theorems about the λ-calculus, an adequate representation method for λ-terms and the concept of substitution is first required. In mathematical proofs this is usually done by referencing the Barendregt Variable Convention [Bar85], which can be used to solve the fundamental problems regarding capturing of free variables, substitution and α-equivalence. In a formal setting, however, these informal conventions must also be adequately represented, and in such a way that formal proofs can follow the approaches indicated by the pen-and-paper ones. Accurately implementing variable binders and non-capturing substitutions turns out to be a particularly tricky and well-known problem in the literature. Many representation methods have been presented, starting with the introduction by Nicolaas G. de Bruijn in 1972 of the concept of de Bruijn indices for his automatic prover AUTOMATH [Bru72]. Other important formalization techniques are the Nominal approach implemented in Isabelle/HOL [Urb08], the Locally Nameless method [Ayd+08], and the Higher-Order Abstract Syntax representation [How10]. An entire project and challenge focusing on the formal correctness of programming language semantics and type systems, where these methods for the treatment of binders can be compared, is the POPLMark challenge. [Ayd+05] This also goes to show how relevant these issues can be in the formalization of more general systems related to λ-calculus. In this thesis we have decided, after initially exploring the other possibilities, to reutilize the Agda implementation based on de Bruijn indices provided by Wadler et al. in [WK19], which constitutes a solid foundation to experiment with λ-calculus and its variations. This also presents us with the possibility of testing an already existing infrastructure and verifying how efficiently it can be interfaced with further constructions.

1.2 Related work

To our knowledge, no formal proof of the results by Komori et al. [KMY14] has yet been presented. The Z-property proof introduced by Dehornoy et al. [DO08], on the other hand, has been formalized in Isabelle/HOL by Julian Nagele, Vincent van Oostrom, and Christian Sternagel in [NOS16] using Nominal Isabelle.
Wadler et al. [WK19] also describe their own development in Agda of the Church-Rosser theorem, by formalizing parallel reduction and the Tait/Martin-Löf proof with their infrastructure based on de Bruijn indices. We present these results in Chapter 5 in order to analyze and then compare them with our own proof.
Another important approach available in Agda is the one established by Copello et al. [CST17]. The authors managed to construct an induction principle based on α-equivalence that directly expresses the Barendregt Variable Convention, where freshness of variables can always be assumed. This method effectively allows them to mimic the classic informal proofs, and enables the use of named variables instead of de Bruijn indices. After proving some lemmas regarding substitutions, they show confluence of β-reduction by following the Takahashi approach with parallel reductions. This development establishes a similar technique as that employed by Nominal Isabelle, with this latter system also providing a high degree of automation both in proof searching and in the automatic application of these tactics.
The library Autosubst presented by Steven Schäfer, Tobias Tebbi, and Gert Smolka [STS15] implements parallel substitutions and λ-terms in Coq, and it provides automation tactics to easily solve those cases where substitutions are directly involved. The authors themselves then use the library to also prove the Church-Rosser theorem as a case study. This infrastructure does not use a nominal representation, and instead employs de Bruijn indices by referencing the σ-calculus and the concept of parallel substitutions established by Abadi et al. [Aba+91]. This paper is in turn referenced by Wadler et al. [WK19] for the implementation of their fundamental work, and its main concepts are also introduced and employed in this thesis.

1.3 Chapter overview

In Chapter 2 we briefly introduce the dependently-typed programming language Agda, by presenting the main elements of its syntax and constructs. While comparing it to other proof assistants, we also provide an overview of the theory behind its use as an interactive theorem prover.

In Chapter 3 we show the usefulness of de Bruijn indices as a formalization tool for λ-calculus, and describe the main infrastructure provided by [WK19] in Agda for this representation method. This foundation constitutes the starting point on which we further develop the higher-level theorems about β-reduction and Takahashi translation.

In Chapter 4 we formally present β-reduction, confluence, and the Church-Rosser theorem. We prove here one of the first important lemmas about β-reduction later used in proofs, the substitutivity of β^*-reduction.

In Chapter 5 we introduce the proof approach developed by Tait-Martin Löf based on parallel reduction. We then include and explain the Agda formalization constructed by [WK19] in order to present its main ideas and later compare this proof with our own.

In Chapter 6 we define the concept of Takahashi translation exposed in [Tak95], and show its usefulness in improving the Tait-Martin Löf proof described in the previous chapter. Takahashi translation represents the central concept on which Chapter 7 and Chapter 8 later derive their confluence proofs without the use of parallel reduction.

In Chapter 7 we present the complete formalization of the confluence proof for β-reduction using the methods described by [KMY14]. This development constitutes the main result of this thesis, and the theorems here proven subsequently provide the foundational properties on which Chapter 8 further elaborates.

In Chapter 8 we formalize the concept of Z-property in Agda as presented by [DO08] and formalized in Isabelle/HOL by [NOS16]. After proving that the Z-property implies confluence for any given generic reduction, we show how some of the fundamental results previously formalized in Chapter 7 can be used to immediately obtain the Z-property for β-reduction, and therefore conclude an even more direct proof of confluence.

1.4 File structure

The files presenting the development of this thesis are publicly available at `https://github.com/iwilare/church-rosser`, and have been organized as follows:

Beta.agda	Contains the basic definitions for β-reduction and β^*-reduction.
BetaSubstitutivity.agda	Contains the substitutivity property for β^*-reduction.
ConfluenceParallel.agda	Contains the Tait/Martin Löf proof formalized by Wadler et al. [WK19].
ConfluenceParallelTakahashi.agda	Contains the improvement provided by Takahashi [Tak95] to the Tait/Martin Löf proof.
ConfluenceTakahashi.agda	Contains the full formalization of the proof presented by Komori et al. [KMY14] for β-reduction.
ConfluenceZ.agda	Contains the confluence proof based on the Z-property by Dehornoy et al. [DO08], formalized in Agda by reusing the properties of the proof by Komori et al. previously proven.
DeBruijn.agda	Contains the definition of λ-terms with de Bruijn indices by Wadler et al.
Parallel.agda	Contains the definition and theorems for parallel reduction by Wadler et al.
Substitution.agda	Contains the fundamental properties for σ-calculus by Wadler et al.
Takahashi.agda	Contains the definition of Takahashi translation.
Z.agda	Contains the statement in Agda of the Z-property presented by Dehornoy et al. and the relations with confluence and semi-confluence.

Chapter 2

Introduction to Agda

In this chapter we present a very general introduction to the language and the way it helps in formalizing mathematical proofs using constructive type theory. For a more detailed introduction to its syntax and features, see [WK19].
Agda is a dependently-typed functional programming language and interactive theorem prover originally developed at Chalmers University by Ulf Norell et al. and first described in his Ph.D. thesis [Nor07]. It can be used as a full-fledged strict programming language with a syntax similar to Haskell, and it has backends that allow compiling programs to JavaScript and GHC Haskell.
What makes Agda stand out from other programming languages is its powerful type system, which is based on dependent types and takes inspiration from Martin-Löf's intuitionistic type theory. In a dependent type system, types can be parameterized according to values (such as concrete integers or strings) and not just to other types, as it would instead happen in the generics system implemented by Java or in the parametric polymorphism on which Haskell is based. As it will be explained in Section 2.3, Agda's expressive type system is precisely what enables its use as a proof assistant, where propositions are expressed as types and proofs as well-typed programs.

2.1 Comparison with other proof assistants

Agda is a relatively new addition to the landscape of interactive theorem provers, having been initially implemented in 1999 by Catarina Coquand and Ulf Norell and then rewritten from scratch in 2007. [BDN09] It sharply contrasts with other proof assistants, such as Isabelle/HOL and Coq, which approach the pro-

cess of theorem proving in profoundly different ways. Isabelle/HOL uses the HOL (Higher Order Logic) language and library to construct proofs. In a similar fashion with its predecessor system LCF, it defines in the ML programming language an abstract data type representing theorems and formulas, and a small kernel of trusted functions that operate on them. These functions correspond to inference rules in higher-order logic and constitute the only possible way to prove new theorems, implementing a sort of symbolic manipulation framework. Therefore, as long as the core functions are correctly implemented, the resulting theorems will also be sound. Isabelle/HOL also comes with both a procedural and a declarative style through which one can write proofs, with the latter being the preferred method. In the procedural style, one only specifies the names of the tactics to be applied, whereas the declarative style uses the proof language Isar to describe a more natural and human-readable explanation of the goals and the results proven. A more detailed and thorough description of the Isabelle/HOL system can be found in [Pau94].

Coq, on the other hand, exploits and uses just as Agda the intuitionistic relationship between proofs and types, and its type system is based on Coquand's Calculus of (Co)Inductive Constructions. [Coq] It also provides like Isabelle/HOL an extensive system to automate theorem proving and try to automatically find proofs. However, Coq uses alongside the programming language Gallina a separate tactic language for writing proof scripts that the programmer has to separately learn and integrate.[1] As it is however noted in [KS15], this approach seems at odds with the spirit of type theory, where a single language is used for both proof and computation.

Even though the usefulness of automatic theorem proving has been one of the main drivers behind the use of Coq and Isabelle/HOL in the formalization of modern mathematics, Agda does not use by construction any tactic language nor intrinsically built-in automation and instead solely relies on user input to provide proofs and well-typed terms. However, some interactive automation mechanisms do exist, such as the one described in [KS15], and have been sometimes utilized to close some proofs in this paper.

[1] There are other more technical differences regarding their respective type structures. For example, Coq has the impredicative universe `Prop`, distinct from `Set`, while Agda only has the latter. For a more detailed discussion, consult [BDN09].

2.2 Interactivity

It is possible to use either the Emacs or the Atom text editors as interactive frontends to exploit the full user interaction and assistance of Agda, enabling functionalities such as trying to automatically find a proof, displaying the current goal, evaluating expressions and their types, etc. It is also an established practice in Agda to frequently use Unicode characters in the names of definitions and theorems, as can be also seen in [WK19], [CST17], and in the Agda standard library itself. Emacs also supports this, by enabling a sort of auto-completion method to type Unicode characters with a standard keyboard. This practice allows the user to better express the concepts used and develop a sort of domain-specific language for the mathematical context that one is proving theorems in. Even though it can sometimes come at the cost of making proofs harder to read for newcomers, we have adopted this style in order to more closely follow the established notation used in papers and proofs related to the domain here treated.

2.3 Constructive type theory

The use of Agda as a proof assistant comes from the Curry-Howard isomorphism, a correspondence first noticed by the American mathematician Haskell Curry and logician William Alvin Howard which delineates the direct relationship between computer programs, the types of such programs, and formal mathematical proofs. Under this correspondence theorems are adequately expressed as types, and proving a certain mathematical statement equates with being able to directly construct a well-typed term (called witness, or proof) of such type. A more complete treatment of this important relationship can be found in [GTL89].
The most well-known and extensively studied example of a sufficiently powerful type theory is Martin-Löf type theory, which closely follows the principles of mathematical constructivism. The Curry-Howard correspondence also partially applies to type systems that computer scientists more commonly work with, even in mainstream languages. For example:

- *Conjunction* corresponds to the more familiar notion of product type, usually known as a tuple: elements of the type $A \wedge B$, or equivalently (A, B), can be constructed by providing a term/proof of type A, and a term/proof of type B. The name "product type" comes from the fact that given the finite types A and B with n and m inhabitants respectively, the type (A, B) has $m * n$ inhabitants in total.

Logic	*Type Theory*
Proposition	Type
Proof of a given proposition	Term of a given type
Structural induction	Recursion with pattern matching
Implication	Function type
Conjunction	Product type (`structs`, tuples)
Disjunction	Sum type (`unions`)
Universal quantification	Dependent product type
Existential quantification	Dependent sum type

Table 2.1: *Main concepts in the Curry-Howard isomorphism between type theory and logic*

- *Disjunction* corresponds to a sum type, more commonly known as a (tagged) union in C, or the `Either` datatype in Haskell: elements of the type $A \vee B$ can be constructed either with an element of type A, appropriately "injected" in the union type, or with an element of type B similarly extended. This injecting function corresponds to the familiar constructors `Left :: a -> Either a b` and `Right :: b -> Either a b` in Haskell. The name "sum type" comes from the same reasoning shown for the product type.

- *Implication* is interpreted as the function type: an element of the function type $A \to B$ is an appropriate programming structure (typically a concrete function) that, given a term/proof of type A, gives in return a term/proof of type B. The act of applying a function to a certain argument therefore naturally corresponds to what is called in logic the modus ponens rule.

- *Negation* is represented with the intuitionistic interpretation of $\neg A$ being defined as the function type $A \to$ False, where False is a type that has no constructors.

If the type system in consideration is rich enough to have dependent types, where types can be defined and depend on values (i.e.: those being quantified), we can also express first-order logic constructs:

- *Universal quantification* becomes a more powerful version of the well-known concept of polymorphism, with the fundamental difference that, contrary to the concept of generics found for example in Java where the generic parameters can only be other types, we can also have normal values and expressions (such

as strings or numbers) as parameters. This mechanism can be also interpreted as "abstracting" a part of a certain proposition and quantifying it as a symbol. Notice how closely this follows the λ-calculus notion of λ-abstraction, with function application as the elimination of such quantification. This construct is also known as a dependent product type, usually denoted with Π.

- *Existential quantification* corresponds to being able to provide a pair of two elements: a concrete object, or "witness", and a proof (another term) that says that the desired proposition holds for the provided witness of the pair. This construct is also known as a dependent sum type, usually denoted with Σ. Note that the concept of pair reminds again to a product type: indeed, when the type of the second element of the pair does not depend on the first, the resulting type is again the familiar notion of a tuple. As it is described in [WK19, Quantifiers], the name "dependent sum type" comes from the fact that, given a finite type A with values $x_1, \ldots, x_n$, and a dependent function B with each of the types $B\ x_1, \ldots, B\ x_n$ having $m_1\ , \ldots, m_n$ distinct members, then the type `Σ[ x ∈ A ] B x` has $m_1 + \cdots + m_n$ inhabitants. A similar reasoning can be applied to the dependent product type previously described.

The construction explained in this last case is the main point of departure from classical mathematics and classical logic because it rules out impredicative proofs (i.e.: proofs that derive existence of a certain mathematical object from assuming its non-existence and deriving a contradiction) since in such a scenario we do not have to concretely provide the mathematical object for which the proposition holds. Furthermore, the principle of the excluded middle of classical logic does not hold in this context, since in order to have a proof of a proposition or its negation one has to actually construct it. For these reasons, Martin-Löf type theory is also called constructive type theory, or intuitionistic type theory, following Brouwer's notion of intuitionistic logic where the law of the excluded middle, along with double negation elimination, purposefully do not hold. [Mar84] The type system used by Agda is specifically based on Zhaohui Luo's unified theory of dependent types (UTT) [Luo94], a type theory close to Martin-Löf's.

2.4 Agda

In this section we will concretely introduce Agda and its syntax, along with the fundamental constructs necessary to understand the proofs presented in this thesis.

2.4.1 Datatypes

The main system that makes it possible to formulate new definitions and concepts in Agda is the concept of inductive datatypes. The basic example of this mechanism is the inductive definition of natural numbers; we create here a new term of type ℕ called `zero`, and a constructor `suc` which, given a natural, returns another natural:

```
-- Inductive definition of naturals
-- Comments in Agda begin with a `--`
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Each inductive definition introduces zero or more concrete terms, called constructors, which are simply optionally dependent functions (or values, which are simply nullary functions) that construct the type being defined. As it happens with the majority of functional programming languages, function application has always the highest precedence, and is expressed by simply specifying the function and then separating arguments with spaces, such as `suc zero`, or `f x y`, in a similar way as with Haskell. In this last example, we can also notice the use of `Set` as the builtin type of other types.[2]

Being Agda a dependently-typed programming language, it is possible to construct datatypes where the type being declared is dependent on values or more complex terms. The classic example in dependent programming is the definition of vectors whose size is statically known at compile time:

[2]In order to avoid Russell's paradox, Agda uses a universe hierarchy where `Set` is actually implicitly indexed by an integer. In this system, $\texttt{Set}_i : \texttt{Set}_{i+1}$, and Set is just a shorthand for $\texttt{Set}_0$.

```
infixr 5 _::_

data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n : ℕ} (x : A) (xs : Vec A n) → Vec A (suc n)
```

This example perfectly shows the main capabilities of Agda's powerful dependent type system, where the type `Vec` is itself in some sense a "higher-order function" that takes a type (also belonging to the type `Set`), a concrete number (with type ℕ), and returns a new type.
We can also employ implicit arguments, denoted by "{" and "}", to avoid having to specify those arguments considered uninteresting every time we use a constructor or a function, in this case `_::_` and the implicit argument `{n : ℕ}`. Using implicit arguments allows us to directly write expressions such as `0 :: 1 :: 2 :: []` without having to specify the size `n` each time. Each argument, however, still fully represents a universally quantified parameter. This useful mechanism forces Agda to try inferring the value of omitted parameters by trying to solve the implicit type constraints (provided by the context in which the expression is placed and by other parameters) necessary to make the program type-check. It is sometimes necessary to make these parameters explicit in certain function applications where Agda cannot infer them by itself, and this is possible by also enclosing the function arguments within curly braces, optionally specifying the name of the implicit parameter.
The use of dependent functions becomes clear from the fact that parameters have their own names in the type definition, so that they can be later used in the type. In this case, the names `x` and `xs` are not strictly necessary, and are merely descriptive; the implicit parameter `n`, however, requires a name so that it can be successively referred to in the expressions `Vec A n` and `Vec A (suc n)`.

2.4.2 Syntactic constructs

As shown in the previous example, Agda also allows the user to specify a so-called *mixfix* definition that makes it possible to easily and flexibly introduce new syntactic constructs. This is specified by the use of underscores to indicate where the extra arguments are syntactically positioned. This mechanism allows the user to define constructs that can be either unary, binary, or even more complex syntactic constructs. In the previous definition, the `_::_` constructor (also called

cons) is defined with `infixr` to be a right-associative binary operator,[3] with a fine-tuned precedence number.

Using the propositions-as-types method, we can define types that encode simple propositional logic. The following example denotes the concept of disjunction, where $A \vee B$ can be proven by either providing a proof of A through the constructor `inj₁`, or by providing a proof of B with `inj₂`:

```
infixr 1 _⊎_

data _⊎_ (A : Set) (B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

The constructors act as introduction mechanisms of the proposition, and pattern matching is used to destruct them into their premises, as it will explained in Section 2.4.3. Through the use of dependent types we can also express when a given relation holds:

```
data _≤_ : ℕ → ℕ → Set where

  z≤n : ∀ {n}
        ---------
    → zero ≤ n

  s≤s : ∀ {m n}
    → m ≤ n
        --------------
    → suc m ≤ suc n
```

In this definition, `z≤n` is to be interpreted as a constructor (which just acts as another function) that gets an implicit parameter `n` and returns a new term of type ≤, as specified. The second constructor is a constructor with two implicit parameters, which then accepts an explicit parameter of type `m ≤ n`. The type checking for each parameter, explicit or not, is done according to the values of the preceding ones. For example, `m` and `n` in this case determine the type of the explicit parameter `m ≤ n`, in such a way that the expression also returns the

[3] The associativity of new operators can be expressed with `infixr`, `infixl`, or `infix` in the case of operators for which no meaningful associativity can be defined.

appropriate type. Note that it is also possible to omit the type of parameters if Agda can infer them from context. Here m and n are automatically inferred to be instances of ℕ from their use in the $\leq$ constructor. The type signature of functions, contrary to other programming languages that employ type inference, cannot however be omitted.
In this case, constructors are implemented as dependent functions that take as arguments propositions (or other values) and give as result a proof that the relation holds. New concepts and propositions are nothing but yet another type, with no specific distinction from the datatype structures commonly found in programming.
Note in the preceding example that two dashes -- are simply used to start a comment. The line just serves as graphic aid to suggest that definitions can be interpreted as inference rules, but it has no intrinsic meaning.

2.4.3 Functions

Agda is a functional programming language, and the main method to operate on the previously defined inductive datatypes is precisely through the use of functions. Functions can be defined through the use of pattern matching, which allows the user to consider and decompose each possible case of the parameters into their constructors, and then state the function value according to each case.

```
infixl 6 _+_

_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

In order to preserve the consistency of the underlying type system, all Agda function definitions must employ well-founded recursion, i.e.: always terminate. Agda tries to check this by ensuring that all recursive calls have, as arguments, terms that are structurally-smaller than the ones specified in the definition.[4] For example, if we were in a given function definition that pattern-matches on the term (a, b), then a would be a valid argument on which to recursively call the function since the argument is *smaller* than the original one. This mechanism,

[4]It is possible with the appropriate language pragma {-# TERMINATING #-} to override the termination checks for a given function. It is obviously not recommended since Agda cannot ensure that the definitions will not lead to logically unsound constructs.

when applied to the previously defined inductive datatype ℕ, perfectly mirrors the classic principle of induction on numbers: a function that operates on a natural number can pattern match with `zero`, the base case, or with the structure `(suc n)`. In this latter case, we can recursively call the function on the smaller argument `n`, which acts as the inductive hypothesis. The use of recursion and pattern matching in the general context of inductive datatypes corresponds to the familiar concept of *structural induction*, and Agda exhaustively checks that every single possible case has been covered. As it will be clear when treating Takahashi translation in Chapter 6, this case-checking mechanism can occasionally give rise to inconveniences.

In the definition of the addition function `_+_`, the recursive call is performed on a syntactically smaller argument so the recursion is guaranteed to terminate. Agda also requires that functions must always be declared before their use in other constructs.[5] Given these basic definitions we can already start proving some simple theorems, such as the following:

```
+-ext : ∀ (x y) → x ≤ x + y
+-ext zero    y = z≤n
+-ext (suc x) y = s≤s (+-ext x y)
```

It is also possible to use the `with` syntax to perform case analysis on an intermediate argument. This powerful mechanism also has the effect of replacing the specified expression inside of the goal in each case.

