UMinho | 2022
Bruna Calisto
Formalization in Coq of the Standardization Theorem for λ-calculus

Universidade do Minho
Escola de Ciências

Bruna Isabel Afonso Carvalho Calisto

Formalization in Coq of the Standardization Theorem for λ-calculus

outubro de 2022

Universidade do Minho
Escola de Ciências

Bruna Isabel Afonso Carvalho Calisto

Formalization in Coq of the Standardization Theorem for λ-calculus

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho efetuado sob a orientação do
Professor Doutor Luís Filipe Ribeiro Pinto

outubro de 2022

Acknowledgements

I would like to thank all people who supported me through different ways in the writing of this dissertation, of which I want to highlight:

My advisor and Doctor Luís Filipe Ribeiro Pinto for all the encouragement and help provided, for all the patience and willingness to guide me throughout the dissertation. Without my advisor, this dissertation would not be accomplished.

To my husband, Jorge Calisto, for the patience and support shown throughout this phase.

To my mother, Susy Calisto, who, in addition to supporting me, helped me in the translation of this dissertation into English.

To the Research Centre of Mathematics of the University of Minho (CMAT) and the Portuguese Foundation for Science and Technology (FCT), for funding this dissertation, through the CMAT Research Grant - UIDB/00013/ 2020 - 02/2021.

Finally, I would like to thank Catarina for helping me with the structure of this dissertation as well as other issues.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Resumo

Formalização em Coq do Teorema da Standardização para o Cálculo-λ

Os *teoremas da standardização* são resultados fundamentais da teoria da redução do Cálculo-λ. Estes resultados estabelecem que um termo t reduz para um termo t' se e só se t reduz para t' seguindo uma sequência de redução específica, dita *standard*. Em particular, estes resultados garantem a completude de certas maneiras específicas de efetuar reduções, e são a base dos resultados sobre *estratégias de avaliação*, nomeadamente *chamada-por-nome* e *chamada-por-valor*, fazendo a ponte entre um cálculo (uma teoria equacional) e uma linguagem de programação.

Esta dissertação apresenta uma formalização no sistema de prova assistida *Coq* do Teorema da Standardização para o Cálculo-λ. Neste sentido, consideramos uma prova deste resultado que extraímos de uma prova de um Teorema da Standardização para um cálculo-λ para lógica modal proposto por Espírito Santo-Pinto-Uustalu, onde *redução standard* é capturada através de uma relação definida indutivamente nos termos-λ, em linha com tratamentos de standardização para o Cálculo-λ por Loader e por Joachimski-Matthes. A implementação da sintaxe dos termos-λ usa os índices de De Bruijn, mas a formalização Coq segue de muito perto a estrutura da prova do Teorema da Standardização (com termos-λ ordinários).

Adicionalmente, esta dissertação considera uma noção independente de *sequência de redução standard* para o Cálculo-λ estudada por Plotkin. Por um lado, provámos que sequências de redução e a abordagem inicial de redução standard como uma relação indutiva nos termos-λ são formas equivalentes de caracterizar redução standard e, por outro, fornecemos uma formalização dessa equivalência em Coq.

Palavras-chave: Chamada-por-nome, chamada-por-valor, sistema de prova Coq, standardização

Abstract

Formalization in Coq of the Standardization Theorem for λ-calculus

Standardization theorems are fundamental results in the theory of reduction of λ-calculus. They establish that a term t reduces to a term t' if and only if t reduces to t' following some specific sequence of reductions said *standard*. In particular, these results guarantee completeness of specific ways of performing reduction, and are at the basis of results about *evaluation strategies*, namely *call-by-name* and *call-by-value*, bridging between calculi (equational theories) and programming languages.

This dissertation presents a formalization in the *Coq proof assistant* of the Standardization Theorem for the call-by-name version of λ-calculus, i.e. ordinary λ-calculus. In this development, we consider a proof of this result that we extracted from a proof of a standardization theorem for a λ-calculus for modal logic Espírito Santo-Pinto-Uustalu, where *standard reduction* is captured via an inductively defined relation on λ-terms, in line with treatments of standardization for λ-calculus by Loader and Joachimski-Matthes. The implementation of the λ-terms syntax uses the De Bruijn indices, but the Coq formalization follows closely the structure of the proof of the Standardization Theorem (with ordinary λ-terms), both in what concerns lemmata and the inductive structure of arguments.

Additionally, this dissertation also considers an independent notion of *standard reduction sequence* for (call-by-name) λ-calculus studied by Plotkin. Firstly, we prove that reduction sequences and the approach of standard reduction as an inductive relation on λ-terms are indeed equivalent ways of characterizing standard reduction. Then, we provide a complete formalization in Coq of this equivalence.

Keywords: Call-by-name, call-by-value, Coq proof assistant, standardization

Contents

List of Figures

Chapter 1

Introduction

λ-calculus and functional programming. λ-calculus was introduced by Alonzo Church in the 1930s, intended as a foundation for mathematics [4]. Church invented this formal system (λ-calculus) and by this via defined the notion of computable function [7]. At about the same time, Turing invented a class of machines (Turing machines) and by this via also defined a notion of computable function [7]. Still in the 1930s, Turing showed that λ-calculus can represent all the functions computable by a Turing machine and vice versa [4, 7]. Meanwhile, Kleene and Rosser also proved that λ-calculus can represent all recursive functions [4]. These equivalences, and the observation that other analysis of computability (such as Post systems [31]) also captured the same class of functions, led to the so-called "Church-Turing Thesis", according to which λ-calculus (just as Turing machines) fully capture the notion of computability [6].

With the invention of computers and programming languages, the importance of λ-calculus became obvious in the design, implementation, and theory of functional programming languages [37]. This formal system has even been qualified as the "smallest universal programming language of the world" [32], since, on the one hand it fully captures computability, and, on the other hand, it consists of a single rule of substitution, which makes it convenient for a rigorous mathematical analysis.

Functional languages are concerned with describing a solution to a problem [10]. Some examples of functional programming languages are Haskell, OCaml, Scheme, SML and LISP [14, 28]. One of the advances provided in λ-calculus is that computations on data types, like trees and syntactic structures, can be represented as expressions in λ-calculus (λ-terms) [6]. Viewed through λ-calculus, the execution mechanism of functional programming languages corresponds to reduction of λ-terms to *normal form*. One of the fundamental results of the theory of reduction of the λ-calculus is the *Standardization Theorem*, which establishes that one λ-term t reduces to another λ-term t if and only if t reduces to t following some specific sequence of reductions, said *standard*. In particular, these kind of results guarantee completeness of specific ways of performing reduction. As mentioned, reduction in λ-calculus considers a single rule of substitution, named (β). We will see later that there maybe situations where the (β)-rule can be applied in different possible ways, which can potentially lead to non-determinism. But, generally, in programming languages we expect determinism of execution. This is one of the reasons for programming languages to adopt specific strategies to evaluate expressions. Two fundamental evaluation strategies are call-by-name (cbn) and call-by-value (cbv), expressing different policies for treating "function call". Roughly, cbn wants to apply the function as soon as possible, whereas cbv only applies the function when the argument is already a "value". Still, it is possible to find simulations of each of the two evaluation strategies by the other, as described in [30]. This work by Plotkin shows also that standardization theorems are useful tools to bridge between functional programming languages, which implement a certain evaluation strategy, and the λ-calculus, which provides an equational theory to reason about such functional programs.

Formalization and proof assistants. Nowadays, we can encode mathematical results in the computer. We call *formalization* to such encoding, and *proof assistant* to a program that implements a meta-logic where mathematical results can be described and which allows to check the correctness of formalizations [41]. Mathematical proofs can be extensive, with many cases to check, even if many of the cases are not interesting and are easy to prove. Also, it is very easy to make a mistake in some part of a proof that can put at risk the veracity of the result under consideration. Automated proof assistants can help us with these type of problems, and therefore are useful tools in the formalization of proofs in Mathematics, but also in the context of the verification of properties of software [2]. Examples of such tools highly used today are the Coq, AGDA and Isabelle proof assistants. For example, the Coq proof assistant (which will be used in this dissertation) implements an higher-order logic based on the Calculus of Inductive Constructions, and it is an interactive tool, where the user can set up a mathematical theory, by defining concepts and stating theorems, and then interactively develop formal proofs of these theorems [39]. An example of a well known result fully formalized in computer is the Four Color Theorem. This theorem states that with at most four colors it is possible to color the regions of any map, so that no two adjacent regions have the same color. This theorem is famous for being the first significative mathematical problem to be formalized using a computer program, namely Coq [15]. Another significative result that has been formalized in the Coq proof assistant is the Feit-Thompson Odd Order Theorem (a result in Group Theory, establishing that every finite group of odd order is solvable).

In the context of the λ-calculus and Type Theory, the literature offers a big collection of mechanized formalizations. For example, in one of the early works in this direction, Huet formalized in Coq results of the residual theory of β-reduction in λ-calculus, including a proof of the Church-Rosser theorem [18]. An even earlier formalization of the Church-Rosser for λ-calculus was developed by Shankar using the Boyer-Moore theorem prover [36], and a later one was developed by Nipkow in Isabelle [27]. Other early works include formalizations in the LEGO proof assistant of results like strong normalization for system F (an extension of simply typed λ-calculus with polymorphism) [1] or of the basic theory of Pure Type Systems (a generalisation of Barendregt's "λ-cube", where the simply-typed lambda-calculus is the "starting corner") [25, 26], addressing in particular the Standardization Theorem.

In order to formalize the theory of λ-calculus or extensions of it, due to the binding mechanism underlying λ-abstraction and the need to address equality of λ-terms up to renaming of bound variables, it is necessary to use some technique to deal with the binders. There are several techniques, such as renaming variables [22], the De Bruijn indices [18], multiple substitution [38], locally nameless [3] and higher-order abstract syntax [29].

Contributions of this dissertation. In this dissertation we consider some results of λ-calculus, concerning standard reduction, using the Coq proof assistant to formalize them. The main result that we formalize is the Standardization Theorem, using the De Bruijn indices technique to deal with binders. There are different ways to define standard reduction in λ-calculus. Here, standard reduction will be given through an inductively defined relation on λ-terms, extracted from a definition in [33] (for a λ-calculus for modal logic), which is in line with the approach followed by Loader and Joachimski-Matthes, where standard reduction is also given as an inductive binary relation, but for λ-terms which allow the application construction to act on a non-empty lists of arguments (not only one argument, as in ordinary λ-terms). So, we needed to start by adapting to λ-calculus the concepts and results leading to the Standardization Theorem in [33]. This is a first small contribution of this dissertation, since these details cannot be found elsewhere. Another contribution of the dissertion is the full formalization in Coq of this proof of the Standardization Theorem. This dissertation also presents a development of a proof of equivalence between the approach we followed to standard reduction (via an inductive relation on λ-terms) and the more common approach considered by Plotkin, based on *standard reduction sequences* [30]. This development and its formalization in Coq is a last contribution of this dissertation.

Plan of the dissertation. Chapter 2 recalls basic concepts and results of the λ-calculus, and informally introduces the call-by-name and call-by-value evaluation strategies. Chapter 3 starts by introducing the relations of call-by-name evaluation and of standard reduction, proves several properties of these relations, and concludes with a proof of the Standardization Theorem. Chapter 4 introduces the λ-calculus with the De Bruijn indices, and presents the Coq formalization of all the results of the previous chapter. Chapter 5 introduces the definition of standard reduction sequence, proves the equivaleFnce between the standard reduction relation and the standard reduction sequences approaches, and presents a full formalization in Coq of this equivalence. Chapter 6 concludes and mentions some topics left open, which can be subject of future work. In Appendices A, B, C and D are the details of proofs of the results of Chapters 2, 3, 4 and 5, respectively. In Appendices E and F is the full code of the formalization of the results of Chapters 4 and 5, respectively, developed under version 8.12.2 of the Coq proof assistant.

Chapter 2

Background on λ-Calculus

In this chapter, we will recall basic material on λ-calculus relevant for this dissertation. We will introduce basic concepts of the λ-calculus regarding syntactical aspects and β-reduction, and we will also recall well known results such as the Substitution Lemma and the Church-Rosser Theorem. Examples are introduced throughout the chapter in order to help understanding notations and definitions. Additionally, we will informally introduce basic evaluation mechanisms for λ-calculus, namely call-by-name and call-by-value evaluations. The concepts and the results recapitulated in this dissertation can be found in many places in literature, such as [5, 16, 17, 21, 22, 24, 35, 40].

2.1 λ-terms and substitution

In λ-calculus there are three kinds of terms: *variables*, *abstractions* and *applications*. The combination of these terms produces the set of λ-*terms*. Basically the abstractions represent functions and an application represents the application of a function to its argument. Formally:

Definition 1. *Let us assume an infinite denumerable* ***set of variables*** V*, and assume also that* $x, y, z\ldots$ *range over this set* V*. The set of* λ*-terms,* Λ*, is defined inductively by:*

1. $V \subseteq \Lambda$;
2. $M \in \Lambda \Rightarrow (\lambda x \cdot M) \in \Lambda \quad$ *(for any* $x \in V$*);*
3. $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$.

In the above definition, a λ-term of the form ($\lambda x \cdot M$) (clause 2) is called a *λ-abstraction*, in which x is said the *parameter* or the *variable* of the abstraction, and M is said the *body* of the abstraction. A λ-term of the form (MN) (clause 3) is called an *application*, where M is said in *function position* and N is said in *argument position*.

Remark 1. *In this dissertation, to avoid heavy parentheses notation, we establish the following conventions for writing λ-terms:*

1. *The outermost parentheses will be omitted. For example, the λ-term MN means (MN);*
2. *Applications associate to the left. Which means, $M_1M_2M_3$ abbreviates $((M_1M_2)\,M_3)$;*
3. *The body of a λ-abstraction extends as far right as possible. Thus, $\lambda x \cdot MN$ means $\lambda x \cdot (MN)$;*
4. *Multiple λ-abstractions can be contracted. For instance, we write $\lambda xyz{\cdot}M$ instead of $(\lambda x{\cdot}\lambda y{\cdot}\lambda z{\cdot}M)$.*

An important operation in the λ-calculus is *substitution* that consists of replacing *free occurrences of a variable* in a λ-term by another λ-term. For example, in the term ($\lambda x \cdot xy$), the occurrence of variable x in its body is bound by the λx binder, so will not count as a free occurrence of x. But the occurrence of the variable y is free, because it is not bound by any λ binder. The set of variables occurring freely in a λ-term can be easily characterized by recursion as follows:

Definition 2. *Let M be a λ-term. We represent the set of* ***free variables*** *by $FV(M)$. This set is recursively defined by:*

1. $FV(x) = \{x\} \quad (x \in V)$;
2. $FV(\lambda x \cdot N) = FV(N) \setminus \{x\} \quad (x \in V, N \in \Lambda)$;
3. $FV(N_1N_2) = FV(N_1) \cup FV(N_2) \quad (N_1, N_2 \in \Lambda)$.

M is said ***closed*** *when $FV(M) = \emptyset$, and is said* ***open*** *otherwise.*

It is obvious to see that when the λ-abstractions $\lambda x \cdot xz$ and $\lambda y \cdot yz$ are regarded as functions, they correspond to the same function, only differing in the concrete name chosen for the parameter. We will say that these two λ-terms are *α-equivalent*. The *α-equivalence* relation is represented by $=_\alpha$, and can be defined starting from a basic rule, called α-rule, which allows the renaming of variables in λ-abstractions [22].

Let us return to the substitution operation. We will write $M[N/x]$ (for x a variable and M,N λ-terms), to stand for the substitution of the free occurrences of x by N in the λ-term M.

A very recurring problem in this operation $M[N/x]$ is *variable capture* which occurs when some free occurrence of a variable y in N ends up in the scope of a binder λy in M. To better understand this phenomenon, let us see one example.

Example 1. *Consider the substitution $(\lambda y \cdot xy)[y/x]$. Applying the substitution operation just described we obtain $\lambda y \cdot yy$. Therefore the occurrence of the variable y that results from the substitution of the free occurrences of x (the blue one) is now bound to the binder λy. In order to avoid this, we should rename the bound variables with fresh variables (occurring nowhere) before making the substitution, i.e.:*

$$(\lambda y \cdot xy)[y/x] =_\alpha (\lambda z \cdot xz)[y/x] = \lambda z \cdot yz$$

.

So, when we apply the substitution operation, we have to be careful in order to avoid the capture of variables substitution, since this can change the intended effect of this operation.

In this dissertation, we will adopt *capture-avoiding* substitution and will work with λ-terms up to α-equivalence. However, when we arrive at the formalization of meta-theory of λ-calculus in Chapters 4 and 5, as the proof assistant cannot simply assume this convention, we will come back to this, and present an alternative to address the possibilities of renaming variables in binders.

Definition 3. *For all M, N in Λ and x in V, $M[N/x]$ represents the λ-term that results from the* ***(capture-avoiding) substitution*** *in M of all free occurrences of x by N and is recursively defined by:*

1. $x[N/x] = N$;
2. $y[N/x] = y,\ y \neq x \quad (y \in V)$;
3. $(\lambda x \cdot M_0)[N/x] = \lambda x \cdot M_0 \quad (M_0 \in \Lambda)$;
4. $(\lambda y \cdot M_0)[N/x] = \lambda z \cdot (M_0[z/y])[N/x], \quad y \neq x,\ z \neq y,\ z \neq x,\ z \notin FV(N) \cup FV(M_0)$ $(y \in V$ *and* $M_0 \in \Lambda)$;
5. $(M_0M_1)[N/x] = M_0[N/x]M_1[N/x] \quad (M_0, M_1 \in \Lambda)$

In the definition above note in clause 4 the renaming of the bound variable y in the λ-abstraction to a fresh variable z in order to prevent variable capture.

A well-known result of λ-calculus is the Substitution Lemma, described below.

Lemma 1. ***(Substitution Lemma):*** *For all x, y in V and M, N, Q in Λ, if $x \neq y$ and $x \notin FV(Q)$, then $(M[N/x])[Q/y] = (M[Q/y])[N[Q/y]/x]$.*

Proof. By induction on the size of M. For variables, the proof follows by case analysis and profits from the assumption $x \notin FV(Q)$. The abstraction and application cases follow routinely from the induction hypotheses. □

2.2 β-reduction

Evaluation of λ-terms will consist of a sequence of *reductions*, where each reduction corresponds to a substitution operation. When we have an application with a λ-abstraction in function position, we can replace in the body of the abstraction its variable by the λ-term in the argument position.
This is called *β-reduction rule*, and its base rule is then:

$$\frac{}{(\lambda x \cdot M)N \to M[N/x]}\ (\beta)$$

As usual, to the left hand side of (β) we call *redex* and the right hand side we call *contractum*.

Full β-reduction allows reduction at any subterm. For this we need to consider the *compatible closure* of the base rule (β):

Definition 4. *The compatible closure of the β-rule (also called **one-step β-reduction**) is denoted by $\to_\beta$ and is inductively defined by the following rules:*

$$\frac{}{(\lambda x \cdot M)N \to M[N/x]}\ (\beta)$$

$$\frac{M \to N}{MP \to NP}\ (\mu) \qquad \frac{M \to N}{PM \to PN}\ (\nu) \qquad \frac{M \to N}{\lambda x \cdot M \to \lambda x \cdot N}\ (\xi)$$

A λ-term M is said to be in *beta normal form* (β-nf) if no β-reduction is possible from it, formally: for no N, $M \to_\beta N$, which is the same of saying that no subterm of M is a β-redex.

Sequencing of β-reductions corresponds to the reflexive and transitive closure of $\to_\beta$:

Definition 5. *The reflexive-transitive closure of $\to_\beta$ is denoted by $\to_\beta^*$ and is inductively defined as:*

$$\frac{M \to_\beta N}{M \to_\beta^* N}\ BASE \qquad \frac{}{M \to_\beta^* M}\ REF \qquad \frac{M \to_\beta^* N \quad N \to_\beta^* P}{M \to_\beta^* P}\ TRANS$$

We defined inductively the relation $\to_\beta$ as the closure of the β-rule w.r.t. to the rules (μ), (ν) and (ξ). The next lemma says that the reflexive and transitive closure of $\to_\beta$ is already closed with respect to these rules, or in other words it is already a relation compatible with the λ-terms syntax:

Lemma 2. *For all M, M' in Λ, if $M \to_\beta^* M'$ then:*

1. *$MN \to_\beta^* M'N$, for all $N \in \Lambda$;*

2. *$NM \to_\beta^* NM'$, for all $N \in \Lambda$;*

3. *$\lambda x \cdot M \to_\beta^* \lambda x \cdot M'$, for all $x \in V$.*

Proof. By induction on $\to_\beta^*$. The proof of the first statement uses rule (μ), the second uses rule (ν), and the last uses rule (ξ). □

When we evaluate a λ-term, it may happen that it has more than one β-redex, and we need to choose the redex that we want to reduce at that moment. In particular, two well-known strategies to select redexes are the *leftmost-outermost reduction* and the *rightmost-innermost reduction*. As the name suggests, in the first one we choose to reduce the *leftmost-outermost redex*, and in the second one we reduce the *rightmost-innermost redex*. Let us illustrate these two strategies at work in the example of the λ-term $M_0 = (\lambda x \cdot xx)((\lambda y \cdot y)(\lambda z \cdot z))$. In the example below, ate each reduction step, we will color in red the λ-abstraction of the selected redex and in blue its argument.

Example 2. *Recall $M_0 = (\lambda x \cdot xx)((\lambda y \cdot y)(\lambda z \cdot z))$. The **leftmost-outermost reduction** of M_0 is as follows:*

$(\lambda x \cdot xx)((\lambda y \cdot y)(\lambda z \cdot z))$
$\to ((\lambda y \cdot y)(\lambda z \cdot z))((\lambda y \cdot y)(\lambda z \cdot z))$
$\to (\lambda z \cdot z)((\lambda y \cdot y)(\lambda z \cdot z))$
$\to (\lambda y \cdot y)(\lambda z \cdot z)$
$\to \lambda z \cdot z$

*Let us now see the **rightmost-innermost reduction** of M_0:*

$(\lambda x \cdot xx)((\lambda y \cdot y)(\lambda z \cdot z))$

$\to (\lambda x \cdot xx)(\lambda z \cdot z)$

$\to (\lambda z \cdot z)(\lambda z \cdot z)$

$\to \lambda z \cdot z$

As we have just illustrated, there may be different ways of evaluating a λ-term. Therefore, β-reduction is *non-deterministic*. An interesting question to pose is: does the way one chooses the β-redex to reduce in the evaluation of a λ-term changes "the final result"? The answer will be "no" [16]. A fundamental result of λ-calculus is the Church-Rosser Theorem establishing that β-reduction is *confluent* (and for this reason is also called the Confluence Theorem):

Theorem 1. ***(Church-Rosser Theorem):*** *For all M, M_1, M_2 in Λ, if $M \to_\beta^* M_1$ and $M \to_\beta^* M_2$, then there exists N in Λ, such that $M_1 \to_\beta^* N$ and $M_2 \to_\beta^* N$.*

This property of $\to_\beta^*$ is also known as the *diamond property*, because it can be depicted graphically as follows [21]:

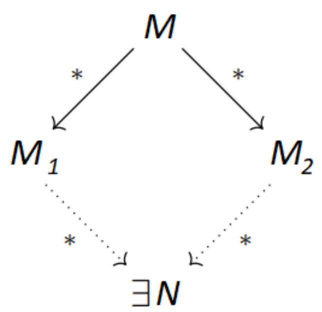

Figure 1: Diamond Property

Since the Church-Rosser Theorem is fundamental in the theory λ-calculus, we can find in the literature multiple proofs of this theorem, such as [5], and we omit it here.

Another important concept in λ-calculus is *normalization*.

Definition 6. *A λ-term N is called a* ***(β-)normal form*** *of a λ-term M when $M \to_\beta^* N$ and N is a β-normal form. A λ-term M is* ***(β-)normalizing*** *when it has a normal form.*

From the Church-Rosser Theorem, we can easily conclude that if a λ-term has a normal form, then this normal form is unique. However, not all terms have normal form. One well-known example is as follows.

Example 3. *Consider the λ-term $\Omega = (\lambda x \cdot xx)(\lambda x \cdot xx)$. When we evaluate Ω the only possible reduction sequence is:*

$$\begin{aligned}\Omega &= (\lambda x \cdot xx)(\lambda x \cdot xx) \\ &\rightarrow_\beta (\lambda x \cdot xx)(\lambda x \cdot xx) \ \ (= \Omega) \\ &\rightarrow_\beta (\lambda x \cdot xx)(\lambda x \cdot xx) \ \ (= \Omega) \\ &\rightarrow_\beta \cdots\end{aligned}$$

As we saw in Example 2, using the rightmost-innermost strategy we arrived at the normal form of the term M_0 in fewer steps than the leftmost-outermost. However in some cases, the former strategy will not even discover the normal form of a λ-term, contrary the leftmost-outermost strategy, as we will illustrate in the next example. Again, in the example we color the λ-abstraction of the redex in red and the argument in blue.

Example 4. *Let us consider the λ-term $M_1 = (\lambda y \cdot z)((\lambda x \cdot xx)(\lambda x \cdot xx))$.*

*Using **leftmost-outermost reduction**, a single step reduces to the normal form of M_1:*

$$(\lambda y \cdot z)((\lambda x \cdot xx)(\lambda x \cdot xx))$$
$$\rightarrow z$$

*Let us now consider the **rightmost-innermost reduction**. In one step, we reduce back to M_1 and this will repeat forever:*

$$(\lambda y \cdot z)((\lambda x \cdot xx)(\lambda x \cdot xx))$$
$$\rightarrow (\lambda y \cdot z)((\lambda x \cdot xx)(\lambda x \cdot xx))$$
$$\rightarrow \ldots$$

So, in this case, this strategy leads to non-termination of the evaluation process.

One last concept we will recall is:

Definition 7. *A λ-term M is **strongly normalizing** if any reduction sequence starting from M is finite.*

Note that not all terms have normal form.

It is obvious that strong normalization implies weak normalization: the last term of any finite sequence starting at a λ-term M must be a normal form and is therefore, the normal form of M. The reverse implication does not hold. Consider the counter-example below.