```
≤-total : ∀ (x y : ℕ) → x ≤ y ⊎ y ≤ x
≤-total zero    _       = inj₁ z≤n
≤-total (suc _) zero    = inj₂ z≤n
≤-total (suc m) (suc n) with ≤-total m n
... | inj₁ m≤n = inj₁ (s≤s m≤n)
... | inj₂ n≤m = inj₂ (s≤s n≤m)
```

Note in the last example the use of the special pattern _, which can be employed for those values that are not referenced later and that can be simply ignored.

It is important to remark that the order considered when applying pattern matching is always first to last case. In interactive mode, Agda also helps the user by signaling when some cases are ”overshadowed” by previous ones.

[5] There nevertheless exists a mechanism to support mutually recursive (but still terminating) definitions.

As with any other functional programming language, Agda allows the use of anonymous functions through the lambda operator. This also can be paired with anonymous case splitting, used here to introduce other common syntactic constructs:

```
case_of_ : ∀ {A B} → A → (A → B) → B
case x of f = f x

filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) =
  case p x of
    λ { true  → x :: filter p xs
      ; false →      filter p xs
      }
```

We can also see here a practical use of the mixfix syntax to introduce complex syntactic constructs, which can be defined for datatypes as well as functions. Another example directly taken from the Agda standard library is the following function, showing the flexibility of the language in creating definitions that can be easier to work with. This example can be interpreted as a construct to decompose (non-dependent) disjunctions:

```
[_,_] : ∀ {A B C} → (A → C) → (B → C) → (A ⊎ B → C)
[ f , g ] (inj₁ x) = f x
[ f , g ] (inj₂ y) = g y
```

2.4.4 Equality

Contrary to many other theorem provers, propositional equality in Agda is not an intrinsically primitive concept, and it can be perfectly defined within the language itself:

```
infix _≡_ 4

data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Following the definition, the only way to construct a proof that two terms of a given type are equal is through the use of the constructor `refl`.

Note how some simple theorems about equality can be easily proven by pattern matching on the sole constructor `refl`: since `refl` requires both arguments to be syntactically the same, Agda can match `y` with the same expression as `x` and effectively simplify the goal.

```
sym : ∀ {A : Set} {x y : A}
  → x ≡ y
    ------
  → y ≡ x
sym refl = refl

trans : ∀ {A : Set} {x y z : A}
  → x ≡ y
  → y ≡ z
    ------
  → y ≡ z
trans refl refl = refl

cong : ∀ {A B : Set} (f : A → B) {x y : A}
  → x ≡ y
    ----------
  → f x ≡ f y
cong f refl = refl

cong₂ : ∀ {A B C : Set} (f : A → B → C) {u x : A} {v y : B}
  → u ≡ x
  → v ≡ y
    --------------
  → f u v ≡ f x y
cong₂ f refl refl = refl

+-suc : ∀ (m n) → m + suc n ≡ suc (m + n)
+-suc zero    n = refl
+-suc (suc m) n = cong suc (+-suc m n)
```

This elegant equality mechanism also allows us to define an entire system of equational reasoning within Agda itself:

```
infix  1 begin_
infixr 2 _≡⟨⟩_ _≡⟨_⟩_
infix  3 _∎

begin_ : ∀ {A} {x y : A}
  → x ≡ y
    ------
  → x ≡ y
begin x≡y = x≡y

_≡⟨⟩_ : ∀ {A} (x : A) {y : A}
  → x ≡ y
    ------
  → x ≡ y
x ≡⟨⟩ x≡y = x≡y

_≡⟨_⟩_ : ∀ {A} (x : A) {y z : A}
  → x ≡ y
  → y ≡ z
    ------
  → x ≡ z
x ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z

_∎ : ∀ {A} (x : A)
    ------
  → x ≡ x
x ∎ = refl
```

The difference between _≡⟨_⟩_ and _≡⟨⟩_ is that the former requires an "external justification" for the rewriting, as represented by the transitive use of equality. In both cases, Agda always tries to apply the definitions of the functions involved so that a common term can be reached.

Without making all the definitions explicit, we can show some practical examples by proving equalities about naturals:

```
+-identityʳ : ∀ (m : ℕ) → m + zero ≡ m
+-identityʳ zero =
  begin
    zero + zero
  ≡⟨⟩
    zero
  ∎
+-identityʳ (suc m) =
  begin
    suc m + zero
  ≡⟨⟩
    suc (m + zero)
  ≡⟨ cong suc (+-identityʳ m) ⟩
    suc m
  ∎

+-comm : ∀ (m n : ℕ) → m + n ≡ n + m
+-comm m zero =
  begin
    m + zero
  ≡⟨ +-identityʳ m ⟩
    m
  ≡⟨⟩
    zero + m
  ∎
+-comm m (suc n) =
  begin
    m + suc n
  ≡⟨ +-suc m n ⟩
    suc (m + n)
  ≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m)
  ≡⟨⟩
    suc n + m
  ∎
```

In the last examples, we need to use the congruence property for equality in order to appropriately "apply" the inductive hypothesis to the subterm inside the `suc` constructor. Sometimes the term we want to rewrite can be a deep subexpression, and that would require us to apply congruence multiple times. As a shorthand for this operation, Agda provides the syntactic sugar `rewrite`, which rewrites the goal with a given equation.[6] For example, we can easily prove the following theorem without having to "narrow down" the point where we want to apply the rewriting rule:

```
suc-comm-eq : ∀ (m n k : ℕ)
  → suc (m + n) ≡ k
    ----------------
  → suc (n + m) ≡ k
suc-comm-eq m n k eq rewrite +-comm m n = eq
```

Note again that the first four arguments to the function (theorem) `suc-comm-eq` are, in order, `m`, `n`, `k`, and the evidence `eq` expressing the hypothesis that `suc (m + n) ≡ k` holds.

2.4.5 Postulates

It is possible to temporarily postulate any principle in Agda using the `postulate` syntax. This can be both used to temporarily assume theorems so that they can be proved them later, or to add new axioms and principles altogether. An extremely useful principle unprovable in Agda that therefore needs to be provided as a postulate is the *extensionality principle*, which states that if two functions are pointwise equal for every possible argument, then they are effectively equal.[7] Wadler et al. extensively use this principle in [WK19, Substitution] to prove the results shown in Section 3.4.1, and we will also employ it in Section 7.7.3 to prove the equivalence of two substitutions. This property can also be found in its more powerful dependent version in the `Axiom.Extensionality.Propositional` module, and it can be expressed as follows:

[6]Internally, this uses the `with` notation previously described.

[7]Note that the converse of the extensionality principle, the η-expansion rule, does not require a specific postulate in Agda and it can be trivially shown.

```
postulate
  extensionality : ∀ {A B : Set} {f g : A → B}
    → (∀ (x : A) → f x ≡ g x)
      -------------------------
    → f ≡ g
```

2.4.6 Existence

Existential quantification can be directly defined in Agda, mirroring the type-theoretical notion of a pair:

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```

This definition uses the underlying principle of dependent pairs, where the first element is a concrete term and the second element is a proof for a proposition applied to that term. Another equivalent way to express this is through the use of records:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst
```

There are other convenient syntactic constructs defined in `Data.Product` used to denote existence. These definitions are slightly easier to use since they do not require the explicit use of functions to quantify propositions. We will employ them to express the notion of confluence in Chapter 5 and Chapter 7, and to formalize the Z-property presented in Chapter 8. We can briefly compare them with the following example:

```
Σ-zero : Σ ℕ (λ x → ∀ {y} → x ≤ y)
Σ-zero = zero , z≤n

Σ-syntax-zero : Σ[ x ∈ ℕ ] ∀ {y} → x ≤ y
Σ-syntax-zero = zero , z≤n

∃-syntax-zero : ∃[ x ] ∀ {y} → x ≤ y
∃-syntax-zero = zero , z≤n
```

2.4.7 Modules

Agda provides a very well-organized module system that allows the user to import, rename, and hide any desired function from the available modules. The module structure directly follows the filesystem organization, with each namespace corresponding to a folder and each module mirroring its filename. All the concepts presented with the previous definitions can be found in the Agda standard library, and can be imported as follows:

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; cong₂; sym)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; _≡⟨_⟩_; _∎)
open import Data.Nat using (ℕ; zero; suc; _+_; _≤_; z≤n; s≤s)
open import Data.Product using (Σ; Σ-syntax; ∃; ∃-syntax; _×_; _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Data.Nat.Properties using (+-assoc; +-identityʳ; +-suc; +-comm)
open import Axiom.Extensionality.Propositional using (Extensionality)
```

Chapter 3

De Bruijn indices and the σ-calculus

In this chapter we introduce the de Bruijn notation for λ-calculus, explain its usefulness in elegantly formalizing and expressing lambda binders, and briefly present the infrastructure implemented by [WK19] that we used as a foundation on which to develop further theorems for untyped λ-calculus. This infrastructural part refers to a paper by Schäfer et al. [STS15], where a library in Coq to automatically reason on de Bruijn terms and substitutions is implemented. This work in turn uses the theoretical foundations presented by Abadi et al. [Aba+91], which introduce the concept of explicit substitutions and the σ-calculus.
We will assume that the reader is familiar with the basic notions of λ-calculus, such as substitution, binders, capturing, α-equivalence. For a more complete treatment of λ-calculus we refer to [Bar85].

3.1 Perspective

When formalizing λ-calculus, the first fundamental choice to make is selecting an appropriate representation for λ-terms and the concept of substitution, crucial in implementing β-reduction. This problem reduces to finding the most reasonable and effective method to implement a famously hard to formalize concept in λ-calculus and the theory of programming languages in general: name binders and how to avoid the capturing of free variables in substitutions. The first main theoretical approach to this problem is represented by the *Barendregt's Variable Convention*, which considers λ-terms up to renaming of bound variables in such a way that no capturing can occur. This greatly simplifies avoiding capturing substitutions and the treatment of binders in proofs.

We here quote the main points of the convention, as stated in [Bar85]:

Convention. Terms that are α congruent are identified. So now we write $\lambda x.x \equiv \lambda y.y$, etc.
Variable convention. If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

These two conventions jointly allow to address the main points that an appropriate representation for λ-terms should provide:

- Identify two terms if each can be transformed into the other by a renaming of its bound variables.

- Consider a λ-term as a representative of its equivalence class.

- Interpret substitution $M[x := N]$ as an operation on the equivalence classes of M and N. This operation can be performed using representatives, provided that the bound variables are named properly according to the variable convention above.

One of the major problems in formalizing mathematical theorems with proof assistants is precisely capturing these informal conventions, which are usually introduced to aid both expositions and proofs of mathematical concepts. Even though these mechanisms lift the mathematician from the burden of having to consider these important but fundamentally tedious details, they unfortunately cannot be omitted in formally-verified proofs. The most-well known mechanism to conveniently formalize and solve these problems in machine-checked proofs is the concept of de Bruijn indices [Bru72], which we present in this chapter along with their formalization in Agda developed by Wadler et al. [WK19].
Other important formalization techniques include the Locally Nameless method [Cha12], the Higher-Order Abstract Syntax representation [How10], the Nominal library implemented in Isabelle/HOL [Urb08], and the approaches presented in Agda by [CST17]. These two latter developments explicitly try to mirror pen-and-paper proofs by providing similar constructions and induction principles as those presented by the Barendregt Variable Convention. Another relevant paper on these topics is [Ayd+08], which further describes the Locally Nameless technique while also giving a survey of the other representation methods.

Classic notation	de Bruijn indices
$\lambda x.x$	$\lambda 0$
$\lambda x.\lambda y.xy$	$\lambda\lambda 10$
$(\lambda x.x)(\lambda f.ff)$	$(\lambda 0)(\lambda 00)$
$\lambda x.\lambda y.(\lambda z.xy)y$	$\lambda\lambda(\lambda 21)0$
$\lambda a.\lambda b.\lambda c.abb$	$\lambda\lambda\lambda 211$
$\lambda x.x(\lambda y.\lambda f.yx(ff)x)$	$\lambda 0(\lambda\lambda 12(00)2)$
$\lambda f.(\lambda x.f)(\lambda y.(\lambda r.ryf)yf)$	$\lambda(\lambda 1)(\lambda(\lambda 012)01)$
$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$	$\lambda(\lambda 1(00))(\lambda 1(00))$

Table 3.1: *Some examples of λ-terms with de Bruijn indices*

3.2 De Bruijn indices

Historically, the first attempt to adequately solve these problems regarding binders and variable capturing for λ-calculus has been the work developed by Nicolaas G. de Bruijn [Bru72] for his automated theorem prover AUTOMATH, one of the first examples of automated computer-assisted theorem proving. This has given rise to the concept of de Bruijn indices, which essentially act as notation to represent λ-calculus terms without having to give names to bound variables. Using this mechanism as a representation method lifts the prover from the many issues of variable renamings and capturing, and makes it easier to implement substitutions. Instead of using symbolic names, variables using the de Bruijn notation[1] are simply expressed through the use of numbers which, intuitively, indicate to the number of λ-abstractions one has to skip to get to the binder that introduces the variable. Note that this is not the same as simply using numbers instead of identifiers: the catch is that λ-abstractions do not express any variable name altogether, therefore also slightly sacrificing the readability of terms. The most important advantage of this notation, however, is that α-equivalent terms become *syntactically* equal, with the concept of α-conversion and the problems related to binders and capturing variable names disappearing altogether.
The way we represent λ-terms in Agda is clearly through the use of a datatype. However, instead of adopting a naïve approach where de Bruijn indexes are represented by plain natural numbers, we can exploit dependent types in such a way the type of each term also reflects the number of free variables that it can refer

[1]We will refer to this mechanism as ”de Bruijn notation”, not to be confused with another representation method that bears a similar name where applications are written in the reverse order and next to the abstraction binder.

to. The idea is that `Term 0` is going to be the type of terms with no free variables (i.e.: closed terms), `Term 1` the set of terms that can refer to one variable above the outermost, and so on. This method also allows us to be more specific in our term representation, since the number of free variables can be known at compile-time. Its use as a natural way to express λ-terms will become clear in Section 3.3 when introducing the concept of parallel substitution of free variables.
Let us indicate[2] with $\#i$ the variable with de Bruijn index i. We say that a term can refer to up to n free variables in the following cases:

- If the term is a variable $\#i$, then i must be a number $0 \leq i < n$.
- If the term is a λ-abstraction, then the body of the abstraction is a term that can refer up to $n + 1$ free variables, since it can also use the newly-introduced variable $\#0$. In accordance with de Bruijn notation, all the indices inside the body of the abstraction will increase by one when referring to outside variables, since they will have to skip the initial abstraction.
- Lastly, if the term is an application, then both subterms must refer to up the same number of free variables.

Note that terms referring to up to n free variables can be also "interpreted" as referring to $n + k$ more variables with $k \geq 0$, since we need not necessarily use all the available indices in our term. We can express this concept of delimited indices with the `Fin` datatype, which is the type of finite types.

```
data Fin : ℕ → Set where
  zero : ∀ {n : ℕ}              → Fin (suc n)
  suc  : ∀ {n : ℕ} (i : Fin n) → Fin (suc n)
```

For instance:

- The type `Fin 0` contains no inhabitant terms.
- The type `Fin 1` contains the term `zero`.
- The type `Fin 2` contains the terms `zero`, `suc zero`.
- The type `Fin 3` contains the terms `zero`, `suc zero`, `suc (suc zero)`.
- ...

[2]Even though the original notation as expressed by de Bruijn started from one, we will use here the zero-indexed notation.

Agda also provides in the standard library the syntactic sugar `0F`, `1F`, `2F`, etc. to indicate the corresponding `Fin` terms. Note that `zero` can belong to any `Fin n`, as it precisely expressed by the definition. This feature, similar to a sort of polymorphic behaviour, is provided by the fact that the quantified parameter `n` remains implicit and gets automatically inferred by Agda according to the context. For example, by setting `n` as `0` we have `0F : Fin (suc 0)`, `0F : Fin (suc (suc 0))`, and similarly with other values.
Given this definition, we can now express the type of λ-terms as follows:

```
infix  9 #_
infixl 7 _·_
infix  6 ƛ_

data Term : ℕ → Set where
    #_  : ∀ {n : ℕ} → Fin n              → Term n
    ƛ_  : ∀ {n : ℕ} → Term (suc n)       → Term n
    _·_ : ∀ {n : ℕ} → Term n → Term n → Term n
```

It is necessary to name the constructor for λ-abstraction with "ƛ" so that it does not conflict with the "λ" already used in Agda for the usual function abstraction. Here are some examples of valid λ-terms with our definition:

```
_ : Term 0
_ = ƛ # 0F
_ : Term 1
_ = ƛ ƛ ƛ # 3F
_ : Term 0
_ = (ƛ # 0F) · (ƛ # 0F · # 0F)
_ : Term 2
_ = ƛ # 1F · (ƛ ƛ # 4F)
_ : Term 0
_ = ƛ # 0F · (ƛ ƛ # 1F · # 2F · (# 0F · # 0F) · # 2F)
_ : Term 2
_ = ƛ (ƛ # 3F) · (ƛ (ƛ # 0F · # 4F · # 3F) · # 0F · # 1F)
_ : Term 4
_ = ƛ (ƛ # 1F · (# 5F · # 5F)) · (ƛ # 0F · (# 4F · # 4F))
```

In a similar way as with the type Fin, we can identify terms with multiple appropriate types:

```
_ : Term 0
_ = ƛ # 0F
_ : Term 5
_ = ƛ # 0F
```

To make these definitions work, Agda is again inferring under the hood the appropriate implicit parameter so that the definition typechecks with the specified type. Agda also informs us that it cannot automatically resolve those cases where the terms do not reflect their expected types:

```
-- The following terms are ill-typed
_ : Term 1
_ = ƛ # 2F
_ : Term 2
_ = ƛ # 1F · (ƛ # 4F)
```

The fact that terms always belong in a sufficiently large context for their free variables is therefore intrinsically checked at compile-time by the type of their constructors. Note that any term with type `Term 0` (the type of closed terms) must necessarily be either a λ-abstraction or an application, since the variable `# 0` has type at least `Term (suc n)`, for any `n`. This is due to the fact that `0F : ∀ {n : ℕ} → Fin (suc n)`, which forces us to use a λ-abstraction to "lower" the size of the Fin parameter so that the term can overall have size zero.

3.2.1 Comparison with the original development

Instead of directly employing the library constructed by Wadler et al. [WK19], we chose to simplify their development in order to describe de Bruijn indices in a more direct fashion. The original architecture is more general, and uses the concept of intrinsically-typed terms where the type Fin represents contexts and terms correspond to a typing judgment with only one possible type. For our specific purposes, we decided to identify a simpler but isomorphic structure, represented by the Fin and Term datatypes, and we then re-purposed their architecture to mirror these less general but easier to read definitions.

3.3 Substitution

We can finally introduce the concept of parallel substitutions for λ-terms described bt Schäfer et al. [STS15] in Coq and implemented in Agda by Wadler et al. [WK19]. We present here the fundamental ideas and lemmas.

Substitution for de Bruijn terms behaves differently than the standard named representation for λ-terms. For example, substituting an index n requires us to refer to the index $n+1$ when entering a λ-abstraction, since all the indices shift accordingly to the fact that we need to account the new λ-abstraction. This operation can be done through the concept of *extending* a substitution, which effectively applies the substitution to indices starting from 1 instead of 0. In practical use, however, there is usually no syntactic need to specify the variable being substituted, since in the context of λ-calculus the variable with index 0 is the one being substituted in β-redexes.

There are other important semantic differences that need to be accounted for because of this representation. For instance, if a term that gets substituted inside an abstraction body refers to some free variable n, that variable index will have to be appropriately increased in order to maintain the reference to the same binder.

$$\lambda 0(\lambda 01)\,\textbf{[}\,\underline{(\lambda 12(\lambda 01))}\,\textbf{]} = \underline{(\lambda 12(\lambda 01))}(\lambda 0\underline{(\lambda 23(\lambda 01))})$$

We provide here an example of β-reduction, with $(\lambda 12(\lambda 01))$ as the term being substituted for. Here, we can immediately notice that the two variables 1 and 2 on the left are free since they do not refer to any previously introduced λ-abstraction, and that 0 and 1 on the right are instead bound. We have underlined the two occurrences where this term gets replaced in the term on the left. When substituting, the two free indices have to be incremented accordingly to the fact that they are being inserted within a new abstraction. Therefore, our substitution function will need to incorporate some concept of ”shifting”, or extending, in such a way that the meaning and reference of free variables is preserved.

3.3.1 Substitutions as functions

In the context of λ-terms, it is both useful and necessary to generalize the concept of a substitution to that of a function associating de Bruijn indices to terms. This is precisely what allows us to easily introduce the previously mentioned concept of extension of a substitution, which considers $n+1$ indices instead of n inside λ-abstractions.

This interpretation of substitutions as functions can be used to imagine a generic substitution as a linear sequence of terms, associating each de Bruijn index with a corresponding term. The substitution is then performed in a parallel fashion, where each index considered is simultaneously substituted in the term.
We can now define both the type of substitutions and the substitution function as follows:

```
Subst : ℕ → ℕ → Set
Subst n m = Fin n → Term m

subst : ∀ {n m} → Subst n m → (Term n → Term m)
subst σ (# x)   = σ x
subst σ (ƛ M)   = ƛ (subst (exts σ) M)
subst σ (M · N) = subst σ M · subst σ N
```

Here, `exts` refers to the extension of a substitution, which formalizes the two operations we previously described. It is defined as follows:

```
exts : ∀ {n m} → Subst n m → Subst (suc n) (suc m)
exts σ zero    = # zero
exts σ (suc x) = rename suc (σ x)
```

The pattern matching makes it so that the extended substitution "ignores" the case of the zero-variable, which would be the one being introduced by the λ-abstraction; by pattern matching on `(suc x)` and then applying the substitution to `x`, it also makes it so that the substitution operates on a decremented index. This is what mirrors the fact that the extension of substitution should be applied on indices starting from one instead of zero. The `rename` function, finally, appropriately shifts by one (through `suc`) all the indices of the free variables of the resulting term `σ x`, in order to preserve their reference. In this context, the extension of a substitution is effectively an extension of both its domain (in terms of indices that a substitution can take) and its codomain (in terms of the possible range of indices that the substituted terms can refer to). This fact is perfectly mirrored in the type signature of the `exts` function, where both `n` and `m` are incremented by one.
We can now present the definition of the `rename` function, along with the type of renamings. A renaming is a function from a given set of de Bruijn indices, bounded by n, to another set of indices bounded by m. One such example of a renaming is the `suc` function, which simply increments all indices by one. The definitions are straightforward:

```
Rename : ℕ → ℕ → Set
Rename n m = Fin n → Fin m

rename : ∀ {n m} → Rename n m → (Term n → Term m)
rename ρ (# x)   = # (ρ x)
rename ρ (ƛ M)   = ƛ (rename (ext ρ) M)
rename ρ (M · N) = rename ρ M · rename ρ N
```

Applying a renaming through the function `rename` effectively alters the number of free variables that a term can refer to, which is why we get `Term m` from `Term n`. Similarly to the case of substitutions, we also have a concept of extension of a renaming:

```
ext : ∀ {n m} → Rename n m → Rename (suc n) (suc m)
ext ρ zero    = zero
ext ρ (suc x) = suc (ρ x)
```

This extension accounts for the fact that renamings can be applied inside the body of a λ-abstraction, thus explaining its use in `rename`. As before, we ignore the case where the index is zero, since it refers to the newly-introduced variable. In the other case, we again decrease the index being considered by pattern matching through `(suc x)`, and then apply the renaming to `x`. We then increment the final result, so that it accounts for the fact that all indices appearing in a λ-abstraction have to be incremented to preserve the intended meaning given by the renaming ρ. Finally, we can express zero-indexed substitution as follows, along with a more familiar syntactic notation that applies it to terms:

```
subst-zero : ∀ {n} → Term n → Subst (suc n) n
subst-zero M zero    = M
subst-zero M (suc x) = # x

infix 8 _[_]

_[_] : ∀ {n} → Term (suc n) → Term n → Term n
M [ N ] = subst (subst-zero N) M
```

The concept of single-term substitution is simply that of a substitution replacing the index 0 with a provided term and decrementing the rest of the variables. This latter mechanism is again implemented through pattern matching and mirrors

the fact that, by eliminating the λ-abstraction, all free variables greater than zero get decremented by one in order to correctly maintain their references. This explains the signature of `subst-zero` as `Subst (suc n) n`, since it effectively eliminates the variable 0 that the abstraction had introduced. This removal of the outermost λ-abstraction is precisely what justifies zero-indexed substitution as the basis of β-reduction.

3.4 σ-calculus

In this section we briefly introduce the main results regarding parallel substitutions formalized by Wadler et al. [WK19]. These properties find their theoretical foundations in the σ-calculus defined by Abadi et al. [Aba+91], which acts as an algebraic mechanism to explicitly reason on substitutions. Aside from being an useful system on its own, it can also be used to elegantly prove the crucial substitution lemma presented by Barendregt [Bar85, Lemma 2.1.16]. We will later employ the foundational theorems described here in Chapter 4 and Chapter 7.
The σ-calculus uses four constructs to define and operate on substitutions:

- The *identity* substitution, which simply takes a de Bruijn index and associates it with a variable with the same index.

  ```
  ids : ∀ {n} → Subst n n
  ids x = # x
  ```

  This gives the following substitution, where we associate the position of each term with the index being considered (with index zero starting on the left, and increasing on the right):

  ```
  # 0, # 1, # 2, ...
  ```

- The *shift* operation ↑ increments all the given indices by one.

  ```
  ↑ : ∀ {n} → Subst n (suc n)
  ↑ x = # (suc x)
  ```

  This gives the substitution sequence:

  ```
  # 1, # 2, # 3, ...
  ```

- The *cons* operation • takes a term and a substitution and "attaches" it as first element of the substitution, shifting all the other terms.

  ```
  infix 6 _•_

  _•_ : ∀ {n m} → Term m → Subst n m → Subst (suc n) m
  (M • σ) zero    = M
  (M • σ) (suc x) = σ x
  ```

  Therefore, given a term `M` and a substitution σ, the consing operation constructs the following sequence:

  ```
  M, σ 0, σ 1, σ 2, ...
  ```

- Finally, the *composition* operation ⨾ functionally composes two substitutions. Even though the original paper by Abadi et al. [Aba+91] uses the classic mathematical order for composition, with τ applied before σ in $\sigma \circ \tau$, Wadler et al. [WK19] chose to implement the clearer concept of forward composition, with σ being the first function to be applied in $\sigma ⨾ \tau$. This definition uses the functional composition defined in the Agda standard library and denoted with "∘":

  ```
  infixr 5 _⨾_

  _⨾_ : ∀ {n k m} → Subst n k → Subst k m → Subst n m
  σ ⨾ τ = ≪ τ ≫ ∘ σ
  ```

The following notation is also used to shorten the use of the `subst` function and express it more conveniently:

```
≪_≫ : ∀ {n m} → Subst n m → Term n → Term m
≪ σ ≫ = subst σ
```

We can also see that any renaming can be equivalently expressed as a substitution that simply "lifts" the returned indices into variables:

```
ren : ∀ {n m} → Rename n m → Subst n m
ren ρ = ids ∘ ρ
```

Notice how our special `subst-zero` substitution can also be defined as `M • ids`, given any term `M`.

3.4.1 σ-calculus equations

The σ-calculus is equipped with its defining equations which operate on substitutions, listed as follows:

```
(sub-head)  ≪ M • σ ≫ (# Z) ≡ M
(sub-tail)  ↑ ⨾ (M • σ) ≡ σ
(sub-η)     (≪ σ ≫ (# Z)) • (↑ ⨾ σ) ≡ σ
(Z-shift)   (# Z) • ↑ ≡ ids

(sub-id)    ≪ ids ≫ M ≡ M
(sub-app)   ≪ σ ≫ (L · M)  ≡ (≪ σ ≫ L) · (≪ σ ≫ M)
(sub-abs)   ≪ σ ≫ (ƛ N) ≡ ƛ ≪ σ ≫ N
(sub-sub)   ≪ τ ≫ ≪ σ ≫ M ≡ ≪ σ ⨾ τ ≫ M

(sub-idL)   ids ⨾ σ ≡ σ
(sub-idR)   σ ⨾ ids ≡ σ
(sub-assoc) (σ ⨾ τ) ⨾ θ ≡ σ ⨾ (τ ⨾ θ)
(sub-dist)  (M • σ) ⨾ τ ≡ (≪ τ ≫ M) • (σ ⨾ τ)
```

The meaning of these equations is self-explanatory and almost directly map one-to-one into Agda. These properties form the basis on which the authors successively develop the following theorems, by using the very same equational reasoning we presented in Section 2.4.4 when introducing Agda:

```
ren-ext : ∀ {n m} {ρ : Rename n m}
  → ren (ext ρ) ≡ exts (ren ρ)

rename-subst-ren : ∀ {n m} {ρ : Rename n m} {M : Term n}
  → rename ρ M ≡ ≪ ren ρ ≫ M

exts-cons-shift : ∀ {n m} {σ : Subst n m}
  → exts σ ≡ (# zero • (σ ⨾ ↑))

subst-zero-cons-ids : ∀ {n} {M : Term n}
  → subst-zero M ≡ (M • ids)

compose-ext : ∀ {n m k} {ρ : Rename m k} {ρ′ : Rename n m}
  → ((ext ρ) ∘ (ext ρ′)) ≡ ext (ρ ∘ ρ′)
```

```
compose-rename : ∀ {n m k} {M : Term n} {ρ : Rename m k} {ρ′ : Rename n m}
  → rename ρ (rename ρ′ M) ≡ rename (ρ ∘ ρ′) M

commute-subst-rename : ∀ {n m}{ M : Term n} {σ : Subst n m}
                        {ρ : ∀ {n} → Rename n (suc n)}
  → (∀ {x : Fin n} → exts σ (ρ x) ≡ rename ρ (σ x))
  → subst (exts σ) (rename ρ M) ≡ rename ρ (subst σ M)

exts-seq : ∀ {n m m′} {σ₁ : Subst n m} {σ₂ : Subst m m′}
  → (exts σ₁ ⨾ exts σ₂) ≡ exts (σ₁ ⨾ σ₂)

rename-subst : ∀ {n m m′} {M : Term n} {ρ : Rename n m} {σ : Subst m m′}
  → ≪ σ ≫ (rename ρ M) ≡ ≪ σ ∘ ρ ≫ M

subst-zero-exts-cons : ∀ {n m} {σ : Subst n m} {M : Term m}
  → exts σ ⨾ subst-zero M ≡ (M • σ)
```

These theorems both elaborate on the σ-calculus foundations and connect them to the definitions of `subst`, `exts`, `rename`, `ext` previously defined on λ-terms.

3.4.2 Fundamental theorems

However, the important results that we employ and require in our theorems regarding substitution are the following lemmas, for which σ-calculus is effectively defined. We provide the entire proofs to show the equational reasoning behind them, while omitting some technical definitions about conjugation that simply allow us to apply the above mentioned properties:

```
subst-commute :
  ∀ {n m} {N : Term (suc n)} {M : Term n} {σ : Subst n m}
    → ≪ exts σ ≫ N [ ≪ σ ≫ M ] ≡ ≪ σ ≫ (N [ M ])
subst-commute {n}{m}{N}{M}{σ} =
  begin
    ≪ exts σ ≫ N [ ≪ σ ≫ M ]
  ≡⟨⟩
    ≪ subst-zero (≪ σ ≫ M) ≫ (≪ exts σ ≫ N)
  ≡⟨ cong-sub {M = ≪ exts σ ≫ N} subst-zero-cons-ids refl ⟩
    ≪ ≪ σ ≫ M • ids ≫ (≪ exts σ ≫ N)
  ≡⟨ sub-sub {M = N} ⟩
    ≪ (exts σ) ⨾ ((≪ σ ≫ M) • ids) ≫ N
  ≡⟨ cong-sub {M = N} (cong-seq exts-cons-shift refl) refl ⟩
  ≪ (# zero • (σ ⨾ ↑)) ⨾ (≪ σ ≫ M • ids) ≫ N
  ≡⟨ cong-sub {M = N} (sub-dist {M = # zero}) refl ⟩
    ≪ ≪ ≪ σ ≫ M • ids ≫ (# zero) • ((σ ⨾ ↑) ⨾ (≪ σ ≫ M • ids)) ≫ N
  ≡⟨⟩
    ≪ ≪ σ ≫ M • ((σ ⨾ ↑) ⨾ (≪ σ ≫ M • ids)) ≫ N
  ≡⟨ cong-sub {M = N} (cong-cons refl (sub-assoc {σ = σ})) refl ⟩
    ≪ ≪ σ ≫ M • (σ ⨾ ↑ ⨾ ≪ σ ≫ M • ids) ≫ N
  ≡⟨ cong-sub {M = N} refl refl ⟩
    ≪ ≪ σ ≫ M • (σ ⨾ ids) ≫ N
  ≡⟨ cong-sub {M = N} (cong-cons refl (sub-idR {σ = σ})) refl ⟩
    ≪ ≪ σ ≫ M • σ ≫ N
  ≡⟨ cong-sub {M = N} (cong-cons refl (sub-idL {σ = σ})) refl ⟩
    ≪ ≪ σ ≫ M • (ids ⨾ σ) ≫ N
  ≡⟨ cong-sub {M = N} (sym sub-dist) refl ⟩
    ≪ M • ids ⨾ σ ≫ N
  ≡⟨ sym (sub-sub {M = N}) ⟩
    ≪ σ ≫ (≪ M • ids ≫ N)
  ≡⟨ cong ≪ σ ≫ (sym (cong-sub {M = N} subst-zero-cons-ids refl)) ⟩
    ≪ σ ≫ (N [ M ])
  ∎
```

```
rename-subst-commute :
  ∀ {n m} {N : Term (suc n)}{ M : Term n} {ρ : Rename n m}
    → (rename (ext ρ) N) [ rename ρ M ] ≡ rename ρ (N [ M ])
rename-subst-commute {n}{m}{N}{M}{ρ} =
  begin
    (rename (ext ρ) N) [ rename ρ M ]
  ≡⟨ cong-sub (cong-sub-zero (rename-subst-ren {M = M}))
                              (rename-subst-ren {M = N}) ⟩
    (≪ ren (ext ρ) ≫ N) [ ≪ ren ρ ≫ M ]
  ≡⟨ cong-sub refl (cong-sub {M = N} ren-ext refl) ⟩
    (≪ exts (ren ρ) ≫ N) [ ≪ ren ρ ≫ M ]
  ≡⟨ subst-commute {N = N} ⟩
    subst (ren ρ) (N [ M ])
  ≡⟨ sym (rename-subst-ren) ⟩
    rename ρ (N [ M ])
  ∎
```

The importance of these theorems comes from the fact that, when read right to left, they allow us to effectively "distribute" a substitution or a renaming inside the 0-indexed substitution subst-zero. These two properties will turn out to form the basis on which to develop further theorems about substitutions and reductions in Chapter 4 and Chapter 7. These lemmas have also been employed in the proofs by Wadler et al. [WK19] presented in Chapter 5.
By introducing a new syntactical construct to indicate the equivalent of 0-indexed substitution for the index 1, we can also recognize within subst-commute a generalized form of the substitution lemma defined in [Bar85, Lemma 2.1.16]:

```
infix 8 _⟦_⟧

_⟦_⟧ : ∀ {n} → Term (suc (suc n)) → Term n → Term (suc n)
_⟦_⟧ N M = subst (exts (subst-zero M)) N

substitution-lemma :
  ∀ {n} {N : Term (suc (suc n))} {M : Term (suc n)} {L : Term n}
    → N [ M ] [ L ] ≡ (N ⟦ L ⟧) [ M [ L ] ]
substitution-lemma {N = N}{M = M}{L = L} =
  sym (subst-commute {N = N}{M = M}{σ = subst-zero L})
```

Chapter 4

The Church-Rosser Theorem

In this chapter we present the definitions required to state the confluence of β-reduction for untyped λ-calculus, a result first presented in 1936 by Alonzo Church and J. Barkley Rosser and famously known as the Church-Rosser theorem. [CR36] We first prove a crucial property later used in proofs, the substitutivity of β^*-reduction, and then introduce the theoretical definition of confluence.

4.1 β-reduction

The most important relation in λ-calculus is β-reduction, and it can be readily defined in Agda as follows, using the previously introduced notion of zero-indexed substitution in the crucial case:

```
infix 3 _→_

data _→_ : ∀ {n} → Term n → Term n → Set where

  →-ξₗ : ∀ {n} {M M′ N : Term n}
    → M → M′
      ---------------
    → M · N → M′ · N

  →-ξᵣ : ∀ {n} {M N N′ : Term n}
    → N → N′
      ---------------
    → M · N → M · N′
```

```
  →-ƛ : ∀ {n} {M M′ : Term (suc n)}
    → M → M′
      -----------
    → ƛ M → ƛ M′

  →-β : ∀ {n} {M : Term (suc n)} {N : Term n}
      --------------------
    → (ƛ M) · N → M [ N ]
```

We now define the reflexive transitive closure of β-reduction, which can either consist of zero reduction steps (the reflexive case) or "adding" a new β-reduction at the start of an existing reduction chain. We will indicate in Agda the transitive closure of β-reduction with ↠, as shown here:

```
infix  3 _↠_
infixr 3 _→⟨_⟩_
infix  4 _∎

data _↠_ : ∀ {n} → Term n → Term n → Set where

  _∎ : ∀ {n} (M : Term n)
      ------
    → M ↠ M

  _→⟨_⟩_ : ∀ {n} {L N : Term n} (M : Term n)
    → M → L
    → L ↠ N
      ------
    → M ↠ N
```

Notice how the term `M` is here provided as an explicit argument, and it is the first syntactic argument of the constructor `_→⟨_⟩_`. This idiom follows the same structure as the reasoning system for equality defined in the Agda standard library and introduced in Section 2.4.4. The term is here made explicit so that the reduction rules applied can be shown step-by-step.
We can prove some similar straightforward properties that β^*-reduction inherits from β-reduction, such as transitivity and the other congruences. The proofs are by induction on the structure of the first reduction:

```
↠-trans : ∀ {n} {M L N : Term n}
  → M ↠ L
    -------
  → M ↠ N
↠-trans (M ∎) M↠M = M↠M
↠-trans (M →⟨ M→L′ ⟩ L′↠L) L↠N =
  M →⟨ M→L′ ⟩ (↠-trans L′↠L L↠N )

↠-congₗ : ∀ {n} {M M′ R : Term n}
  → M ↠ M′
    ---------------
  → M · R ↠ M′ · R
↠-congₗ {M = M}{R = R} (M ∎) = M · R ∎
↠-congₗ {M = M}{R = R} (M →⟨ M→L ⟩ L↠M′)
  = M · R →⟨ →-ξₗ M→L ⟩ ↠-congₗ L↠M′

↠-congᵣ : ∀ {n} {M M′ L : Term n}
  → M ↠ M′
    ---------------
  → L · M ↠ L · M′
↠-congᵣ {M = M}{L = L} (M ∎) = L · M ∎
↠-congᵣ {M = M}{L = L} (M →⟨ M→L ⟩ L↠M′)
  = L · M →⟨ →-ξᵣ M→L ⟩ ↠-congᵣ L↠M′

↠-cong-ƛ : ∀ {n} {M M′ : Term (suc n)}
  → M ↠ M′
    -----------
  → ƛ M ↠ ƛ M′
↠-cong-ƛ (M ∎) = ƛ M ∎
↠-cong-ƛ (M →⟨ M→L ⟩ L↠N′)
  = ƛ M →⟨ →-ƛ M→L ⟩ ↠-cong-ƛ L↠N′

↠-cong : ∀ {n} {M M′ N N′ : Term n}
  → M ↠ M′
  → N ↠ N′
    ----------------
  → M · N ↠ M′ · N′
↠-cong M↠M′ N↠N′ = ↠-trans (↠-congₗ M↠M′) (↠-congᵣ N↠N′)
```

Note that in the case of simple β-reduction these properties are not theorems, but definitions required in order to be able to contract existing redexes inside λ-terms (i.e.: β-reduction is essentially the *compatible closure* of the β-reduction rule $((\lambda x.M)N, M[x := N])$, as defined in [Bar85, Definition 3.1.4]). The same congruence properties also hold for β^*-reduction precisely because β-reduction is defined upon them. The definitions presented so far can also be found verbatim in [WK19], up to renaming.