Example 5. *Consider the λ-term $MN\Omega$. Where $M = \lambda xy \cdot x$, $N = \lambda x \cdot x$ and $\Omega = (\lambda x \cdot xx)(\lambda x \cdot xx)$. We can obtain different reduction sequences starting from this term, depending on the redex that we evaluate first. We will give two different possible sequences.*

1. *One such reduction sequence is:*

$$
\begin{aligned}
MN\Omega &= (\lambda xy \cdot x)(\lambda x \cdot x)\Omega \\
&\rightarrow_\beta (\lambda yx \cdot x)\Omega \\
&\rightarrow_\beta \lambda x \cdot x
\end{aligned}
$$

   and $\lambda x \cdot x$ is a normal form. Therefore, $MN\Omega$ is weakly normalizing.

2. *Another possible way of evaluate the term $MN\Omega$ is to evaluate first Ω. But, as we have seen before, this term admits an infinite reduction sequence:*

$$
\begin{aligned}
MN\Omega &= (\lambda xy \cdot x)(\lambda x \cdot x)\Omega \\
&\rightarrow_\beta MN\Omega \\
&\rightarrow_\beta MN\Omega \\
&\rightarrow_\beta \cdots
\end{aligned}
$$

   so the term $MN\Omega$ is not strongly normalizing.

Observe that, in the previous example, the first reduction sequence obeys to the leftmost-outermost strategy, whereas the second one obeys to the rightmost-innermost strategy. So, $MN\Omega$ is an example of a λ-term whose normal form can be reached by the letfmost-outermost strategy, but not by the rightmost-innermost strategy.

In fact, there is a fundamental theorem in the reduction theory of λ-calculus, known as the *Leftmost Reduction Theorem* which establishes that, if a λ-term has a normal form, then the leftmost-outermost reduction strategy will find it [20].

2.3 Call-by-name and call-by-value

In the previous section we saw the concept of (full) β-reduction. As already observed, this reduction relation is non-deterministic, because a λ-term may have multiple β-redexes. Functional programming languages are based on β-reduction and, therefore, their implementations need to fix an *evaluation mechanism*, to tell which redex should be chosen at any given moment of the reduction process, turning this process deterministic. Typically, evaluation mechanisms express different policies for treating "function call", and have in mind efficiency considerations. Between all the calling mechanisms of functional programming, one can highlight two basic ones, namely, the call-by-name (cbn) and the call-by-value (cbv) mechanisms.

Call-by-name. This form of evaluation is also known as the *normal order reduction* and corresponds to the choice of the leftmost-outermost redex (in the sense we have exemplified before). Basically, it evaluates first the main expression and then the subexpressions [35]. But this form of evaluation can be too expensive in practice because it can be repeating the same reductions unnecessarily. Let us illustrate this with one example. Consider the λ-term $M_2 = (\lambda x \cdot ((xy)x)x)((\lambda z \cdot z)w)$. Its evaluation under call-by-name is as follows (we color in red and blue, respectively, the terms in function and in argument position of the redex being reduced at each step):

$$
\begin{aligned}
&(\lambda x \cdot ((xy)x)x)((\lambda z \cdot z)w)\\
&\quad \to (((\lambda z \cdot z)w)y)((\lambda z \cdot z)w))((\lambda z \cdot z)w)\\
&\quad \to (((wy)(((\lambda z \cdot z)w)))((\lambda z \cdot z)w)\\
&\quad \to (((wy)w)(((\lambda z \cdot z)w)))\\
&\quad \to ((wy)w)w \qquad \text{(4 steps)}
\end{aligned}
$$

So, the argument $(\lambda z \cdot z)w$ of M_2 ends up evaluated three times.

Call-by-value. This evaluation strategy is also known as *applicative order reduction* [35]. While in call-by-name evaluation reduction of the main expression is the first to occur, in call-by-value, basically, we evaluate the subexpressions first and only reduce the main application after reducing the internal redexes [35] (and so is closer to the spirit of rightmost-innermost reduction). More concretely, call-by-value evaluation requires an argument to be reduced to a value before a function can be applied to it. So, this evaluation mechanism can actually be defined on top of a restricted β-rule, namely:

$$\frac{}{(\lambda \cdot M)V \to M[V/x]}\ (\beta_v)$$

.

where V is a *value*, and only variables and λ-abstractions are considered to be values.

An advantage of the call-by-value strategy is that arguments are only evaluated once. For example, if we return to the λ-term M_2 above, under the call-by-value strategy, reduction will proceed as follows (again, at each step we color the redex):

$$
\begin{aligned}
&(\lambda x \cdot ((xy)x)x)((\lambda z \cdot z)w) \\
&\quad \to (\lambda x \cdot ((xy)x)x)w \\
&\quad \to ((wy)w)w \qquad \text{(2 steps)}
\end{aligned}
$$

So the reduction of $(\lambda z \cdot z)w$ is not repeated as above (in cbn). However, in call-by-value the arguments will always be evaluated, even when they will not be used. For example, consider the λ-term $M_3 = (\lambda x \cdot y)(\lambda wz \cdot wz)wz$. Its evaluation under call-by-value is as follows:

$$
\begin{aligned}
&(\lambda x \cdot y)(\lambda wz \cdot wz)wz \\
&\quad \to (\lambda x \cdot y)(\lambda z \cdot wzz) \\
&\quad \to \lambda z \cdot wzz \qquad \text{(2 steps)}
\end{aligned}
$$

Note that under call-by-name M_3 reduces in a single step to y.

Another important remark is that the concept of normal form under call-by-name and call-by-value is different. To better understand the differences, consider the example below.

Example 6. *Consider the λ-term $M_4 = (\lambda x \cdot x)(yz)$.*

Note that this term has no β_v-redex and is therefore a normal form with respect to call-by-value, but under call-by-name it reduces in one step to yz, resulting in a different normal form.

Chapter 3

λ-calculus and the Standardization Theorem

As illustrated in the previous chapter, there are different types of evaluation mechanisms for λ-terms. Throughout this dissertation, we will concentrate on the call-by-name variant of the λ-calculus, based on the ordinary β-rule of λ-calculus. In this Chapter we will define the *call-by-name evaluation relation*, and the *standard reduction relation* on λ-terms. In order to prove the Standardization Theorem, we will also establish several auxiliary properties concerning the two defined relations.

3.1 Call-by-name evaluation

We start by considering a sub-relation of $\rightarrow_\beta$ given by the closure of the β-rule under the closure rule (μ) only, i.e.:

Definition 8. $\rightarrow_n$ *(**one step call-by-name evaluation**) is the binary relation in λ-terms given inductively by:*

$$\frac{}{(\lambda x \cdot M)N \rightarrow_n M[N/x]}\ (\beta) \qquad \frac{M \rightarrow_n N}{MP \rightarrow_n NP}\ (\mu)$$

*The **call-by-name evaluation relation** is then the relation $\rightarrow_n^*$, i.e. the reflexive and transitive closure of $\rightarrow_n$.*

Example 7. *Let $M := (\lambda x \cdot x)y$, which is a β-redex. Whereas $Mz \rightarrow_n yz$ (with the help of closure rule (μ)), it is not the case $zM \rightarrow_n zy$. Of course, $zM \rightarrow_\beta zy$, but for this we need the closure rule (ν),* *which is not allowed for* $\rightarrow_n$.

An effect of the limitation to the closure rule (μ) is that β-reduction becomes deterministic that is call-by-name evaluation is a deterministic relation. The β-redex that can be reduced at one given moment corresponds to the *leftmost-outermost redex*, found in the function position of the given application.

Example 8. *Given λ-terms M_1, M_2,*

$$M_0 := (\lambda xy \cdot y)M_1M_2 \rightarrow_n (\lambda y \cdot y)M_2$$

Note that regardless of M_1 and M_2, only this reduction of M_0 is possible under call-by-name. Then,

$$(\lambda y \cdot y)M_2 \rightarrow_n M_2$$

and, again, (and regardless of M_2) this is the only possible way of continuing reduction under call-by-name.

We end this section establishing some properties of the call-by-name evaluation relation that will become useful later.

Lemma 3. *The following rule is admissible, that is, for all M_1, M_2, N in Λ:*

$$\frac{M_1 \rightarrow_n^* M_2}{M_1N \rightarrow_n^* M_2N}$$

Proof. The proof is by induction on $M_1 \rightarrow_n^* M_2$. In the base case of $\rightarrow_n$ we make use of the closure rule (μ). □

Lemma 4. *The following rules are admissible:*

$$\frac{M_1 \rightarrow_n M_2}{M_1[N/x] \rightarrow_n M_2[N/x]} \qquad \frac{M_1 \rightarrow_n^* M_2}{M_1[N/x] \rightarrow_n^* M_2[N/x]}$$

Proof. The proof of the admissibility of the first rule is an induction on $M_1 \rightarrow_n M_2$. The (β) case of $\rightarrow_n$ uses the Substitution Lemma 1. The proof of admissibility of the second one is by induction on $M_1 \rightarrow_n^* M_2$. The base case relative to $\rightarrow_n$, follows immediately from the first admissible rule. □

3.2 Standardization relation and admissible rules

The Standardization Theorem establishes that M reduces to N if and only if M reduces to N in a standard way. The specification of *reducing in a standard way* can be made by using an inductive definition of a binary relation of *standard reduction*. This approach has been independently by Loader [23] and by Joachimski-Matthes [19].

In this dissertation we will follow this approach, and characterize what reductions are accepted as standard by axiomatizing the relation "reduces in a standard way", as a binary relation on λ-terms, to which we call the *standard reduction relation*, and for this we will actually follow directly what is done in [33].

It should be noted that, as in [33], we will consider a standard reduction relation defined on the original syntax on λ-terms, rather than on a syntax of λ-terms where the application constructor can act on a list of arguments, as is done in [19, 23].

Definition 9. *The **standard reduction relation** is the binary relation on λ-terms, which we denote by $\Rightarrow_n$, and is inductively defined by:*

$$\frac{}{x \Rightarrow_n x}\ VAR \qquad \frac{M \Rightarrow_n N}{\lambda x \cdot M \Rightarrow_n \lambda x \cdot N}\ ABS \qquad \frac{M \Rightarrow_n M' \quad N \Rightarrow_n N'}{MN \Rightarrow_n M'N'}\ APL$$

$$\frac{M \rightarrow_n^* \lambda x \cdot M' \quad M'[N/x] \Rightarrow_n P}{MN \Rightarrow_n P}\ RDX$$

Now we make some remarks of standard reduction rules. The key rule is RDX. In this rule, we reduce under call-by-name evaluation the λ-term M that is in the function position until we find an abstraction $(\lambda x \cdot M')$. Hence, a β-redex $(\lambda x \cdot M')N$ is found and can be contracted to $M'[N/x]$. Then, if this contractum reduces in a standard way to a λ-term P, the original application of MN also reduces in a standard way to P. Note also that in the APL rule, we can choose to reduce "first" $M \Rightarrow_n M'$, or $N \Rightarrow_n N'$, but these reductions can also be done in parallel.

Recall the two reduction steps in Example 8 leading from $M_0 := (\lambda xy \cdot y)M_1M_2$ to M_2 and for simplicity fix M_2 to be w respectively. This reduction sequence is actually associated to a standard reduction, and hence we have $M_0 \Rightarrow_n w$, which can be justified by the following derivation:

$$\frac{\dfrac{}{(\lambda xy \cdot y)M_1 \rightarrow_n^* \lambda y \cdot y}\ (\beta) \qquad \dfrac{}{w \Rightarrow_n w}\ VAR}{(\lambda xy \cdot y)M_1 w \Rightarrow_n w}\ RDX$$

In order to prove the Standardization Theorem, we will first show the admissibility of the rules for the standard reduction relation in Figure 2. In fact, in the proof of the Standardization Theorem, we will only use directly rules (1), (7) and (8). However, to prove the admissibility of (7) we will use the remaining rules.

$$\frac{}{M \Rightarrow_n M}(1) \quad \frac{M \Rightarrow_n M' \quad N \Rightarrow_n N'}{M[N/x] \Rightarrow_n M'[N'/x]}(2) \quad \frac{M \to_n N \Rightarrow_n P}{M \Rightarrow_n P}(3)$$

$$\frac{M \to_n^* N \Rightarrow_n P}{M \Rightarrow_n P}(4) \quad \frac{M \Rightarrow_n \lambda x \cdot M' \quad N \Rightarrow_n N'}{MN \Rightarrow_n M'[N'/x]}(5)$$

$$\frac{M \Rightarrow_n (\lambda x \cdot M')N'}{M \Rightarrow_n M'[N'/x]}(6) \quad \frac{M \Rightarrow_n N \to_\beta P}{M \Rightarrow_n P}(7) \quad \frac{M \Rightarrow_n N \to_\beta^* P}{M \Rightarrow_n P}(8)$$

Figure 2: Admissible rules for $\Rightarrow_n$

The following lemmata establish the admissibility of rules in Figure 2. More detailed proofs of these lemmas can be found in Appendix B.

Lemma 5. *The rules* (1) *and* (2) *of Figure 2 are admissible.*

Proof. The proof of the admissibility of rule (1) is by an easy induction on M. The other one, (2), is by induction on $M \Rightarrow_n M'$. The RDX case requires the Substitution Lemma and uses the second admissible rule of Lemma 4. □

Lemma 6. *The rules* (3) *and* (4) *of Figure 2 are admissible.*

Proof. The proof of the admissibility of (3) is by induction on $M \to_n N$. The (β) case of $\to_n$ we make use of the RDX rule. The (μ) case explores all the possible subcases of the hypothesis $N \Rightarrow_n P$. The admissibility of (4) is proved by induction on $M \to_n^* N$. The base case of $\to_n^*$ we make use of rule (3). □

Lemma 7. *The rules* (5) *and* (6) *of Figure 2 are admissible.*

Proof. The proof of the admissibility of (5) is by induction on $M \Rightarrow_n \lambda x \cdot M'$. The cases VAR and APL are impossible. ABS case uses rules (2) and (4). RDX uses the first point of Lemma 2 and rule (4). The admissibility of (6) is proved by induction on $M \Rightarrow_n (\lambda x \cdot M')N'$. The VAR and ABS cases are impossible. Use is made of (5) in the APL case. □

Lemma 8. *The rules* (7) *and* (8) *of Figure 2 are admissible.*

Proof. The proof of the admissibility of (7) is by induction on $M \Rightarrow_n N$. Use is made of (6). The admissibility of (8) is proved by induction on $N \to_\beta^* P$. The base case of $\to_\beta^*$ requires rule (7). □

As we mentioned before, our proof of the Standardization Theorem can be extracted from the proof in [33] of standardization for a λ-calculus for modal logic, namely λ_b-calculus. This proof identifies a collection of admissible rules for the standard relation for λ_b ($\Rightarrow_b$) in Figure 9 on [33]. Note that we can obtain from these rules the rules in Figure 2 by: replacing $\Rightarrow_b$ by $\Rightarrow_n$, $\to_{we}$ by $\to_n$, $\to_{\beta_b}$ by $\to_\beta$, and omitting the modal constructors box and the ϵ. Note however two differences. Our rule (8) has no corresponding rule in Figure 9 in [33], but it is just a matter of convenience because it is immediately obtained by induction once we have rule (7). The second difference is that in our proof we found no need to use a rule corresponding to rule (2) of [33] so we have omitted such rule. More interestingly, it should be remarked that whereas the proof of rule (6) of [33] (corresponding to our rule (5)) uses a subinduction on $N \Rightarrow_b box(N')$, in our (simplified) setting of the ordinary λ-calculus we found no need to such subinduction.

3.3 Standardization Theorem

Now we are ready to prove the Standardization Theorem. On the one hand, we will show soundness of standard reduction, i.e., if M standardly reduces to N, then M β-reduces to N. The true content of the Standardization Theorem is however the converse, establishing that whenever a term N can be reached by β-reduction from a term M it is in relation to M through standard reduction.

Theorem 2. ***(Standardization Theorem)*** *For all* M, N *in* Λ, $M \to_\beta^* N$ *iff* $M \Rightarrow_n N$.

Proof. The "if" direction (soundness) follows by induction on $M \Rightarrow_n N$.

The VAR case just uses the fact that $\to_\beta^*$ denotes the reflexive, transitive closure of $\to_\beta$, that in particular is reflexive.

In the ABS case, $M = \lambda x \cdot M'$ and $N = \lambda x \cdot N'$, for some x in V and M', N' in Λ and $M' \Rightarrow_n N'$. From the induction hypothesis $M' \to_\beta^* N'$ and by the third point of Lemma 2 we obtain $\lambda x \cdot M' \to_\beta^* \lambda x \cdot N'$.

In the APL case, $M = M'N'$ and $N = M''N''$, for some M', N', M'', N'' in Λ and $M' \Rightarrow_n M''$ and $N' \Rightarrow_n N''$. By induction hypothesis $M' \rightarrow^*_\beta M''$. Then, by the first point of Lemma 2, follows:

$$\begin{aligned} M'N' &\rightarrow^*_\beta M''N' \\ &\rightarrow^*_\beta M''N'' \end{aligned}$$

The last relation is justified by induction hypothesis $N' \rightarrow^*_\beta N''$ and by the second point of Lemma 2. Finally we conclude $M'N' \rightarrow^*_\beta M''N''$ by using the fact that $\rightarrow^*_\beta$ denotes the reflexive and transitive closure of $\rightarrow_\beta$, which in particular is transitive.

In the RDX case, $M = QS$, for some Q, S in Λ and $Q \rightarrow^*_n \lambda x \cdot Q'$ (for some x, Q') and $Q'[S/x]\,_nN$. By induction hypothesis $Q'[S/x] \rightarrow^*_\beta N$. From the hypothesis $Q \rightarrow^*_n \lambda x \cdot Q'$, follows $Q \rightarrow^*_\beta \lambda x \cdot Q'$ by the fact that $\rightarrow^*_n \subseteq \rightarrow^*_\beta$. Then by the first point of Lemma 2 follows:

$$\begin{aligned} QS &\rightarrow^*_\beta (\lambda x \cdot Q')S \\ &\rightarrow_\beta Q'[S/x] \\ &\rightarrow^*_\beta N \end{aligned}$$

The last relation is justified by induction hypothesis and the previous one is of course, justified by the rule (β). Then we conclude $QS \rightarrow^*_\beta N$ using the fact that $\rightarrow^*_\beta$ is transitive.

Now we tern to the "only if"direction (completeness). Having shown the admissibility of rules (1), (7) and (8) of Figure 2, this will be a simple induction on $M \rightarrow^*_\beta N$.

In the base case relative to $\rightarrow_\beta$, we have the hypothesis $M \rightarrow_\beta N$. Since from rule (1) $M \Rightarrow_n M$, by applying (7) we obtain $M \Rightarrow_n N$.

The reflexive case follows immediately from rule (1).

In the transitive case, we have the hypotheses $M \rightarrow^*_\beta P$ and $P \rightarrow^*_\beta N$. From $M \rightarrow^*_\beta P$ follows by induction hypothesis $M \Rightarrow_n P$. Applying rule (8) with $M \Rightarrow_n P$ and $P \rightarrow^*_\beta N$ we obtain $M \Rightarrow_n N$.

□

Remark 2. *From the proof that $\Rightarrow_n$ is contained in $\rightarrow^*_\beta$ (soudness) we can extract a notion of standard reduction sequence. A good example of this idea is found by looking back to the RDX case. In this case we have argued that QS reduces in a standard way to N. For this we implicitly build a sequence of reductions*

$$QS \rightarrow^*_\beta (\lambda x \cdot Q')S \rightarrow_\beta Q'[S/x] \rightarrow^*_\beta N$$

which will be standard, once we impose $QS \to_\beta^* (\lambda x \cdot Q')S$ *and* $Q'[S/x] \to_\beta^* N$ *are themselves standard. This idea will be fully developed in Chapter 5.*

An immediate corollary of the Standardization Theorem is transitivity of the relation $\Rightarrow_n$, which will be useful later in Chapter 5.

Corollary 1. ***(Transitivity of*** $\Rightarrow_n$***)*** *For all* M, P, N *in* Λ, *if* $M \Rightarrow_n P$ *and* $P \Rightarrow_n N$, *then* $M \Rightarrow_n N$.

Proof. We have by hypothesis $P \Rightarrow_n N$. By the Standardization Theorem follows immediately $P \to_\beta^* N$. Then we apply (8) to obtain $M \Rightarrow_n N$.

□

Chapter 4

Formalization in Coq of the Standardization Theorem

In this chapter, we will provide a complete formalization of the proof of the Standardization Theorem developed in the previous chapter. As already mentioned, when we want to formalize meta-theoretic results of the λ-calculus or, in general, of languages which allow binders, we need to find a way to take into account that expressions should be treated up to renaming of bound variables. There are several techniques to address this question. For example, Section 2 of [3] offers an interesting survey of such techniques. In this survey, the techniques are classified as “concrete” (also called “first-order”) or as “higher-order” approaches, basically depending on whether variables acquire a concrete first-order representation, typically based on natural numbers or names, or whether binders are represented as meta-language functions in an higher-order setting. Concrete approaches include the “named representation”, the usual approach followed on paper, where names are used to represent variables, but then requires to work under α-equivalence, which raises difficulties in formal developments, like the fact that capture-avoiding substitution cannot be given by structural recursion. An approach still with names that avoids this particular difficulty is found in [11] and is based on *multiple substitution* [38], an operation that can be given by structural recursion and where bound variables are always renamed in parallel with substitutions. Another concrete approach (used since the early efforts of formalization of languages with binders) is the *De Bruijn indices* technique, where variables are represented by natural numbers indicating its depth relatively to its binder [9]. Another concrete approach is the so-called *locally nameless* technique, where free variables are represented by names, but bound variables are represented through De Bruijn indices, attempting to conjoin benefits of both named and De Bruijn indices techniques [3]. *Higher-order abstract syntax* is a prototypical example of a higher-order approach to binding representation [29]. In this representation, a λ-abstraction is represented as an higher-order function, whose argument is a function that can be thought of as a function ready to substitute an argument passed to the body of the abstraction.

In this dissertation we have chosen to use the De Bruijn indices technique for the representation of binders. Since this is a widely used technique in formalization of meta-theoretic results of the λ-calculus and our prior experience with proof assistants was rather short, it was very useful to benefit from the enormous collection of material available in the literature on this technique. In this sense, it was possible to adapt to our setting the formalization of essential concepts and results concerning the syntax of λ-terms, the β-reduction rule and the Substitution Lemma. For this, we followed closely Huet [18] for the basic definitions around the syntax of λ-terms and of β-reduction, but we also directly profited from other works, such as [27], by Nipkow, and [8], by Berghofer-Urban, for the formalization of the Substitution Lemma. But, as we progressed in our formalization effort, it turned out that, once we defined all the basic infrastructure around de Bruijn indices, we could follow very closely the structure of the proof of the Standardization Theorem with ordinary λ-terms, both in what concerns lemmata and the inductive structure of arguments.

4.1 A λ-calculus with De Bruijn indices

In this section we will introduce a λ-calculus with the De Bruijn indices, that we named λ_{dB}, which we use in our formalization. Throughout this section, together with the definition of basic concepts of λ_{dB}, we immediately present their respective formalizations in Coq. In Section 2.1, we defined λ-terms and the respective substitution operation. In this section will do the same but for the corresponding concepts using the De Bruijn indices. In a first contact, λ-terms with the De Bruijn indices are not so intuitive to understand and the substitution operation becomes rather complex. For this reason we will present quite some examples throughout the section.

Definition 10. *The set of **λ-terms with the De Bruijn indices**, Λ_{dB}, is defined inductively by:*

1. $i \in \Lambda_{dB} \quad (i \in \mathbb{N}_0)$;
2. $M \in \Lambda_{dB} \Rightarrow (\lambda \cdot M) \in \Lambda_{dB}$;
3. $M, N \in \Lambda_{dB} \Rightarrow (MN) \in \Lambda_{dB}$.

In the above definition, i (belonging to $\mathbb{N}_0$) is called a **De Bruijn index** and roughly corresponds to a variable of the λ-calculus. In $\lambda \cdot M$, M is said is the **scope** of the displayed occurrence of λ.

Using the Coq proof assistant we define the set of λ-terms Λ_{dB} inductively as follows:

```
Inductive lambda : Set :=
| Ref : nat → lambda
| Abs : lambda → lambda
| App : lambda → lambda → lambda.
```

Note that the constructor `Ref` is used to represent De Bruijn indices (resorting to the representation of $\mathbb{N}_0$ in Coq via `nat`), the constructor `Abs` is used to represent abstractions ($\lambda \cdot M$) and the constructor `App` is used to represent applications (MN).

Remark 3. *The conventions referred to in Remark 1 remain in this chapter, and for successive abstractions we will omit the* $\cdot$*. For example, the* λ*-term* $\lambda\lambda\lambda \cdot 013$ *abbreviates* $(\lambda \cdot (\lambda \cdot (\lambda \cdot 013)))$*, hence the scope of the third occurrence of* λ *is* 013*, the scope of the second occurrence of* λ *is* $\lambda \cdot 013$ *and the scope of the first occurrence of* λ *is* $\lambda \cdot (\lambda \cdot 013)$*.*

Definition 11. *An occurrence of an index* i *is said* ***bound*** *if it is inside the scope of an abstraction (*λ*), otherwise it is said* ***free****.*

To better understand the definition, consider the example below:

Example 9. *Consider the De Bruijn* λ*-term:* $(\lambda\lambda \cdot 01)0$*. As we can see, the red* λ*-binder (the first occurrence of* λ*) binds the only occurrence of index* 1 *and the blue one (the second* λ *occurrence) binds the first occurrence of index* 0*. The second occurrence of index* 0 *is a free one.*

The base β-reduction rule in λ_{dB} is given by:

$$(\lambda \cdot M)N \to M[0 := N] \ \ (\beta)$$

where $M[0 := N]$ stands for a substitution operation that wants to replace free occurrences of index 0 in M by N. This operation of substitution is tricky and complex:

1. as just said, in $M[0 := N]$ we want to replace by N, all free occurrences of index 0 in M;

2. however, we must take into account that inside M, if we have traversed k λ's, 0 will actually correspond to index k, and additionally we need to update the indices of N accordingly, in order to prevent index capture;

3. finally, we should keep in mind that, as the outer λ of $\lambda \cdot M$ is being removed, the indices in its scope should be decreased by 1.

To better understand the β-reduction rule, we describe it in detail below in one example.

Example 10. *Let M_0 be the following De Bruijn λ-term: $(\lambda\lambda \cdot 31(\lambda \cdot 02))(\lambda \cdot 50)$.*

Start by noting that in M_0 each occurrence of λ binds the index with the same colour, and the indices in black are free indices.

According to the β-rule M_0 reduces to $M[0 := N]$, where M is $\lambda\lambda \cdot 31(\lambda \cdot 02)$, and N is $\lambda \cdot 50$. First we have to find in M all free occurrences of index 0, as well as other occurrences of indices that also represent index 0. So, we will have to replace the occurrences of indices 1 and 2. Because the occurrence of 1 is inside the blue λ this occurrence should be replaced by $\lambda \cdot 60$ (note here that 5 was updated to 6 in N). Because the occurrence of 2 is inside the two λ's (the blue and purple occurrences), this occurrence should be replaced by $\lambda \cdot 70$ (again note that 5 was updated to 7 in N). Finally, the only free occurrence of an index in M, namely 3, should be decreased by 1. So the final result is: $\lambda \cdot 2(\lambda \cdot 60)(\lambda \cdot 0(\lambda \cdot 70))$.

As we describe above, in the course of the substitution operation some indices may need to be updated. To make these updates, we will define the *lifting operation* that will be denoted by $\Uparrow_k$. This operation updates the indices of free occurrences of indices across k levels of extra binders in term N, in order to avoid index capture. This operation is defined as follows:

Definition 12. *Given $k \in \mathbb{N}_0$, the **lifting function** $\Uparrow_k$ is defined recursively by:*

- $\Uparrow_k i = \begin{cases} i, & \text{if } i < k \\ i+1, & \text{otherwise} \end{cases}$

- $\Uparrow_k (\lambda \cdot M) = \lambda \cdot \Uparrow_{k+1} M$

- $\Uparrow_k (M_1 M_2) = \Uparrow_k M_1 \Uparrow_k M_2$

In $\Uparrow_k$ the parameter k will represent the number of λ's traversed. Notice that when the index is bound $(i < k)$, the index is not changed. When the index is free $(i \geq k)$, the corresponding index is lifted by 1. In our Coq development, the lifting function is implemented as follows:

```
Fixpoint lift_rec (L : lambda) : nat → lambda :=
fun k : nat ⇒
match L with
| Ref i ⇒ Ref (relocate i k)
| Abs M ⇒ Abs (lift_rec M (S k))
| App M N ⇒ App (lift_rec M k) (lift_rec N k)
end.

Definition lift (N : lambda) := lift_rec N 0.
```

In the Coq code above, `relocate i k` stands for the implementation of the function that returns the value i if $k > i$ and $i + 1$ otherwise. Also, we define in Coq `lift` to represent the special case $\Uparrow_0$ of the lifting operation.

Now that we have defined the lifting function, we will turn to the definition of the *substitution function*.

Definition 13. *For De Bruijn λ-terms M, N and De Bruijn index k the* ***substitution function*** $M[k := N]$ *is recursively defined by:*

- $i[k := N] = \begin{cases} i - 1, \text{ if } k < i \\ N, \text{ if } k = i \\ i, \text{ if } k > i \end{cases}$
- $(\lambda \cdot M_1)[k := N] = \lambda \cdot M_1[k + 1 :=\Uparrow_0 N]$
- $(M_1 M_2)[k := N] = M_1[k := N]M_2[k := N]$

Note that in the case of $M[k := N]$ where M is index i we need to compare indices i and k and we can have one more option than in the variable case of substitution with ordinary λ-terms. The additional case corresponds to $k < i$ where we decrease i by 1 because, as explained before, this substitution operation will be used in the context of β-reduction.

In the Coq code below to implement the substitution function we use an auxiliary function `insert_Ref` to perform all the action needed at the base case of substitution:

```

Definition insert_Ref (N : lambda) (i k : nat) :=
match compare k i with

 (* k<i *) | inleft (left _) ⇒ Ref (pred i)
 (* k=i *) | inleft _ ⇒ N
 (* k>i *) | _ ⇒ Ref i
end.

Fixpoint subst_rec (L : lambda) : lambda → nat → lambda :=
fun (N : lambda) (k : nat) ⇒
match L with
| Ref i ⇒ insert_Ref N i k
| Abs M ⇒ Abs (subst_rec M (lift_rec N 0) (S k))
| App M M' ⇒ App (subst_rec M N k) (subst_rec M' N k)
end.

Definition subst (N M : lambda) := subst_rec M N 0.
```

Recall that $\rightarrow_\beta$ stands for the compatible closure of the base β-rule. In λ_{dB}, $\rightarrow_\beta$ is defined analogously, and now the closure rules are as follows:

$$\frac{M \rightarrow N}{MP \rightarrow NP}\ (\mu) \qquad \frac{M \rightarrow N}{PM \rightarrow PN}\ (\nu) \qquad \frac{M \rightarrow N}{\lambda \cdot M \rightarrow \lambda \cdot N}\ (\xi)$$

In Coq the representation of $\rightarrow_\beta$ for De Bruijn λ-terms is as follows:

```
Inductive red1 : lambda → lambda → Prop :=
| beta : forall M N : lambda, red1 (App (Abs M) N) (subst N M)
| abs_red : forall M N : lambda, red1 M N → red1 (Abs M) (Abs N)
| app_red_l :
    forall M1 N1 : lambda,
```

```
    red1 M1 N1 → forall M2 : lambda, red1 (App M1 M2) (App N1 M2)
  | app_red_r :
    forall M2 N2 : lambda,
    red1 M2 N2 → forall M1 : lambda, red1 (App M1 M2) (App M1 N2).
```

In particular, note the encoding of the base β-rule of λ_{dB}, making use of the `subst` Coq function defined before as a particular case of substitution in De Bruijn λ-terms.

Next we will see a representation of the $\longrightarrow_n$ relation for λ_{dB}. Recall that $\longrightarrow_n$ should correspond to a sub-relation of $\longrightarrow_\beta$, obtained by closing the base β-rule under rule (μ) only.

```
 (* → n *)
Inductive name_eval_1 : lambda → lambda → Prop :=
  | beta_name_eval : forall M N : lambda, name_eval_1 (App (Abs M) N) (subst N M)
  | app_red_name_eval_1 :
    forall M1 N1 : lambda,
    name_eval_1 M1 N1 → forall M2 : lambda, name_eval_1 (App M1 M2) (App N1 M2).
```

The relation $\longrightarrow_\beta^*$ for De Bruijn λ-terms (as for ordinary λ-terms) is the reflexive-transitive closure of $\longrightarrow_\beta$ and can thus be represented in Coq as follows:

```
 Inductive red : lambda → lambda → Prop :=
  | one_step_red : forall M N : lambda, red1 M N → red M N
  | refl_red : forall M : lambda, red M M
  | trans_red : forall M N P : lambda, red M N → red N P → red M P.
```

Finally, call-by-name evaluation for De Bruijn λ-terms is the reflexive-transitive closure of $\longrightarrow_n$ (like for ordinary λ-terms) and it is represented in Coq by:

```
(* Transitive closure of → n *)
Inductive name_eval : lambda → lambda → Prop :=
  | one_step_name_eval : forall M N : lambda, name_eval_1 M N → name_eval M N
  | refl_name_eval : forall M : lambda, name_eval M M
```

```
| trans_name_eval : forall M N P : lambda, name_eval M N → name_eval N P → name_eval M P.
```

4.2 The Substitution Lemma

An important and well-known result of the λ-calculus is the Substitution Lemma. We already proved this lemma in a previous chapter (Lemma 1). However, its statement for De Bruijn λ-terms is subtler, and a proof of it becomes very involved as we will see below. In the organization of the proof of the Substitution Lemma shown here we followed closely [27], but this proof also profited from the argument for this lemma in [8].

As we have already mentioned, the substitution operation uses an auxiliary lifting function ($\Uparrow_k$). The Substitution Lemma will require the next three auxiliary lemmas involving the lifting function. After each of these lemmas we show the respective formalization in Coq of its statement.

Lemma 9. *For all M, N in Λ_{dB} and k in $\mathbb{N}_0$, $(\Uparrow_k M)[k := N] = M$.*

Proof. By an easy induction on M. □

```
Lemma prop_1 : forall M N : lambda, forall k : nat, subst_rec (lift_rec M k) N k = M.
```

Lemma 10. *For all M in Λ_{dB} and k, i in $\mathbb{N}_0$, if $i \geq k$, then $\Uparrow_{i+1} (\Uparrow_k M) =\Uparrow_k (\Uparrow_i M)$.*

Proof. By an easy induction on M. □

```
Lemma prop_2 : forall M : lambda, forall k i : nat, k<=i → lift_rec (lift_rec M k) (S i) =
lift_rec (lift_rec M i) k.
```

Lemma 11. *For all M, N in Λ_{dB} and k, i in $\mathbb{N}_0$, if $i \geq k$, then $\Uparrow_k (M[i := N]) = (\Uparrow_k M)[i+1 :=\Uparrow_k N]$.*

Proof. By an easy induction on M. Use is made of Lemma 10 in the abstraction case. □

```
Lemma prop_3 : forall M N : lambda, forall k i : nat, k<=i → lift_rec (subst_rec M N i) k =
subst_rec (lift_rec M k) (lift_rec N k) (S i).
```

The next lemma will not be used in the proof of the Substitution Lemma, however it will be useful later.

Lemma 12. *For all M, N in Λ_{dB} and k, i in $\mathbb{N}_0$, if $i \geq k$, then $\Uparrow_i (M[k := N]) = (\Uparrow_{i+1} M)[k :=\Uparrow_i N]$.*

Proof. By an easy induction on M. Again, the abstraction case uses Lemma 10. □

```
Lemma prop_4 : forall M N : lambda, forall k i : nat, k<=i → lift_rec (subst_rec M N k) i =
subst_rec (lift_rec M (S i)) (lift_rec N i) k.
```

Now we are ready to prove the Substitution Lemma for De Bruijn λ-terms (Lemma 13 below). In order to help understanding its statement, we recall first the Substitution Lemma for ordinary λ-terms:

if $x \neq y$ and x not free in Q, then $(M[N/x])[Q/y] = (M[Q/y])[N[Q/y]/x]$.

A direct comparison of the two statements shows that in the De Bruijn case we need (additionally) to increase by one the index for the inner substitution and lift by k the free indices of Q. Note also that the statement only holds for De Bruijn indices $i \geq k$.

Lemma 13. *(**Substitution Lemma for De Bruijn λ-terms**) For all M, N, Q in Λ_{dB} and i, k in $\mathbb{N}_0$, if $i \geq k$, then*

$$M[k := N][i := Q] = M[i + 1 :=\Uparrow_k Q][k := N[i := Q]]$$

Proof. By induction on M. The index case requires Lemma 9. In the abstraction case, use is made of Lemmas 10 and 11. □

The formalization in Coq of the statement of the Substitution Lemma is thus:

```
Lemma substitution_lemma : forall M N Q : lambda, forall i k : nat, k<=i →
subst_rec (subst_rec M N k) Q i =
subst_rec (subst_rec M (lift_rec Q k) (S i)) (subst_rec N Q i) k.
```

4.3 Standard reduction relation and admissible rules

This section corresponds to Section 3.2, but now using De Bruijn λ-terms. In particular, we will establish that the rules for standard reduction in Section 3.2 (Figure 2) have admissible analogous for De Bruijn λ-terms. Furthermore, the proofs of the latter are similar to those in Section 3.2, with the exception of rule (2), which will require some new auxiliary lemmas.

We start with the analogue to Lemma 2 for De Bruijn λ-terms. We will omit its proof, as it follows directly the proof of the mentioned lemma (an induction on $M \rightarrow_{\beta}^{*} M'$):

Lemma 14. *For all M, M', N in Λ_{dB}, if $M \rightarrow_{\beta}^{*} M'$ then:*

1. $MN \rightarrow_{\beta}^{*} M'N$
2. $NM \rightarrow_{\beta}^{*} NM'$
3. $\lambda \cdot M \rightarrow_{\beta}^{*} \lambda \cdot M'$

In Coq this lemma reads as follows:

```
Lemma right_apl_red : forall M1 M2 N : lambda, red M1 M2 → red (App M1 N) (App M2 N).

Lemma left_apl_red : forall M1 M2 N : lambda, red M1 M2 → red (App N M1) (App N M2).

Lemma center_abs_red : forall M1 M2 : lambda, red M1 M2 → red (Abs M1) (Abs M2).
```

Now, we define the standard reduction relation $\Rightarrow_n$ for De Bruijn λ-terms:

Definition 14. $\Rightarrow_n$ *for De Bruijn λ-terms is given inductively by the following rules:*

$$\frac{}{i \Rightarrow_n i}\ VAR \qquad \frac{M \Rightarrow_n N}{\lambda \cdot M \Rightarrow_n \lambda \cdot N}\ ABS \qquad \frac{M \Rightarrow_n M' \quad N \Rightarrow_n N'}{MN \Rightarrow_n M'N'}\ APL$$

$$\frac{M \rightarrow_n^{*} \lambda \cdot M' \quad M'[0 := N] \Rightarrow_n P}{MN \Rightarrow_n P}\ RDX$$

The formalization in Coq is thus:

```
  (* Standard reduction ⇒ n *)
Inductive standard_red : lambda → lambda → Prop :=
  | VAR : forall i : nat, standard_red (Ref i) (Ref i)
  | ABS : forall M N : lambda, standard_red M N → standard_red (Abs M) (Abs N)
  | APL : forall M1 M2 N1 N2 : lambda, standard_red M1 M2 → standard_red N1 N2 →
  standard_red (App M1 N1) (App M2 N2)
  | RDX : forall M1 M2 N P : lambda, name_eval (M1) (Abs M2) → standard_red (subst N M2) (P)
  → standard_red (App M1 N) (P).
```

The proof of the Standardization Theorem for De Bruijn λ-terms will follow directly the proof of this result for ordinary λ-terms. Figure 3 shows the collection of rules about the standard reduction relation for De Bruijn λ-terms that will play the role of the respective rules in Figure 2 for ordinary λ-terms. As anticipated, the proofs of the admissibility of the rules in Figure 3 are very similar to the proofs of the admissibility for the corresponding rules for ordinary λ-terms. For this reason we will omit details of the proofs of the rules in Figure 3, except for rule (2) which shows relevant differences. Indeed, to prove the admissibility of rule (2), we need the collection of auxiliary lemmas shown in Figure 4, whose admissibility is established in the following three lemmas.

$$\frac{}{M \Rightarrow_n M}(1) \quad \frac{M \Rightarrow_n M' \quad N \Rightarrow_n N'}{M[i := N] \Rightarrow_n M'[i := N']}(2) \quad \frac{M \rightarrow_n N \Rightarrow_n P}{M \Rightarrow_n P}(3)$$

$$\frac{M \rightarrow_n^* N \Rightarrow_n P}{M \Rightarrow_n P}(4) \quad \frac{M \Rightarrow_n \lambda \cdot M' \quad N \Rightarrow_n N'}{MN \Rightarrow_n M'[0 := N']}(5)$$

$$\frac{M \Rightarrow_n (\lambda \cdot M')N'}{M \Rightarrow_n M'[0 := N']}(6) \quad \frac{M \Rightarrow_n N \rightarrow_\beta P}{M \Rightarrow_n P}(7) \quad \frac{M \Rightarrow_n N \rightarrow_\beta^* P}{M \Rightarrow_n P}(8)$$

Figure 3: Admissible rules of $\Rightarrow_n$ for λ_{dB}

Lemma 15. *The rules (aux_1) and (aux_2) on Figure 4 are admissible.*

Proof. The proof of the admissibility of rule (aux_1) is by induction on $M_1 \rightarrow_n M_2$. The (β) case uses

$$\frac{M_1 \to_n M_2}{(\Uparrow_k M_1) \to_n (\Uparrow_k M_2)}(aux_1) \quad \frac{M \to_n^* N}{(\Uparrow_k M) \to_n^* (\Uparrow_k N)}(aux_2) \quad \frac{M \Rightarrow_n N}{(\Uparrow_k M) \Rightarrow_n (\Uparrow_k N)}(aux_3)$$

$$\frac{M_1 \to_n M_2}{M_1[i{:=}N] \to_n M_2[i{:=}N]}(aux_4) \quad \frac{M_1 \to_n^* M_2}{M_1[i{:=}N] \to_n^* M_2[i{:=}N]}(aux_5)$$

Figure 4: Auxiliary admissible rules for λ_{dB}

Lemma 12. The (μ) case follows from induction hypothesis. The proof of the admissibility of rule (aux_2) is by induction on $M \to_n^* N$. The base case follows immediately from the rule (aux_1).

□

The statements in Coq of the admissibility of rules aux_1 and aux_2 are therefore:

```
Lemma lift_1: forall M N : lambda, forall i:nat, name_eval_1 M N →
name_eval_1 (lift_rec M i) (lift_rec N i).

Lemma lift_n: forall M N : lambda, name_eval M N → forall i :nat,
name_eval (lift_rec M i) (lift_rec N i).
```

Lemma 16. *Rule (aux_3) of Figure 4 is admissible.*

Proof. By induction on $M \Rightarrow_n N$. The VAR case follows from the admissible rule (1) on Figure 3 (proved ahead in Lemma 18). The RDX case requires Lemmas 15 and 12.

□

In Coq this lemma reads as follows:

```
Lemma lift_i : forall N1 N2 : lambda, standard_red N1 N2 → forall i: nat,
standard_red (lift_rec N1 i) (lift_rec N2 i).
```

The next two lemmas correspond to Lemma 4 for ordinary λ-terms and their proofs are similar to those constructed for the latter lemma.

Lemma 17. *The rules (aux_4) and (aux_5) of Figure 4 are admissible.*

Proof. The admissibility of aux_4 follows by induction on $M_1 \to_n M_2$ and needs the Substitution Lemma (Lemma 13). The admissibility of rule aux_5 follows by induction on $M_1 \to_n^* M_2$. □

The statements of the admissibility of these two rules in Coq is:

```
Lemma subs_name_eval_1 : forall M1 M2 N : lambda, forall i : nat, name_eval_1 M1 M2 →
name_eval_1 (subst_rec M1 N i) (subst_rec M2 N i).

Lemma subs_name_eval : forall M1 M2 N : lambda, forall i : nat, name_eval M1 M2 →
name_eval (subst_rec M1 N i) (subst_rec M2 N i).
```

The next four lemmas establish the admissibility of the rules in Figure 3, and each of them is followed by the respective codification in Coq. As said, the proofs of the admissibility of these proves are similar to those of the respective rules for ordinary λ-terms, and are therefore omitted. The only exception will be rule (2) which requires some of the admissible rules of Figure 4.

Lemma 18. *The rules* (1) *and* (2) *of Figure 3 are admissible.*

Proof. The proof of the admissibility of (2) is by induction on $M \Rightarrow_n M'$. The ABS case follows from rule (aux_3). The RDX case requires the Substitution Lemma (Lemma 13) plus rule (aux_5).

□

```
Lemma rule_1 : forall M : lambda, standard_red M M.

Lemma rule_2 : forall M1 M2 : lambda, standard_red M1 M2 → forall N1 N2 : lambda,
standard_red N1 N2 → forall i:nat, standard_red (subst_rec M1 N1 i) (subst_rec M2 N2 i).
```

Lemma 19. *The rules* (3) *and* (4) *of Figure 3 are admissible.*

```
Lemma rule_3 : forall M N : lambda, name_eval_1 M N → forall P : lambda, standard_red N P
→ standard_red M P.

Lemma rule_4 : forall M N P : lambda, name_eval M N → standard_red N P → standard_red M P.
```

Lemma 20. *The rules* (5) *and* (6) *of Figure 3 are admissible.*

```
Lemma rule_5 : forall M1 M2 N1 N2 : lambda, standard_red M1 (Abs M2) → standard_red N1 N2
→ standard_red (App M1 N1) (subst N2 M2).

Lemma rule_6 : forall M1 M3 N0 : lambda, standard_red M1 (App (Abs M3) (N0)) →
standard_red M1 (subst N0 M3).
```

In the Coq code above, recall that `subst N M` has been defined as `subst_rec M N 0`.

Lemma 21. *The rules* (7) *and* (8) *of Figure 3 are admissible.*

```
Lemma rule_7 : forall M N : lambda, standard_red M N → forall P : lambda, red1 N P →
standard_red M P.

Lemma rule_8 : forall M N P : lambda, standard_red M N → red N P → standard_red M P.
```

Theorem 3. ***(Standardization Theorem with Bruijn indices):*** *In* λ_{dB}, *for all* M, N *in* Λ_{dB},

$M \to_{\beta}^{*} N$ *iff* $M \Rightarrow_{n} N$.

Since the proof of this theorem is very similar to the proof of the Standardization Theorem developed in Section 3.3 for ordinary λ-terms, instead of giving its details we show directly its formalization in the Coq proof assistant. As mentioned before, this formalization follows very closely the structure of the proof on paper for λ_{dB}-terms:

```
Theorem standardization : forall M N : lambda, red M N ↔ standard_red M N.
Proof.
split.

(*"Only if" direction: *)
intro H. induction H.
(*Base case: *)
assert (H1: standard_red M M).
```

```
apply rule_1.
pose proof rule_7 as pp.
specialize pp with (1 := H1) (2 := H); trivial.
(*Reflexice case: *)
apply rule_1.
(*Transitive case: *)
pose proof rule_8 as pp.
specialize pp with (1 := IHred1) (2 := H0); trivial.

(*"If" direction: *)
intro H. induction H.
(* VAR case: M = Ref i and N = Ref i *)
apply refl_red.
(* ABS case: M = Abs M' and N = Abs N' *)
apply red_abs. trivial.
(* APL case: M = App M1 N1 and N = M2 N2 *)
assert (H1: red (App M1 N1) (App M2 N1)).
apply red_appl. trivial.
assert (H2: red (App M2 N1) (App M2 N2)).
apply red_appr. trivial.
apply trans_red with (App M2 N1). trivial. trivial.
(* RDX case: M = App M1 N*)
assert (H1: red M1 (Abs M2)).
induction H.
apply one_step_red.
induction H.
apply beta.
apply app_red_l. trivial.
apply refl_red.
apply trans_red with (N0); trivial.
assert (H2: red (App M1 N) (App (Abs M2) N)).
apply red_appl. trivial.
assert (H3: red1 (App (Abs M2) N) (subst N M2)).
```

```
apply beta.
assert (H4: red (subst N M2) P). trivial. apply trans_red with (App (Abs M2) N).
trivial. apply trans_red with (subst N M2).
apply one_step_red in H3.
 trivial. trivial.
Qed.
```

Transitivity of the relation $\Rightarrow_n$ is an immediate corollary of the Standardization Theorem, as for ordinary λ-calculus. Since the proof of this is also similar to the one for λ-calculus (Corollary 1), we omit it here.

Corollary 2. *For all M, P, N in Λ, if $M \Rightarrow_n P$ and $P \Rightarrow_n N$, then $M \Rightarrow_n N$.*

In Coq this corollary reads as follows:

```
Theorem rule_9 : forall M N P : lambda,
standard_red M N → standard_red N P → standard_red M P.
```

Chapter 5

Standard Reduction Sequences

In this dissertation, we approach standard reduction via an inductive binary relation on λ-terms. As mentioned in Section 3.2, we follow very closely Espírito Santo-Pinto-Uustalu [33] in order to define standard reduction ($\Rightarrow_n$) and to prove the Standardization Theorem. A more traditional approach to standard reduction is via standard reduction sequences, such as suggested by Plotkin [30]. In this chapter we will formalize the equivalence of these two approaches. We will start by developing in Section 5.1 the theory of reduction sequences (concepts and properties). Then in Section 5.2, we will formalize in Coq all their theory, as well as the equivalence of the two approaches to standard reduction.

5.1 Theory

Naturally, the representation of standard reduction sequences will be through lists of λ-terms. For our purpose, it suffices to consider finite lists of λ-terms. So:

Definition 15. *The set $L(\Lambda)$ of* ***lists of λ-terms*** *is defined inductively by the grammar:*

$$L \ ::= \ [] \ | \ M :: L$$

In the definition above, as throughout this section, M,N,P,M',M_1, etc will range over λ-terms. Also, we will assume that L,L',L_1,L_2, etc will range over lists of λ-terms.

As usual, given a list $M :: L$, we say the λ-term M is its head, and the list L is its tail.

The *appending* of two lists is defined as usual:

Definition 16. *Given lists of λ-terms L_1 and L_2, the* ***append*** *function App is recursively defined on lists by:*

$$App(L_1, L_2) = \begin{cases} L_2, & \textit{if } L_1 = [] \\ M :: App(L_1', L_2), & \textit{if } L_1 = M :: L_1' \end{cases}$$

A basic property of the append function needed below is associativity:

Lemma 22. *For all L_1, L_2, L_3 in $L(\Lambda)$,*

$$App(App(L_1, L_2), L_3) = App(L_1, App(L_2, L_3)).$$

Proof. By induction on the list L_1.

□

In what follows, we often represent the appending of lists by :: (using infix notation) and drop parentheses when there are successive append operations, restoring parentheses as convenient (since append is associative). Additionally, we often represent singleton lists by writing its unique λ-term. For example, for lists L_1, L_2 and for λ-term M, the notation $L_1 :: M :: L_2$ will represent the list $App(L_1, App(M :: [], L_2)) = App(App(L_1, M :: []), L_2)$.

Definition 17. *Given a list of λ-terms L, and a variable x, we define the function* ***Abs*** *by recursion on lists:*

$$Abs(x, L) = \begin{cases} [], & \textit{if } L = [] \\ \lambda x \cdot M :: Abs(x, L'), & \textit{if } L = M :: L' \end{cases}$$

So the $Abs(x, L)$ function prefixes each λ-term of L by the binder λx, which means that eventual free occurrences of x in λ-terms of L will become bound.

Definition 18. *Given a list L of λ-terms and a λ-term N, we define the function Apl_a by recursion on lists:*

$$Apl_a(L, N) = \begin{cases} [], & \textit{if } L = [] \\ MN :: Apl_a(L', N), & \textit{if } L = M :: L' \end{cases}$$

So, $Apl_a(L, N)$ creates an application MN out of each λ-term M in L.