4.2 Substitutivity of β^*-reduction

One of the first non-trivial lemmas regarding β^*-reduction that we need in further proofs is the fact that it respects substitution:

Theorem 4.2.1 (Substitutivity of β^*-reduction). For all λ-terms M, N and any variable x, given $M \to_\beta^* M'$ and $N \to_\beta^* N'$, then:

$$M[x := N] \to_\beta^* M'[x := N']$$

We will refer to this property as *substitutivity of β^*-reduction*, also called (in the context of parallel reduction) *"strong substitutivity"* in [STS15, Lemma 3] and stated without name in [Bar85, Proposition 2.1.17 (iii)]. This theorem will be required in those inductive cases where β-reduction is applied, since it is effectively defined upon the special case of 0-indexed substitution. We make the reader aware of two conventions employed in the rest of this thesis: first, we will denote theorems regarding the transitive closure of relations with an "s" at the end of the name in order to distinguish them with their single-step version, for example using "`betas`" in the case of β-reduction. This convention will also be used in Chapter 5 for parallel reduction and its transitive closure. Secondly, we shall omit variables when substitutions refer to the first 0-indexed variable, in order to be consistent with the notation employed with de Bruijn indices.
The theorem can now be stated in Agda as follows:

```
sub-betas : ∀ {n} {M M′ : Term (suc n)} {N N′ : Term n}
  → M ↠ M′
  → N ↠ N′
    --------------------
  → M [ N ] ↠ M′ [ N′ ]
```

This is one of the first examples where proofs based on the de Bruijn representation start becoming non-trivial, especially when compared to the classic pen-and-paper ones. This is obvious from the fact that we are effectively entering the domain of substitutions, along with their interaction with β-reduction.
Note how the maximum number of free variables for M' must be one more than that of the term M being substituted, a fact both reflected here in the implicit type parameters and, most importantly, in the type signature of the `_[_]` operator.
If we were to directly try proving this theorem by induction on the structure of the term M, as one would do in a standard proof, we would immediately encounter difficulties. In the case where M is a λ-abstraction we would have to prove the following statement:

```
ƛ subst (exts (subst-zero N)) M ↠ ƛ subst (exts (subst-zero N′)) M′
```

Our inductive hypothesis is simply not strong enough, since we need to operate on *the extension* of a substitution, `exts (subst-zero N)`, and then reach `exts (subst-zero N′)` given the fact that $N \to_\beta^* N'$. This guides us towards a generalization of the theorem that uses a generic substitution σ so that we can also account for the extension of any substitution. However, the reduction $N \to_\beta^* N'$ also needs to be extended, since we are not treating the specific substitution `subst-zero N` with the term `N` anymore. This can be done by introducing the concept that substitutions themselves can β-reduce, through the notion of *pointwise reduction*.

4.2.1 Pointwise β^*-reduction

Definition 4.2.1 (Pointwise reduction). A substitution σ pointwise β-reduces to the substitution σ' if, for all possible x, we have that $\sigma\ x \to_\beta^* \sigma'\ x$.

This same generalization technique is also used in [WK19, Confluence] and [STS15] to prove an equivalent lemma in the case of parallel reduction. This important proof approach constitutes a recurring theme when operating with substitutions. In terms of notation, we shall use a subscript s in the Agda code to denote that the operation is defined for substitutions:

```
infix 3 _↠ₛ_

_↠ₛ_ : ∀ {n m} → Subst n m → Subst n m → Set
σ ↠ₛ σ′ = ∀ {x} → σ x ↠ σ′ x
```

Given this shortcut to express the pointwise reduction of two substitutions σ and σ', we can show the generalized form of our theorem by induction on the structure of the $M \to_\beta^* M'$ reduction:

```
subst-betas : ∀ {n m} {σ σ′ : Subst n m} {M M′ : Term n}
  → σ ↠ₛ σ′
  → M ↠ M′
    ------------------------
  → subst σ M ↠ subst σ′ M′
subst-betas σ↠σ′ (M ∎) = subst-betas-sub {M = M} σ↠σ′
subst-betas {σ = σ} σ↠σ′ (M →⟨ M→L ⟩ L↠M′) =
  subst σ M →⟨ subst-beta-term M→L ⟩ subst-betas σ↠σ′ L↠M′
```

This leads to two cases: either the term does not reduce and the substitution does (subst-betas-sub), or we naturally obtain the opposite situation for one-step β-reduction (subst-beta-term) which we then connect with the inductive hypothesis. This latter theorem turns out to be easier, and proceeds by induction on the β-reduction:

```
subst-beta-term : ∀ {n m} {M M′ : Term n} {σ : Subst n m}
  → M → M′
    -----------------------
  → subst σ M → subst σ M′
subst-beta-term (→-ƛ M→M′) = →-ƛ (subst-beta-term M→M′)
subst-beta-term (→-ξₗ M→M′) = →-ξₗ (subst-beta-term M→M′)
subst-beta-term (→-ξᵣ N→N′) = →-ξᵣ (subst-beta-term N→N′)
subst-beta-term {σ = σ} (→-β {M = M}{N = N})
  rewrite sym (subst-commute {N = M}{M = N}{σ = σ}) = →-β
```

The first three cases are trivial, and they are simply the congruence definitions for β-reduction paired with the inductive hypothesis. In the last crucial case we can finally rely on the theorem for substitutions subst-commute described in Section 3.4.2, which allows us to decompose the single-term substitution.

We use the `rewrite` syntax to apply this equality,[1] with the goal reducing from:

```
(ƛ subst (exts σ) M) · subst σ N —→ subst σ (M [ N ])
```

to:

```
(ƛ subst (exts σ) M) · subst σ N —→ (subst (exts σ) M) [ subst σ N ]
```

This last expression is just one specific case of the β-reduction rule —→-β, which we can use to conclude the proof.
The other case where the term remains fixed proceeds by induction on the structure of the term M, since we cannot meaningfully decompose the pointwise reduction:

```
subst-betas-sub : ∀ {n m} {M : Term n} {σ σ′ : Subst n m}
  → σ —↠ₛ σ′
    ------------------------
  → subst σ M —↠ subst σ′ M
subst-betas-sub {M = # x} σ↠σ′ = σ↠σ′
subst-betas-sub {M = M · N} σ↠σ′ =
  ↠-cong (subst-betas-sub {M = M} σ↠σ′)
         (subst-betas-sub {M = N} σ↠σ′)
subst-betas-sub {M = ƛ M} σ↠σ′ =
  ↠-cong-ƛ (subst-betas-sub {M = M}
               (λ {x} → betas-subst-exts σ↠σ′ {x = x}))
```

4.2.2 β^*-reduction and renamings

The case where M is a λ-abstraction now requires the following lemma `betas-subst-exts`, which simply lifts the reduction of two substitutions to the reduction of their extensions:

```
betas-subst-exts : ∀ {n m} {σ σ′ : Subst n m}
  → σ —↠ₛ σ′
    ------------------
  → exts σ —↠ₛ exts σ′
betas-subst-exts σ↠σ′ {zero}  = # zero ∎
betas-subst-exts σ↠σ′ {suc x} = betas-rename σ↠σ′
```

[1] The use of the sym property for equality is necessary here because the simplification direction of the equation is relevant, and sym allows us to obtain the symmetric version of any equality.

The proof is by induction on the (implicit) index provided to the substitution σ. Since the extension of a substitution leaves the 0-index case untouched, we only need to consider the case where the result of the substitution is renamed with suc. We can, however, generalize this case to any renaming function ρ, and then proceed by induction on the relation:

```
betas-rename : ∀ {n m} {ρ : Rename n m} {M M′ : Term n}
  → M ↠ M′
     -------------------------
  → rename ρ M ↠ rename ρ M′
betas-rename {ρ = ρ} (M ∎) = rename ρ M ∎
betas-rename {ρ = ρ} (M —→⟨ M—→L ⟩ L—↠M′) =
  rename ρ M —→⟨ beta-rename M—→L ⟩ betas-rename L—↠M′
```

This lemma simply consists in a repeated application of its version for one-step β-reduction, which we can state as follows:

```
beta-rename : ∀ {n m} {ρ : Rename n m} {M M′ : Term n}
  → M —→ M′
     -------------------------
  → rename ρ M —→ rename ρ M′
beta-rename (—→-ƛ M—→M′) = —→-ƛ (beta-rename M—→M′)
beta-rename (—→-ξₗ M—→M′) = —→-ξₗ (beta-rename M—→M′)
beta-rename (—→-ξᵣ N—→N′) = —→-ξᵣ (beta-rename N—→N′)
beta-rename {ρ = ρ} (—→-β {M = M}{N = N})
  rewrite sym (rename-subst-commute {N = M}{M = N}{ρ = ρ}) = —→-β
```

The proof is straightforward, and in the substitution case we can again employ one of the foundational theorems previously derived with σ-calculus, in this case `rename-subst-commute`. This allows us to "decompose" the substitution and conclude the case with the β-reduction rule —→-β. We can notice a striking structural similarity with the lemma `subst-beta-term`; here renaming takes the place of substitution and β^*-reduction that of β-reduction, with reduction being solely applied to the term.

We can now prove the original substitutivity theorem as a special case of the more general `subst-betas`. We first need to show the pointwise β-reduction of `subst-zero` given the reduction on the applied term:

```
betas-subst-zero : ∀ {n} {M M′ : Term n}
  → M ↠ M′
     ------------------------------
  → subst-zero M ↠ₛ subst-zero M′
betas-subst-zero M↠M′ {zero}  = M↠M′
betas-subst-zero M↠M′ {suc x} = # x ∎
```

Finally, we have our specific case of 0-indexed substitution:

```
sub-betas : ∀ {n} {M M′ : Term (suc n)} {N N′ : Term n}
  → M ↠ M′
  → N ↠ N′
     --------------------
  → M [ N ] ↠ M′ [ N′ ]
sub-betas M↠M′ N↠N′ = subst-betas (betas-subst-zero N↠N′) M↠M′
```

This concludes the proof of the substitutivity theorem. These properties about β^*-reduction will be concretely used in the main proof presented in Chapter 7.

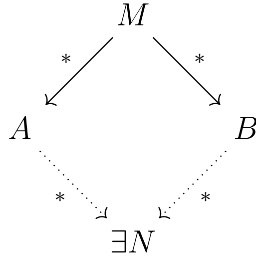
4.3 Church-Rosser Theorem

Having formally defined β-reduction and β^*-reduction, we can now give the theoretical definitions necessary to state the Church-Rosser theorem.

Definition 4.3.1 (Diamond property). A relation $\rightarrow$ is said to have or satisfy the *diamond property* if, given a term M such that $M \rightarrow A$ and $M \rightarrow B$, then there exists a term N such that $A \rightarrow N$ and $B \rightarrow N$.

Definition 4.3.2 (Confluence). A relation $\rightarrow$ is said to be *confluent*, or having the Church-Rosser property, if, given a term M such that $M \rightarrow^* A$ and $M \rightarrow^* B$, there exists a term N such that $A \rightarrow^* N$ and $B \rightarrow^* N$. Alternatively, a relation $\rightarrow$ is *confluent* iff its reflexive transitive closure satisfies the diamond property.

Confluence can be visually expressed through a commutative diagram as follows:



That is, given any two reductions stemming from the same term, there exists another term and two other reductions converging to it that "unify" the original ones. Interpreting this theorem constructively means that, given any two reductions starting from a common term M, we need to construct a function that provides us three things: a concrete term N, and two reductions $A \to^* N$ and $B \to^* N$ in an appropriate form which depends on the N provided.
One important theoretical consequence of confluence is for example the uniqueness of normal forms, when they exist:

Definition 4.3.3 (Normal form). A term N is the *normal form* of a term M with respect to $\to$ if $M \to^* N$ and there exists no term N' for which $N \to N'$.

Theorem 4.3.1 (Uniqueness of a normal form). If a term M has a normal norm N with respect to a confluent relation $\to$, then it is unique.

Proof. Suppose that M has two different normal forms N_1 and N_2 which do not further reduce to any term. We have that $M \to^* N_1$ and $M \to^* N_2$, so by confluence there exists a term N for which $N_1 \to^* N$ and $N_2 \to^* N$. Since both N_1 and N_2 can perform no other reduction steps, both reductions must be in the reflexive case, therefore $N_1 = N = N_2$. □

Confluence is studied in many other kinds of rewriting systems besides β-reduction and the λ-calculus, and it is one of the most fundamental properties checked for when considering any relation that can be interpreted as some sort of reduction. The Church-Rosser theorem for untyped λ-calculus is precisely the statement that β-reduction is confluent:

Theorem 4.3.2 (Church-Rosser theorem). β-reduction is confluent.

In the following chapters we present and gradually analyze the various proof approaches for this fundamental property of λ-calculus.

Chapter 5

The Tait/Martin-Löf proof and parallel reduction

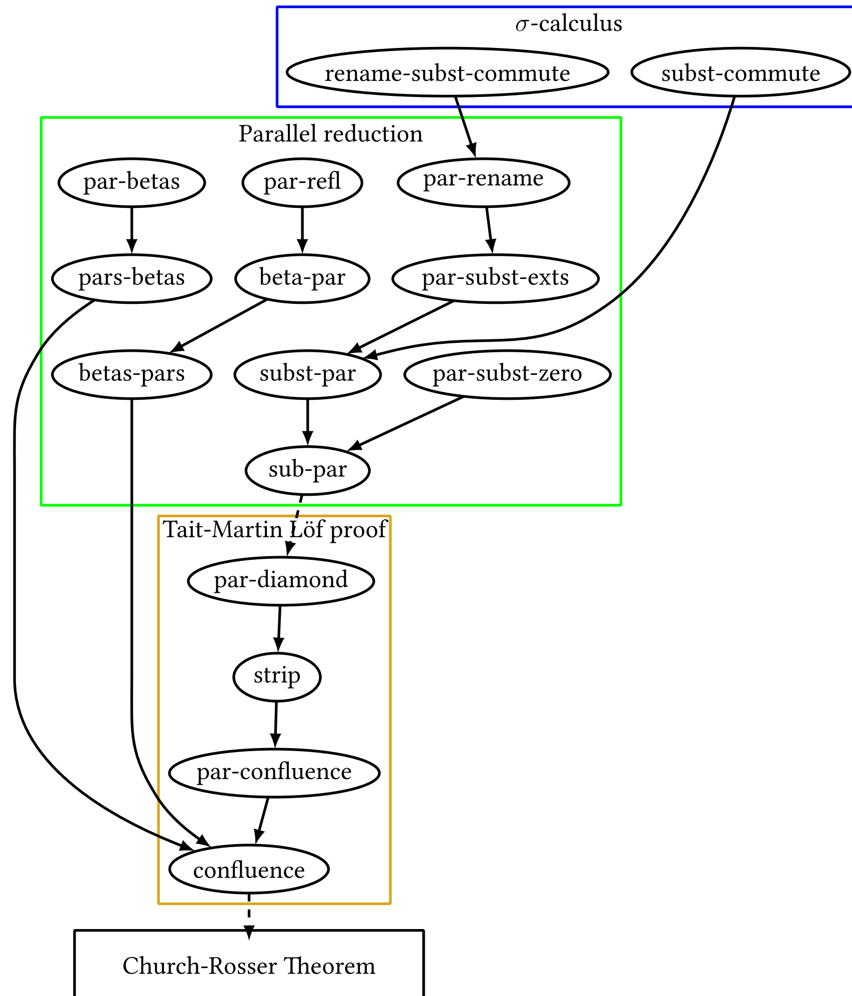
In this chapter we present the proof of the Church-Rosser theorem by Tait and Martin-Löf, along with its complete formalization in Agda developed by Wadler et al. [WK19]. This proof approach is considered the most well-known confluence proof for β-reduction, and can be found in [HS86] and [Bar85]. We include and briefly explain here the development by Wadler et al. in order to highlight its main concepts and then compare their formalization with our own. The central passage of the proof will be further treated in Chapter 6 with the improvement presented by Takahashi in [Tak95].

5.1 Main idea

The idea of the Tait-Martin Löf proof is to define a notion of *parallel reduction* $\rightrightarrows$ that can contract multiple redexes at the same time, and for which confluence is easier to prove. It is then shown that the confluence of $\rightrightarrows$ implies that of $\rightrightarrows^*$, a result also known as the strip lemma. Finally, we can prove that $\rightrightarrows^*$ and $\rightarrow_\beta^*$ actually denote the same relation, thus concluding the proof.

5.2 Proof overview

We present here an overview of the main theorems formalized by Wadler et al. [WK19], showing the theorem dependencies in the form of a directed acyclic graph:


σ-calculus
rename-subst-commute
subst-commute
Parallel reduction
par-betas
par-refl
par-rename
pars-betas
beta-par
par-subst-exts
betas-pars
subst-par
par-subst-zero
sub-par
Tait-Martin Löf proof
par-diamond
strip
par-confluence
confluence
Church-Rosser Theorem

5.3 Parallel reduction

The central definition employed in the proof is that of the so-called parallel reduction. As it can be directly seen in its formal statement, the crucial difference with β^*-reduction is that it can contract multiple redexes ”simultaneously”, by combining the results of previous parallel reductions with the substitution step of β-reduction. The definition in Agda of this relation is straightforward:

```
infix 3 _⇉_

data _⇉_ : ∀ {n} → Term n → Term n → Set where

  ⇉-c : ∀ {n} {x : Fin n}
        ----------
      → # x ⇉ # x

  ⇉-ƛ : ∀ {n} {M M′ : Term (suc n)}
      → M ⇉ M′
        -----------
      → ƛ M ⇉ ƛ M′

  ⇉-ξ : ∀ {n} {M M′ N N′ : Term n}
      → M ⇉ M′
      → N ⇉ N′
        ----------------
      → M · N ⇉ M′ · N′

  ⇉-β : ∀ {n} {M M′ : Term (suc n)} {N N′ : Term n}
      → M ⇉ M′
      → N ⇉ N′
        ----------------------
      → (ƛ M) · N ⇉ M′ [ N′ ]
```

By using the same development and technique with pointwise reduction that we have followed for the substitutivity of β^*-reduction `subst-betas`, the authors prove the equivalent theorem for parallel reduction. We include the required properties as follows, without going into the specific proof details:

```
par-subst : ∀{n m} → Subst n m → Subst n m → Set
par-subst σ σ′ = ∀ {x} → σ x ⇉ σ′ x

par-rename : ∀ {n m} {ρ : Rename n m} {M M′ : Term n}
  → M ⇉ M′
    ------------------------
  → rename ρ M ⇉ rename ρ M′
par-rename ⇉-c           = ⇉-c
par-rename (⇉-ƛ p)       = ⇉-ƛ (par-rename p)
par-rename (⇉-ξ p₁ p₂) = ⇉-ξ (par-rename p₁) (par-rename p₂)
par-rename {n}{m}{ρ} (⇉-β {n}{N}{N′}{M}{M′} p₁ p₂)
    with ⇉-β (par-rename {ρ = ext ρ} p₁) (par-rename {ρ = ρ} p₂)
... | G rewrite rename-subst-commute {n}{m}{N′}{M′}{ρ} = G

par-subst-exts : ∀ {n m} {σ σ′ : Subst n m}
  → par-subst σ σ′
    ----------------------------
  → par-subst (exts σ) (exts σ′)
par-subst-exts s {x = zero} = ⇉-c
par-subst-exts s {x = suc x} = par-rename s

subst-par : ∀ {n m} {σ σ′ : Subst n m} {M M′ : Term n}
  → par-subst σ σ′
  → M ⇉ M′
    -----------------------
  → subst σ M ⇉ subst σ′ M′
subst-par {M = # x} s ⇉-c = s
subst-par {n}{m}{σ}{σ′} {ƛ N} s (⇉-ƛ p) =
  ⇉-ƛ (subst-par {σ = exts σ}{σ′ = exts σ′}
          (λ {x} → par-subst-exts s {x = x}) p)
subst-par {M = L · M} s (⇉-ξ p₁ p₂) =
  ⇉-ξ (subst-par s p₁) (subst-par s p₂)
subst-par {n}{m}{σ}{σ′} {(ƛ N) · M} s (⇉-β {M′ = M′}{N′ = N′} p₁ p₂)
    with ⇉-β (subst-par {σ = exts σ}{σ′ = exts σ′}{M = N}
                        (λ {x} → par-subst-exts s {x = x}) p₁)
               (subst-par {σ = σ} s p₂)
... | G rewrite subst-commute {N = M′}{M = N′}{σ = σ′} = G
```

The case of parallel reduction for substitutivity turns out to require fewer lemmas than that of β^*-reduction, since it does not rely on an underlying reduction in the same way that β^*-reduction does with its single-step version. For this reason, subst-par can proceed by induction on both the relation and the structure of the term, while subst-betas has a linear proof by simply inducting on the β^*-reduction. After establishing the theorem in its general form for pointwise reduction, the case for zero-indexed substitutions comes as a corollary:

```
par-subst-zero : ∀ {n} {M M′ : Term n}
  → M ⇉ M′
    ----------------------------------------
  → par-subst (subst-zero M) (subst-zero M′)
par-subst-zero M⇉M′ {zero} = M⇉M′
par-subst-zero M⇉M′ {suc x} = ⇉-c

sub-par : ∀ {n} {M M′ : Term (suc n)} {N N′ : Term n}
  → M ⇉ M′
  → N ⇉ N′
    --------------------
  → M [ N ] ⇉ M′ [ N′ ]
sub-par M⇉M′ N⇉N′ = subst-par (par-subst-zero N⇉N′) M⇉M′
```

An intermediate lemma par-subst-zero to show the pointwise-reduction of subst-zero is necessary to conclude the proof, in a similar way as with the case of β^*-reduction. As we will see in Section 5.5, the substitutivity of parallel reduction is precisely the fundamental component necessary to prove its confluence.

5.4 Relations between parallel reduction and β-reduction

We now have to formally establish the connections between parallel reduction and β-reduction. Firstly, we have that β-reduction trivially implies parallel reduction. The opposite is clearly not true, since parallel reduction can do more than one single reduction step. Moreover, parallel reduction implies β^*-reduction, which simply takes more steps to do what the parallel one does more compactly. The proofs developed by [WK19] of the mentioned theorems are straightforward, and do not require any new property except reflexivity of parallel reduction and transitivity of β-reduction. We omit here the complete proofs for brevity.
First, we have that parallel reduction is reflexive:

```
par-refl : ∀ {n} {M : Term n}
    ------
  → M ⇉ M
```

β-reduction implies parallel reduction:

```
beta-par : ∀ {n} {M N : Term n}
  → M ⟶ N
    ------
  → M ⇉ N
```

Parallel reduction implies β^*-reduction:

```
par-betas : ∀ {n} {M N : Term n}
  → M ⇉ N
    ------
  → M ↠ N
```

The converse of `par-betas` is unfortunately not true: β^*-reduction can perform many reductions even in already reduced terms, while parallel reduction can only perform ”shallow” reductions on the initially available redexes.
β^*-reduction and parallel reduction are directly related through the reflexive transitive closure of the latter. Their equivalence is shown by simply applying the previous theorems:

```
infix  3 _⇛*_
infixr 3 _⇛⟨_⟩_
infix  4 _∎

data _⇛*_ : ∀ {n} → Term n → Term n → Set where

  _∎ : ∀ {n} (M : Term n)
       -------
    → M ⇛* M

  _⇛⟨_⟩_ : ∀ {n} {L N : Term n} (M : Term n)
    → M ⇛  L
    → L ⇛* N
       -------
    → M ⇛* N
```

β^*-reduction implies parallel reduction:

```
betas-pars : ∀ {n} {M N : Term n}
  → M ↠ N
     -------
  → M ⇛* N
betas-pars (M ∎) = M ∎
betas-pars (M →⟨ b ⟩ bs) = M ⇛⟨ beta-par b ⟩ betas-pars bs
```

And in the other direction:

```
pars-betas : ∀ {n} {M N : Term n}
  → M ⇛* N
     -------
  → M ↠ N
pars-betas (M ∎) = M ∎
pars-betas (M ⇛⟨ p ⟩ ps) = ↠-trans (par-betas p) (pars-betas ps)
```

This shows that, as stated in the proof outline, the transitive reflexive closure of parallel reduction is indeed equivalent to β^*-reduction. The crucial property is now showing that parallel reduction is confluent, and that its confluence therefore implies that of β-reduction.

5.5 Diamond lemma for parallel reduction

Continuing the proof, it is shown in [WK19] that it is possible to directly prove the diamond lemma for parallel reduction by induction on both reductions:

```
par-diamond : ∀ {n} {M N N′ : Term n}
  →         M ⇉ N → M  ⇉ N′
     --------------------------
  → ∃[ L ] (N ⇉ L × N′ ⇉ L)
par-diamond (⇉-c {x = x}) ⇉-c = # x , ⇉-c , ⇉-c
par-diamond (⇉-ƛ p1) (⇉-ƛ p2)
    with par-diamond p1 p2
... | L′ , p3 , p4 =
      ƛ L′ , ⇉-ƛ p3 , ⇉-ƛ p4
par-diamond (⇉-ξ p1 p3) (⇉-ξ p2 p4)
    with par-diamond p1 p2
... | L₃ , p5 , p6
      with par-diamond p3 p4
...   | M₃ , p7 , p8 =
        L₃ · M₃ , ⇉-ξ p5 p7 , ⇉-ξ p6 p8
par-diamond (⇉-ξ (⇉-ƛ p1) p3) (⇉-β p2 p4)
    with par-diamond p1 p2
... | N₃ , p5 , p6
      with par-diamond p3 p4
...   | M₃ , p7 , p8 =
        N₃ [ M₃ ] , ⇉-β p5 p7 , sub-par p6 p8
par-diamond (⇉-β p1 p3) (⇉-ξ (⇉-ƛ p2) p4)
    with par-diamond p1 p2
... | N₃ , p5 , p6
      with par-diamond p3 p4
...   | M₃ , p7 , p8 =
        N₃ [ M₃ ] , sub-par p5  p7 , ⇉-β p6 p8
par-diamond (⇉-β p1 p3) (⇉-β p2 p4)
    with par-diamond p1 p2
... | N₃ , p5 , p6
      with par-diamond p3 p4
...   | M₃ , p7 , p8 =
        N₃ [ M₃ ] , sub-par p5 p7 , sub-par p6 p8
```

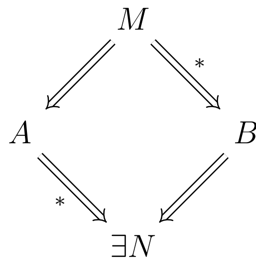
Note that in each of the cases the only theorem required is precisely the property that parallel reduction respects substitution, formalized in `sub-par`. This proof is quite articulated, and we present a small improvement for this lemma by introducing the Takahashi translation method in Chapter 6. The original authors also refer to this technique in [WK19, Confluence, Notes], but state that they chose not to adopt it because they felt that the proof was simple enough based solely on parallel reduction.

5.6 Strip lemma

Having presented the diamond lemma for parallel reduction, we now need to formally show that the same property holds for its transitive reflexive closure. This theorem, however, requires an intermediate result known as the strip lemma, which we can state as follows:

Lemma 5.6.1 (Strip lemma). Given a term M and two reductions $M \rightrightarrows A$ and $M \rightrightarrows^* B$, then there exists a term N such that $A \rightrightarrows^* N$and $B \rightrightarrows N$.

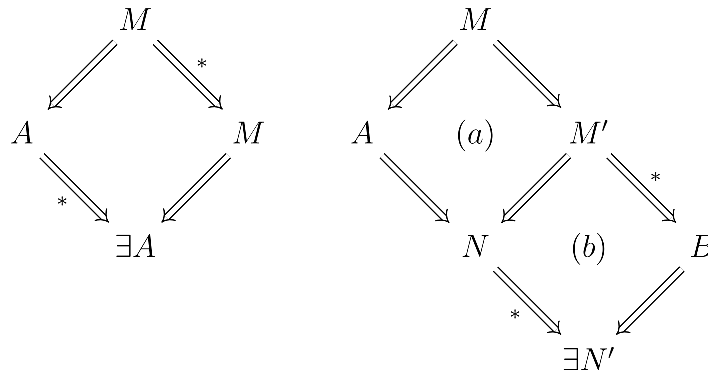
Intuitively, the strip lemma firstly converts parallel reduction into its transitive reflexive closure on one axis. We can denote this property as follows, indicating parallel reductions with double arrows:



The proof is by induction on the structure of the multiple-steps parallel reduction:

```
strip : ∀ {n} {M A B : Term n}
  →         M ⇉  A → M ⇉* B
    ---------------------------
  → ∃[ N ] (A ⇉* N × B ⇉  N)
strip {A = A} M⇉A (M ∎) = A , (A ∎) , M⇉A
strip {A = A} M⇉A (M ⇉⟨ M⇉M′ ⟩ M′⇉*B)
    with par-diamond M⇉A M⇉M′
... | N , A⇉N , M′⇉N
      with strip M′⇉N M′⇉*B
...   | N′ , N⇉*N′                 , B⇉N′ =
        N′ , (A ⇉⟨ A⇉N ⟩ N⇉*N′) , B⇉N′
```

We can also show how the proof works through the use of commuting diagrams. The base case is shown on the left, where the multiple-steps parallel reduction simply does no reductions; the inductive case is shown on the right, where we suppose that the reduction $M \rightrightarrows^* B$ has the form $M \rightrightarrows M' \rightrightarrows^* B$:



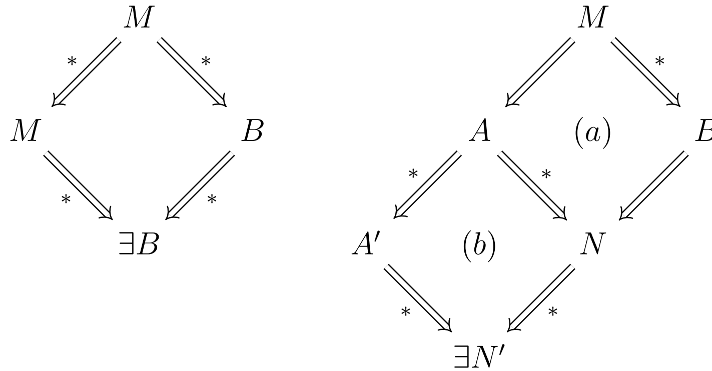
In the base case, M parallel reduces to itself so A is the unifying term. The inductive step is also easily proved by having (a) commute due to the diamond lemma for parallel reduction, and (b) by inductive hypothesis.

5.7 Confluence of parallel reduction

The confluence of parallel reduction follows a very similar proof:

```
par-confluence : ∀ {n} {M A B : Term n}
  →          M ⇉* A → M ⇉* B
     --------------------------
  → ∃[ N ] (A ⇉* N × B ⇉* N)
par-confluence {B = B} (M ∎) M⇉*B = B , M⇉*B , (B ∎)
par-confluence {B = B} (M ⇉⟨ M⇉A ⟩ A⇉*A′) M⇉*B
    with strip M⇉A M⇉*B
... | N , A⇉*N , B⇉N
      with par-confluence A⇉*A′ A⇉*N
...     | N′ , A′⇉*N′ , N⇉*N′ =
          N′ , A′⇉*N′ , (B ⇉⟨ B⇉N ⟩ N⇉*N′)
```

We can again graphically visualize it as follows:



Similarly as with the strip lemma, in the base case M parallel reduces to itself, so B is the unifying term. The inductive step is shown by having (a) commute with the strip lemma, and (b) by inductive hypothesis. Note how the orientation of the rectangle (i.e.: the reduction that we use induction on) must be the opposite of that used by the strip lemma so that we can apply it in the inductive case.

5.8 Confluence of β-reduction

Finally, we can prove that β-reduction is confluent through the confluence of parallel reduction, using the following sequence of steps: as input we have two β^*-reductions, and we can immediately convert them to $\rightrightarrows^*$ reductions; then, we apply the confluence of parallel reduction to obtain a term N and two more reductions $A \rightrightarrows^* N$ and $B \rightrightarrows^* N$; finally, we convert these two reductions back to β^*-reductions. The conversions between the reductions are handled by the previous theorems `betas-pars` and `pars-betas` stated in Section 5.4.
The proof is as follows:

```
confluence : ∀ {n} {M A B : Term n}
  →          M ↠ A → M ↠ B
     -------------------------
  → ∃[ N ] (A ↠ N × B ↠ N)
confluence M↠A M↠B
    with par-confluence (betas-pars M↠A) (betas-pars M↠B)
... | N , A⇉*N , B⇉*N =
      N , pars-betas A⇉*N , pars-betas B⇉*N
```

This concludes the Tait/Martin Löf proof as formalized by Wadler et al. [WK19].

Chapter 6

Takahashi translation

In this chapter we introduce the concept of Takahashi translation, first defined by Masako Takahashi in [Tak95]. We then show its use in simplifying the previously presented Tait and Martin-Löf confluence proof for β-reduction, briefly developing on the formalization by Wadler et al. [WK19].

6.1 Definition

Definition 6.1.1 (Takahashi translation). The *Takahashi translation* or *full-superdevelopment* of a λ-term M is written as M^*, and in the context of β-reduction it is defined as follows by always using the first applicable rule:

```
infix 8 _*

_* : ∀ {n} → Term n → Term n
(# x) *         = # x
(ƛ M) *         = ƛ M *
((ƛ M) · N) * = M * [ N * ]
(L · N) *       = L * · N *
```

Intuitively, the Takahashi translation of a term M "marks" all the existing redexes of a term and β-reduces them, without further reducing the result of substitutions except for the already marked ones. In a way, the reductions that this operator applies are simultaneous. As it is however noted in [KMY14], the reduction strictly speaking considers contractions in an innermost-first order.

This mechanism is what gives it the name of *full-superdevelopment* function, since it mirrors the concept of developments as presented in [Bar85], along with the related finiteness of developments theorem.
Note that the Takahashi translation of a term does not necessarily correspond to its normal form, since the reduction is "shallow" in the same sense that parallel reduction is (i.e.: the new redexes that have not been "marked" will still not be reduced by the translation). For this reason, Takahashi translation can be also interpreted as a parallel reduction which always chooses to contract redexes whenever possible.

6.1.1 Pattern overloading in Agda

This definition of Takahashi translation, as it is formalized in Agda, presents a peculiar characteristic. When the term is a β-redex, the third case matches and β-reduction is performed. Otherwise, the last case is applied and Takahashi translation is recursively applied on the subterms. Here, the pattern `L` can either represent a variable or an application, and is thus overloaded with two different cases that nevertheless follow the same definition. The case where the term on the left is a λ-abstraction must come before this last one, because it would otherwise be captured[1] by the more general pattern `L`. And this important remark about the application order of rules has also been stated in [KMY14], and Agda always tries to use the first applicable definition. As it will be clear in proofs using Takahashi translation, this unfortunately leads to unnecessary case expansion and the duplication of proof cases which could be handled by a single case. This is because Agda cannot understand that it can apply the last definition if the β-redex case has already been treated; this in turn requires a full case expansion of all three possible cases, so that the appropriate definition can be applied. This recurring difficulty appears in all those instances where Takahashi translation is inductively applied on terms, and leads to some proofs having identical cases.
A similar definition for Takahashi translation can be found in the Church-Rosser proof of [CST17], where they chose to show all the cases explicitly. Unfortunately, this still does not solve the technical problems deriving from this pattern overloading, and parts of the proof still need to be be repeated for the two cases.

[1] These instances of pattern shadowing and overloading are shown in Agda by highlighting the affected case with a light or dark gray background, respectively.

6.2 Revisiting the confluence of parallel reduction

The fundamental theorem about parallel reduction that Takahashi translation enables us to prove is the following property, which corresponds to Theorem 5 in the original paper by [Tak95]. The proof is by induction on the structure of the reduction:

```
par-triangle : ∀ {n} {M N : Term n}
  → M ⇛ N
    --------
  → N ⇛ M *
par-triangle {M = # x} ⇛-c = ⇛-c
par-triangle {M = ƛ M} (⇛-ƛ M⇛N) = ⇛-ƛ (par-triangle M⇛N)
par-triangle {M = (ƛ M) · N} (⇛-β M⇛M′ N⇛N′) =
  sub-par (par-triangle M⇛M′) (par-triangle N⇛N′)
par-triangle {M = # _   · N} (⇛-ξ M⇛M′ N⇛N′) =
  ⇛-ξ (par-triangle M⇛M′) (par-triangle N⇛N′)
par-triangle {M = _ · _ · N} (⇛-ξ M⇛M′ N⇛N′) =
  ⇛-ξ (par-triangle M⇛M′) (par-triangle N⇛N′)
par-triangle {M = (ƛ _) · N} (⇛-ξ (⇛-ƛ M⇛M′) N⇛N′) =
  ⇛-β (par-triangle M⇛M′) (par-triangle N⇛N′)
```

The idea behind this lemma is that Takahashi translation enables us to complete the redexes that any given parallel reduction does not consider. This allows to provide a final term that directly relates to the starting one, in a way "reversing" the direction of the arrow. This property has also been called the *triangle property* in [Ter03], due to the fact that it can be expressed as follows:

M

N

M^*

This also gives Takahashi translation the name of *maximal parallel reduction function*, as stated in [STS15]. The term M^* is maximal in the sense that it is always possible to extend an arbitrary parallel reduction $M \Rrightarrow N$ into $N \Rrightarrow M^*$ by contracting the remaining redexes.

6.2.1 Diamond lemma for parallel reduction

The triangle property is what allows us to even more directly prove the diamond property for parallel reduction:

```
par-diamond : ∀ {n} {M A B : Term n}
  →         M ⇉ A → M ⇉ B
     -------------------------
  → ∃[ N ] (A ⇉ N × B ⇉ N)
par-diamond {M = M} M⇉A M⇉B =
  M * , par-triangle M⇉A , par-triangle M⇉B
```

In this proof, we explicitly provide the single term that the two reductions converge to, namely M^*, without having to consider any case in terms of reductions. This method effectively constitutes a sharpening of the result presented by Tait/Martin-Löf, since not only we can prove the existence of a confluent term, but we can also explicitly construct the term itself in a way that does not depend on the specific reduction steps applied. The proof can then proceed as usual using the previously shown theorems to obtain the confluence of β-reduction.

6.3 Comparison with the previous proof

The use of Takahashi translation greatly simplifies the proof of the diamond lemma for parallel reduction, and constitutes an useful concept that will be further expanded upon in Chapter 7 and Chapter 8.
In the same way as with the direct case-splitting version `par-diamond` provided by [WK19], the only fundamental property about substitutions and β-reduction required here is `sub-par`, the notion that parallel reduction respects substitution.
The main conceptual improvement of this proof is the fact that the concretely provided term unifying the two reductions does not depend on the terms A and B nor on the reductions, and remains the same in all cases. This also avoids the need for induction in the actual confluence proof, which simply provides the confluent term and the two reductions using the triangle lemma.

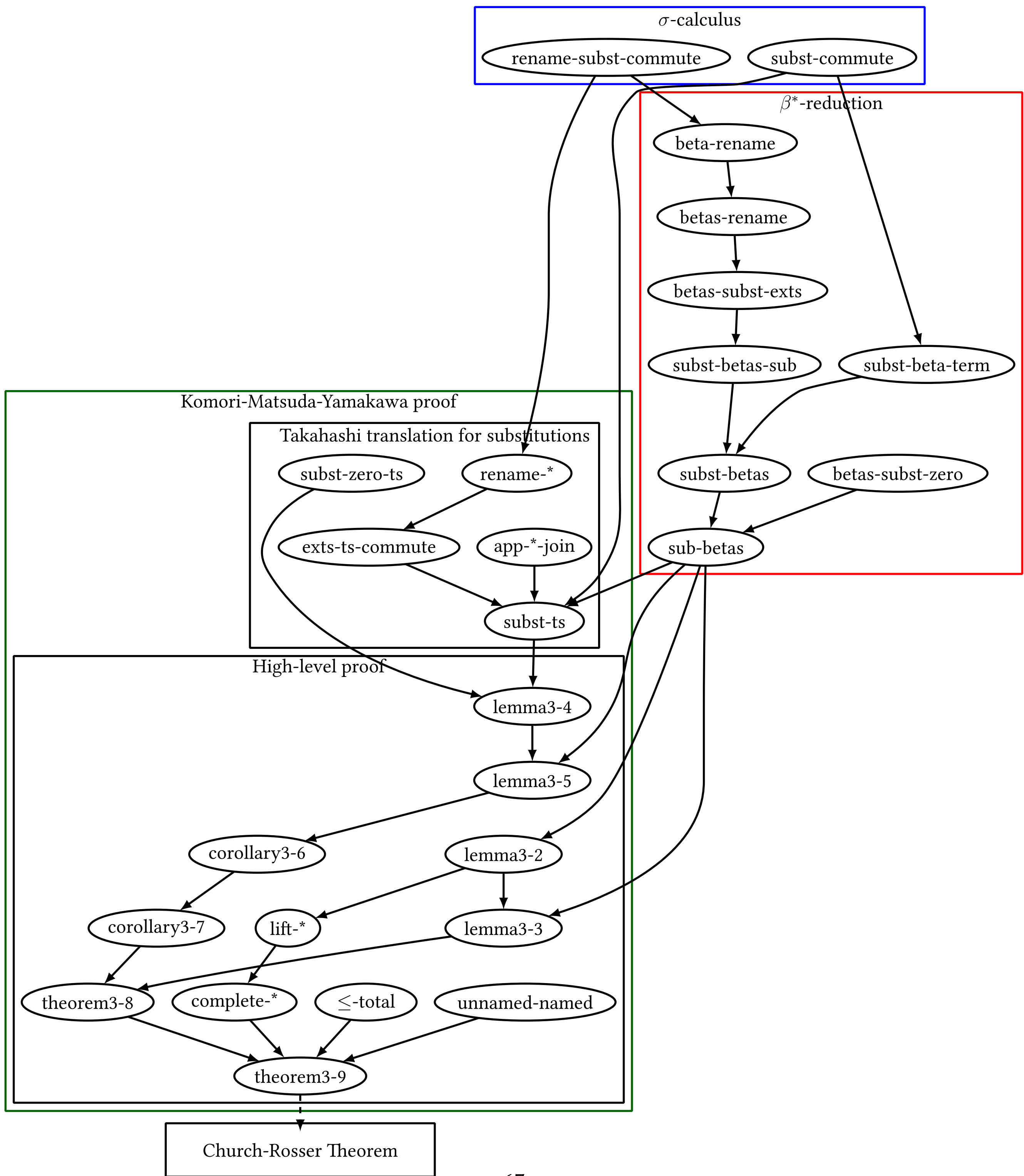
Chapter 7

The Komori-Matsuda-Yamakawa proof

In this chapter we present the novel approach exposed by Yuichi Komori, Naosuke Matsuda, and Fumika Yamakawa in [KMY14], along with our complete formalization for the confluence of β^*-reduction. Alongside the classic proof, we explain in-depth the Agda implementation and how it compares to the original development. The names of the theorems presented in our formalization directly correspond with the names used in the paper.