Definition 19. *Given a list L of λ-terms and a λ-term M, we define the function Apl_f by recursion on lists:*

$$Apl_f(M, L) = \begin{cases} [], & \textit{if } L = [] \\ MN :: Apl_f(M, L'), & \textit{if } L = N :: L' \end{cases}$$

Analogously to Apl_a, $Apl_f(M, L)$ creates an application MN out of each λ-term N in L.

The next two lemmas will be useful later. They establish how the functions just defined interact with lists appending.

Lemma 23. *For all $L_1, L_2 \in L(\Lambda)$, and $x \in V$, $Abs(x, L_1 :: L_2) = Abs(x, L_1) :: Abs(x, L_2)$*

Proof. By induction on L_1.

□

Lemma 24. *For all $L_1, L_2 \in L(\Lambda)$,*

1. $Apl_f(M, L_1 :: L_2) = Apl_f(M, L_1) :: Apl_f(M, L_2)$
2. $Apl_a(L_1 :: L_2, M) = Apl_a(L_1, M) :: Apl_a(L_2, M)$

Proof. Both items follows by induction on L_1.

□

Now we are ready to define standard reduction sequences. We follow Plotkin's definition [30].

Definition 20. ***Standard reduction sequences (s.r.s.)*** *is a predicate on lists of λ-terms given inductively by:*

$$\frac{}{x\ \ s.r.s.}\ VAR' \qquad \frac{L\ \ s.r.s.}{Abs(x, L)\ \ s.r.s.}\ ABS' \qquad \frac{N_1 \to_n N_2 \quad N_2 :: L\ \ s.r.s.}{N_1 :: (N_2 :: L)\ \ s.r.s.}\ RDX'$$

$$\frac{L :: M\ \ s.r.s. \quad N :: L'\ \ s.r.s.}{Apl_a(L, N) :: MN :: Apl_f(M, L')\ \ s.r.s.}\ APL'$$

A sensible alternative to the rule APL' above could have been:

$$\frac{M :: L\ \ s.r.s. \quad L' :: N\ \ s.r.s.}{Apl_f(M, L') :: MN :: Apl_a(L, N)\ \ s.r.s.}\ APL''$$

To better understand this two alternative rules, let us consider one example:

Example 11. *Let us consider standard reduction sequences $L_1 = M_1 :: M_2 :: M_3$ and $L_2 = N_1 :: N_2 :: N_3$.*

- *First, we apply APL' to L_1 and L_2:*

$$\frac{(M_1 :: M_2) :: M_3 \ \ s.r.s. \quad N_1 :: (N_2 :: N_3) \ \ s.r.s.}{Apl_a(M_1 :: M_2, N_1) :: M_3N_1 :: Apl_f(M_3, N_2 :: N_3) \ \ s.r.s.} \ APL'$$

Note that, $Apl_a(M_1 :: M_2, N_1) :: M_3N_1 :: Apl_f(M_3, N_2 :: N_3)$ is the list

$$L_3 = M_1N_1 :: M_2N_1 :: M_3N_1 :: M_3N_2 :: M_3N_3.$$

- *Now, we apply APL'' to L_1 and L_2:*

$$\frac{M_1 :: (M_2 :: M_3) \ \ s.r.s. \quad (N_1 :: N_2) :: N_3 \ \ s.r.s.}{Apl_f(M_1, N_1 :: N_2) :: M_1N_3 :: Apl_a(M_2 :: M_3, N_3) \ \ s.r.s.} \ APL''$$

Note that, $Apl_f(M_1, N_1 :: N_2) :: M_1N_3 :: Apl_a(M_2 :: M_3, N_3)$ is the list

$$L_4 = M_1N_1 :: M_1N_2 :: M_1N_3 :: M_2N_3 :: M_3N_3.$$

Note that, the first and last terms of lists L_3 and L_4 coincide, but the middle terms are different. Although L_4 is not s.r.s. according to our definition still it is a sensible reduction sequence, since there is no interaction between the terms in fuction and in arguments position along the list L_4 (as in L_3).

The next lemma establishes that singleton lists are standard reduction sequences.

Lemma 25. *For all M in Λ, M s.r.s.*

Proof. By induction on M.

□

The following two lemmas will be useful later and establish that certain subsequences of a standard reduction sequence are still standard reduction sequences with specific shapes.

Lemma 26. *For all $M, N \in \Lambda$ and $L \in L(\Lambda)$, if $M :: N :: L$ s.r.s., then $N :: L$ s.r.s.*

Proof. By induction on: $M :: N :: L$ s.r.s..

$\square$

Lemma 27. *For all M, N in Λ and L in $L(\Lambda)$, if $M :: N :: L$ s.r.s., then $M :: N$ s.r.s.*

Proof. By induction on: $M :: N :: L$ s.r.s.

$\square$

Below we will make use of the lemma that follows, which in fact, is a particular case of our final result.

Lemma 28. *For all M, N in Λ, if $M :: N$ s.r.s., then $M \Rightarrow_n N$.*

Proof. By induction on : $M :: N$ s.r.s. The VAR' case is impossible. The APL' case requires the admissible rule (1) of Figure 2 and the RDX' case uses the Standardization Theorem. $\square$

In order to facilitate the proof of our main theorem, we will make use of an alternative way to characterize the reflexive and transitive closure of the evaluation relation $\rightarrow_n$ on λ-terms.

Definition 21. $\rightarrow^*_{n_1}$ *is the binary relation on λ-terms given inductively by:*

$$\frac{}{M \rightarrow^*_{n_1} M}\ REF' \qquad \frac{M \rightarrow_n N \quad N \rightarrow^*_{n_1} P}{M \rightarrow^*_{n_1} P}\ BASE/TRANS'$$

Lemma 29. *For all M, N and P in Λ,*

$$\frac{M \rightarrow^*_{n_1} N \quad N \rightarrow^*_{n_1} P}{M \rightarrow^*_{n_1} P}$$

Proof. The proof is by induction on $M \rightarrow^*_{n_1} N$. The REF' case, follows immediately from the hypothesis. The $BASE/TRANS'$ case follows from induction hypothesis and by $BASE/TRANS'$. $\square$

Now we can prove that $\rightarrow^*_{n_1}$ is indeed the same as our initial relation $\rightarrow^*_n$:

Lemma 30. *For all M and N in Λ, $M \rightarrow^*_n N$ iff $M \rightarrow^*_{n_1} N$.*

Proof. The "only if" direction is proved by induction on $M \rightarrow^*_n N$. Use is made of the previous lemma in the transitive case. The "if" direction is proved by induction on $M \rightarrow^*_{n_1} N$. $\square$

Now we are ready to prove the key relations between standard reduction sequences and the standard reduction relation $\Rightarrow_n$.

Theorem 4. *For all M, N in Λ,*

1. *If $M \Rightarrow_n N$, then $M = N$ or for some list L, $M :: L :: N$ is a standard reduction sequence (s.r.s.);*
2. *For any $M :: L$ s.r.s., $L = []$ or $L = L' :: N$ (for some list L' and term N), and $M \Rightarrow_n N$.*

Proof. The proof of 1. is by induction on induction on $M \Rightarrow_n N$. The proof of 2. is by induction on L. Use is made of Lemmas 26, 27, 28 and Corollary 1.

□

As an easy corollary of the previous theorem, we can finally establish the equivalence between the standard reduction relation ($\Rightarrow_n$) and standard reduction sequences (s.r.s.).

Corollary 3. *For all M, N in Λ, $M \Rightarrow_n N$ iff $(M = N$ or $M :: L :: N$ s.r.s, for some list $L)$.*

Proof. In order to prove this corollary, we will prove separately both directions of the equivalence.

In the "only if" direction we have by hypothesis,

$M = N$ or $M :: L :: N$ s.r.s, for some list L.

If $M = N$, then $M \Rightarrow_n M$ follows immediately from the admissible rule (1) of Figure 3. If $M :: L :: N$ s.r.s, for some list L, then by the second statement of Theorem 4, follows

$$L :: N = [] \quad \vee \quad \exists L' \in L(\Lambda), N' \in \lambda, (L :: N = L' :: N' \quad \wedge \quad M \Rightarrow_n N).$$

The hypothesis $L :: N = []$ is impossible. Then, remains the hypothesis,

$\exists L' \in L(\Lambda), N' \in \Lambda, (L :: N = L' :: N' \quad \wedge \quad M \Rightarrow_n N)$

which in particular gives $M \Rightarrow_n N$.

The "if" direction follows immediately from clause 1 of Theorem 4.

□

5.2 Formalization in Coq

This section briefly presents the formalization in Coq of some of the definitions and main results described in the previous section. This will use several definitions and results whose formalization was presented in Chapter 4. In particular, recall that λ-terms are represented via De Bruijn λ-terms. The full details of this Coq formalization can be found in Appendix F.

We will start with the Coq definitions of lists of De Bruijn λ-terms and of several functions operating on these lists, and will then state some results about these functions.

Lists of De Bruijn λ-terms are represented in Coq through the following inductive definition:

```
Inductive term_list : Set :=
| nil
| cons (M : lambda) (L : term_list).
```

To avoid heavy notation, we will usually write `cons M L` as `M :: L`, and `nil` as `[]`:

```
Notation "M :: L" := (cons M L).
Notation "[ ]" := nil.
```

The concatenation of two lists L_1 and L_2 is defined in Coq as follows:

```
Fixpoint app (L1 L2 : term_list) : term_list :=
match L1 with
| nil ⇒ L2
| h :: t ⇒ h :: (app t L2)
end.

Notation "L1 · L2" := (app L1 L2) (at level 50) : type_scope.
```

The Coq function `Abs_list`, that follows implements the Abs function of Definition 17:

```
Fixpoint Abs_list (L : term_list) : term_list :=
  match L with
```

```
  | nil ⇒ nil
  | M :: L1 ⇒ Abs M :: Abs_list (L1)
  end.
```

The Coq functions `Apl_arg` and `Apl_fun` that follow implement the functions Apl_a and Apl_f of Definitions 18 and 19, respectively:

```
Fixpoint Apl_arg (L : term_list) : lambda → term_list :=
  fun N : lambda ⇒
  match L with
  | nil ⇒ nil
  | M :: L1 ⇒ (App M N) :: (Apl_arg L1 N)
  end.


Fixpoint Apl_fun (L : term_list) : lambda → term_list :=
  fun M : lambda ⇒
  match L with
  | nil ⇒ nil
  | N :: L1 ⇒ (App M N) :: (Apl_fun L1 M)
  end.
```

The statement in Coq that concatenation is associative is as follows:

```
Lemma concatenate_assoc : forall L1 L2 L3 : term_list, (L1 · L2) · L3 = L1 · (L2 · L3).
```

The next two Coq Lemmas correspond to Lemma 23 and to the first clause of Lemma 24 respectively:

```
Lemma abs_lists : forall L1 L2 : term_list, Abs_list (L1 · L2) = Abs_list L1 · Abs_list L2.
```

```
Lemma apl_fun_lists : forall L1 L2 : term_list, forall N : lambda, Apl_fun (L1 · L2) N =
(Apl_fun L1 N) · (Apl_fun L2 N).
```

Now, we turn to the representation of standard reduction sequences. This concept is formalized in Coq as an inductive predicate:

```
Inductive standard_red_seq : term_list → Prop :=
  | VAR' : forall i : nat, standard_red_seq ((Ref i) :: [])
  | ABS' : forall L : term_list, standard_red_seq L → standard_red_seq (Abs_list L)
  | APL' : forall L1 L2 : term_list, forall M N : lambda, standard_red_seq (L1 · (M :: []))
  → standard_red_seq (N :: L2) →
  standard_red_seq (Apl_arg L1 N · (( App M N) :: []) · Apl_fun L2 M )
  | RDX' : forall N1 N2 : lambda, forall L : term_list, name_eval_1 N1 N2 →
  standard_red_seq (N2 :: L) → standard_red_seq (N1 :: (N2 :: L)).
```

Next we show the Coq statement of the Lemmas 25 to 28 in the previous section:

```
Lemma single_list_srs : forall M : lambda, standard_red_seq (M :: [ ]).
```

```
Lemma aux_2 : forall M N : lambda, forall L : term_list, standard_red_seq (M :: N :: L) →
standard_red_seq (N :: L).
```

```
Lemma aux_6 : forall M N : lambda, standard_red_seq (M :: (N :: [])) → standard_red M N.
```

```
Lemma aux_10 : forall M N : lambda, forall L : term_list, standard_red_seq (M :: N :: L) →
standard_red_seq (M :: N :: []).
```

We are now ready to address the formlization of the key results relating the standard reduction relation and standard reduction sequences, namely parts 1 and 2 of Theorem 4. These results are stated in Coq as follows, respectively:

```
Lemma standard_red_1 : forall M N : lambda, standard_red M N → M = N ∨
(exists L : term_list, standard_red_seq (M :: L · (N :: []))).

```

```
Lemma standard_red_2 : forall L : term_list, forall M : lambda, standard_red_seq (M :: L) ⟶
( L=[] ∨ (exists N : lambda, exists L' : term_list, L = L' · (N :: []) ∧ standard_red M N)).
```

Finally, the equivalence of the two approaches to standard reduction (Corollary 3) is formalized as follows:

```
Lemma s_r_s_equiv: forall M N : lambda, standard_red M N ↔
(M = N ∨ exists L : term_list, standard_red_seq (M :: L · (N :: []) )).
```

Chapter 6

Conclusion

Concluding remarks. In this dissertation, we presented a formalization in the Coq proof assistant of a proof of the Standardization Theorem for λ-calculus that we extracted from a proof of a Standardization Theorem for a λ-calculus for modal logic [33]. The approach followed is in line with treatments of standardization for λ-calculus by Loader and Joachimski - Matthes, where standard reduction is captured via an inductively defined relation, but differs from them in that our standard relation is over λ-terms with ordinary (unary) applications, rather than with applications that allow multiple arguments. Although distinct, we show that the approach to standardization we follow is equivalent to the more traditional one, based on standard reduction sequences (as considered in work by Plotkin [30]), providing a formalization of this equivalence in Coq.

Our formalization used a representation of the binders via De Bruijn indices. In principle, there should be no major difficulty in adapting this formalization to work with other techniques for dealing with binders. The initial reasons to opt for this technique were rather pragmatic ones (a big body of literature and developments of formalizations of meta-theory of λ-calculus and extensions available in the literature), but it turned out that, once the basic structure for working on top of De Bruijn indices was set up, the Coq formalization of the proof of the Standardization Theorem could follow very closely the structure of the paper proof of this result, both in what concerns lemmata and the inductive structure of the arguments.

As expected in formalizations efforts, the complete formalization in Coq of the proof of the Standardization Theorem (developed beforehand on paper) reinforced our confidence on the paper proof, for example, ensuring that our inductive arguments (typically needing the analysis of multiple cases) did not miss any case. Additionally, our formalization in Coq helped in identifying small aspects of the proof on paper that needed more attention or could have been done differently, or even to reuse some Coq code when arguments had a similar structure. One example of the latter was the proof the admissibility of rule (8) (Lemma 8) that resulted from an immediate adaptation of the Coq code to prove the admissibility of rule (4) (Lemma 6).

Related work. In the literature, other efforts to formalize the Standardization Theorem for λ-calculus include proofs based on Kashima [20] such as [12, 16], where a notion of *β-reducibility with a standard sequence* is captured by an inductively defined reduction relation. What sets our development apart from these efforts is essentially the way in which the standard reduction is captured. In particular, the definition of standard sequence in Kashima [20] uses two binary relations, *head reduction in application* and *standard*, that are defined on the set of λ-terms and are the keys to the main proof. In [12] the same two relations are defined but, in order to formalize the proof, they use multiple substitution. In [16] the technique chosen to formalize all the theory was also the De Bruijn indices, but they adopt a system of *reference by pointers* (lists of steps). Another early effort worth highlighting is that of McKinna-Pollack [26]. This work also proves the Standardization Theorem, but using the proof assistant LEGO. In order to formalize λ-terms, this work uses named variables, based on syntactically distinguishing free from bound variables, following a suggestion by Coquand in [13].

Future work. A natural follow-up on this dissertation would be to test all the ideas in this dissertation on Plotkin's call-by-value λ-calculus [34]. On the one hand, notice that the proof of standardization formalized in this dissertation was extracted from a proof of standardization for the λ_{b}-calculus for modal logic, and a refinement of this calculus studied in [34] (called $\lambda_{\between}$) allows to obtain as a corollary the Standardization Theorem for Plotkin's cbv λ-calculus. We expect that the overall ideas involved in the proof of the Standardization Theorem in this dissertation (including the admissible rules for standard reduction) can be adapted to work for Plotkin's cbv λ-calculus. On the other hand, in such a formalization of standardization for Plotkin's cbv λ-calculus we could immediately profit from all the basic infrastructure of the De Bruijn λ-terms that is already set up. Another natural follow-up (that could also immediately benefit from the work in this dissertation) would be to address the formalization of the Standardization Theorem for the modal calculus λ_{b}, or for its refined versions $\lambda_{b\cancel{b}}$ or $\lambda_{\between}$ considered in [34]. Since from the Standardization Theorem for $\lambda_{\between}$ it is possible to obtain as corollaries the Standardization Theorem for the cbn and for the cbv λ-calculus [34], a complementary and rather different (and big) challange could then be to formalize the additional collection of concepts and results involved in these alternative proofs of standardization for cbn and cbv λ-calculus.

As mentioned before, the approach followed in this dissertation to formalize standard reduction as an inductive relation is in line with the one followed by Joachimski and Matthes in [19]. In this paper, the λ-calculus treated is actually an extension of ordinary λ-calculus with the so-called *generalised applications*. This calculus needs an additional rule (on top of β) to perform reduction (the π-rule). So, another interesting challenge could be to try to adapt the ideas in this dissertation to obtain a formalized proof of the Standardization Theorem for this λ-calculi with generalized applications. We would expect such a proof to show some small differences w.r.t. the proof of standardization for this calculus in [19], because this proof uses *multiple application* ("lists of generalised arguments"), and in our development we confine to unary application, as in the ordinary syntax of λ-calculi.

Appendix A

In this Appendix we have the details of the proofs of some results described in Chapter 2.

Lemma 1. (Substitution Lemma): For all x, y in V and M, N, Q in Λ, if $x \neq y$ and $x \notin FV(Q)$, then $(M[N/x])[Q/y] = (M[Q/y])[N[Q/y]/x]$.

Proof. The proof of this lemma is an induction on the size of M, given as usual by: $size(z) = 1$, $size(\lambda z.M_0) = 1 + size(M_0)$ and $size(M_1 M_2) = size(M_1) + size(M_2)$.

- $M = z$
  - $z = x$:

    Left-side:

    $(x[N/x])[Q/y] = N[Q/y]$

    Right-side:

    $(x[Q/y])[N[Q/y]/x] = x[N[Q/y]/x] = N[Q/y]$
  - $z = y$

    Left-side:

    $(y[N/x])[Q/y] = y[Q/y] = Q$

    Right-side:

    $(y[Q/y])[N[Q/y]/x] = Q[N[Q/y]/x] = Q$

    The last equality is valid because x is not free in Q.
  - $z \neq x$ and $z \neq y$

Left-side:

$(z[N/x])[Q/y] = z[Q/y] = z$

Right-side:

$(z[Q/y])[N[Q/y]/x] = z[N[Q/y]/x] = z$

- $M = \lambda x \cdot M'$

  Left-side:

  $((\lambda x \cdot M')[N/x])[Q/y] =_{*1} (\lambda x \cdot M')[Q/y]$

  $(*1)$ by Definition 3

  Right-side:

  $((\lambda x \cdot M')[Q/y])[N[Q/y]/x] =_{*1} (\lambda z \cdot (M'[z/x])[Q/y])[N[Q/y]/x]) =_{*2}$
  $(\lambda z \cdot (M'[z/x])[N/x])[Q/y] =_{*3} (\lambda z \cdot M'[z/x])[Q/y] =_{\alpha} (\lambda x \cdot M')[Q/y]$

  $(*1)$ by Definition 3

  $(*2)$ by induction hypothesis (note that $size(M') = size(M'[z/x])$)

  $(*3)$ $x \notin FV(M'[z/x])$

- $M = \lambda w \cdot M'$, where $w \neq x$

  Left-side:

  $((\lambda w \cdot M')[N/x])[Q/y] =_{*1} (\lambda z \cdot (M'[z/w])[N/x])[Q/y]$

  $(*1)$ by Definition 3

  Right-side:

  $((\lambda w \cdot M')[Q/y])[N[Q/y]/x] =_{*1} ((\lambda z \cdot M'[z/w])[Q/y])[N[Q/y]/x] =_{*2}$
  $\lambda z \cdot (M'[z/w][N/x])[Q/y]$

  $(*1)$ by Definition 3

$(*2)$ by induction hypothesis (note that $size(M') = size(M'[z/w])$)

- $M = M'M''$

  By induction hypothesis: $(M'[N/x])[Q/y] = (M'[Q/y])[N[Q/y]/x]$ and $(M''[N/x])[Q/y] = (M''[Q/y])[N[Q/y]/x]$.

  Left-side:

  $((M'M'')[N/x])[Q/y] =_{*1} ((M'[N/x])[Q/y])((M''[N/x])[Q/y]) =_{*2}$

  $((M'[Q/y])[N[Q/y]/x])((M''[N/x])[Q/y]) =_{*3}$

  $((M'[Q/y])[N[Q/y]/x])((M''[Q/y])[N[Q/y]/x])$

  $(*1)$ by Definition 3

  $(*2)$ by induction hypothesis

  $(*3)$ by induction hypothesis

  Right-side:

  $((M'M'')[Q/y])[N[Q/y]/x] =_{*1} ((M'[Q/y])[N[Q/y]/x])((M''[Q/y])[N[Q/y]/x])$

  $(*1)$ by Definition 3

□

Lemma 2. For all M, M' in Λ, if $M \to_\beta^* M'$ then:

1. $MN \to_\beta^* M'N$, for all $N \in \Lambda$;
2. $NM \to_\beta^* NM'$, for all $N \in \Lambda$;
3. $\lambda x \cdot M \to_\beta^* \lambda x \cdot M'$, for all $x \in V$.

Proof. Proof of 1. The proof is an easy induction on $M \to_\beta^* M'$.

In the base case, we apply the rule (μ) to the hypothesis $M \to_\beta M'$ and obtain $MN \to_\beta M'N$. Finally using the fact that $\to_\beta \subseteq \to_\beta^*$, we conclude $MN \to_\beta^* M'N$.

The reflexive case follows immediately by the fact that $\to_\beta^*$ is reflexive. Then we conclude $MN \to_\beta^* MN$.

In the transitive case, we suppose by hypotheses $M \to_\beta^* P$ and $P \to_\beta^* M'$. By induction hypotheses, for all N' in Λ, $MN' \to_\beta^* PN'$ and for all N'' in Λ, $PN'' \to_\beta^* M'N''$. Using the fact that $\to_\beta^*$ is transitive, and take $N' = N$ and $N'' = N$, we conclude $MN \to_\beta^* M'N$.

Proof of 2. The proof is an easy induction on $M \to_\beta^* M'$.

In the base case, we apply the rule (ν) to the hypothesis $M \to_\beta M'$ and obtain $NM \to_\beta NM'$. Then we conclude $NM \to_\beta^* NM'$ using the fact that $\to_\beta \subseteq \to_\beta^*$.

The reflexive case just uses the fact that $\to_\beta^*$ is reflexive to conclude $NM \to_\beta^* NM$.

In the transitive case, we suppose by hypotheses $M \to_\beta^* P$ and $P \to_\beta^* M'$. By induction hypotheses, for all N' in Λ, $N'M \to_\beta^* N'P$ and for all N'' in Λ, $N''P \to_\beta^* N''M'$. Using the fact that $\to_\beta^*$ is transitive, and take $N' = N$ and $N'' = N$, we conclude $NM \to_\beta^* NM'$.

Proof of 3. The proof is an easy induction on $M \to_\beta^* M'$.

In the base case, we apply the rule (ξ) to the hypothesis $M \to_\beta M'$ and obtain $\lambda x \cdot M \to_\beta \lambda x \cdot M'$. We conclude $\lambda x \cdot M \to_\beta^* \lambda x \cdot M'$, just using the fact that $\to_\beta \subseteq \to_\beta^*$.

The reflexive case follows immediately by the fact that $\to_\beta^*$ is reflexive to conclude $\lambda x \cdot M \to_\beta^* \lambda x \cdot M$.

In the transitive case, we suppose by hypotheses $M \to_\beta^* P$ and $P \to_\beta^* M'$. By induction hypotheses $\lambda x \cdot M \to_\beta^* \lambda x \cdot P$ and $\lambda x \cdot P \to_\beta^* \lambda x \cdot M'$. Using the fact that $\to_\beta^*$ is transitive, we conclude $\lambda x \cdot M \to_\beta^* \lambda x \cdot M'$.

$\square$

Appendix B

In this Appendix we have the details of the proofs of some results described in Chapter 3.

Lemma 3. The following rule is admissible, that is, for all M_1, M_2, N in Λ:

$$\frac{M_1 \to_n^* M_2}{M_1 N \to_n^* M_2 N}$$

Proof. By induction on $M_1 \to_n^* M_2$.

In the base case, we have by hypothesis $M_1 \to_n M_2$. Then by (μ) follows immediately:

$$M_1 N \to_n M_2 N \subseteq M_1 N \to_n^* M_2 N$$

The reflexive case follows immediately by the fact $\to_n^*$ is reflexive, to conclude $M_1 N \to_n^* M_1 N$.

In the transitive case, we suppose by hypotheses $M_1 \to_n^* M_3$ and $M_3 \to_n^* M_2$. By induction hypotheses $M_1 N \to_n^* M_3 N$ and $M_3 N \to_n^* M_2 N$. Then we conclude $M_1 N \to_n^* M_2 N$ by using the fact that $\to_n^*$ is transitive.

□

Lemma 4. The following rules are admissible:

$$\frac{M_1 \to_n M_2}{M_1[N/x] \to_n M_2[N/x]} \qquad \frac{M_1 \to_n^* M_2}{M_1[N/x] \to_n^* M_2[N/x]}$$

Proof. Proof of the admissibility of the first rule The proof is an induction on $M_1 \to_n M_2$.