7.1 Proof overview

We present here the general outline of the theorems formalized in this proof, along with their dependencies:


σ-calculus
rename-subst-commute
subst-commute
β*-reduction
beta-rename
betas-rename
betas-subst-exts
subst-betas-sub
subst-beta-term
subst-betas
betas-subst-zero
sub-betas
Komori-Matsuda-Yamakawa proof
Takahashi translation for substitutions
subst-zero-ts
rename-*
exts-ts-commute
app-*-join
subst-ts
High-level proof
lemma3-4
lemma3-5
corollary3-6
lemma3-2
corollary3-7
lift-*
lemma3-3
theorem3-8
complete-*
≤-total
unnamed-named
theorem3-9
Church-Rosser Theorem

7.2 Main concepts

One important advantage of this method is the fact that it does not require the definition of parallel reduction, and only employs Takahashi translation and its iterated version. As stated by the authors, this approach can be useful to prove confluence of those systems where parallel reduction is harder to treat.
Firstly, we inductively define the following straightforward concepts of iterated Takahashi translation and iterated β-reduction:

```
infix 8 _*⁽_⁾

_*⁽_⁾ : ∀ {n} → Term n → ℕ → Term n
M *⁽ zero ⁾  = M
M *⁽ suc k ⁾ = (M *) *⁽ k ⁾

data _↠⁽_⁾_ : ∀ {n} → Term n → ℕ → Term n → Set where

  _∎ : ∀ {n} (M : Term n)
       ---------------
     → M ↠⁽ zero ⁾ M

  _→⁽⁾⟨_⟩_ : ∀ {n} {N L : Term n} {k : ℕ} (M : Term n)
     → M → L
     → L ↠⁽ k ⁾ N
       ----------------
     → M ↠⁽ suc k ⁾ N
```

In term of notation, we postpone as apex the number of times that the iteration occurs. We have the following basic lemma to convert the iteration naturally induced by β^*-reduction into an explicitly quantified one. This lemma has been left implicit in the original paper, but the proof is trivial by induction on the β^*-reduction and clearly does not require any previous property:

```
unnamed-named : ∀ {n} {M N : Term n}
  →          M ↠      N
    --------------------
  → ∃[ m ] (M ↠⁽ m ⁾ N)
unnamed-named (M ∎) = zero , (M ∎)
unnamed-named (M →⟨ M→L ⟩ L↠N) with unnamed-named L↠N
... | m' , L↠ᵐ'N = suc m' , (M →⁽⁾⟨ M→L ⟩ L↠ᵐ'N)
```

7.3 Fundamental theorems for confluence

With these definitions, we can presented the following two central theorems of the proof by Komori et al. [KMY14], Lemma 3.2 and Theorem 3.8:

Lemma 3.2. Every λ-term β^*-reduces to its Takahashi translation:

$$M \to_\beta^* M^*$$

Intuitively, this property comes from the fact that β^*-reduction can manually contract the redexes treated by Takahashi translation. This first theorem can be proven fairly easily by induction[1] on the structure of the term M, using in the crucial case the substitutivity of β^*-reduction `sub-betas` that we previously proved in Section 4.2:

```
lemma3-2 : ∀ {n} {M : Term n}
  → M ↠ M *
lemma3-2 {M = # x}         = # x ∎
lemma3-2 {M = ƛ _}         = ↠-cong-ƛ lemma3-2
lemma3-2 {M = # _ · _}     = ↠-congᵣ  lemma3-2
lemma3-2 {M = _ · _ · _} = ↠-cong   lemma3-2 lemma3-2
lemma3-2 {M = (ƛ M) · N} =
  (ƛ M) · N →⟨ →-β ⟩ sub-betas {M = M} lemma3-2 lemma3-2
```

Note that, in the case where M is an application and has a variable on the left side, another valid proof could have been `↠-cong lemma3-2 lemma3-2`, mirroring the same proof of the case below. This further case expansion on the left is precisely the additional case splitting necessary because of Takahashi translation, as previously explained in Section 6.1.1. However, since `# x` is equal to itself under Takahashi translation,[2] only the right side requires the application of the inductive hypothesis.

[1] Another possible proof could have reused the triangle property `theorem5` shown in the previous chapter, by using transitivity of parallel reduction and then converting it into β^*-reduction. However, the proof presented here is considerably simpler and more direct.

[2] We could have also used the reflexive case of β^*-reduction, equivalently.

Two important corollaries of Lemma 3.2 not explicitly proven in the original proof but necessary for our formal development are the following properties:

```
lift-* : ∀ {n} (M : Term n) (m : ℕ)
  → M ↠ M *⁽ m ⁾
lift-* M zero    = M ∎
lift-* M (suc m) = ↠-trans lemma3-2 (lift-* (M *) m)

complete-* : ∀ {k} (M : Term k) {n m : ℕ}
  → n ≤ m
    ------------------
  → M *⁽ n ⁾ ↠ M *⁽ m ⁾
complete-* M {m = m} z≤n = lift-* M m
complete-* M (s≤s k) = complete-* (M *) k
```

These last proofs follow by the repeated application of Lemma 3.2, and they allow us to "lift" as many times as necessary a term M^{*n} into a reduced one M^{*m} provided that $n \leq m$.
The next fundamental theorem presented by [KMY14] is Theorem 3.8, which turns out to be the backbone of the proof:

Theorem 3.8. For all λ-terms M and N:

$$M \to_\beta^n N \Longrightarrow N \to_\beta^* M^{*n}$$

Proving it requires a series of intermediate lemmas, and we will present the complete proof in Section 7.5. Note the similarity of this theorem, which effectively "reverses" the direction of the reduction, with the triangle property `par-triangle` used in the confluence proof for parallel reduction with Takahashi translation. This theorem expands on this previously proven property by directly considering the case of β^*-reduction and further quantifying the results. The intuition behind this important theorem is that n steps of β^*-reduction always perform less (or equal) reductions than an equally iterated Takahashi translation. The remaining reductions necessary to unify the two terms are simply performed by repeating the unquantified β^*-reduction on the right as many times as necessary for N to catch up with the Takahashi-translated term.

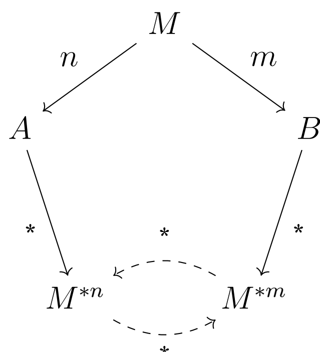
7.4 Confluence of β-reduction

We immediately present here the main theorem of the paper, the confluence of β-reduction, in order to show how the previous lemmas together constitute the core of the proof:

```
theorem3-9 : ∀ {n} {M A B : Term n}
  →         M ↠ A → M ↠ B
     -------------------------
  → ∃[ N ] (A ↠ N × B ↠ N)
theorem3-9 {M = M} M↠A M↠B =
    let n , M↠ⁿA = unnamed-named M↠A
        m , M↠ᵐB = unnamed-named M↠B
        A↠M*ⁿ   = theorem3-8 M↠ⁿA
        B↠M*ᵐ  = theorem3-8 M↠ᵐB
     in [ (λ n≤m →
             let M*ⁿ↠M*ᵐ : M *⁽ n ⁾ ↠ M *⁽ m ⁾
                 M*ⁿ↠M*ᵐ = complete-* n≤m
              in M *⁽ m ⁾ , ↠-trans A↠M*ⁿ M*ⁿ↠M*ᵐ , B↠M*ᵐ)
        , (λ m≤n →
             let M*ᵐ↠M*ⁿ : M *⁽ m ⁾ ↠ M *⁽ n ⁾
                 M*ᵐ↠M*ⁿ = complete-* m≤n
              in M *⁽ n ⁾ , A↠M*ⁿ , ↠-trans B↠M*ᵐ M*ᵐ↠M*ⁿ)
        ] (≤-total n m)
```

The idea is as follows: firstly, we transform the implicit reductions into explicit ones using `unnamed-named`. Then, we apply `theorem3-8` to "move" the number of β-reductions into the same number of Takahashi translations and reverse the term order in each reduction. We proceed by cases on whether $n \leq m$ or $m \leq n$ by using the fact that $\leq$ is a total relation, and consider the two possibilities using the disjunction decomposition `[_,_]`. Both of these definitions have been outlined in Chapter 2 and can also be found in the Agda standard library. Let us suppose that $n \leq m$: then, by using `complete-*` we can extend the shorter reduction $A \to_\beta^* M^{*n}$ with the remaining iterations of Takahashi translation in order to reach the common term M^{*m}, thus completing the proof. The other case is symmetric by completing the other reduction. The two cases jointly show that the unifying term being provided is always $M^{*\max\{n,m\}}$.

The proof can be shown with the following diagram, where the last dashed reduction depends on which of the two final terms is the smallest one (i.e.: which of the two reductions needs to be completed):



This result even further improves on the advancements presented by Takahashi translation in the case of parallel reduction. Not only is the definition of parallel reduction here unneeded, but the unifying term is specified even more precisely than the one previously given in the case of parallel reduction. The two final reductions still do not rely on the specific form of reduction steps previously applied, and jointly specify the confluent term.

7.5 Central theorems

In order to prove the Theorem 3.8 previously presented, we state the following two crucial theorems:

7.5.1 Lemma 3.3

Lemma 3.3. For all λ-terms M and N:

$$M \to_\beta N \Longrightarrow N \to_\beta^* M^*$$

The proof is quite articulated, and proceeds by induction on the β-reduction. However, since the proof uses nothing more than the previously proven `lemma3-2` and the substitutivity of β^*-reduction `sub-betas`, we can directly present it as follows:

```
lemma3-3 : ∀ {n} {M N : Term n}
  → M → N
    --------
  → N ↠ M *
lemma3-3 {M = # _} ()
lemma3-3 {M = ƛ M}           (→-ƛ M→M′) = ↠-cong-ƛ (lemma3-3 M→M′)
lemma3-3 {M = # _     · N} (→-ξr N→N′) = ↠-congr (lemma3-3 N→N′)
lemma3-3 {M = _ · _   · N} (→-ξr N→N′) = ↠-cong lemma3-2 (lemma3-3 N→N′)
lemma3-3 {M = M₁ · M₂ · _} (→-ξl M→M′) = ↠-cong (lemma3-3 M↠M′) lemma3-2
lemma3-3 {M = (ƛ M)   · N}  →-β = sub-betas {M = M} lemma3-2 lemma3-2
lemma3-3 {M = (ƛ M)   · N} (→-ξr {N′ = N′} N→N′)
  = (ƛ M ) · N′ →⟨ →-β ⟩ sub-betas {M = M} lemma3-2 (lemma3-3 N→N′)
lemma3-3 {M = (ƛ M)   · N} (→-ξl (→-ƛ {M′ = M′} M→M′))
  = (ƛ M′) · N  →⟨ →-β ⟩ sub-betas {N = N} (lemma3-3 M→M′) lemma3-2
```

Intuitively, this lemma can be interpreted as the base case of Theorem 3.8 where $n = 1$ and a single Takahashi translation is performed. Alternatively, it can be seen as a one-step version of the triangle lemma. Note how the input reduction can only be a single reduction step; a β^*-reduction might apply too many contractions for a single Takahashi translation, which only operates shallowly as we previously discussed in Chapter 6. The use of the absurd pattern `()` in the case where M is a variable is justified by the fact that no β-reduction step can be provided, so no constructor is available for the input reduction.

7.5.2 Lemma 3.5

The second central theorem is considerably harder to prove, and for now we only provide the definition:

Lemma 3.5. For all λ-terms M and N:

$$M \to_\beta N \Longrightarrow M^* \to_\beta^* N^*$$

These two properties are the core lemmas that enable us to show Theorem 3.8. As we will show in Chapter 8, these same results will be later used to introduce another confluence proof, which will be directly obtained through the use of the so-called Z-property.
Having stated these two fundamental properties, we can now conclude the confluence proof of Theorem 3.8. We proceed through the following series of straightforward intermediate corollaries.

7.5.3 Proving Theorem 3.8

As a direct consequence of Lemma 3.5, we begin by stating that Takahashi translation is monotonic with respect to multi-step β^*-reduction:

```
corollary3-6 : ∀ {n} {M N : Term n}
  → M   ↠ N
    ----------
  → M * ↠ N *
corollary3-6 (M ∎) = M * ∎
corollary3-6 (M →⟨ M→L ⟩ L↠N) =
  ↠-trans (lemma3-5 M→L) (corollary3-6 L↠N)
```

The following result is obtained through the repeated application of the first:

```
corollary3-7 : ∀ {n} {M N : Term n} (m : ℕ)
  → M        ↠ N
    ------------------
  → M *⁽ m ⁾ ↠ N *⁽ m ⁾
corollary3-7 zero    M↠N = M↠N
corollary3-7 (suc m) M↠N = corollary3-7 m (corollary3-6 M↠N)
```

Finally, we use this theorem along with `lemma3-3` previously shown in Section 7.5.1 to conclude the proof.

```
theorem3-8 : ∀ {n} {M N : Term n} {m : ℕ}
  → M ↠⁽ m ⁾ N
    ------------
  → N ↠ M *⁽ m ⁾
theorem3-8 {m = zero}  (M ∎) = M ∎
theorem3-8 {m = suc m} (M →⁽⁾⟨ M→L ⟩ L↠ᵐN) =
  ↠-trans (theorem3-8 L↠ᵐN) (corollary3-7 m (lemma3-3 M→L))
```

7.5.4 Proving Lemma 3.5

The last theorem relating β-reduction and β^*-reduction is now Lemma 3.5, which proceeds by a refined case analysis on the relation and the term structure in the same way as with `lemma3-3`:

```
lemma3-5 : ∀ {n} {M N : Term n}
  → M   → N
    ----------
  → M * ↠ N *
lemma3-5 {M = # x} ()
lemma3-5 {M = ƛ M}          (→-ƛ  M→M′) = ↠-cong-ƛ (lemma3-5 M→M′)
lemma3-5 {M = # _      · N} (→-ξᵣ M₂→M′) = ↠-congᵣ (lemma3-5 M₂→M′)
lemma3-5 {M = (ƛ M)    · N}  →-β = lemma3-4 M N
lemma3-5 {M = (ƛ M)    · N} (→-ξₗ (→-ƛ M→M′)) =
  sub-betas (lemma3-5 M→M′) (N * ∎)
lemma3-5 {M = (ƛ M)    · N} (→-ξᵣ N→N′) =
  sub-betas (M * ∎) (lemma3-5 N→N′)
lemma3-5 {M = M₁ · M₂ · _} (→-ξᵣ N→N′) =
  ↠-congᵣ (lemma3-5 N→N′)
lemma3-5 {M = M₁ · M₂ · _} (→-ξₗ {M′ = # x}        M₁M₂→#x) =
  ↠-congₗ (lemma3-5 M₁M₂→#x)
lemma3-5 {M = M₁ · M₂ · _} (→-ξₗ {M′ = M′₁ · M′₂} M₁M₂→M′₁M′₂) =
  ↠-congₗ (lemma3-5 M₁M₂→M′₁M′₂)
lemma3-5 {M = M₁ · M₂ · N} (→-ξₗ {M′ = ƛ M′}       M₁M₂→ƛM′) =
  ↠-trans (↠-congₗ (lemma3-5 M₁M₂→ƛM′))
          ((ƛ M′ *) · N →⟨ →-β ⟩ subst (subst-zero (N *)) (M′ *) ∎)
```

In the last three cases, the further splitting of `M′` into its components is necessary because the goal would be in the following form (the required type is indicated in the comment):

```
lemma3-5 {M = M₁ · M₂ · _} (→-ξₗ r) =
  {! (M₁ · M₂) * · N * ↠ (M′ · N) * !}
```

Unfortunately, as we previously discussed, Agda cannot apply any case of Takahashi translation since it cannot verify whether `M′` is a λ-abstraction or not. We therefore need to make the argument `M′` of the →-ξ_l constructor explicit and then consider each possibility. The last case where `M′` is a λ-abstraction is the most articulated one, since Takahashi translation applies the β-contraction. Here, we have as goal:

```
lemma3-5 {M = M₁ · M₂ · N} (→-ξₗ {M′ = ƛ M′} M₁M₂→ƛM′) =
  {! (M₁ · M₂) * · N * ↠ (M′ *) [ N * ] !}
```

Having as inductive hypothesis `(M₁ · M₂)*` ↠ `ƛ M′ *`, we can append the term `N*` on the right using `↠-cong`$_l$, and then apply one β-reduction step to obtain `(M′ *) [ N * ]` as final term on the right side. This last passage is described in the last line of the proof, and the rule application can be automatically completed by Agda since it only requires known type constructors. Even though this makes this last passage slightly harder to read, we again chose to leave the automated form in order to highlight this possibility. The other subcases need to be duplicated because of the Takahashi translation overloading.
The remaining cases of this theorem worth considering are the variable and the β-reduction ones, since the rest can be solved by simply applying the inductive hypothesis through congruence or `sub-betas`. The case where M is a variable is again impossible in a similar way to `lemma3-3`, since no reduction can be provided. The case where β-reduction applies is the crucial one, and is directly refactored into the separate Lemma 3.4 which we now introduce.

7.6 Lemma 3.4

This theorem is the last high-level property presented in the proof that we need to show, and it relates Takahashi translation and β^*-reduction with substitutions:

Lemma 3.4. For all λ-terms M and N:

$$M^*[N^*] \to_\beta^* M[N]^*$$

Intuitively, this theorem is not trivial. In the original paper this statement is expressed in the unexpanded (but easier to understand) form of $((\lambda M)N)^* \to_\beta^* (M[N])^*$, and the proof simply states that the following sub-properties can be verified by induction on the structure of the term M:

1. *if N is not a λ-abstraction then* $M^*[N^*] \to_\beta^* (M[N])^*$
2. $M^*[\lambda N_1^*] \to_\beta^* (M[\lambda N_1])^*$

However, proving these facts turns out to be extremely hard in our formal setting. Constructing the proof of Lemma 3.4 constituted the major technical difficulty in conceptualizing and then completing the proof structure described in this thesis.

Tentative proof by induction

A first naïve approach to prove this crucial theorem would be to directly proceed by induction on the structure of M, as it is also specified in the paper. This intuition is justified by the fact that M is precisely the term on which both Takahashi translation and the substitution are applied:

```
lemma3-4-ind : ∀ {n} (M : Term (suc n)) (N : Term n)
  → M * [ N * ] ↠ (M [ N ]) *
```

or, with expanded definitions:

```
  → subst (subst-zero (N *)) (M *) ↠ (subst (subst-zero N) M) *
```

Following the induction, the case where M is a variable can be easily handled by checking whether the index is zero, to see if the substitution is performed:

```
lemma3-4-ind (# zero) N = N * ∎
lemma3-4-ind (# suc x) N = # x ∎
```

However, the case where M is a λ-abstraction requires us to prove a different but fundamentally related version of the theorem, namely, the case where the term being substituted is in fact on the index 1. This case necessarily starts with an application of `↠-cong-ƛ` in order to elide the λ-abstractions:

```
lemma3-4-ind (ƛ M) N = ↠-cong-ƛ (lemma3-4-ind-exts M N)
```

The necessary theorem to solve this case indeed turns out to be the following:

```
  →  subst (exts (subst-zero (N *))) (M *)
  ↠ (subst (exts (subst-zero N)) M) *
```

or, in a more readable form:

```
lemma3-4-ind-exts : ∀ {n} (M : Term (suc (suc n))) (N : Term n)
  → M * ⟦ N * ⟧ ↠ (M ⟦ N ⟧) *
```

The fact that this lemma refers to 1-indexed substitution can be expressed with the special parentheses introduced in Section 3.4.2. This immediately suggests us that a generalization of our 0-indexed theorem is necessary, so that we can consider the theorem for all extensions of `subst-zero` and possibly find an even more general form of this property. This similarly mirrors the proofs of the substitutivity of β-reduction `sub-betas` and parallel reduction `par-betas`, where without generalizations one would apparently need to recursively prove different versions of the same theorem.
Notice that in the crucial case where Takahashi translation performs β-reduction we could actually apply (along with the inductive hypothesis and our "extended" version of the theorem) another property that is similarly specialized for 0-indexed substitutions, namely the `substitution-lemma` that we presented in Section 3.4.2 as a corollary of the more general `subst-commute`:

```
lemma3-4-ind ((ƛ M₁) · M₂) N
  rewrite substitution-lemma {M = M₁ *}{N = M₂ *}{L = N *} =
    sub-betas (lemma3-4-ind-exts M₁ N) (lemma3-4-ind M₂ N)
```

This also suggests us, in retrospect, that the crucial theorem resolving this case in the generalized setting is precisely going to be `subst-commute`.

7.7 Takahashi translation for substitutions

The generalization that we need to find cannot however be related to a reduction, as it happened when proving the substitutivity of β-reduction `sub-betas` and parallel reduction `par-betas`; here we simply have no reduction to generalize against in the arguments. Instead, the novel concept that we introduce is that of Takahashi translation of an entire substitution σ:

Definition 7.7.1 (Takahashi translation for substitutions). Given a substitution σ, the *Takahashi translation of* σ is defined as a substitution σ^* such that for all x we have:

$$\sigma^* \ x = (\sigma \ x)^*$$

In other words, σ^* is the pointwise Takahashi translation for σ. We will follow the same notation of using a small s in the Agda code, as in our definition of pointwise β^*-reduction, to denote that the operation is extended to substitutions. This technical definition is the key element in completing the proof. The original theorem can now be expressed in a general form where the Takahashi translation of a generic substitution σ takes the place of the 0-indexed `subst-zero (N *)`.