In the (β) case we have by hypothesis $(\lambda y \cdot M)N_0 \to_n M[N_0/y]$. We want to prove $((\lambda y \cdot M)N_0)[N/x] \to_n (M[N_0/y])[N/x]$. By Definition 3 follows the equalities:

$$((\lambda y \cdot M)N_0)[N/x] = (\lambda y \cdot M)[N/x]N_0[N/x] = \lambda y \cdot (M[N/x])N_0[N/x]$$
$$\rightarrow_n (M[N/x])[N_0[N/x]/y]$$

where the last reduction is justified by rule (β).

By the Substitution Lemma (1) follows $(M[N_0/y])[N/x] = (M[N/x])[N_0[N/x]/y]$.

In the (μ) case, we want to prove $(M_0M_3)[N/x] \rightarrow_n (M_4M_3)[N/x]$. By hypothesis $M_0 \rightarrow_n M_4$. By Definition 3, $(M_0M_3)[N/x] = M_0[N/x]M_3[N/x]$ and $(M_4M_3)[N/x] = M_4[N/x]M_3[N/x]$. By induction hypothesis $M_0[N/x] \rightarrow_n M_4[N/x]$. Use is made of (μ) to conclude $M_0[N/x]M_3[N/x] \rightarrow_n M_4[N/x]M_3[N/x]$.

Proof of the admissibility of the second rule. The proof is by induction on $M_1 \rightarrow_n^* M_2$.

In the base case, we have by hypothesis $M_1 \rightarrow_n M_2$. Then by the previous admissible rule follows:

$$M_1[N/x] \rightarrow_n M_2[N/x] \subseteq M_1[N/x] \rightarrow_n^* M_2[N/x]$$

The reflexive case follows immediately by the fact $\rightarrow_n^*$ is reflexive, to conclude $M_1[N/x] \rightarrow_n^* M_1[N/x]$.

In the transitive case, we suppose by hypotheses $M_1 \rightarrow_n^* M_3$ and $M_3 \rightarrow_n^* M_2$. By induction hypotheses $M_1[N/x] \rightarrow_n^* M_3[N/x]$ and $M_3[N/x] \rightarrow_n^* M_2[N/x]$. Then we conclude $M_1[N/x] \rightarrow_n^* M_2[N/x]$ by using the fact that $\rightarrow_n^*$ is transitive.

□

Lemma 5. The rules (1) and (2) of Figure 2 are admissible.

Proof. Proof of the admissibility of (1). The proof is an induction on M.

The case where M is a variable follows immediately from rule VAR.

The case where $M = \lambda x \cdot M'$, follows by ABS and the induction hypothesis $M' \Rightarrow_n M'$ to conclude $\lambda x \cdot M' \Rightarrow_n \lambda x \cdot M'$.

The case where $M = M'N'$, follows by APL and the induction hypotheses $M' \Rightarrow_n M'$ and $N' \Rightarrow_n N'$, to obtain $M'N' \Rightarrow_n M'N'$.

Proof of the admissibility of (2). The proof is an induction on $M \Rightarrow_n M'$.

By inversion on the VAR case follows two possible subcases, or M and M' are equal to the variable that we want to replace x, or are different.

In the first one, by Definition 3:

$x[N/x] = N$

$x[N'/x] = N'$

Then by hypothesis $N \Rightarrow_n N'$.

In the second one ($y \neq x$), using the Definition 3:

$y[N/x] = y$

$y[N'/x] = y$

Then by VAR follows $y \Rightarrow_n y$.

In the ABS case, $M = \lambda y \cdot Q$ and $M' = \lambda y \cdot Q'$. By hypothesis $Q \Rightarrow_n Q'$. By Definition 3:

$(\lambda y \cdot Q)[N/x] = \lambda y \cdot (Q[N/x])$

$(\lambda y \cdot Q')[N'/x] = \lambda y \cdot (Q'[N'/x])$

By induction hypothesis $Q[N/x] \Rightarrow_n Q'[N'/x]$. Applying this induction hypothesis in rule ABS follows $\lambda y \cdot (Q[N/x]) \Rightarrow_n \lambda y \cdot (Q'[N'/x])$.

In the APL case, $M = QS$ and $M' = Q'S'$. By hypotheses $Q \Rightarrow_n Q'$ and $S \Rightarrow_n S'$. By Definition 3:

$(QS)[N/x] = (Q[N/x])(S[N/x])$

$(Q'S')[N'/x] = (Q'[N'/x])(S'[N'/x])$

By induction hypotheses, $Q[N/x] \Rightarrow_n Q'[N'/x]$ and $S[N/x] \Rightarrow_n S'[N'/x]$. From the induction hypotheses and rule APL follows immediately $(Q[N/x])(S[N/x]) \Rightarrow_n (Q'[N'/x])(S'[N'/x])$.

In the RDX case, $M = QS$. By hypotheses $Q \to_n^* \lambda y \cdot Q'$ and $Q'[S/y] \Rightarrow_n M'$. By the hypothesis $Q \to_n^* \lambda y \cdot Q'$ and Lemma 4, follows $Q[N/x] \to_n^* (\lambda y \cdot Q')[N/x]$.

By Definition 3, $(\lambda y \cdot Q')[N/x] = \lambda y \cdot (Q'[N/x])$. By the hypotheses $Q'[S/y] \Rightarrow_n M'$ and $N \Rightarrow_n N'$ follows by induction hypothesis $(Q'[S/y])[N/x] \Rightarrow_n M'[N'/x]$. By the Substitution Lemma, $(Q'[S/y])[N/x] = (Q'[N/x])[S[N/x]/y]$. From the hypotheses $Q[N/x] \to_n^* \lambda y \cdot (Q'[N/x])$ and $(Q'[N/x])(S[N/x]/y] \Rightarrow_n M'[N'/x]$ and rule RDX follows $(Q[N/x])(S[N/x]) \Rightarrow_n M'[N'/x]$.

□

Lemma 6. The rules (3) and (4) of Figure 2 are admissible.

Proof. Proof of the admissibility of (3). The proof is by induction on $M \to_n N$.

In the (β) case, $M = (\lambda x \cdot Q)S$. By hypothesis $Q[S/x] \Rightarrow_n P$. Using the fact that $\to_n^*$ is reflexive, follows $\lambda x \cdot Q \to_n^* \lambda x \cdot Q$. Then by rule RDX follows $(\lambda x \cdot Q)S \Rightarrow_n P$.

In the (μ) case $M = QR$ and $N = Q'R$. By inversion on the hypothesis $Q'R \Rightarrow_n P$ we have two possible subcases, APL and RDX.

In the first one, $P = Q''R'$. By the hypotheses $Q \to_n Q' \Rightarrow_n Q''$ and by induction hypothesis follows $Q \Rightarrow_n Q''$. Applying rule APL to the hypotheses $Q \Rightarrow_n Q''$ and $R \Rightarrow_n R'$ follows $QR \Rightarrow_n Q''R'$.

In the second one, we have by hypotheses $Q' \to_n^* \lambda x \cdot Q''$ and $Q''[R/x] \Rightarrow_n P$. Using the fact that $Q \to_n Q'$ and $Q' \to_n^* \lambda x \cdot Q''$ and $\to_n^*$ is transitive, follows $Q \to_n^* \lambda x \cdot Q''$. Finally applying the rule RDX with $Q \to_n^* \lambda x \cdot Q''$ and $Q''[R/x] \Rightarrow_n P$ we conclude $QR \Rightarrow_n P$.

Proof of the admissibility of (4) . The proof is by induction on $M \to_n^* N$.

The base case follows immediately from (3).

The reflexive case follows immediately by the hypothesis $M \Rightarrow_n P$.

In the transitive case, we suppose by hypotheses $M \to_n^* Q$ and $Q \to_n^* N$. By induction hypothesis associated with the hypothesis $Q \to_n^* N \Rightarrow_n P$ follows $Q \Rightarrow_n P$. Then by induction hypothesis associated with $M \to_n^* Q$ and $Q \Rightarrow_n P$ we conclude $M \Rightarrow_n P$.

□

Lemma 7. The rules (5) and (6) of Figure 2 are admissible.

Proof. Proof of the admissibility of (5). The proof is by induction on $M \Rightarrow_n \lambda x \cdot M'$.

We only have two possible cases, the ABS and the RDX.

In the first one, $M = \lambda x \cdot Q$. Applying the rule (β) we have that $(\lambda x \cdot Q)N \rightarrow_\beta Q[N/x]$. Then using the fact that $(\beta) \subseteq \rightarrow_n^*$ follows:

$$\begin{aligned}(\lambda x \cdot Q)N &\rightarrow_n^* Q[N/x] \\ &\Rightarrow_n M'[N'/x]\end{aligned}$$

The last relation is justified by (2) with the hypotheses $Q \Rightarrow_n M'$ and $N \Rightarrow_n N'$. Then using (4) follows $(\lambda x \cdot Q)N \Rightarrow_n M'[N'/x]$.

In the second one, $M = QS$. By Lemma 2 and uses the fact $\rightarrow_n^* \subseteq \rightarrow_\beta^*$ and the hypothesis $Q \rightarrow_n^* \lambda y \cdot Q'$ follows:

$$\begin{aligned}QS &\rightarrow_n^* (\lambda y \cdot Q')S \\ &\rightarrow_n^* Q'[S/y]\end{aligned}$$

The last relation is justified by rule (β) and the fact that $(\beta) \subseteq \rightarrow_n^*$

Using the fact that $\rightarrow_n^*$ is transitive we conclude $QS \rightarrow_n^* Q'[S/y]$. Then by the first point of Lemma 2, with the hypothesis $QS \rightarrow_n^* Q'[S/y]$ and the fact that $\rightarrow_n^* \subseteq \rightarrow_\beta^*$ follows:

$$\begin{aligned}(QS)N &\rightarrow_n^* (Q'[S/y])N \\ &\Rightarrow_n M'[N'/x]\end{aligned}$$

The last relation is justified by induction hypothesis associated with $Q'[S/y] \Rightarrow_n \lambda x \cdot M'$ and the hypothesis $N \Rightarrow_n N'$. Then $(QS)N \Rightarrow_n M'[N'/x]$ follows immediately from (4).

Proof of the admissibility of (6). The proof of the admissibility of (6) is by induction on $M \Rightarrow_n (\lambda x \cdot M')N'$.

In this induction we only have two possible cases, the APL and the RDX.

In the first one M have the form QP. Then $QP \Rightarrow_n M'[N'/x]$ follows immediately from (5) using the hypothesis $Q \Rightarrow_n \lambda x \cdot M'$ and $P \Rightarrow_n N'$.

In the second, M have the form QR. We have by hypothesis $Q \rightarrow_n^* \lambda y \cdot Q'$ and $Q'[R/y] \Rightarrow_n (\lambda x \cdot M')N'$. By induction hypothesis associated to the hypothesis $Q'[R/y] \Rightarrow_n (\lambda x \cdot M')N'$ follows $Q'[R/y] \Rightarrow_n M'[N'/x]$. Finally applying RDX to the hypothesis $Q \rightarrow_n^* \lambda y \cdot Q'$ and $Q'[R/y] \Rightarrow_n M'[N'/x]$ we conclude $QR \Rightarrow_n M'[N'/x]$.

□

Lemma 8. The rules (7) and (8) of Figure 2 are admissible.

Proof. Proof of the admissibility of (7). The proof is by induction on $M \Rightarrow_n N$.

The VAR case is impossible.

In ABS case, $M = \lambda x \cdot M'$ and $N = \lambda x \cdot N'$. then by inversion on $\lambda x \cdot N' \rightarrow_\beta P$, follows the (ξ) subcase, where $P = \lambda x \cdot N''$. By induction hypothesis associated to the hypotheses $M' \Rightarrow_n N' \rightarrow_\beta N''$ follows $M' \Rightarrow_n N''$. Then by ABS we conclude $\lambda x \cdot M' \Rightarrow_n \lambda x \cdot N''$.

In APL case, $M = QS$ and $N = Q'S'$. Then by inversion on $Q'S' \rightarrow_\beta P$ we have three possible subcases, (β), (μ) and (ν).

In the first one, $Q' = \lambda x \cdot Q''$. Applying APL with the hypotheses $Q \Rightarrow_n \lambda x \cdot Q''$ and $S \Rightarrow_n S'$, we have $QS \Rightarrow_n (\lambda x \cdot Q'')S'$. Then $QS \Rightarrow_n Q''[S'/x]$ follows immediatly by (6).

In the second one, $P = RS'$. Then by induction hypothesis associated to the hypotheses $Q \Rightarrow_n Q' \rightarrow_\beta R$ follows $Q \Rightarrow_n R$. Finally applying the APL with the hypotheses $Q \Rightarrow_n R$ and $S \Rightarrow_n S'$ we conclude $QS \Rightarrow_n RS'$.

In the last one, $P = Q'R$. By induction hypothesis associated to the hypotheses $S \Rightarrow_n S' \rightarrow_{\beta_n} R$ follows $S \Rightarrow_n R$. Then applying APL with the hypotheses $Q \Rightarrow_n Q'$ and $S \Rightarrow_n R$ we conclude $QS \Rightarrow_n Q'R$.

In the RDX case, $M = QS$. Using the induction hypothesis associated to the hypotheses $Q'[S/y] \Rightarrow_n N \rightarrow_{\beta_n} P$ follows $Q'[S/y] \Rightarrow_n P$. Finally applying RDX to the hypotheses $Q \rightarrow_n^* \lambda y \cdot Q'$ and $Q'[S/y] \Rightarrow_n P$, we conclude $QS \Rightarrow_n P$.

Proof of the admissibility of (8). The proof is by induction on $N \rightarrow_\beta^* P$.

The base case follows immediately from (7).

The reflexive case follows immediately by the hypothesis $M \Rightarrow_n N$.

In the transitive case, we suppose by hypotheses $N \rightarrow_\beta^* P'$ and $P' \rightarrow_\beta^* P$. By induction hypothesis associated with the hypothesis $M \Rightarrow_n N \rightarrow_\beta^* P'$ follows $M \Rightarrow_n P'$. Then by induction hypothesis associated with $M \Rightarrow_n P' \rightarrow_\beta^* P$ we conclude $M \Rightarrow_n P$.

□

Appendix C

In this Appendix we have the details of the proofs of some results described in Chapter 4.

Lemma 9. For all M, N in Λ_{dB} and k in $\mathbb{N}_0$, $(\Uparrow_k M)[k := N] = M$.

Proof. The proof of this lemma is an induction on M.

- $M = n$

  - subcase $n < k$:

    $(\Uparrow_k n)[k := N] =_{*1} n[k := N] =_{*2} n$

    (*1) by Definition 12 and $n < k$

    (*2) by Definition 13 and $n < k$

  - subcase $n \geq k$:

    $(\Uparrow_k n)[k := N] =_{*1} n + 1[k := N] =_{*2} n$

    (*1) by Definition 12 and $n \geq k$

    (*2) by Definition 13 and $n \geq k \Rightarrow n + 1 > k$

- $M = \lambda \cdot M'$

  By induction hypothesis: $(\Uparrow_k M')[k := N] = M'$

  $(\Uparrow_k \lambda{\cdot}M')[k := N] =_{*1} (\lambda{\cdot} \Uparrow_{k+1} M')[k := N] =_{*2} \lambda{\cdot}(\Uparrow_{k+1} M'[k{+}1 :=\Uparrow_0 N]) =_{*3} \lambda{\cdot}M'$

  (*1) by Definition 12

(*2) by Definition 13

(*3) by induction hypothesis

- $M = M'M''$

  By induction hypothesis: $(\Uparrow_k M')[k := N] = M'$ and $(\Uparrow_k M'')[k := N] = M''$

  $(\Uparrow_k M'M'')[k := N] =_{*1} ((\Uparrow_k M')(\Uparrow_k M''))[k := N] =_{*2} (\Uparrow_k M')[k := N](\Uparrow_k M'')[k := N] =_{*3} M'(\Uparrow_k M'')[k := N] =_{*4} M'M''$

  (*1) by Definition 12

  (*2) by Definition 13

  (*3) by induction hypothesis

  (*4) by induction hypothesis

□

Lemma 10. For all M in Λ_{dB} and k, i in $\mathbb{N}_0$, if $i \geq k$, then $\Uparrow_{i+1} (\Uparrow_k M) = \Uparrow_k (\Uparrow_i M)$.

Proof. The proof of this lemma is an induction on M.

- $M = n$

  - subcase $n < k$:

    Left-side:

    $\Uparrow_{i+1} (\Uparrow_k n) =_{*1} \Uparrow_{i+1} n =_{*2} n$

    (*1) by Definition 12 and $n < k$

    (*2) by Definition 12 and $(n < i \wedge i \geq k \Rightarrow n < i + 1)$

    Right-side:

    $\Uparrow_k (\Uparrow_i n) =_{*1} \Uparrow_k n =_{*2} n$

    (*1) by Definition 12 and $(n < k \wedge i \geq k \Rightarrow n < i)$

(*2) by Definition 12 and $n < k$

- subcase $n \geq k$ and $n < i$:

  Left-side:

  $\Uparrow_{i+1} (\Uparrow_k n) =_{*1} \Uparrow_{i+1} (n+1) =_{*2} n+1$

  (*1) by Definition 12 and $n \geq k$

  (*2) by Definition 12 and ($n < i \Rightarrow n+1 < i+1$)

  Right-side:

  $\Uparrow_k (\Uparrow_i n) =_{*1} \Uparrow_k n =_{*2} n+1$

  (*1) by Definition 12 and $n < i$

  (*2) by Definition 12 and ($n \geq k \Rightarrow n+1 > k$)

- subcase $n \geq k$ and $n \geq i$:

  Left-side:

  $\Uparrow_{i+1} (\Uparrow_k n) =_{*1} \Uparrow_{i+1} (n+1) =_{*2} n+2$

  (*1) by Definition 12 and $n \geq k$

  (*2) by Definition 12 and ($n \geq i \Rightarrow n+1 \geq i+1$)

  Right-side:

  $\Uparrow_k (\Uparrow_i n) =_{*1} \Uparrow_k (n+1) =_{*2} n+2$

  (*1) by Definition 12 and $n \geq i$

  (*2) by Definition 12 and ($n \geq k \Rightarrow n+1 > k$)

- $M = \lambda \cdot M'$

  By induction hypothesis: $\Uparrow_{i+1} (\Uparrow_k M') = \Uparrow_k (\Uparrow_i M')$

  Left-side:

$\Uparrow_{i+1} (\Uparrow_k \lambda{\cdot}M') =_{*1} \Uparrow_{i+1} (\lambda{\cdot} \Uparrow_{k+1} M') =_{*2} \lambda{\cdot}(\Uparrow_{i+2} (\Uparrow_{k+1} M')) =_{*3} \lambda{\cdot}(\Uparrow_{k+1} (\Uparrow_{i+1} M'))$

(*1) by Definition 12

(*2) by Definition 12

(*3) by induction hypothesis

Right-side:

$\Uparrow_k (\Uparrow_i \lambda \cdot M') =_{*1} \Uparrow_k (\lambda \Uparrow_{i+1} M') =_{*2} \lambda(\Uparrow_{k+1} (\Uparrow_{i+1} M'))$

(*1) by Definition 12

(*2) by Definition 12

- $M = M'M''$

  By induction hypothesis: $\Uparrow_{i+1} (\Uparrow_k M') = \Uparrow_k (\Uparrow_i M')$ and $\Uparrow_{i+1} (\Uparrow_k M'') = \Uparrow_k (\Uparrow_i M'')$

  Left-side:

  $\Uparrow_{i+1} (\Uparrow_k M'M'') =_{*1} \Uparrow_{i+1} ((\Uparrow_k M')(\Uparrow_k M'')) =_{*2} (\Uparrow_{i+1} (\Uparrow_k M'))(\Uparrow_{i+1} (\Uparrow_k M'')) =_{*3} (\Uparrow_k (\Uparrow_i M'))(\Uparrow_{i+1} (\Uparrow_k M'')) =_{*4} (\Uparrow_k (\Uparrow_i M'))(\Uparrow_k (\Uparrow_i M''))$

  (*1) by Definition 12

  (*2) by Definition 12

  (*3) by induction hypothesis

  (*4) by induction hypothesis

  Right-side:

  $\Uparrow_k (\Uparrow_i M'M'') =_{*1} \Uparrow_k ((\Uparrow_i M')(\Uparrow_i M'')) =_{*2} (\Uparrow_k (\Uparrow_i M'))(\Uparrow_k (\Uparrow_i M''))$

  (*1) by Definition 12

  (*2) by Definition 12

□

Lemma 11. For all M, N in Λ_{dB} and k, i in $\mathbb{N}_0$, if $i \geq k$, then $\Uparrow_k (M[i := N]) = (\Uparrow_k M)[i+1 :=\Uparrow_k N]$.

Proof. The proof of this lemma is an induction on M.

- $M = n$
  - subcase $n < i$ and $n < k$:

    Left-side:

    $\Uparrow_k (n[i := N]) =_{*1}\Uparrow_k n =_{*2} n$

    (*1) by Definition 13 and $n < i$

    (*2) by Definition 12 and $n < k$

    Right-side:

    $(\Uparrow_k n)[i + 1 :=\Uparrow_k N] =_{*1} n[i + 1 :=\Uparrow_k N] =_{*2} n$

    (*1) by Definition 12 and $n < k$

    (*2) by Definition 13 and ($n < i \Rightarrow n < i + 1$)

  - subcase $n < i$ and $n \geq k$:

    Left-side:

    $\Uparrow_k (n[i := N]) =_{*1}\Uparrow_k n =_{*2} n + 1$

    (*1) by Definition 13 and $n < i$

    (*2) by Definition 12 and $n \geq k$

    Right-side:

    $(\Uparrow_k n)[i + 1 :=\Uparrow_k N] =_{*1} (n + 1)[i + 1 :=\Uparrow_k N] =_{*2} n + 1$

    (*1) by Definition 12 and $n \geq k$

    (*2) by Definition 13 and ($n < i \Rightarrow n + 1 < i + 1$)

- subcase $n = i$ and $n < k$: This subcase is impossible because $i \geq k$.
- subcase $n = i$ and $n \geq k$:

  Left-side:

  $\Uparrow_k (n[i := N]) =_{*1} \Uparrow_k N$

  (*1) by Definition 13 and $n = i$

  Right-side:

  $(\Uparrow_k n)[i + 1 := \Uparrow_k N] =_{*1} (n + 1)[i + 1 := \Uparrow_k N] =_{*2} \Uparrow_k N$

  (*1) by Definition 12 and $n \geq k$

  (*2) by Definition 13 and ($n = i \Rightarrow n + 1 = i + 1$)

- subcase $n > i$ and $n < k$: This subcase is impossible because $i \geq k$.
- subcase $n > i$ and $n = k$: This subcase is impossible because $i \geq k$.
- subcase $n > i$ and $n > k$:

  Left-side:

  $\Uparrow_k (n[i := N]) =_{*1} \Uparrow_k (n - 1) =_2 n$

  (*1) by Definition 13 and $n > i$

  (*2) by Definition 12 and ($n > k \Rightarrow n - 1 \geq k$)

  Right-side:

  $(\Uparrow_k n)[i + 1 := \Uparrow_k N] =_{*1} (n + 1)[i + 1 := \Uparrow_k N] =_{*2} n$

  (*1) by Definition 12 and $n > k$

  (*2) by Definition 13 and ($n > i \Rightarrow n + 1 > i + 1$)

- $M = \lambda \cdot M'$

  By induction hypothesis: $\Uparrow_k (M'[i := N]) = (\Uparrow_k M')[i + 1 := \Uparrow_k N]$

Left-side:

$\Uparrow_k ((\lambda \cdot M')[i := N]) =_{*1} \Uparrow_k (\lambda \cdot (M'[i+1 :=\Uparrow_0 N])) =_{*2} \lambda \cdot \Uparrow_{k+1} (M'[i+1 :=\Uparrow_0 N]) =_{*3} \lambda \cdot ((\Uparrow_{k+1} M')[i+2 :=\Uparrow_{k+1} (\Uparrow_0 N)])$

(*1) by Definition 13

(*2) by Definition 12

(*3) by induction hypothesis

Right-side:

$(\Uparrow_k \lambda \cdot M')[i+1 :=\Uparrow_k N] =_{*1} (\lambda \cdot \Uparrow_{k+1} M')[i+1 :=\Uparrow_k N] =_{*2} \lambda \cdot ((\Uparrow_{k+1} M')[i+2 :=\Uparrow_0 (\Uparrow_k N)]) =_{*3} \lambda \cdot ((\Uparrow_{k+1} M')[i+2 :=\Uparrow_{k+1} (\Uparrow_0 N)])$

(*1) by Definition 12

(*2) by Definition 13

(*3) by Lemma 10

- $M = M'M''$

  By induction hypothesis: $\Uparrow_k (M'[i := N]) = (\Uparrow_k M')[i+1 :=\Uparrow_k N]$ and $\Uparrow_k (M''[i := N]) = (\Uparrow_k M'')[i+1 :=\Uparrow_k N]$

  Left-side:

  $\Uparrow_k ((M'M'')[i := N]) =_{*1} \Uparrow_k (M'[i := N]M''[i := N]) =_{*2} \Uparrow_k (M'[i := N]) \Uparrow_k (M''[i := N]) =_{*3} (\Uparrow_k M')[i+1 :=\Uparrow_k N] \Uparrow_k (M''[i := N]) =_{*4} (\Uparrow_k M')[i+1 :=\Uparrow_k N](\Uparrow_k M'')[i+1 :=\Uparrow_k N]$

  (*1) by Definition 13

  (*2) by Definition 12

  (*3) by induction hypothesis

  (*4) by induction hypothesis

  Right-side:

$(\Uparrow_k (M'M''))[i+1 :=\Uparrow_k N] =_{*1} ((\Uparrow_k M')(\Uparrow_k M''))[i+1 :=\Uparrow_k N] =_{*2} (\Uparrow_k M')[i+1 :=\Uparrow_k N](\Uparrow_k M'')[i+1 :=\Uparrow_k N]$

(*1) by Definition 12

(*2) by Definition 13

□

Lemma 12. For all M, N in Λ_{dB} and k, i in $\mathbb{N}_0$, if $i \geq k$, then $\Uparrow_i (M[k := N]) = (\Uparrow_{i+1} M)[k :=\Uparrow_i N]$.

Proof. The proof of this lemma is an induction on M.

- $M = n$
  - subcase $n < k$ and $n < i$:

    Left-side:

    $\Uparrow_i (n[k := N]) =_{*1}\Uparrow_i n =_{*2} n$

    (*1) by Definition 13 and $n < k$

    (*2) by Definition 12 and $n < i$

    Right-side:

    $(\Uparrow_{i+1} n)[k :=\Uparrow_i N] =_{*1} n[k :=\Uparrow_i N] =_{*2} n$

    (*1) by Definition 12 and $(n < i \Rightarrow n < i + 1)$

    (*2) by Definition 13 and $n < k$
  - subcase $n < k$ and $n \geq i$: This subcase is impossible because $i \geq k$.
  - subcase $n = k$ and $n > i$: This subcase is impossible because $i \geq k$.
  - subcase $n = k$ and $n < i$:

    Left-side:

$\Uparrow_i (n[k := N]) =_{*1} \Uparrow_i N$

(*1) by Definition 13 and $n = k$

Right-side:

$(\Uparrow_{i+1} n)[k := \Uparrow_i N] =_{*1} n[k := \Uparrow_i N] =_{*2} \Uparrow_i N$

(*1) by Definition 12 and ($n < i \Rightarrow n < i + 1$)

(*2) by Definition 13 and $n = k$

- subcase $n = k$ and $n = i$:

  Left-side:

  $\Uparrow_i (n[k := N]) =_{*1} \Uparrow_i N$

  (*1) by Definition 13 and $n = k$

  Right-side:

  $(\Uparrow_{i+1} n)[k := \Uparrow_i N] =_{*1} n[k := \Uparrow_i N] =_{*2} \Uparrow_i N$

  (*1) by Definition 12 and ($n = i \Rightarrow n < i + 1$)

  (*2) by Definition 13 and $n = k$

- $M = \lambda \cdot M'$

  By induction hypothesis: $\Uparrow_i (M'[k := N]) = (\Uparrow_{i+1} M')[k := \Uparrow_i N]$

  Left-side:

  $\Uparrow_i (\lambda \cdot M'[k := N]) =_{*1} \Uparrow_i \lambda \cdot (M'[k + 1 := \Uparrow_0 N]) =_{*2} \lambda \cdot (\Uparrow_{i+1} (M'[k + 1 := \Uparrow_0 N])) =_{*3}$
  $=_{*3} \lambda \cdot ((\Uparrow_{i+2} M')[k + 1 := \Uparrow_{i+1} (\Uparrow_0 N)])$

  (*1) by Definition 13

  (*2) by Definition 12

  (*3) by induction hypothesis

Right-side:

$(\Uparrow_{i+1} (\lambda \cdot M'))[k :=\Uparrow_i N] =_{*1} (\lambda\cdot \Uparrow_{i+2} M')[k :=\Uparrow_i N] =_{*2} \lambda \cdot ((\Uparrow_{i+2} M')[k + 1 :=\Uparrow_0 (\Uparrow_i N)]) =_{*3}$

$=_{*3} \lambda \cdot ((\Uparrow_{i+2} M')[k + 1 :=\Uparrow_{i+1} (\Uparrow_0 N)])$

(*1) by Definition 12

(*2) by Definition 13

(*3) by Lemma 10

- $M = M'M''$

  By induction hypothesis: $\Uparrow_i (M'[k := N]) = (\Uparrow_{i+1} M')[k :=\Uparrow_i N]$ and $\Uparrow_i (M''[k := N]) =$

  $= (\Uparrow_{i+1} M'')[k :=\Uparrow_i N]$

  Left-side:

  $\Uparrow_i (M'M''[k := N]) =_{*1}\Uparrow_i (M'[k := N]M''[k := N]) =_{*2} (\Uparrow_i (M'[k := N]))(\Uparrow_i (M''[k := N])) =_{*3}$

  $=_{*3} ((\Uparrow_{i+1} M')[k :=\Uparrow_i N])(\Uparrow_i (M''[k := N])) =_{*4} ((\Uparrow_{i+1} M')[k :=\Uparrow_i N])((\Uparrow_{i+1} M'')[k :=\Uparrow_i N])$

  (*1) by Definition 13

  (*2) by Definition 12

  (*3) by induction hypothesis

  (*4) by induction hypothesis

  Right-side:

  $(\Uparrow_{i+1} M'M'')[k :=\Uparrow_i N] =_{*1} (\Uparrow_{i+1} M' \Uparrow_{i+1} M'')[k :=\Uparrow_i N] =_{*2} ((\Uparrow_{i+1} M')[k :=\Uparrow_i N])((\Uparrow_{i+1} M'')[k :=\Uparrow_i N])$

  (*1) by Definition 12

  (*2) by Definition 13

□

Lemma 13. (**Substitution Lemma for De Bruijn λ-terms**) For all M, N, Q in Λ_{dB} and i, k in $\mathbb{N}_0$, if $i \geq k$, then

$$M[k := N][i := Q] = M[i+1 :=\Uparrow_k Q][k := N[i := Q]]$$

Proof. The proof of this lemma is an induction on M.

- $M = n$
  - subcase $n < k$ and $n < i$:

    Left-side:

    $(n[k := N])[i := Q] =_{*1} n[i := Q] =_{*2} n$

    (*1) by Definition 13 and $n < k$

    (*2) by Definition 13 and $n < i$

    Right-side:

    $(n[i+1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*1} n[k := N[i := Q]] =_{*2} n$

    (*1) by Definition 13 and ($n < i \Rightarrow n < i + 1$)

    (*2) by Definition 13 and $n < k$
  - subcase $n < k$ and $n = i$: This subcase is impossible because $i \geq k$.
  - subcase $n < k$ and $n > i$: This subcase is impossible because $i \geq k$.
  - subcase $n = k$ and $n < i$:

    Left-side:

    $(n[k := N])[i := Q] =_{*1} N[i := Q]$

    (*1) by Definition 13 and $n = k$

    Right-side:

$(n[i+1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*1} n[k := N[i := Q]] =_{*2} N[i := Q]$

(*1) by Definition 13 and $(n < i \Rightarrow n < i+1)$

(*2) by Definition 13 and $n = k$

- subcase $n = k$ and $n = i$:

  Left-side:

  $(n[k := N])[i := Q] =_{*1} N[i := Q]$

  (*1) by Definition 13 and $n = k$

  Right-side:

  $(n[i+1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*1} n[k := N[i := Q]] =_{*2} N[i := Q]$

  (*1) by Definition 13 and $(n = i \Rightarrow n < i+1)$

  (*2) by Definition 13 and $n = k$

- subcase $n = k$ and $n > i$: This subcase is impossible because $i \geq k$.

- subcase $n > k$ and $n < i$:

  Left-side:

  $(n[k := N])[i := Q] =_{*1} n-1[i := Q] =_{*2} n-1$

  (*1) by Definition 13 and $n > k$

  (*2) by Definition 13 and $(n < i \Rightarrow n-1 < i)$

  Right-side:

  $(n[i+1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*1} n[k := N[i := Q]] =_{*2} n-1$

  (*1) by Definition 13 and $(n < i \Rightarrow n < i+1)$

  (*2) by Definition 13 and $n > k$

- subcase $n > k$ and $n = i$:

  Left-side:

$(n[k := N])[i := Q] =_{*1} n - 1[i := Q] =_{*2} n - 1$

(*1) by Definition 13 and $n > k$

(*2) by Definition 13 and ($n = i \Rightarrow n - 1 < i$)

Right-side:

$(n[i + 1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*1} n[k := N[i := Q]] =_{*2} n - 1$

(*1) by Definition 13 and ($n < i \Rightarrow n < i + 1$)

(*2) by Definition 13 and ($n > k$)

- subcase $n > k$, $n > i$ and $n - 1 = i$:

  Left-side:

  $(n[k := N])[i := Q] =_{*1} n - 1[i := Q] =_{*2} Q$

  (*1) by Definition 13 and $n > k$

  (*2) by Definition 13 and ($n - 1 = i \Rightarrow n - 1 < i$)

  Right-side:

  $(n[i + 1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*1} (\Uparrow_k Q)[k := N[i := Q]] =_{*2} Q$

  (*1) by Definition 13 and ($n - 1 = i \Rightarrow n = i + 1$)

  (*2) by Lemma 9

- subcase $n > k$, $n > i$ and $n - 1 > i$:

  Left-side:

  $(n[k := N])[i := Q] =_{*1} n - 1[i := Q] =_{*2} n - 2$

  (*1) by Definition 13 and $n > k$

  (*2) by Definition 13 and $n - 1 > i$

  Right-side:

  $(n[i + 1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*1} (n - 1)[k := N[i := Q]] =_{*2} n - 2$

(*1) by Definition 13 and ($n - 1 > i \Rightarrow n > i + 1$)

(*2) by Definition 13 and ($n - 1 > i \wedge i \geq k \Rightarrow n - 1 > k$)

- subcase $n > k$, $n > i$ and $n - 1 = i$:

  Left-side:

  $(n[k := N])[i := Q] =_{*1} n - 1[i := Q] =_{*2} Q$

  (*1) by Definition 13 and $n > k$

  (*2) by Definition 13 and $n - 1 = i$

  Right-side:

  $(n[i + 1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*1} \Uparrow_k Q[k := N[i := Q]] =_{*2} Q$

  (*1) by Definition 13 and ($n - 1 = i \Rightarrow n = i + 1$)

  (*2) by Lemma 9

- $M = \lambda \cdot M'$

  By induction hypothesis: $M'[k := N][i := Q] = M'[i + 1 :=\Uparrow_k Q][k := N[i := Q]]$

  Left-side:

  $(\lambda \cdot M')[k := N][i := Q] =_{*1} (\lambda \cdot (M'[k + 1 :=\Uparrow_0 N]))[i := Q] =_{*2} \lambda \cdot ((M'[k + 1 :=\Uparrow_0 N])[i + 1 :=\Uparrow_0 Q]) =_{*3} \lambda \cdot (M'[i + 2 :=\Uparrow_{k+1} (\Uparrow_0 Q)][k + 1 := (\Uparrow_0 N)[i + 1 :=\Uparrow_0 Q]])$

  (*1) by Definition 13

  (*2) by Definition 13

  (*3) by induction hypothesis

  Right-side:

  $(\lambda \cdot M')[i + 1 :=\Uparrow_k Q][k := N[i := Q]] =_{*1} (\lambda \cdot (M'[i + 2 :=\Uparrow_0 (\Uparrow_k Q)]))[k := N[i := Q]] =_{*2} \lambda \cdot (M'[i + 2 :=\Uparrow_0 (\Uparrow_k Q)][k + 1 :=\Uparrow_0 (N[i := Q])]) =_{*3} \lambda \cdot (M'[i + 2 :=\Uparrow_{k+1} (\Uparrow_0 Q)][k + 1 :=\Uparrow_0 (N[i := Q])]) =_{*4} \lambda \cdot (M'[i + 2 :=\Uparrow_{k+1} (\Uparrow_0 Q)][k + 1 := (\Uparrow_0 N)[i + 1 :=\Uparrow_0 Q]])$

(*1) by Definition 13

(*2) by Definition 13

(*3) by Lemma 10

(*4) by Lemma 11

- $M = M'M''$

  By induction hypothesis: $M'[k := N][i := Q] = M'[i+1 :=\Uparrow_k Q][k := N[i := Q]]$ and

  $M''[k := N][i := Q] = M''[i+1 :=\Uparrow_k Q][k := N[i := Q]]$

  Left-side:

  $(M'M'')[k := N][i := Q] =_{*1} (M'[k := N]M''[k := N])[i := Q] =_{*2} (M'[k := N][i := Q])(M''[k := N][i := Q]) =_{*3} (M'[i+1 :=\Uparrow_k Q][k := N[i := Q]])(M''[k := N][i := Q]) =_{*4} (M'[i+1 :=\Uparrow_k Q][k := N[i := Q]])(M''[i+1 :=\Uparrow_k Q][k := N[i := Q]])$

  (*1) by Definition 13

  (*2) by Definition 13

  (*3) by induction hypothesis

  (*4) by induction hypothesis

  Right-side:

  $(M'M'')[i+1 :=\Uparrow_k Q][k := N[i := Q]] =_{*1} (M'[i+1 :=\Uparrow_k Q]M''[i+1 :=\Uparrow_k Q])[k := N[i := Q]] =_{*2} (M'[i+1 :=\Uparrow_k Q][k := N[i := Q]])(M''[i+1 :=\Uparrow_k Q][k := N[i := Q]])$

  (*1) by Definition 13

  (*2) by Definition 13

□

Lemma 15. The rules (aux_1) and (aux_2) on Figure 4 are admissible.

Proof. Proof of the admissibility of (aux_1). The proof is an induction on $M_1 \rightarrow_n M_2$.

In the (β) case, $M_1 = (\lambda \cdot M_0)M_3$ and $M_2 = M_0[0 := M_3]$. We want to prove $\Uparrow_k ((\lambda \cdot M_0)M_3) \rightarrow_n \Uparrow_k (M_0[0 := M_3])$. By Definition 12:

$$\begin{aligned}\Uparrow_k ((\lambda \cdot M_0)M_3) = \Uparrow_k (\lambda \cdot M_0) \Uparrow_k M_3 = (\lambda \cdot \Uparrow_{k+1} M_0) \Uparrow_k M_3 \\ \rightarrow_n (\Uparrow_{k+1} M_0)[0 := \Uparrow_k M_3] \\ = \Uparrow_k (M_0[0 := M_3])\end{aligned}$$

where the last reduction follows immediately from (β), and the last equailty is justified by Lemma 12.

In the (μ) case, $M_1 = M_0M_3$ and $M_2 = M_4M_3$. We want to prove $\Uparrow_k (M_0M_3) \rightarrow_n \Uparrow_k (M_4M_3)$. By Definition, 12 $\Uparrow_k (M_0M_3) = (\Uparrow_k M_0)(\Uparrow_k M_3)$ and $\Uparrow_k (M_4M_3) = (\Uparrow_k M_4)(\Uparrow_k M_3)$. By induction hypothesis associated to the hypothesis $M_0 \rightarrow_n M_4$ follows $(\Uparrow_k M_0) \rightarrow_n (\Uparrow_k M_4)$. Then $(\Uparrow_k M_0)(\Uparrow_k M_3) \rightarrow_n (\Uparrow_k M_4)(\Uparrow_k M_3)$ follows immediately from (μ).

Proof of the admissibility of (aux_2). The proof is an induction on $M \rightarrow_n^* N$.

In the base case we have by hypothesis $M \rightarrow_n N$. Then by the previous admissible rule follows:

$$(\Uparrow_k M) \rightarrow_n (\Uparrow_k N) \subseteq (\Uparrow_k M) \rightarrow_n^* (\Uparrow_k N)$$

The reflexive case just uses the fact that $\rightarrow_n^*$ is reflexive.

In the transitive case we have by hypothesis $M \rightarrow_n^* P \rightarrow_n^* N$. By induction hypothesis associated to the hypothesis $M \rightarrow_n^* P$ follows $\Uparrow_k M \rightarrow_n^* \Uparrow_k P$. And using the induction hypothesis associated to the hypothesis $P \rightarrow_n^* N$ follows $\Uparrow_k P \rightarrow_n^* \Uparrow_k N$. Using the fact that $\rightarrow_n^*$ is transitive we conclude $\Uparrow_k M \rightarrow_n^* \Uparrow_k N$.

□

Lemma 16. Rule (aux_3) of Figure 4 is admissible.

Proof. By induction on $M \Rightarrow_n N$.

The VAR case follows immediately from rule (1).

In the ABS case $M = \lambda \cdot M_0$ and $N = \lambda \cdot N_0$. We want to prove, $\Uparrow_k (\lambda \cdot M_0) \Rightarrow_n (\lambda \cdot N_0)$. By Definition 12:

$$\Uparrow_k (\lambda \cdot M_0) = \lambda \cdot (\Uparrow_{k+1} M_0)$$

$\Uparrow_k (\lambda \cdot N_0) = \lambda \cdot (\Uparrow_{k+1} N_0)$

From induction hypothesis, $\Uparrow_{k+1} M_0 \Rightarrow_n \Uparrow_{k+1} N_0$. Then we conclude by ABS, $\lambda \cdot (\Uparrow_{k+1} M_0) \Rightarrow_n \lambda \cdot (\Uparrow_{k+1} N_0)$.

In the APL case, $M = M_1 N_1$ and $N = M_2 N_2$. We want to prove $\Uparrow_k (M_1 N_1) \Rightarrow_n \Uparrow_k (M_2 N_2)$. From Definition 12, we have the equalities:

$\Uparrow_k (M_1 N_1) = \Uparrow_k M_1 \Uparrow_k N_1$

$\Uparrow_k (M_2 N_2) = \Uparrow_k M_2 \Uparrow_k N_2$

From induction hypothesis associated with the hypothesis $M_1 \Rightarrow_n M_2$ follows $\Uparrow_k M_1 \Rightarrow_n \Uparrow_k M_2$. And associated with the hypothesis $N_1 \Rightarrow_n N_2$ follows $\Uparrow_k N_1 \Rightarrow_n \Uparrow_k N_2$. Finally we apply APL to conclude $\Uparrow_k M_1 \Uparrow_k N_1 \Rightarrow_n \Uparrow_k M_2 \Uparrow_k N_2$.

In the RDX case, $M = M_1 P$. We want to prove $\Uparrow_k (M_1 P) \Rightarrow_n \Uparrow_k N$ and by Definition 12 $\Uparrow_k (M_1 P) = \Uparrow_k M_1 \Uparrow_k P$. Applying Lemma 15 to the hypothesis $M_1 \rightarrow_n^* \lambda \cdot M_2$ we obtain:

$$\begin{aligned} \Uparrow_k M_1 &\rightarrow_n^* \Uparrow_k (\lambda \cdot M_2) \\ &= \lambda \cdot (\Uparrow_{k+1} M_2) \end{aligned}$$

Where the last equality is justified by Definition 12. By Lemma 12 and using the fact that $k \geq 0$, we have $\Uparrow_k (M_2[0 := P]) = (\Uparrow_{k+1} M_2)[0 := \Uparrow_k P]$. By induction hypothesis $\Uparrow_k (M_2[0 := P]) \Rightarrow_n \Uparrow_k N$. Applying the RDX rule with the hypothesis $\Uparrow_k M_1 \rightarrow_n^* \lambda \cdot (\Uparrow_{k+1} M_2)$ and $(\Uparrow_{k+1} M_2)[0 := \Uparrow_k P] \Rightarrow_n \Uparrow_k N$ we conclude $\Uparrow_k M_1 \Uparrow_k P \Rightarrow_n \Uparrow_k N$.

□

Lemma 18. The rules (1) and (2) of Figure 3 are admissible.

Proof. Proof of the admissibility of (1)*.* The proof of this rule is very similar to the proof of the first admissible rule of Lemma 5. For this reason the details of the proof will be omitted.

Proof of the admissibility of (2). The proof of the admissibility of (2) is by induction on $M \Rightarrow_n M'$.

In the VAR case, we have three possible cases, $i < i_0$, $i = i_0$ or $i > i_0$.

In the first one, by Definition 13:

$$i_0[N/i] = i_0 - 1$$

$$i_0[N'/i] = i_0 - 1$$

Then by VAR, $i_0 - 1 \Rightarrow_n i_0 - 1$.

In the second one by Definition 13:

$$i_0[N/i] = N$$

$$i_0[N'/i] = N'$$

Then by hypothesis $N \Rightarrow_n N'$.

In the last one, by Definition 13:

$$i_0[N/i] = i_0$$

$$i_0[N'/i] = i_0$$

Then from VAR follows $i_0 \Rightarrow_n i_0$.

In ABS case, $M = \lambda{\cdot}M_1$ and $M' = \lambda{\cdot}M_1'$. We want to prove, $(\lambda{\cdot}M_1)[i := N] \Rightarrow_n (\lambda{\cdot}M_1')[i := N']$. By Definition 13 follows the equalities:

$$(\lambda \cdot M_1)[i := N] = \lambda \cdot (M_1[i + 1 :=\Uparrow_0 N])$$

$$(\lambda \cdot M_1')[i := N'] = \lambda \cdot (M_1'[i + 1 :=\Uparrow_0 N'])$$

By Lemma 16 and take $k = 0$ with the hypothesis $N \Rightarrow_n N'$ follows $(\Uparrow_0 N) \Rightarrow_n (\Uparrow_0 N')$. By induction hypothesis associated to the hypothesis $M_1 \Rightarrow_n M_1'$ and the hypothesis $(\Uparrow_0 N) \Rightarrow_n (\Uparrow_0 N')$ follows $\forall i_0 \in \mathbb{N},\quad M_1[i_0 :=\Uparrow_0 N] \Rightarrow_n M_1'[i_0 :=\Uparrow_0 N']$. Take $i_0 = i + 1$, $M_1[i + 1 :=\Uparrow_0 N] \Rightarrow_n M_1'[i + 1 :=\Uparrow_0 N']$. Finally by ABS, $\lambda \cdot (M_1[i + 1 :=\Uparrow_0 N]) \Rightarrow_n \lambda \cdot (M_1'[i + 1 :=\Uparrow_0 N'])$.

In the APL case, $M = M_1M_3$ and $M' = M_2M_4$. We want to prove, $(M_1M_3)[i := N] \Rightarrow_n (M_2M_4)[i := N']$. By Definition 13 follows:

$$(M_1M_3)[i := N] = (M_1[i := N])(M_3[i := N])$$

$$(M_2M_4)[i := N'] = (M_2[i := N'])(M_4[i := N'])$$

By induction hypothesis, $M_1[i := N] \Rightarrow_n M_2[i := N']$ and $M_3[i := N] \Rightarrow_n M_4[i := N']$. Then from APL we conclude $(M_1[i := N])(M_3[i := N]) \Rightarrow_n (M_2[i := N'])(M_4[i := N'])$.

In the RDX case, $M = M_1M_3$. and we want to prove $M_1M_3[i := N] \Rightarrow_n M'[i := N']$. By Definition 13 $M_1M_3[i := N] = M_1[i := N]M_3[i := N]$. By induction hypothesis associated to the hypothesis $M_2[0 := M_3] \Rightarrow_n M'$ and $N \Rightarrow_n N'$ follows, $(M_2[0 := M_3])[i := N] \Rightarrow_n M'[i := N']$. By the Substitution Lemma 13:

$$(M_2[0 := M_3])[i := N] = (M_2[i + 1 :=\Uparrow_0 N])[(M_3[i := N]) := 0]$$

Applying (aux_5) to the hypothesis $M_1 \rightarrow_n^* \lambda \cdot M_2$ follows:

$$\begin{aligned} M_1[i := N] &\rightarrow_n^* (\lambda \cdot M_2)[i := N] \\ &= \lambda \cdot (M_2[i + 1 :=\Uparrow_0 N]) \end{aligned}$$

where the last equality is justified by Definition 13. Finally by RDX with the hypothesis:

$$M_1[i := N] \rightarrow_n^* \lambda \cdot (M_2[i + 1 :=\Uparrow_0 N])$$

$$(M_2[i + 1 :=\Uparrow_0 N])[(M_3[i := N]) := 0] \Rightarrow_n M'[i := N']$$

we conclude, $M_1[i := N]M_3[i := N] \Rightarrow_n M'[i := N']$.

□

Appendix D

In this Appendix we have the details of the proofs of some results described in Chapter 5.

Lemma 22. For all L_1, L_2, L_3 in $L(\Lambda)$,

$$App(App(L_1, L_2), L_3) = App(L_1, App(L_2, L_3)).$$

Proof. By induction on list L_1.

In the case where L_1 is the empty list, we want to prove, $App(App([], L_2), L_3) = App([], App(L_2, L_3))$. The equality is prove by developing both sides of the equality:

Left-side:

$App(App([], L_2), L_3) =_{*1} App(L_2, L_3)$

$(*1)$ by Definition 16

Right-side:

$App([], App(L_2, L_3)) =_{*1} App(L_2, L_3)$

$(*1)$ by Definition 16

In case where $L_1 = M :: L_1'$, for some $M \in \lambda$-term and $L_1' \in \Gamma$. We want to prove,

$$App(App(M :: L_1', L_2), L_3) = App(M :: L_1', App(L_2, L_3))$$

.

By induction hypothesis, $App(App(L_1', L2), L_3) = App(L_1', App(L_2, L_3))$ .Once again, the equality is prove by developing both sides of the equality:

Left-side:

$App(App(M :: L'_1, L_2), L_3) =_{*1} App(M :: App(L'_1, L_2), L_3) =_{*2} M :: App(App(L'_1, L_2), L_3) =_{*3} M :: App(L'_1, App(L_2, L_3))$

($*1$) by Definition 16

($*2$) by Definition 16

($*3$) by induction hypothesis

Right-side:

$App(M :: L'_1, App(L_2, L_3)) =_{*1} M :: App(L'_1, App(L_2, L_3))$

($*1$) by Definition 16

□

Lemma 23. For all L_1, $L_2 \in L(\Lambda)$, and $x \in V$, $Abs(x, L_1 :: L_2) = Abs(x, L_1) :: Abs(x, L_2)$

Proof. By induction on L_1.

In the case where $L_1 = []$, we want to prove, $Abs(x, [] :: L2) = Abs(x, []) :: Abs(x, L2)$. The equality is proved by developing both sides of the equality:

Left-side:

$Abs(x, [] :: L2) = Abs(x, L2)$

Right-side:

$Abs(x, []) :: Abs(x, L2) = [] :: Abs(x, L2) = Abs(x, L2)$

In the case where $L_1 = M :: L'_1$, for some $M \in \lambda$-term and $L'_1 \in \Gamma$. We want to prove, $Abs(x, (M :: L'_1) :: L2) = Abs(x, (M :: L'_1)) :: Abs(x, L2)$. The prove is made by developing both sides of the equality:

Left-side:

$Abs(x, (M :: L'_1) :: L2) = \lambda x \cdot M :: Abs(x, L'_1 :: L2) = \lambda x \cdot M :: Abs(x, L'_1) :: Abs(x, L2)$

Right-side:

$Abs(x, (M :: L'_1)) :: Abs(x, L2) = \lambda x \cdot M :: Abs(x, L'_1) :: Abs(x, L2)$

The first equality of both sides follows from Definition 17, and the second one from left-side follows from the induction hypothesis, $Abs(x, L'_1 :: L2) = Abs(x, L'_1) :: Abs(x, L2)$.