7.7.1 Generalized Lemma 3.4

We can now state the generalization of Lemma 3.4 as follows:

```
subst-ts : ∀ {n m} (σ : Subst n m) (M : Term n)
  → subst (σ *ˢ) (M *) ↠ (subst σ M) *
subst-ts σ (# x) = σ x * ∎
subst-ts σ (ƛ M)
  rewrite exts-ts-commute σ =
    ↠-cong-ƛ (subst-ts (exts σ) M)
subst-ts σ (# x · N) =
  ↠-trans (↠-congᵣ (subst-ts σ N))
          (app-*-join (σ x) (subst σ N))
subst-ts σ (M₁ · M₂ · N) =
  ↠-cong (subst-ts σ (M₁ · M₂)) (subst-ts σ N)
subst-ts σ ((ƛ M) · N)
  rewrite sym (subst-commute {N = M *}{M = N *}{σ = σ *ˢ})
        | exts-ts-commute σ
        = sub-betas (subst-ts (exts σ) M) (subst-ts σ N)
```

The proof of the theorem now does, indeed, proceed by induction on the term M as stated in the original pen-and-paper development.
We analyze each case of the proof in detail.

(Case `# x`) By definition of Takahashi translation (where variables remain unchanged) and application of `subst` on variables (where due to σ ^{*s} we apply Takahashi translation on the result of σ `x`) we have σ `x` * on the left side. On the right side we also have σ `x` by application of `subst` on variables and then σ `x` * by the outer Takahashi translation. The case is finally solved with the reflexive case of β^*-reduction.

(Case `ƛ M`) Through the application of Takahashi translation on the left and the two cases of `subst` for λ-abstractions we have as goal:

```
ƛ subst (exts (σ *ˢ)) (M *) —↠ ƛ subst (exts σ) M *
```

We can temporarily conjecture a theorem `exts-ts-commute` that allows us to commute the extension `exts` with the Takahashi translation of a given substitution, and then apply it (through `rewrite`) to get the following:

```
ƛ subst ((exts σ) *ˢ)) (M *) —↠ ƛ subst (exts σ) M *
```

But this is exactly the inductive hypothesis applied to the term `M` and with σ equal to the substitution `(exts σ)`, up to the λ-abstraction congruence.

(Case `(M₁ · M₂ · N)`) We first treat the easier of the three cases where the left term is expanded. In the case where `M` is an application `(M₁ · M₂)` we have as simplified goal:

```
    subst (σ *ˢ) ((M₁ · M₂) *) · subst (σ *ˢ) (N *)
—↠ (subst σ M₁ · subst σ M₂) * · subst σ N *
```

We can now apply the inductive hypothesis twice by using the double congruence for applications `—↠-cong`. Compared with the original goal, the simplification of the right-side term is possible due to the fact that `(M₁ · M₂)` is structurally checked not to be a λ-abstraction. This in turn allows Takahashi translation to split the outermost application in two separate subterms.

(Case `(ƛ M · N)`**)** In the crucial case of β-reduction, because of the two Takahashi translations we have as goal:

```
   subst (σ *ˢ) ((M *) [ N * ])
↠ (subst (exts σ) M *) [ subst σ N * ]
```

We can now leverage the σ-calculus properties by using (the symmetric version of) `subst-commute` in order to "distribute" the substitution with Takahashi translation over both `(M *)` and `(N *)` on the left side:

```
  (subst (exts (σ *ˢ)) (M *)) [ subst (σ *ˢ) (N *) ]
↠ (subst (exts σ) M *) [ subst σ N * ]
```

We then now use another `rewrite` to apply on the left side the equality `exts-ts-commute` we previously hypothesized:

```
  (subst ((exts σ) *ˢ) (M *)) [ subst (σ *ˢ) (N *) ]
↠ (subst (exts σ) M *) [ subst σ N * ]
```

Finally, we can apply the substitutivity of β^*-reduction with the two inductive hypotheses `(subst-ts (exts σ) M)` and `(subst-ts σ N)` to close the case.

(Case `(# x · N)`**)** Finally, we show the case where M is a variable. By applying `subst` on both sides we have as goal:

```
(σ *ˢ) x · subst (σ *ˢ) (N *) ↠ (σ x · subst σ N) *
```

On the left side, we can use the right congruence property `↠-cong`$_r$ with the inductive hypothesis, and simply apply the definition of Takahashi translation for the substitution σ on the left:

```
σ x * · subst σ N * ↠ (σ x · subst σ N) *
```

However, unlike the previous case where the left side of M is another application, we cannot automatically decompose the right side into two subterms by applying Takahashi translation (and indeed Agda in this case does not perform any simplification), because we cannot know beforehand whether σ `x` is going to be a λ-abstraction or not.

Now we need to prove what seems to be a particularly hard lemma that "joins" together the application of two Takahashi-translated terms with the translation of the entire application. We will then be able to apply this property by simply using the transitivity of β^*-reduction `↠-trans`. We temporarily hypothesize and name this theorem `app-*-join`, which allows us to finally close the proof.

7.7.2 Lemma 3.4 for applications

The theorem `app-*-join` used in the last case of the previous proof bears a considerable similarity (and presumably, difficulty) with the original Lemma 3.4 itself, where a term containing two separate Takahashi translations β^*-reduces to a single translated term. As it has been shown in the previous case analysis, this effectively constitutes the first interesting instance where Takahashi translation non-trivially treats differently the two overloaded case patterns, which should in theory behave in completely similar ways. Note that this theorem cannot even be applied in the case of M having another application on the left side, but it is strictly necessary in this case where the left side of the application is a variable. The theorem actually turns out to have an interesting proof. Since there are no substitutions involved, we can tentatively proceed by induction on the structure of the term M:

```
app-*-join : ∀ {n} (M N : Term n)
  → M * · N * ↠ (M · N) *
app-*-join (# x) N = # x · N * ∎
app-*-join (ƛ M) N =
  (ƛ M *) · N * →⟨ →-β ⟩ subst (subst-zero (N *)) (M *) ∎
app-*-join (M₁ · M₂) N = (M₁ · M₂) * · N * ∎
```

What seemed to be a particularly tricky lemma can, in fact, be completely solved by Agda automatically (except for the induction split, which always needs to be manually performed), since it requires no specific theorem except the usual constructors of β^*-reduction. Furthermore, notice how no case that further decomposes M is required, since M already appears as a left-subterm in the application on the right side. We chose to leave this proof in its automated form (especially in the second case, where the definition of 0-indexed substitution gets automatically expanded) in order to highlight this automation step.

7.7.3 Takahashi translation and renamings

Our next theorem is the commutativity between Takahashi translation and extension of substitutions that we hypothesized in order to solve the preceding cases. Since we are considering an equality between substitutions, which are just functions from de Bruijn indices to terms, we can use the `extensionality` postulate defined in Section 2.4.5. By showing that the two substitutions give the same result on every possible input, this principle allows us to conclude that the two substitutions denote the same function and are therefore interchangeable.[3]

```
exts-ts-commute : ∀ {n m} (σ : Subst n m)
  → exts (σ *ˢ) ≡ (exts σ) *ˢ
exts-ts-commute {n} σ = extensionality exts-ts-commute′
  where
    exts-ts-commute′ : (x : Fin (suc n))
      → (exts (σ *ˢ)) x ≡ ((exts σ) *ˢ) x
    exts-ts-commute′ zero    = refl
    exts-ts-commute′ (suc x) = rename-* suc (σ x)
```

In order apply `extensionality`, we define an auxiliary function that explicitly inducts on the index `x`. As per the definition of `exts`, when the index is greater than zero (that is, we effectively apply the substitution σ and then shift all the indices of the result by one) we have as goal the following statement:

```
rename suc ((σ *ˢ) x) ≡ (exts σ *ˢ) (suc x)
```

Through a manual simplification of the goal, however, we can obtain the following easier to conceptualize property, which simply states that shifting by one all variables does not essentially alter the result of Takahashi translation and in fact commutes with it:

```
rename suc ((σ x) *) ≡ rename suc (σ x) *
```

We can actually prove a more general version of this theorem, by generalizing over the renaming applied (in this case `suc`) with a generic renaming function ρ, and abstracting the term `σ x` with a generic `M`:

[3] This theorem could be expressed even more explicitly as a commutation between the two functions `exts` and `_*ˢ` using the function application operator `_∘_`. The proof would then proceed through a double application of the `extensionality` principle, one time for σ (without using any induction) and another time for the index `x`.

```
rename-* : ∀ {n m} (ρ : Rename n m) (M : Term n)
  → rename ρ (M *) ≡ rename ρ M *
rename-* ρ (# _) = refl
rename-* ρ (ƛ M) = cong ƛ_ (rename-* (ext ρ) M)
rename-* ρ (# _ · N)      = cong₂ _·_ refl (rename-* ρ N)
rename-* ρ (M₁ · M₂ · N) = cong₂ _·_ (rename-* ρ (M₁ · M₂))
                                     (rename-* ρ N)
rename-* ρ ((ƛ M) · N)
  rewrite sym (rename-subst-commute {N = M *}{M = N *}{ρ = ρ})
          | rename-* (ext ρ) M
          | rename-* ρ N
          = refl
```

This theorem is trivial in the usual cases. Note that, since we effectively entered the domain of equalities, the `cong` and `cong₂` properties refer to equality congruences, and are not to be confused with the congruences for β^*-reduction used thus far. In the last case, we have the following goal:

```
   rename ρ ((M *) [ N * ])
≡ (rename (ext ρ) M *) [ rename ρ N * ]
```

In a similar way as with `subst-ts`, we apply on the left side `rename-subst-commute` (which is just a version for renamings of the previously applied `subst-commute`) in order to distribute the renaming and obtain:

```
   rename (ext ρ) (M *) [ rename ρ (N *)) ]
≡ (rename (ext ρ) M *) [ rename ρ N * ]
```

By applying the inductive hypothesis twice with ρ and its extension (`ext ρ`) in the respective cases of N and M, we can finally conclude the proof.

7.7.4 Special case of 0-indexed substitution

Now that we have finally proven the general case, we need to relate it back to our original Lemma 3.4. In order to be able to apply `subst-ts` we need to first convert `subst-zero (N *)` into a Takahashi-translated substitution, as follows:

```
subst-zero-ts : ∀ {n} {N : Term n}
  → subst-zero (N *) ≡ (subst-zero N) *ˢ
subst-zero-ts =
  extensionality (λ { zero → refl ; (suc x) → refl })
```

Since we are again treating equality of substitutions, we need to apply the principle of `extensionality` and then case split on the index the substitutions operate on. Since the proof is direct by applying the definition of `subst-zero` in each case, we can use an anonymous case-splitting λ-expression to easily close the proof. Lemma 3.4 can now be proved by applying this rewriting rule, and then use its general version `subst-ts`:

```
lemma3-4 : ∀ {n} (M : Term (suc n)) (N : Term n)
  → M * [ N * ] ↠ (M [ N ]) *
lemma3-4 M N
  rewrite subst-zero-ts {N = N}
    = subst-ts (subst-zero N) M
```

This concludes the entire confluence proof by Komori et al. [KMY14].

7.8 Proof remarks

The formalization we presented can be broadly divided in two general subareas: a series of high-level theorems that constitute the core of the proof, and another group of infrastructural theorems that relate Takahashi translation with de Bruijn indices and parallel substitutions. This latter part is required in order to prove the crucial Lemma 3.4, which directly interacts with substitutions and the representation of λ-terms themselves. As we have seen throughout the development, the necessary formalization steps do not always exactly mirror the tactics and principles used to prove theorems in the same way that they are usually carried out in the classic pen-and-paper setting. Indeed, it is sometimes necessary to consider the fact that proofs rest on secondary yet crucial technical assumptions, starting from substitutions and the concept of extension, up until the concept of renamings and the properties related to these definitions. However, after settling these fundamental theorems, the high level proofs proceed relatively with ease, as it can be also witnessed by their length.

Chapter 8

The Z-property proof

In this chapter we present the confluence proof for β-reduction with the use of the so-called Z-property, first introduced by Dehornoy et al. [DO08]. This proof has also been formalized in Nagele et al. [NOS16] using the Isabelle/HOL theorem prover. The main method consists in proving that for any given relation the Z-property implies semi-confluence and therefore confluence. As it will be shown, the fundamental theorems necessary to establish the Z-property for the case of β-reduction have already been incidentally proved in the previous chapter, and they can be used to obtain an even more direct proof of confluence. Similarly as with the proof by Komori et al. [KMY14], these results do not require the definition of a parallel reduction relation.

8.1 Generic reflexive transitive closure

In order to formally define the Z-property, we first need to construct a general setting to treat the reflexive transitive closure of any generic relation $\rightarrow$.
We can use some definitions from the standard library (represented by `Reflexive` and `Trans`) to more clearly expose the properties of reflexivity and transitivity:

```
data Star {t r} {T : Set t} (R : Rel T r) : Rel T (t ⊔ r) where
  ε   : Reflexive (Star R)
  _▷_ : Trans (Star R) R (Star R)
```

Compared with the previous definitions of _↠_ and _⇉*_, we follow a similar mechanism of defining reflexivity and the addition of one transitivity step. However, instead of having transitivity as "adding" one reduction step at the beginning of the reduction, we consider here the additional step to be appended at the end of the reduction. This convention is also implicitly followed in [NOS16]. It turns out that defining transitivity in this way, opposite to our earlier use of β^*-relation and parallel reduction star, considerably simplifies the proof that the Z-property implies semi-confluence. The proof that semi-confluence implies confluence remains essentially the same. A similar definition for the generic transitive reflexive closure is already present in the Agda standard library under the module `Relation.Binary.Construct.Closure.ReflexiveTransitive`, but it does not follow the definition we have established here. We nevertheless adopt from it similar names and notation. Having established our generic definition, we can trivially derive the proper transitivity of any star relation:

```
_▷▷_ : ∀ {t r} {T : Set t} {R : Rel T r} {M L N : T}
  → Star R M L
  → Star R L N
  → Star R M N
ML ▷▷ ε = ML
ML ▷▷ (LL′ ▷ L′N) = (ML ▷▷ LL′) ▷ L′N
```

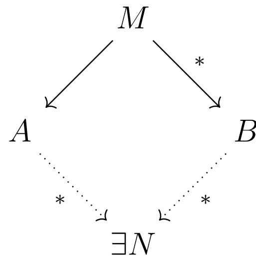
On a more technical note, because of our use of the `Rel` datatype (also defined in the Agda standard library) to express the notion of relation, Agda requires us to explicitly treat the concept of type universe levels in the signature of both definitions presented so far. This is achieved through the use of the implicit level variables `t`, `r` and the upper bound operator _⊔_. We will not further cover here the specific details since they do not particularly affect the proof.

8.2 Semi-confluence

As we will see in Section 8.3, it is easier to prove that the Z-property implies semi-confluence, rather than full confluence. Although seemingly weaker at first, semi-confluence in turn implies and is indeed equivalent to full confluence. We define this property as follows:

Definition 8.2.1 (Semi-confluence). A relation $\rightarrow$ is said to be *semi-confluent*, if, given a term M such that $M \rightarrow A$ and $M \rightarrow^* B$, there exists a term N such that $A \rightarrow^* N$ and $B \rightarrow^* N$.

We can be visually express this notion as follows:



Note that, contrary to full confluence, one of the two given reductions is not the reflexive transitive closure but a single reduction step.
The notion of semi-confluence for any generic relation R can be directly formalized with the following predicate:

```
Semi-Confluence : ∀ {t r} {T : Set t} (R : Rel T r) → Set (t ⊔ r)
Semi-Confluence R = ∀ {M A B}
  →               R M A → Star R M B
     ---------------------------------
  → ∃[ N ] (Star R A N × Star R B N)
```

It is useful to compare such definition with that of full confluence:

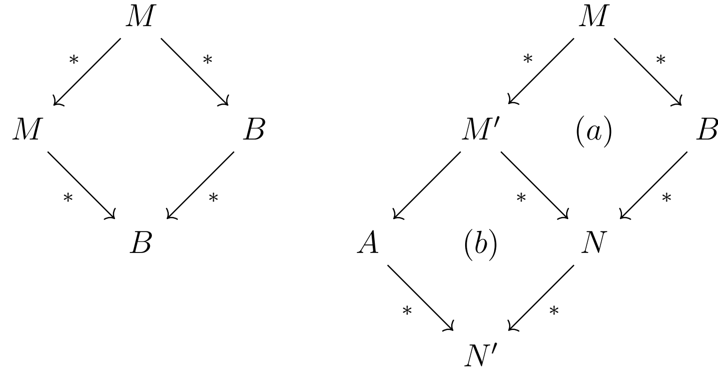
```
Confluence : ∀ {t r} {T : Set t} (R : Rel T r) → Set (t ⊔ r)
Confluence R = ∀ {M A B}
  →          Star R M A → Star R M B
     ---------------------------------
  → ∃[ N ] (Star R A N × Star R B N)
```

We chose not to use this last formalization in the previous chapters because it makes the proofs slightly harder to read, and it is unnecessarily general for our case that specifically treats β-reduction.

The proof that semi-confluence implies confluence now proceeds by induction on the left reduction provided by the `Confluence` statement:

```
semi-to-confluence : ∀ {t r} {T : Set t} {R : Rel T r} →
  Semi-Confluence R → Confluence R
semi-to-confluence sc {B = B} ε M*B = B , M*B , ε
semi-to-confluence sc (M*M′ ▷ M′A) M*B
    with semi-to-confluence sc M*M′ M*B
... | N , M′*N , B*N
    with sc M′A M′*N
... | N′ , A*N′ , N*N′
    = N′ , A*N′ , B*N ▷▷ N*N′
```

We can follow the proof with diagrams, indicating the base case on the right and the inductive one on the left:



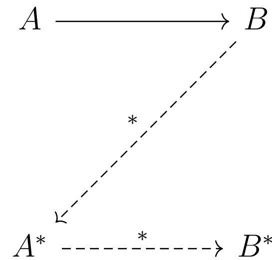
On the left we have the trivial case where M reduces to itself. On the right we have $M \to^* M' \to A$ according to our definition of transitivity for `Star`, with (a) commuting by inductive hypothesis and (b) commuting by semi-confluence.

8.3 Z-property

Having defined the general concept of reflexive transitive closure, we can now state the Z-property:

Definition 8.3.1 (Z-property). A relation $\to$ on T is said to have the *Z-property* if there exists a map $* : T \to T$ such that $A \to B$ implies $B \to^* A^*$ and $A^* \to^* B^*$, for any A and B. We refer to the implication of the two reductions as first Z-property and second Z-property, respectively.

The name of this definition again comes from the fact that the diagrammatic form for this statement resembles the letter Z:



We can now present its formalization in Agda:

```
Z : ∀ {t r} {T : Set t} (R : Rel T r) → Set (t ⊔ r)
Z {t}{r}{T} R =
  Σ[ _* ∈ (T → T) ]
    (∀ {A B : T} → R A B → Star {t}{r} R B     (A *)) ×
    (∀ {A B : T} → R A B → Star {t}{r} R (A *) (B *))
```

First, we have an easy lemma stating that a relation with the Z-property is monotonic with respect to the map $*$, i.e.: $A \rightarrow^* B$ implies $A^* \rightarrow^* B^*$. The proof is trivial by repeated application of the second Z-property:

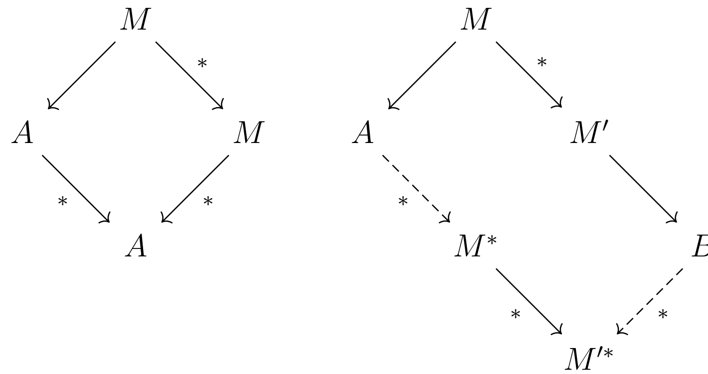
```
z-monotonic : ∀ {t r} {T : Set t} {R : Rel T r}
  → (z : Z R)
    -----------------------------------------
  → (∀ {A B} → Star R A B
             → Star R (proj₁ z A) (proj₁ z B))
z-monotonic z ε = ε
z-monotonic z@(_* , Z₁ , Z₂) (A*A′ ▷ A′B) =
  z-monotonic z A*A′ ▷▷ Z₂ A′B
```

The projection function `proj₁` is used here to simply "extract" the map `*` from the evidence that the Z-property holds, and it is defined in the Agda standard library module `Data.Product`. The `@` syntax is used in the function body to give a placeholder name to this evidence, which is first decomposed and then passed as it is to the inductive hypothesis.