□

Lemma 24. For all $L_1, L_2 \in L(\Lambda)$,

1. $Apl_f(M, L_1 :: L_2) = Apl_f(M, L_1) :: Apl_f(M, L_2)$
2. $Apl_a(L_1 :: L_2, M) = Apl_a(L_1, M) :: Apl_f(L_2, M)$

Proof. Proof of 1. The proof is an induction on L_1.

In the case where $L_1 = []$, we want to prove, $Apl_f(M, [] :: L_2) = Apl_f(M, []) :: Apl_f(M, L_2)$. The prove is made by developing both sides of the equality.

Left-side:

$Apl_f(M, [] :: L_2) = Apl_f(M, L_2)$

Right-side:

$Apl_f(M, []) :: Apl_f(M, L_2) = [] :: Apl_f(M, L_2) = Apl_f(M, L_2)$

The first equality from the right-side follows by Definition 19.

In the case where $L_1 = M' :: L'_1$, for some $M' \in \lambda$-term and $L'_1 \in \Gamma$, we want to prove, $Apl_f(M, (M' :: L'_1) :: L_2) = Apl_f(M, M' :: L'_1) :: Apl_f(M, L2)$. Once again the prove is made by developing both sides of the equality:

Left-side:

$Apl_f(M, (M' :: L'_1) :: L_2) = MM' :: Apl_f(M, L'_1 :: L_2) = MM' :: Apl_f(M, L'_1) :: Apl_f(M, L_2)$

Right-side:

$Apl_f(M, M' :: L'_1) :: Apl_f(M, L2) = MM' :: Apl_f(M, L'_1) :: Apl_f(M, L_2)$

The second equality of the left-side follows from the induction hypothesis, $Apl_f(M, L'_1 :: L2) = Apl_f(M, L'_1) :: Apl_f(M, L_2)$. The others follows from Definition 19.

Proof of 2. The proof is also an induction on L_1 and is analogous to the previous proof.

□

Lemma 25. For all M in λ-term, M s.r.s.

Proof. The proof of this lemma is an induction on M.

The case where M is a variable follows immediately by VAR'.

In the case where M is an abstraction, M have the form $M = \lambda x \cdot M'$. We want to prove $\lambda x \cdot M'$ s.r.s. By induction hypothesis, M' s.r.s. Then by ABS' follows $Abs(x, M')\ \ s.r.s.$ By Definition 17, $Abs(x, M')\ \ s.r.s. = \lambda x \cdot M'\ \ s.r.s.$

In the case where M is an application, $M = M_1 M_2$. We want to prove $M_1 M_2$ s.r.s. By induction hypothesis, M_1 s.r.s. and M_2 s.r.s. It is obvious that, $[] :: M_1 = M_1$ and $M_2 :: [] = M_2$. Then by APL' follows $Apl_a([], M_2) :: M_1 M_2 :: Apl_f(M_1, [])\ \ s.r.s.$. finally by Definitions 18 and 19 follows:

$$(Apl_a([], M_2) :: M_1 M_2 :: Apl_f(M_1, []))\ \ s.r.s. = M_1 M_2\ \ s.r.s.$$

□

Lemma 26. For all M, $N \in \Lambda$ and $L \in L(\Lambda)$, if $M :: N :: L$ s.r.s., then $N :: L$ s.r.s.

Proof. The proof of this lemma is an induction on $M :: N :: L$ s.r.s.

The VAR' case is impossible.

In the ABS' case, $Abs(x, M_0 :: N_0 :: L_0)\ \ s.r.s.$. By hypothesis, $M_0 :: N_0 :: L_0\ \ s.r.s.$. By Definition 17, $Abs(x, M_0 :: N_0 :: L_0) = \lambda x \cdot M_0 :: \lambda x \cdot N_0 :: Abs(x, L_0)$. So, $M = \lambda x \cdot M_0$, $N = \lambda x \cdot N_0$ and $L = Abs(x, L_0)$. By induction hypothesis, $N_0 :: L_0$ s.r.s.. Then by ABS' follows $Abs(x, N_0 :: L_0)\ \ s.r.s.$. Finally by Definition 17 follows:

$$Abs(x, N_0 :: L_0) = \lambda x \cdot N_0 :: Abs(x, L_0).$$

In the APL' case, $Apl_a(L_0, N_0) :: M_0 N_0 :: Apl_f(M_0, L'_0)\ \ s.r.s.$. By hypothesis $L_0 :: M_0\ \ s.r.s.$ and $N_0 :: L'_0\ \ s.r.s.$. Then when we analyse all possible subcases we have the subcase where $L_0 = M_1 :: L_1$ and the subcase where $L_0 = []$ and $L'_0 = N_1 :: L'_1$.

In the first one, by Definition 18 follows the equality:

$$Apl_a(M_1 :: L_1, N_0) :: M_0 N_0 :: Apl_f(M_0, L'_0) = M_1 N_0 :: Apl_a(L_1, N_0) :: M_0 N_0 :: Apl_f(M_0, L'_0)$$

By inversion on L_1 follows, $L_1 = []$ or $L_1 = M_2 :: L_2$.

If $L_1 = []$, follows the equality:

$M_1N_0 :: Apl_a([], N_0) :: M_0N_0 :: Apl_f(M_0, L'_0)\ \ s.r.s. = M_1N_0 :: M_0N_0 :: Apl_f(M_0, L'_0)$

Then we have, $M = M_1N_0$, $N = M_0N_0$ and $L = Apl_f(M_0, L'_0)$.

Finally by APL' with the hypothesis, $M_0\ \ s.r.s.$ and $N_0 :: L'_0\ \ s.r.s.$ follows:

$M_0N_0 :: Apl_f(M_0, L'_0)\ \ s.r.s..$

If $L_1 = M_2 :: L_2$, by Definition 18 follows the equality:

$Apl_a(M_1 :: M_2 :: L_2, N_0) :: M_0N_0 :: Apl_f(M_0, L'_0)\ \ s.r.s. = M_1N_0 :: M_2N_0 :: Apl_a(L_2, N_0) :: M_0N_0 :: Apl_f(M_0, L'_0) s.r.s.$

Then we have, $M = M_1N_0$, $N = M_2N_0$ and $L = Apl_a(L_2, N_0) :: M_0N_0 :: Apl_f(M_0, L'_0)$.

By induction hypothesis, associated to the hypothesis $M_1 :: M_2 :: L_2 :: M_0$ follows:

$M_2 :: L_2 :: M_0$ s.r.s.

Finally from applying the hypotheses $M_2 :: L_2 :: M_0\ \ s.r.s.$ and $N_0 :: L'_0\ \ s.r.s.$ to APL' follows:

$Apl_a(M_2 :: L_2, N_0) :: M_0N_0 :: Apl_f(M_0, L'_0)\ \ s.r.s. = M_2N_0 :: Apl_a(L_2, N_0) :: M_0N_0 :: Apl_f(M_0, L'_0) s.r.s.$

The equality is justified by Definition 18.

In the second possible subcase, we have $L_0 = []$ and $L'_0 = N_1 :: L'_1$. By Definition 19 follows the equality:

$Apl_a([], N_0) :: M_0N_0 :: Apl_f(M_0, N_1 :: L'_1)\ \ s.r.s. = M_0N_0 :: M_0N_1 :: Apl_f(M_0, L'_1),$

By induction hypothesis, associated to the hypothesis $N_0 :: N_1 :: L'_1$ s.r.s. follows, $N_1 :: L'_1$ s.r.s. Finally applying the hypothesis $M_0\ \ s.r.s.$ and $N_1 :: L'_1\ \ s.r.s.$ to APL', we conclude:

$Apl_a([], N_1) :: M_0N_1 :: Apl_f(M_0, L'_1) = M_0N_1 :: Apl_f(M_0, L'_1)$

In the RDX' case, from the hypothesis, follows immediately $N :: L$ s.r.s.

□

Lemma 27. For all M, N in Λ and L in $L(\Lambda)$, if $M :: N :: L$ s.r.s., then $M :: N$ s.r.s.

Proof. The proof of this Lemma is an induction on $M :: N :: L$ s.r.s.

The VAR' case is impossible.

In the ABS' case, $Abs(x, M_0 :: N_0 :: L_0)\ \ s.r.s.$ By hypothesis, $M_0 :: N_0 :: L_0\ \ s.r.s.$ By Definition 17, $Abs(x, M_0 :: N_0 :: L_0) = \lambda x \cdot M_0 :: \lambda x \cdot N_0 :: Abs(x, L_0)$. So, $M = \lambda x \cdot M_0$, $N = \lambda x \cdot N_0$ and $L = Abs(x, L_0)$. By induction hypothesis, $M_0 :: N_0$ s.r.s. From applying the hypothesis $M_0 :: N_0\ \ s.r.s.$ to ABS' follows:

$$Abs(x, M_0 :: N_0)\ \ s.r.s. = \lambda x \cdot M_0 :: \lambda x \cdot N_0$$

where the equality is justified by Definition 17.

In the APL' case, $Apl_a(L_0, N_0) :: M_0N_0 :: Apl_f(M_0, L'_0)\ \ s.r.s.$. By hypotheses, $L_0 :: M_0\ \ s.r.s.$ and $N_0 :: L'_0\ \ s.r.s.$.

Then we analyse all possible subcases for the lists L_0 and L'_0.

The subcase where, $L_0 = []$ and $L'_0 = []$ is impossible.

If $L_0 = []$ and $L'_0 = N_1 :: L'_1$, by Definition 19 follows the equality:

$$Apl_a([], N_0) :: M_0N_0 :: Apl_f(M_0, N_1 :: L'_1) = M_0N_0 :: M_0N_1 :: Apl_f(M_0, L'_1)$$

Where, $M = M_0N_0$, $N = M_0N_1$ and $L = Apl_f(M_0, L'_1)$. Then by induction hypothesis associated to the hypothesis, $N_0 :: N_1 :: L'_1$ s.r.s. follows, $N_0 :: N_1$ s.r.s.

Finally applying the hypotheses $M_0\ \ s.r.s.$ and $N_0 :: N_1\ \ s.r.s.$ to APL' follows:

$$Apl_a([], N_0) :: M_0N_0 :: Apl_f(M_0, N_1)\ \ s.r.s. = M_0N_0 :: M_0, N_1\ \ s.r.s.$$

The equality is justified by Definition 19.

If $L_0 = M_1 :: L_1$ and $L'_0 = []$, then by Definition 18 follows:

$$Apl_a(M_1 :: L_1, N_0) :: M_0N_0 :: Apl_f(M_0, []) = M_1N_0 :: Apl_a(L_1, N_0) :: M_0N_0$$

Now one of two things can happen, $L_1 = []$ or $L_1 = M_2 :: L_2$.

In the first one, we have the equality:

$M_1N_0 :: Apl_a([], N_0) :: M_0N_0 \ \ s.r.s. = M_1N_0 :: M_0N_0 \ \ s.r.s.$

Where, $M = M_1N_0$, $N = M_0N_0$ and $L = []$.

In the second one, by Definition 18 follows the equality:

$M_1N_0 :: Apl_a(M_2 :: L_2, N_0) :: M_0N_0 :: Apl_f(M_0, []) \ \ s.r.s. = M_1N_0 :: M_2N_0 :: Apl_a(L_2, N_0) :: M_0N_0 \ \ s.r.s.$

Where $M = M_1N_0$, $N = M_2N_0$ and $L = Apl_a(L_2, N_0) :: M_0N_0$.

By induction hypothesis, associated to the hypothesis, $M_1 :: M_2 :: L_2 :: M_0 \ \ s.r.s.$ follows, $M_1 :: M_2 \ \ s.r.s.$

Finally applying the hypotheses, $M_1 :: M_2 \ \ s.r.s.$ and $N_0 \ \ s.r.s.$ to APL' follows:

$Apl_a(M_1, N_0) :: M_2N_0 :: Apl_a(M_2, []) \ \ s.r.s. = M_1N_0 :: M_2N_0 \ \ s.r.s.$

The equality is justified by Definition 18.

If $L_0 = M_1 :: L_1$ and $L'_0 = N_1 :: L'_1$, then by Definitions 18 and 19 follows the equality:

$Apl_a(M_1 :: L_1, N_0) :: M_0N_0 :: Apl_f(M_0, N_1 :: L'_1) \ \ s.r.s. = M_1N_0 :: Apl_a(L_1, N_0) :: M_0N_0 :: M_0N_1 :: Apl_f(M_0, L'_1) \ \ s.r.s.$

Now one of two things can happen, $L_1 = []$ or $L_1 = M_2 :: L_2$.

In the first one, we have:

$M_1N_0 :: M_0N_0 :: M_0N_1 :: Apl_f(M_0, L'_1) \ \ s.r.s.$

Where $M = M_1N_0$, $N = M_0N_0$ and $L = M_0N_1 :: Apl_f(M_0, L'_1)$.

Finally applying the hypotheses $M_1 :: M_0 \ \ s.r.s.$ and $N_0 :: [] \ \ s.r.s.$ to APL' follows:

$Apl_a(M_1, N_0) :: M_0N_0 :: Apl_f(M_0, []) = M_1N_0 :: M_0N_0$

The equality is justified by Definition 18.

In the second one, by Definitions 18 and 19 follows:

$Apl_a(M_1 :: M_2 :: L_2, N_0) :: M_0 N_0 :: Apl_f(M_0, N_1 :: L_1') \ s.r.s. = M_1 N_0 :: M_2 N_0 :: Apl_a(L_2, N_0) :: M_0 N_0 :: M_0 N_1 :: Apl_f(M_0, L_1') \ s.r.s.$

Where $M = M_1 N_0$, $N = M_2 N_0$ and $L = Apl_a(L_2, N_0) :: M_0 N_0 :: M_0 N_1 :: Apl_f(M_0, L_1')$.

By induction hypothesis, associated to the hypothesis, $M_1 :: M_2 :: L_2 :: M_0 \ s.r.s.$, follows $M_1 :: M_2 \ s.r.s.$.

Finally applying the hypotheses $M_1 :: M_2 \ s.r.s.$ and $N_0 \ s.r.s.$ to APL' follows:

$Apl_a(M_1, N_0) :: M_2 N_0 :: Apl_f(M_2, []) \ s.r.s. = M_1 N_0 :: M_2 N_0 \ s.r.s.$

The equality is justified by Definition 18.

In the RDX' case, $M :: (N :: L) \ s.r.s.$. By hypotheses, $M \to_n N$ and $N :: L \ s.r.s.$

Then applying the hypotheses $M \to_n N$ and $N :: [] \ s.r.s.$ to RDX' follows, $M :: N \ s.r.s.$

□

Lemma 28. For all M, N in Λ, if $M :: N$ s.r.s., then $M \Rightarrow_n N$.

Proof. The proof of this Lemma is an induction on $M :: N$ s.r.s.

The VAR' case is impossible.′

In the ABS' case, $Abs(x, M_0 :: N_0) \ s.r.s.$. By hypothesis $M_0 :: N_0 \ s.r.s.$. By Definition 17 follows:

$$Abs(x, M_0 :: N_0) = \lambda x \cdot M_0 :: \lambda x \cdot N_0$$

So, $M = \lambda x \cdot M_0$ and $N = \lambda x \cdot N_0$. By induction hypothesis, $M_0 \Rightarrow_n N_0$. Then by ABS follows $\lambda x \cdot M_0 \Rightarrow_n \lambda x \cdot N_0$.

In the APl' case, $Apl_a(L_1, N_1) :: M_1 N_1 :: Apl_f(M_1, L_2) \ s.r.s.$. By hypothesis, $L_1 :: M_1 \ s.r.s.$ and $N_1 :: L_2 \ s.r.s.$. In this case we have two possible subcases, $L_1 = M_2$ and $L_2 = []$, or $L_1 = []$ and $L_2 = N_2$.

In the first one, $M = M_2 N_1$ and $N = M_1 N_1$. By induction hypothesis, $M_2 \Rightarrow_n M_1$. By (1), follows $N_1 \Rightarrow_n N_1$. Then by APL, we conclude $M_2 N_1 \Rightarrow_n M_1 N_1$.

In the second one, $M = M_1 N_1$ and $N = M_1 N_2$. By induction hypothesis, $N_1 \Rightarrow_n N_2$. Then by (1) follows $M_1 \Rightarrow_n M_1$. Finally by APL we conclude, $M_1 N_1 \Rightarrow_n M_1 N_2$.

In the RDX' case, we have by hypothesis $M \to_n N$ and $N :: [] \ s.r.s.$. We conclude that $M \Rightarrow_n N$ using the Standardization Theorem and using the fact that $\to_n \subseteq \to_n^* \subseteq \to_\beta^*$.

□

Lemma 29. For all M, N and P in Λ,

$$\frac{M \to_{n_1}^* N \quad N \to_{n_1}^* P}{M \to_{n_1}^* P}$$

Proof. By induction on $M \to_{n_1}^* N$.

The REF' case, follows immediately from the hypothesis $M \to_{n_1}^* P$.

In the $BASE/TRANS'$, we have by hypothesis $M \to_n Q$, $Q \to_{n_1}^* N$ and $N \to_{n_1}^* P$.

By induction hypothesis associated to the hypotheses $Q \to_{n_1}^* N$ and $N \to_{n_1}^* P$ follows:

$$Q \to_{n_1}^* P$$

Then by $BASE/TRANS'$ associated to $M \to_n Q$ and $Q \to_{n_1}^* P$ follows, $M \to_{n_1}^* P$.

□

Lemma 30. For all M and N in Λ, $M \to_n^* N$ iff $M \to_{n_1}^* N$.

Proof. In order to prove this Lemma, we will prove both directions of the equivalence.

The "only if"direction is proved by induction on $M \to_n^* N$.

In the base case, we have by hypothesis $M \to_n N$. By REF' follows $N \to_{n_1}^* N$. Then applying $BASE/TRAN'$ with the hypotheses $M \to_n N$ and $N \to_{n_1}^* N$ follows:

$$M \to_{n_1}^* N.$$

The reflexive case follows immediately from REF' to conclude, $M \to_{n_1}^* M$

In the transitive case, we have by hypothesis, $M \to_n^* P$ and $P \to_n^* N$. By induction hypotheses follows, $M \to_{n_1}^* P$ and $P \to_{n_1}^* N$. Then from Lemma 29 follows immediately:

$$M \to_{n_1}^* N.$$

The "if"direction is proved by induction on $M \to_{n_1}^* N$.

The REF' case follows immediately from REF, to conclude $M \to_n^* M$.

In the $BASE/TRANS'$ case, we have by hypothesis $M \to_n P$ and $P \to_{n_1}^* N$. It is easy to see that:

$$M \to_n P \subseteq M \to_n^* P$$

By induction hypothesis associated to the hypothesis $P \rightarrow_{n_1}^* N$ follows $P \rightarrow_n^* N$. Applying $TRANS$ with the hypothesis $M \rightarrow_n^* P$ and $P \rightarrow_n^* N$ follows:

$$M \rightarrow_n^* N$$

□

Theorem 4. For all M, N in Λ,

1. If $M \Rightarrow_n N$, then $M = N$ or for some list L, $M :: L :: N$ is a standard reduction sequence (s.r.s.);

2. For any $M :: L$ s.r.s., $L = []$ or $L = L' :: N$ (for some list L' and term N), and $M \Rightarrow_n N$.

Proof. Proof of 1. The proof is a induction on $M \Rightarrow_n N$.

In the VAR case we have $x = x$.

In the ABS case ($M = \lambda x \cdot M'$ and $N = \lambda x \cdot N'$), we have by induction hypotheses $M' = N'$ or $M' :: L' :: N'$ s.r.s, for some list L'.

In the first subcase, follows immediately:

$$\lambda x \cdot M' = \lambda x \cdot N' \Leftrightarrow \lambda x \cdot M' = \lambda x \cdot M'$$

In the second one, by ABS' follows, $Abs(x, M' :: L' :: N')\ \ s.r.s.$ Then by Definition 17 and Lemma 23, we have the equalities:

$$Abs(x, M' :: L' :: N') = Abs(x, M') :: Abs(x, L') :: Abs(x, N') = \lambda x \cdot M' :: Abs(x, L') :: \lambda x \cdot N'$$

Then just take, $L = Abs(x, L')$ to obtain $\lambda x \cdot M' :: Abs(x, L') :: \lambda x \cdot N'$ s.r.s.

In the APL case, $M = M_1 N_1$ and $N = M_2 N_2$. We have by induction hypothesis associated to the hypothesis $M_1 \Rightarrow_n M_2$, ($(M_1 :: L_1) :: M_2$ s.r.s., for some list L_1, or $M_1 = M_2$).

In the first subcase, we have $(M_1 :: L_1) :: M_2$ s.r.s., for some list L_1. Then by induction hypothesis associated to the hypothesis $N_1 \Rightarrow_n N_2$ follows, ($(N_1 :: L_2) :: N_2$ s.r.s., for some list L_2, or $N_1 = N_2$).

If $(N_1 :: L_2) :: N_2$ s.r.s., for some list L_2. Then by APL' follows:

$$Apl_a(M_1 :: L_1, N_1)@M_2N_1@Apl_f(M_2, L_2 :: N_2)\ \ s.r.s. = (M_1N_1) :: Apl_a(L_1, N_1) :: M_2N_1 :: Apl_f(M_2, L_2) :: (M_2N_2)\ \ s.r.s.$$

The last equality is justified by Definitions 18 and 19.

Then just take, $L = Apl_a(L_1, N_1) :: M_2N_1 :: Apl_f(M_2, L_2)$.

If $N_1 = N_2$, by APL' follows:

$$Apl_a(M_1 :: L_1, N_1) :: M_2N_1 :: Apl_f(M_2, []) \ \ s.r.s. = M_1N_1 :: Apl_a(L_1, N_1) :: M_2N_2 \ \ s.r.s.$$

The last equality is justified by Definition 18 an by the hypothesis, $N_1 = N_2$.

Then just take, $L = Apl_a(L_1, N_1)$.

In the second subcase , we have the hypothesis $M_1 = M_2$. Then by induction hypothesis associated to the hypothesis $N_1 \Rightarrow_n N_2$ follows, $(N_1 :: L_2) :: N_2$ s.r.s., for some list L_2, or $N_1 = N_2$).

If $(N_1 :: L_2) :: N_2$ s.r.s., for some list L_2. By APL' follows:

$$Apl_a([], N_1)@M_1N_1@Apl_f(M_1, L_1 :: N_2) \ \ s.r.s. = M_1N_1 :: Apl_f(M_1, L_1) :: M_2N_2 \ \ s.r.s.$$

The equality is justified by Definition 19 and the hypothesis $M_1 = M_2$.

Then just take, $L = Apl_f(M_1, L_1)$.

Finally if $N_1 = N_2$, follows immediately:

$$M_1N_1 = M_1N_2 \Leftrightarrow M_1N_2 = M_1N_2$$

In the RDX case, we have $M = QS$. By induction hypothesis associated to the hypothesis $Q'[S/x] \Rightarrow_n N$ follows, $Q'[S/x] :: L' :: N$ s.r.s., for some list L', or $Q'[S/x] = N$.

In the first subcase, we have $Q'[S/x] :: L' :: N$ s.r.s.

Then by subinduction on $Q \rightarrow_n^* \lambda x \cdot Q'$, follows two possible subcases, the reflexive or the base/-transitive.

In the reflexive subcase, we have by hypothesis $Q = \lambda x \cdot Q'$.

By the β reduction rule (β) follows:

$$(\lambda x \cdot Q')S \rightarrow_n Q'[S/x]$$

Then by RDX' applying with the hypothesis $(\lambda x \cdot Q')S \rightarrow_n Q'[S/x]$ and $Q'[S/x] \Rightarrow_n N$, follows, $(\lambda x \cdot Q')S :: (Q'[S/x] :: (L' :: N))\ \ s.r.s.$

Then just take, $L = Q'[S/x] :: L'$.

In the base/transitive case, we have by hypothesis $Q \rightarrow_n P$ and $P \rightarrow_n^* \lambda x \cdot Q'$.

Then applying RDX to the hypothesis $P \rightarrow_n^* \lambda x \cdot Q'$ and $Q'[S/x] \Rightarrow_n N$, follows, $PS \Rightarrow_n N$. By induction hypothesis:

$PS :: L_1 :: N\ \ s.r.s.$, for some list L_1, or $PS = N$

If $PS :: L_1 :: N\ \ s.r.s.$, for some list L_1, from the hypothesis $Q \rightarrow_n P$ and by (μ) follows, $QS \rightarrow_n PS$.

Then applying the hypotheses $QS \rightarrow_n PS$ and $PS :: (L_1 :: N)\ \ s.r.s.$ to RDX' follows, $QS :: (PS :: (L_1 :: N))\ \ s.r.s.$

The we just take, $L = PS :: L_1 :: N$.

If $PS = N$, from the hypothesis $Q \rightarrow_n P$ and (μ) follows, $QS \rightarrow_n PS$.

Then by applying the hypotheses $QS \rightarrow_n PS$ and $PS\ \ s.r.s.$ to RDX' follows, $QS :: PS\ \ s.r.s.$

Then we just take, $QS :: L :: PS$, for $L = []$.

In the subcase where, $Q'[S/x] = N$, by subinduction on $Q \rightarrow_n^* \lambda x \cdot Q'$, follows two possible subcases, the reflexive or the base/transitive.

In the reflexive subcase, we have by hypothesis $Q = \lambda x \cdot Q'$.

By the β reduction rule (β) follows:

$(\lambda x \cdot Q')S \rightarrow_n Q'[S/x]$

Then by RDX', $(\lambda x \cdot Q')S :: Q'[S/x]\ \ s.r.s.$.

Then just take, $(\lambda x \cdot Q')S :: L :: Q'[S/x]\ \ s.r.s.$, for $L = []$.

In the base/transitive subcase, we have by hypothesis $Q \rightarrow_n P$ and $P \rightarrow_n^* \lambda x \cdot Q'$. Then applying RDX to the hypothesis $P \rightarrow_n^* \lambda x \cdot Q'$ and $Q'[S/x] \Rightarrow_n N$, follows, $PS \Rightarrow_n N$. Then by induction hypothesis:

$PS :: L_1 :: Q'[S/x]$, for some list L_1, or $PS = Q'[S/x]$

If $PS :: L_1 :: Q'[S/x]$, for some list L_1, by the hypothesis $Q \rightarrow_n P$ and (μ) follows, $QS \rightarrow_n PS$.