Now we finally have that Z-property is sufficient for semi-confluence. The proof proceeds by cases on the structure of the left reduction of semi-confluence:

```
z-semi-confluence : ∀ {t r} {T : Set t} {R : Rel T r} →
  Z R → Semi-Confluence R
z-semi-confluence   (_* , Z₁ , Z₂) {A = A} MA ε = A , ε , (ε ▷ MA)
z-semi-confluence z@(_* , Z₁ , Z₂) MA (_▷_ {j = M′} M*M′ M′B) =
   M′ * , Z₁ MA ▷▷ z-monotonic z M*M′ , Z₁ M′B
```

Graphically, we can express the two cases with the following diagrams:



In the case where there are no reductions, we simply lift the left reduction into its reflexive transitive closure to complete the case. In the other case, the reduction has the structure $M \to^* M' \to B$. We apply the first Z-property twice on the two one-step reductions (with the results being the two dashed arrows), and then apply monotonicity to the rest of the star reduction to get $M^* \to^* M'^*$. By a final application of transitivity, we have provided the two unifying reductions, and thus completed the proof. Note that the inductive hypothesis was not necessary, and that the second Z-property is not directly applied here but it is embedded into the monotonicity lemma. The advantage of considering `Star` with the additional one-step reduction at the end of the chain becomes clear here, since the term M' effectively becomes the joining term through the use of the Z-property map.
Finally, we simply combine the previous lemmas to have our result:

```
z-confluence : ∀ {t r} {T : Set t} {R : Rel T r} →
  Z R → Confluence R
z-confluence z = semi-to-confluence (z-semi-confluence z)
```

8.4 Z-property for β-reduction

As it was implicit in the name we chose to the map described in the Z-property, we actually have that Takahashi translation (quoted as the full-superdevelopment function in [NOS16]) indeed is a map that provides the Z-property for β-reduction. As a technical detail, we need to prove that our generic version of the reflexive transitive closure (with its inverted transitivity) is indeed equivalent to our definition of β^*-reduction `_↠_` presented in Section 4.1. We also need a trivial intermediate lemma stating we can append a reduction to the left of any star reduction:

```
star-left :  ∀ {n} {M L N : Term n}
  → M ⟶ L
  → Star _⟶_ L N
    -------------
  → Star _⟶_ M N
star-left ML ε = ε ▷ ML
star-left ML (L*N′ ▷ N′N) =
  star-left ML L*N′ ▷ N′N

betas-star : ∀ {n} {M N : Term n}
  → M ↠ N
    -------------
  → Star _⟶_ M N
betas-star (M ∎) = ε
betas-star (M ⟶⟨ M⟶M′ ⟩ M′↠N) =
  star-left M⟶M′ (betas-star M′↠N)
```

Finally, we can show that Lemma 3.3 and Lemma 3.5 introduced in the paper by Komori et al. [KMY14] actually represent, respectively, the first and second Z-properties for the case of β-reduction. We can reuse the proofs `lemma3-3` and `lemma3-5` presented in Chapter 7 to manually construct the evidence that the Z-property holds, and ultimately derive confluence:

```
z-confluence-beta : ∀ {n} → Confluence (_⟶_ {n})
z-confluence-beta = z-confluence ( _*
                                 , betas-star ∘ lemma3-3
                                 , betas-star ∘ lemma3-5)
```

This concludes our last proof of the Church-Rosser theorem.

8.5 Comparison with the Komori-Matsuda-Yamakawa proof

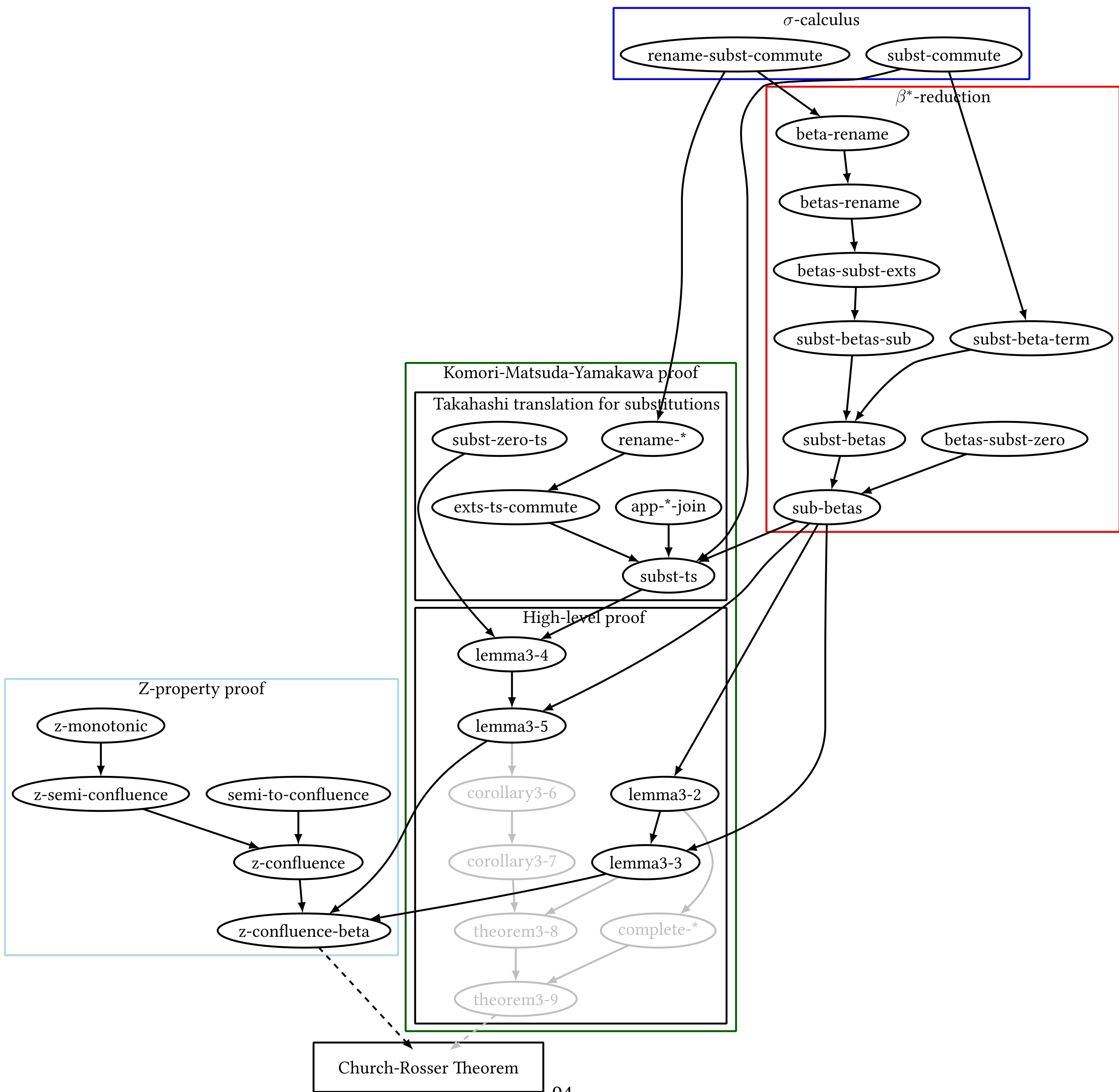
This proof is considerably shorter in terms of lemmas and properties used, and follows a more direct approach than the one presented by Komori et al. in [KMY14]. Similarly as with the Tait/Martin-Löf proof, the proof nevertheless comes at the cost of not providing the more precisely quantified results about the joining term. The existence of the Z-property actually further sharpens an observation remarked in [KMY14, Theorem 3.10], where the authors state that their proof can be generalized to any reduction $\rightarrow$ that satisfies the following properties:

- $\langle A \rangle \;\; A \rightarrow^* A^*$
- $\langle B \rangle \;\; A \rightarrow B \Longrightarrow B \rightarrow^* A^*$
- $\langle C \rangle \;\; A \rightarrow B \Longrightarrow A^* \rightarrow^* B^*$

However, as we have shown in Agda and [NOS16] have proven in Isabelle/HOL, properties $\langle B \rangle$ and $\langle C \rangle$ are indeed sufficient to prove confluence. Property $\langle A \rangle$, which we formalized in `lemma3-2`, is still required in order to prove `lift-*` and `complete-*`; these two theorems are then used to "complete" the remaining Takahashi translations in the final step of `theorem3-9`. Therefore, it seems that property $\langle A \rangle$ is indeed necessary in order to have the more precisely quantified results shown in [KMY14], but it is not strictly needed to prove confluence in general. In our specific case of β-reduction, property $\langle A \rangle$ is still an essential lemma used to prove property $\langle B \rangle$, represented by `lemma3-3`.

8.6 Proof overview

We present here a structural overview of the proof, with the specific purpose of highlighting and dividing the parts necessary to establish the two Z-properties for β-reduction, and those specific for the proof by Komori et al. [KMY14]. For brevity, we omit some intermediate lemmas of this latter section while still showing the main theorems:


σ-calculus
rename-subst-commute
subst-commute
β*-reduction
beta-rename
betas-rename
betas-subst-exts
subst-betas-sub
subst-beta-term
subst-betas
betas-subst-zero
sub-betas
Komori-Matsuda-Yamakawa proof
Takahashi translation for substitutions
subst-zero-ts
rename-*
exts-ts-commute
app-*-join
subst-ts
High-level proof
lemma3-4
lemma3-5
corollary3-6
corollary3-7
lemma3-2
lemma3-3
theorem3-8
complete-*
theorem3-9
Z-property proof
z-monotonic
z-semi-confluence
semi-to-confluence
z-confluence
z-confluence-beta
Church-Rosser Theorem

Chapter 9

Conclusion

9.1 Theoretic perspective

To our knowledge, no formal proof of the results presented by Komori et al. [KMY14] can be found in the extensive literature concerning the formalization of the Church-Rosser theorem. The developments we presented in Chapter 7 of this thesis further certify the validity of these important advancements in the context of rewriting systems and confluence. Presenting a formalization of their results also provides future readers a complete perspective on the proof, clarifying and thoroughly proving correct those details that might have remained unspecified or left to the reader. Indeed, Lemma 3.4 relating substitutions and Takahashi translation, whose proof in the original paper had only been sketched, effectively constituted the hardest property to adequately formalize in our setting.
As we have shown in Chapter 8, the theorems proven in this last proof unknowingly coincided with the fundamental lemmas also used by Nagele et al. [NOS16] in their formalization. This coincidence has been completely accidental, and we discovered this latter paper only at a later time. Our developments also provide a perspective on how the properties presented by the authors in Isabelle/HOL, using the Nominal approach, can be shown in Agda with the different context of de Bruijn indices. The more explicit proof detail that Agda naturally induces, compared with the profound automation provided by Isabelle/HOL, also provides insight in those crucial steps that might be lost in a completely automatic development. This gives us a comparative sense on how difficult a complete manual formalization can be when considering proofs regarding λ-calculus, substitutions and binders.

9.2 Implementation

As we have experimented in this thesis, the infrastructure implemented by Wadler et al. in [WK19] provides an elegant foundation on which to develop more general proofs about λ-calculus. Unfortunately, the lack of tactics and powerful automation mechanisms in Agda comes at the cost of having to concretely interact with the representation method used for λ-terms, in this case de Bruijn indices. This interaction in turn requires a complete understanding of the functions and theorems used in the lower-level layers, especially when the properties considered directly interact with substitutions and therefore renamings.
This is a general characteristic of Agda proofs, where each detail has to be explicitly indicated and even automation provides results in the form of explicit proofs. This perspective stands in sharp contrast with the black-box and higher-level approaches available in Isabelle, for example with Nominal Isabelle [Urb08], or in Coq with the automation tactics provided by the Autosubst library [STS15], where the low-level details are automatically handled by the tools. Many other libraries handling binders and variables can be found mentioned in these papers, showing how important and ubiquitous these representation issues can be.
The approach taken by Agda is simply that of library reuse, where the user only interfaces themselves with the high-level theorems of the library. The hardest part of the formalization, however, still remains the conceptual shift in having to reconsider everything in terms of de Bruijn indices and inevitably being aware of the functions that operate on them; the need for pointwise reduction and pointwise Takahashi translation is certainly not obvious at first glance, and it required a detailed understanding of the underlying concepts of renamings, parallel substitutions, and extensions. Fortunately, the use of the `subst-commute` and `rename-subst-commute` theorems, developed through the framework of σ-calculus, still allowed us to maintain a fairly high abstraction level in handling de Bruijn indices by solving for us the crucial substitution cases.
After overcoming these technical and conceptual difficulties, the proofs nevertheless proceed with relative ease. Having to understand the implementation details can also provide the user with a clearer vision of the overall proof, and even interfacing with the underlying library can occasionally constitute an engaging proof activity that leads to lower-level yet elegant theorems.
Interactivity also plays an important role in the use of a proof assistant; for example, Agda can aid the user in keeping track of the goal to prove, automatically splitting the inductive cases, and sometimes even providing a limited form of proof automation with the automatic proof searcher Agsy.

This gives rise to an explorative dynamic between the prover and the interface, where the tool interactively assists the user in constructing proofs. These facilities have been extensively used during the development of this thesis, and have provided a useful guidance in proving the results here presented.

9.3 Future work

The most immediate further improvement for this thesis is the additional treatment of η-reduction. At the end of their paper, Komori et al. [KMY14] also outline some crucial definitions and proof remarks treating the specific case of $\beta\eta$-reduction, where a term $\lambda x.Mx$ can be reduced to M if x does not occur free in M. In our formal setting with de Bruijn indices this would be represented as $\lambda M0$, with a similarly defined constraint that requires the variable referring to the outermost binder not to appear inside M. In the case of $\beta\eta$-reduction, the authors also propose alternative and more precise definitions for Takahashi translation than those originally established by Takahashi [Tak95], by introducing the additional requirement for Mx not to be a redex. Further exploration on this topic is therefore required, first by adequately formalizing these constraints with de Bruijn indices, and successively in extending and constructing the confluence proof in order to account for this additional reduction.
Another possible extension of this development is dealing with different representation techniques for λ-terms. This would help in verifying the logical independence of the theorems so far presented from the implementation used, and give further insights on how easily the same results can be obtained with alternative methods. This would especially concern those properties connecting Takahashi translation and substitutions, as exemplified by `lemma3-4`.
The results proven by Komori et al. could also be shown using different proof assistants, such as Coq or Isabelle/HOL. By reusing the lemmas already provided by Nagele et al. [NOS16], it would not be too difficult to complete the remaining theorems required to obtain the quantified confluence formalized in this thesis.
In conclusion, this formalization experience gave us the possibility to explore how different proof approaches can target the same fundamental results.
Analyzing theorems with the detail level offered by Agda also allowed us to get a more refined comprehension on how the proofs proceed on a small scale, while also gaining insight on the connections between these two different approaches.
We hope that the results presented here will further enrich the well-documented literature and the formalizations available for this fundamental theorem.

Ringraziamenti

Vorrei innanzitutto ringraziare il professor Ugo de' Liguoro e il dottor Riccardo Treglia per la loro costante disponibilità e per il loro fondamentale aiuto nell'organizzare il lavoro presentato in questa tesi. Ringrazio la mia famiglia, la mia ragazza Amalia e tutti gli amici che da sempre sono stati al mio fianco in questo percorso. 頑張ります！

Bibliography

[CR36] Alonzo Church and J. B. Rosser. "Some Properties of Conversion". In: *Transactions of the American Mathematical Society* 39.3 (1936), pp. 472–482. ISSN: 00029947. URL: http://www.jstor.org/stable/1989762.

[Bru72] Nicolaas G. de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. URL: https://doi.org/10.1016/1385-7258(72)90034-0.

[Mar84] Per Martin-Löf. *Intuitionistic type theory*. Vol. 1. Studies in proof theory. Bibliopolis, 1984. ISBN: 978-88-7088-228-5.

[Bar85] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*. Vol. 103. Studies in logic and the foundations of mathematics. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

[Sha88] Natarajan Shankar. "A mechanical proof of the Church-Rosser theorem". In: *J. ACM* 35.3 (1988), pp. 475–522. DOI: 10.1145/44483.44484.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Vol. 7. Cambridge Tracts in Theoretical Computer Science. Cambridge, 1989.

[Aba+91] Martín Abadi et al. "Explicit Substitutions". In: *J. Funct. Program.* 1.4 (1991), pp. 375–416. DOI: 10.1017/S0956796800000186.

[Pfe92] Frank Pfenning. *A Proof of the Church-Rosser Theorem and its Representation in a Logical Framework*. 1992.

[Hue94] Gérard P. Huet. "Residual Theory in lambda-Calculus: A Formal Development". In: *J. Funct. Program.* 4.3 (1994), pp. 371–394. DOI: 10.1017/S0956796800001106.

[Luo94] Zhaohui Luo. *Computation and reasoning - a type theory for computer science*. Vol. 11. International series of monographs on computer science. Oxford University Press, 1994. ISBN: 978-0-19-853835-6.

[Pau94] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*. Vol. 828. Lecture Notes in Computer Science. Springer, 1994. ISBN: 3-540-58244-4. DOI: 10.1007/BFb0030541.

[Ras95] Ole Rasmussen. *The Church-Rosser Theorem in Isabelle: A Proof Porting Experiment*. Tech. rep. 364. Computer Laboratory, University of Cambridge, May 1995. URL: %5Curl%7Bhttp://www.cl.cam.ac.uk:80/ftp/papers/reports/TR364-or200-church-rosser-isabelle.ps.gz%7D.

[Tak95] Masako Takahashi. "Parallel Reductions in λ-Calculus". In: *Inf. Comput.* 118.1 (1995), pp. 120–127. DOI: 10.1006/inco.1995.1057.

[Nip96] Tobias Nipkow. "More Church-Rosser Proofs (in Isabelle/HOL)". In: *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*. Ed. by Michael A. McRobbie and John K. Slaney. Vol. 1104. Lecture Notes in Computer Science. Springer, 1996, pp. 733–747. DOI: 10.1007/3-540-61511-3_125.

[MP99] James McKinna and Robert Pollack. "Some Lambda Calculus and Type Theory Formalized". In: *J. Autom. Reasoning* 23.3-4 (1999), pp. 373–409. DOI: 10.1023/A:1006294005493.

[Hom01] Peter V. Homeier. *A Proof of the Church-Rosser Theorem for the Lambda Calculus in Higher Order Logic*. US Department of Defense, 2001.

[VB01] René Vestergaard and James Brotherston. "A Formalised First-Order Confluence Proof for the lambda-Calculus Using One-Sorted Variable Names". In: *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*. Ed. by Aart Middeldorp. Vol. 2051. Lecture Notes in Computer Science. Springer, 2001, pp. 306–321. DOI: 10.1007/3-540-45127-7_23.

[Ter03] Terese. *Term Rewriting Systems*. Vol. 55. Cambridge tracts in theoretical computer science. Cambridge University Press, 2003. ISBN: 978-0-521-39115-3.

[Ayd+05] Brian E. Aydemir et al. “Mechanized Metatheory for the Masses: The PoplMark Challenge”. In: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*. Ed. by Joe Hurd and Thomas F. Melham. Vol. 3603. Lecture Notes in Computer Science. Springer, 2005, pp. 50–65. DOI: 10.1007/11541868_4.

[Nor07] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. 2007.

[Ayd+08] Brian E. Aydemir et al. “Engineering Formal Metatheory”. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 3–15. DOI: 10.1145/1328438.1328443.

[DO08] Patrick Dehornoy and Vincent van Oostrom. “Z: Proving confluence by monotonic single-step upperbound functions”. In: *Logical Models of Reasoning and Computation (LMRC-08)*. 2008.

[Urb08] Christian Urban. “Nominal Techniques in Isabelle/HOL”. In: *J. Autom. Reasoning* 40.4 (2008), pp. 327–356. DOI: 10.1007/s10817-008-9097-2.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. “A Brief Overview of Agda - A Functional Language with Dependent Types”. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6.

[CH09] Felice Cardone and J. Roger Hindley. “Lambda-Calculus and Combinators in the 20th Century”. In: *Logic from Russell to Church*. Ed. by Dov M. Gabbay and John Woods. Vol. 5. Handbook of the History of Logic. Elsevier, 2009, pp. 723–817. DOI: 10.1016/S1874-5857(09)70018-4.

[How10] Douglas J. Howe. "Higher-Order Abstract Syntax in Isabelle/HOL". In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 481–484. DOI: 10.1007/978-3-642-14052-5_33.

[Acc12] Beniamino Accattoli. "Proof Pearl: Abella Formalization of λ-Calculus Cube Property". In: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*. Ed. by Chris Hawblitzel and Dale Miller. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 173–187. DOI: 10.1007/978-3-642-35308-6_15.

[Cha12] Arthur Charguéraud. "The Locally Nameless Representation". In: *J. Autom. Reasoning* 49.3 (2012), pp. 363–408. DOI: 10.1007/s10817-011-9225-2.

[KMY14] Yuichi Komori, Naosuke Matsuda, and Fumika Yamakawa. "A Simplified Proof of the Church-Rosser Theorem". In: *Studia Logica* 102.1 (2014), pp. 175–183. DOI: 10.1007/s11225-013-9470-y.

[KS15] Pepijn Kokke and Wouter Swierstra. "Auto in Agda - Programming Proof Search Using Reflection". In: *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. Ed. by Ralf Hinze and Janis Voigtländer. Vol. 9129. Lecture Notes in Computer Science. Springer, 2015, pp. 276–301. DOI: 10.1007/978-3-319-19797-5_14.

[STS15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. "Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions". In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 359–374. DOI: 10.1007/978-3-319-22102-1_24.

[NOS16] 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". In: *CoRR* abs/1609.03139 (2016). arXiv: 1609.03139.

[CST17] 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: *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017*. Ed. by Sandra Alves and Renata Wasserman. Vol. 338. Electronic Notes in Theoretical Computer Science. Elsevier, 2017, pp. 79–95. DOI: 10.1016/j.entcs.2018.10.006.

[WK19] Philip Wadler and Wen Kokke. *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk/. 2019.

[Coq] The Coq Development Team. *The Coq Proof Assistant Reference Manual - Version V8.4, 2012*. URL: http://coq.inria.fr.