Then applying the hypotheses $QS \rightarrow_n PS$ and $PS :: (L_1 :: Q'[S/x])\ \ s.r.s.$ to RDX' follows:

$QS :: (PS :: (L_1 :: Q'[S/x]))\ \ s.r.s.$

Then we just take, $QS :: L :: Q'[S/x]$ s.r.s., for $L = PS :: L_1$.

If $PS = Q'[S/x]$, by the hypothesis $Q \rightarrow_n P$ and (μ) follows, $QS \rightarrow_n PS$. Then applying the hypotheses $QS \rightarrow_n PS$ and $PS\ \ s.r.s.$ to RDX' follows, $QS :: PS\ \ s.r.s.$.

The we just take, $QS :: L :: PS$, for $L = []$.

Proof of 2. The proof is a induction on L, and consists in find a list L' and a term N that satisfies the equality ($L = L' :: N$) and the relation ($M \Rightarrow_n N$).

The case where $L = []$ is trivial.

In case where $L = M_0 :: L_0$, for some list L_0, by Lemma 26 and the hypothesis $M :: M_0 :: L_0\ \ s.r.s.$, follows $M_0 :: L_0\ \ s.r.s.$. By induction hypothesis, associated to the hypothesis $M_0 :: L_0\ \ s.r.s.$ follows:

$L_0 = []$ or ($L_0 = L'_0 :: N_0$, for some L'_0 list and λ-term N_0, and $M_0 \Rightarrow_n N_0$).

In the first subcase, we just consider $L' = []$ and $N = M_0$. Then by Lemma 28 with the hypothesis $M :: M_0\ \ s.r.s.$, follows, $M \Rightarrow_n M_0$.

In the second one, we consider $L' = M_0 :: L'_0$ and $N = N_0$. Then from applying Lemma 27 to the hypothesis $M :: M_0 :: L'_0 :: N_0\ \ s.r.s.$ follows, $M :: M_0\ \ s.r.s.$ The by Lemma 28 follows immediately $M \Rightarrow_n M_0$. Finally applying Lemma 1, to the hypothesis, $M \Rightarrow_n M_0$ and $M_0 \Rightarrow_n N_0$ follows $M \Rightarrow_n N_0$.

□

Appendix E

This appendix contains the full Coq code for the theory of λ-calculus with the De Bruijn indices, and the formalization of all concepts and results corresponding to Chapter 4, such as the relations of call-by-name evaluation and of standard reduction and several properties of these relations. The code below was developed under version 8.12.2 of the Coq proof assistant.

```
(*------------------------- Arithmetic tests ------------------------- *)

Require Import Arith.

(* Pattern-matching lemmas for comparing 2 naturals *)

Definition test : forall n m : nat, {n <= m} + {n > m}.
Proof.
simple induction n; simple induction m; simpl in |- *; auto with arith.
intros m' H'; elim (H m'); auto with arith.
Defined.

Definition le_lt : forall n m : nat, n <= m → {n < m} + {n = m}.
Proof.
simple induction n; simple induction m; simpl in |- *; auto with arith.
intros m' H1 H2; elim (H m'); auto with arith.
Defined.

Definition compare : forall n m : nat, {n < m} + {n = m} + {n > m}.
```

```
Proof.
intros n m; elim (test n m); auto with arith.
left; apply le_lt; trivial with arith.
Defined.

(*--------------- Lambda terms with de Bruijn's indices ---------------*)

(* Lambda terms with de Bruijn's indices *)

Inductive lambda : Set :=
  | Ref : nat → lambda
  | Abs : lambda → lambda
  | App : lambda → lambda → lambda.

(*-------------------------- Lifting --------------------------*)

Definition relocate (i k : nat) :=
  match test k i with


      (* k<=i *) | left _ ⇒ S i
   (* k>i  *) | _ ⇒ i
  end.

Fixpoint lift_rec (L : lambda) : nat → lambda :=
  fun k : nat ⇒
  match L with
  | Ref i ⇒ Ref (relocate i k)
  | Abs M ⇒ Abs (lift_rec M (S k))
  | App M N ⇒ App (lift_rec M k) (lift_rec N k)
  end.

Definition lift (N : lambda) := lift_rec N 0.

```

```
(*-------------------------- Substitution --------------------------*)

Definition insert_Ref (N : lambda) (i k : nat) :=
  match compare k i with

      (* k<i *) | inleft (left _) ⇒ Ref (pred i)
   (* k=i *) | inleft _ ⇒ N
   (* k>i *) | _ ⇒ Ref i
  end.

Fixpoint subst_rec (L : lambda) : lambda → nat → lambda :=
  fun (N : lambda) (k : nat) ⇒
  match L with
  | Ref i ⇒ insert_Ref N i k
  | Abs M ⇒ Abs (subst_rec M (lift_rec N 0) (S k))
  | App M M' ⇒ App (subst_rec M N k) (subst_rec M' N k)
  end.

Definition subst (N M : lambda) := subst_rec M N 0.

(*------------------------- one step beta-reduction -------------------------*)

Inductive red1 : lambda → lambda → Prop :=
  | beta : forall M N : lambda, red1 (App (Abs M) N) (subst N M)
  | abs_red : forall M N : lambda, red1 M N → red1 (Abs M) (Abs N)
  | app_red_l :
      forall M1 N1 : lambda,
      red1 M1 N1 → forall M2 : lambda, red1 (App M1 M2) (App N1 M2)
  | app_red_r :
      forall M2 N2 : lambda,
      red1 M2 N2 → forall M1 : lambda, red1 (App M1 M2) (App M1 N2).

(*---------------- Reflevixe-transitive closure of beta-reduction ----------------*)
```

```
Inductive red : lambda → lambda → Prop :=
  | one_step_red : forall M N : lambda, red1 M N → red M N
  | refl_red : forall M : lambda, red M M
  | trans_red : forall M N P : lambda, red M N → red N P → red M P.

(*--------------------- Auxiliar Lemmas for beta-reduction ---------------------*)

Lemma red_appl :
 forall M M' : lambda,
 red M M' → forall N : lambda, red (App M N) (App M' N).
Proof.
simple induction 1; intros.
apply one_step_red; apply app_red_l; trivial.
apply refl_red.
apply trans_red with (App N N0); trivial.
Qed.

Lemma red_appr :
 forall M M' : lambda,
 red M M' → forall N : lambda, red (App N M) (App N M').
Proof.
simple induction 1; intros.
apply one_step_red; apply app_red_r; trivial.
apply refl_red.
apply trans_red with (App N0 N); trivial.
Qed.

Lemma red_abs : forall M M' : lambda, red M M' → red (Abs M) (Abs M').
Proof.
simple induction 1; intros.
apply one_step_red; apply abs_red; trivial.
apply refl_red.
```

```
apply trans_red with (Abs N);  trivial.
Qed.

(*---------------------- one step cbn evaluation ⟶ n ----------------------*)

Inductive name_eval_1 : lambda ⟶ lambda ⟶ Prop :=
  | beta_name_eval : forall M N : lambda, name_eval_1 (App (Abs M) N) (subst N M)
  | app_red_name_eval_1 :
      forall M1 N1 : lambda,
      name_eval_1 M1 N1 ⟶ forall M2 : lambda, name_eval_1 (App M1 M2) (App N1 M2).

(*----- Call-by-name evaluation: Reflexive-transitive closure of ⟶ n -----*)

Inductive name_eval : lambda ⟶ lambda ⟶ Prop :=
  | one_step_name_eval : forall M N : lambda, name_eval_1 M N ⟶ name_eval M N
  | refl_name_eval : forall M : lambda, name_eval M M
  | trans_name_eval : forall M N P : lambda, name_eval M N ⟶ name_eval N P ⟶ name_eval M P.

(*--------------------------- Auxiliar Lemma for cbn ------------------------*)

Lemma right_apl_n : forall M1 M2 N : lambda,
name_eval M1 M2 ⟶ name_eval (App M1 N) (App M2 N).
Proof.
intros M1 M2 N H. induction H.
(* Base case: *)
apply one_step_name_eval.
apply app_red_name_eval_1; trivial.
apply refl_name_eval.
apply trans_name_eval with (App N0 N);  trivial.
Qed.

(*--------------------------- Standard reduction ( ⟹ n ) ---------------------------*)

```

```
Inductive standard_red : lambda → lambda → Prop :=
  | VAR : forall i : nat, standard_red (Ref i) (Ref i)
  | ABS : forall M N : lambda, standard_red M N → standard_red (Abs M) (Abs N)
  | APL : forall M1 M2 N1 N2 : lambda, standard_red M1 M2 → standard_red N1 N2 →
  standard_red (App M1 N1) (App M2 N2)
  | RDX : forall M1 M2 N P : lambda, name_eval (M1) (Abs M2) → standard_red (subst N M2) (P)
  → standard_red (App M1 N) (P).

(*------------------- Properties of substitution and lifting -------------------*)

Require Import Lia.

Lemma prop_1 : forall M N : lambda, forall k : nat, subst_rec (lift_rec M k) N k = M.
Proof.
induction M.

(*VAR case: *)
intros N k.
unfold lift_rec.
unfold relocate.
destruct (test k n) eqn:H0.
    (* subcase k <= n: *)
simpl.
unfold insert_Ref.
destruct (compare k (S n)) eqn:H1.
destruct s.
          (*subsubcase k < S n: *)
simpl. trivial.
          (* subcases k = S n and k > S n, are impossible! *)
lia. lia.
    (* subcase k > n: *)
simpl.
unfold insert_Ref.
```

```
destruct (compare k n) eqn:H1.
destruct s.
         (* subcases k < n and k = n, are impossible! *)
lia. lia.
         (* subcases k > n : *)
trivial.

(*ABS case: *)
intros N k.
simpl.
assert (H: subst_rec (lift_rec M (S k)) (lift_rec N 0) (S k) = M).
apply IHM.
rewrite → H. trivial.

(*APL case: *)
intros N k.
simpl.
rewrite → IHM1.
rewrite → IHM2.
trivial.

Qed.


Lemma prop_2 : forall M : lambda, forall k i : nat,
k<=i → lift_rec (lift_rec M k) (S i) = lift_rec (lift_rec M i) k.
Proof.
induction M.

(*VAR case: *)
intros k i H.
simpl.
unfold relocate.
```

```
destruct (test k n) eqn:H0.
    (* subcase k < = n : *)
destruct (test i n) eqn:H1.
           (* subcase i < = n : *)
destruct (test (S i) (S n)) eqn:H2.
                  (* subcase S i < = S n : *)
destruct (test k (S n)) eqn:H3.
                           (*subcase k < = S n : *)
trivial.
                           (* subcase k > S n is impossible: *)
lia.
                  (* subcase S i > S n : *)
destruct (test k (S n)) eqn:H4.
                           (* subcase k < = S n is impossible: *)
lia.
                           (* subcase k > S n : *)
trivial.
            (* subcase i>n : *)
destruct (test (S i) (S n)) eqn:H2.
                  (* subcase S i < = S n is impossible: *)
lia.
                  (* subcase S i > S n : *)
destruct (test k n) eqn:H3.
trivial. lia.
    (* subcase k > n : *)
destruct (test (S i) n) eqn:H1.
           (* subcase S i < = n is impossible : *)
lia.
           (* subcase S i > n : *)
destruct (test i n) eqn:H2.
                  (* subcase i < = n is impossible : *)
lia.
                  (* subcase i > n : *)
```

```
destruct (test k n) eqn:H3. lia. trivial.

(*ABS case: *)
intros k i H.
simpl.
assert (H0: (S k) <= (S i) ).
lia.
assert (H1: (lift_rec (lift_rec M (S k)) (S (S i))) = (lift_rec (lift_rec M (S i)) (S k))).
pose proof IHM as pp.
specialize pp with (l:= H0). trivial.
rewrite <- H1. trivial.

(*APL case: *)
intros k i H.
simpl.
rewrite -> IHM1.
rewrite -> IHM2.
trivial. trivial. trivial.
Qed.

(* If n > 0, then S(n-1) = n *)
Lemma pred_n : forall n : nat, n>0 -> S (Init.Nat.pred n) = n.
Proof.
intro n. intro H.
induction n.
(* H: 0 > 0 is absurd *)
lia.
(* H: S n > 0 *)
simpl. trivial.
Qed.


Lemma prop_3 : forall M N : lambda, forall k i : nat,
```

```
k<=i -> lift_rec (subst_rec M N i) k = subst_rec (lift_rec M k) (lift_rec N k) (S i).
Proof.
induction M.

(*VAR case: *)
intros N k i H.
unfold subst_rec at 1.
unfold insert_Ref at 1.
destruct (compare i n) eqn:H0.
destruct s.
    (* subcase i < n : *)
unfold lift_rec at 1.
unfold relocate at 1.
destruct (test k (Init.Nat.pred n)) eqn:H1.
         (* subcase k <= n-1 : *)
unfold lift_rec at 1.
unfold relocate at 1.
destruct (test k n) eqn:H2.
              (* subcase k <= n : *)
simpl.
unfold insert_Ref at 1.
destruct (compare (S i) (S n)) eqn:H3.
destruct s.
                     (* subcase S i < S n : *)
simpl.
assert (H4: S (Init.Nat.pred n)=n).
apply pred_n. lia.
rewrite -> H4. trivial.
                     (* subcase S i = S n and S i > S n, are impossible: *)
lia. lia.
              (* subcase k > n is impossible: *)
lia.
         (* subcase k > n-1 is impossible: *)
```

```
lia.
    (* subcase i = n : *)
unfold lift_rec at 2.
unfold relocate.
destruct (test k n) eqn:H1.
              (* subcase k <= n : *)
simpl.
unfold insert_Ref.
destruct (compare (S i) (S n)) eqn:H2.
destruct s.
                        (* subcase S i < S n is impossible: *)
lia.
                        (* subcase S i = S n : *)
trivial.
                        (* subcase S i > S n is impossible: *)
lia.
              (* subcase k > n is impossible: *)
lia.
    (* subcase i > n : *)
unfold lift_rec at 1.
unfold relocate.
destruct (test k n) eqn:H1.
              (* subcase k <= n : *)
unfold lift_rec at 1.
unfold relocate.
destruct (test k n) eqn:H2.
simpl.
unfold insert_Ref.
destruct (compare (S i) (S n)) eqn: H3.
destruct s.
                        (* subcase S i < S n is impossible: *)
lia.
                        (* subcase S i = S n is impossible: *)
```

```
lia.
                    (* subcase S i > S n : *)
trivial.
lia.
             (* subcase k > n : *)
unfold lift_rec at 1.
unfold relocate.
destruct (test k n) eqn:H2. lia.
simpl.
unfold insert_Ref.
destruct (compare (S i) n) eqn:H3.
destruct s.
                    (*subcase S i < n and S i = n are impossible: *)
lia. lia.
                    (*subcase S i > n : *)
trivial.

(*ABS case: *)
intros N k i H.
simpl.
assert (H0 : 0<=k).
lia.
assert (H1 : lift_rec (lift_rec N 0) (S k) = lift_rec (lift_rec N k) 0).
pose proof prop_2 as pp.
specialize pp with (1 := H0). trivial.
rewrite <- H1.
assert (H2: (S k) <= (S i) ).
lia.
assert (H3 : (lift_rec (subst_rec M (lift_rec N 0) (S i)) (S k)) =
(subst_rec (lift_rec M (S k)) (lift_rec (lift_rec N 0) (S k)) (S (S i)))).
pose proof IHM as pp.
specialize pp with (1:= H2). trivial.
rewrite <- H3. trivial.
```

```
(*APL case: *)
intros N k i H.
simpl.
rewrite ⟶ IHM1.
rewrite ⟶ IHM2.
trivial. trivial. trivial.

Qed.


Lemma prop_4 : forall M N : lambda, forall k i : nat,
k<=i ⟶ lift_rec (subst_rec M N k) i = subst_rec (lift_rec M (S i)) (lift_rec N i) k.
Proof.
induction M.

(*VAR case: *)
intros N k i H.
unfold subst_rec at 1.
unfold insert_Ref.
destruct (compare k n) eqn:H0.
destruct s.
   (* subcase k < n *)
unfold lift_rec at 1.
unfold relocate.
destruct (test i (Init.Nat.pred n)) eqn:H1.
unfold lift_rec at 1.
unfold relocate.
destruct (test (S i) n) eqn:H2.
unfold subst_rec.
unfold insert_Ref.
destruct (compare k (S n)) eqn:H3.
destruct s. simpl.
```

```
assert (H4: S (Init.Nat.pred n)=n).
apply pred_n. lia.
rewrite → H4. trivial. lia. lia. lia.
unfold lift_rec at 1.
unfold relocate.
destruct (test (S i) n) eqn:H2.
unfold subst_rec.
unfold insert_Ref.
destruct (compare k (S n)) eqn:H3.
destruct s. lia. lia. lia.
unfold subst_rec.
unfold insert_Ref.
destruct (compare k n) eqn:H3.
destruct s. trivial. lia. lia.
     (* subcase k = n *)
unfold lift_rec at 2.
unfold relocate.
destruct (test (S i) n) eqn:H1.
unfold subst_rec.
unfold insert_Ref.
destruct (compare k (S n)) eqn:H2.
destruct s. lia. trivial. lia.
unfold subst_rec.
unfold insert_Ref.
destruct (compare k n) eqn:H2.
destruct s. lia. trivial. lia.
     (* subcase k > n*)
unfold lift_rec at 2.
unfold relocate.
destruct (test (S i) n) eqn:H1.
unfold subst_rec.
unfold insert_Ref.
destruct (compare k (S n)) eqn:H2.
```

```
destruct s. lia. lia. lia.
unfold subst_rec.
unfold insert_Ref.
destruct (compare k n) eqn:H2.
destruct s. lia. lia.
simpl.
unfold relocate.
destruct (test i n) eqn:H3.
lia. trivial.

(*ABS case: *)
intros N k i H.
simpl.
assert (H0 : 0<=i).
lia.
assert (H1 : lift_rec (lift_rec N 0) (S i) = lift_rec (lift_rec N i) 0).
pose proof prop_2 as pp.
specialize pp with (1 := H0). trivial.
rewrite ← H1.
assert (H2: (S k) <= (S i) ).
lia.
assert (H3 : (lift_rec (subst_rec M (lift_rec N 0) (S k)) (S i)) =
(subst_rec (lift_rec M (S (S i))) (lift_rec (lift_rec N 0) (S i)) (S k))).
pose proof IHM as pp.
specialize pp with (1:= H2). trivial.
rewrite ← H3. trivial.

(*APL case: *)
intros N k i H.
simpl.
rewrite → IHM1.
rewrite → IHM2.
trivial. trivial. trivial.
```

```
Qed.

(*-------------------------------------------------- -------------------------------*)

(*------------------------------- Substitution Lemma -------------------------------*)

Lemma substitution_lemma : forall M N Q : lambda, forall i k : nat,
k<=i -> subst_rec (subst_rec M N k) Q i =
subst_rec (subst_rec M (lift_rec Q k) (S i)) (subst_rec N Q i) k.
Proof.
induction M.

(*VAR case: *)
intros N Q i k H.
unfold subst_rec at 2.
unfold insert_Ref.
destruct (compare k n) eqn:H0.
destruct s.
    (* k < n *)
unfold subst_rec at 3.
unfold insert_Ref.
destruct (compare (S i) n) eqn:H1.
destruct s.
unfold subst_rec at 2.
unfold insert_Ref.
destruct (compare k (Init.Nat.pred n)) eqn:H2.
destruct s.
unfold subst_rec.
unfold insert_Ref.
destruct (compare i (Init.Nat.pred n)) eqn:H3.
destruct s. trivial. lia. lia. lia. lia.
unfold subst_rec at 1.
```

```
unfold insert_Ref.
destruct (compare i (Init.Nat.pred n)) eqn:H3.
destruct s. lia.
assert (H4: subst_rec (lift_rec Q k) (subst_rec N Q i) k = Q).
apply prop_1.
rewrite → H4. trivial.
lia.
unfold subst_rec at 1.
unfold insert_Ref.
destruct (compare i (Init.Nat.pred n)) eqn:H2.
destruct s. lia. lia.
unfold subst_rec at 1.
unfold insert_Ref.
destruct (compare k n) eqn:H3.
destruct s. trivial. lia. lia.
     (* k = n *)
unfold subst_rec at 3.
unfold insert_Ref.
destruct (compare (S i) n) eqn:H1.
destruct s.
lia. lia.
unfold subst_rec at 2.
unfold insert_Ref.
destruct (compare k n) eqn:H2.
destruct s. lia. trivial. lia.
     (* k > n *)
unfold subst_rec at 3.
unfold insert_Ref.
destruct (compare (S i) n) eqn:H1.
destruct s.
lia. lia.
unfold subst_rec at 2.
unfold insert_Ref.
```

```
destruct (compare k n) eqn:H2.
destruct s. lia. lia.
unfold subst_rec.
unfold insert_Ref.
destruct (compare i n) eqn:H3.
destruct s. lia. lia. trivial.

(*ABS case: *)
intros N Q i k H.
simpl.
assert (H1 : 0 <= k).
lia.
assert (H2 : lift_rec (lift_rec Q 0) (S k) = lift_rec (lift_rec Q k) 0).
pose proof prop_2 as pp.
specialize pp with (1 := H1). trivial.
rewrite <- H2.
assert (H3 : 0 <= i).
lia.
assert (H4 : lift_rec (subst_rec N Q i) 0 = subst_rec (lift_rec N 0) (lift_rec Q 0) (S i)).
pose proof prop_3 as pp.
specialize pp with (1 := H3). trivial.
rewrite -> H4.
assert (H5 : (S k) <= (S i)). lia.
assert (H6 : (subst_rec (subst_rec M (lift_rec N 0) (S k)) (lift_rec Q 0) (S i)) =
subst_rec (subst_rec M (lift_rec (lift_rec Q 0) (S k)) (S (S i))) (subst_rec (lift_rec N 0)
(lift_rec Q 0) (S i)) (S k)).
pose proof IHM as pp.
specialize pp with (1 := H5). trivial.
rewrite -> H6. trivial.


(* APL case: *)
intros N Q i k H.
```

```
simpl.
rewrite ← IHM1.
rewrite ← IHM2.
trivial.
lia.
lia.

Qed.

(*--------------------------------------------------------------------------------*)

(*------------------------- Admissible rules (1) to (8) for ⇒ n --------------------*)

Lemma rule_1 : forall M : lambda, standard_red M M.
Proof.
intro M. induction M.
(*M = Ref n *)
apply VAR.
(*M = Abs M *)
apply ABS. trivial.
(*M = M1 M2 *)
apply APL. trivial. trivial.
Qed.

    (*---------- Auxiliar Lemmas to prove Rule 2 -------------*)

Lemma lift_1: forall M N : lambda, forall i:nat, name_eval_1 M N →
name_eval_1 (lift_rec M i) (lift_rec N i).
Proof.
simple induction 1.
intros M0 N0.
unfold subst.
rewrite prop_4; auto with arith.
```

```
unfold lift_rec at 1.
apply beta_name_eval.
intros.
unfold lift_rec.
apply app_red_name_eval_1; auto with arith.

Qed.


Lemma lift_n: forall M N : lambda, name_eval M N ->
forall i :nat, name_eval (lift_rec M i) (lift_rec N i).
Proof.
simple induction 1; intros.

(* Base case: *)
apply one_step_name_eval.
apply lift_1.
trivial.

(* Reflexice case: *)
apply refl_name_eval.

(* Transitive case: *)
apply trans_name_eval with ((lift_rec N0 i)).
auto. auto.

Qed.

Lemma lift_i : forall N1 N2 : lambda, standard_red N1 N2 ->
forall i: nat, standard_red (lift_rec N1 i) (lift_rec N2 i).
Proof.
intro N1. intro N2. intro H.
induction H.
```

```
(*VAR case: *)
intro i0. apply rule_1.

(*ABS case: *)
intro i. simpl.
assert (H1: standard_red (lift_rec M (S i)) (lift_rec N (S i))).
apply IHstandard_red.
pose proof ABS as pp.
specialize pp with (1:= H1). trivial.

(*APL case: *)
intro i.
simpl.
assert (H1: standard_red (lift_rec M1 i) (lift_rec M2 i)).
apply IHstandard_red1.
assert (H2: standard_red (lift_rec N1 i) (lift_rec N2 i)).
apply IHstandard_red2.
pose proof APL as pp.
specialize pp with (1:= H1) (2:= H2). trivial.

(*RDX case: *)
intro i. simpl.
assert (H1: name_eval (lift_rec M1 i) (lift_rec (Abs M2) i) ).
apply lift_n. trivial.
assert (H2: name_eval (lift_rec M1 i) (Abs (lift_rec M2 (S i))) ).
simpl in H1. trivial.
assert (H3: lift_rec (subst_rec M2 N 0) i = subst_rec (lift_rec M2 (S i)) (lift_rec N i) 0 ).
apply prop_4. lia.
assert (H4: standard_red (lift_rec (subst N M2) i) (lift_rec P i)).
trivial. unfold subst in H4.
rewrite → H3 in H4.
pose proof RDX as pp.
```

```
specialize pp with (1:= H2) (2:= H4). trivial.

Qed.

Lemma subs_name_eval_1 : forall M1 M2 N : lambda, forall i : nat, name_eval_1 M1 M2 →
name_eval_1 (subst_rec M1 N i) (subst_rec M2 N i).
Proof.
simple induction 1.

(* beta case: *)
intros.
unfold subst.
rewrite substitution_lemma; auto with arith.
unfold subst_rec at 1.
apply beta_name_eval.
(* \mu case: *)
intros.
apply app_red_name_eval_1; auto with arith.
Qed.

Lemma subs_name_eval : forall M1 M2 N : lambda, forall i : nat, name_eval M1 M2 →
name_eval (subst_rec M1 N i) (subst_rec M2 N i).
Proof.
simple induction 1; intros.

(* Base case: *)
apply one_step_name_eval.
apply subs_name_eval_1. trivial.

(* Reflexice case: *)
apply refl_name_eval.

(* Transitive case: *)
```

```
apply trans_name_eval with ((subst_rec N0 N i)).
auto. auto.

Qed.

    (*-------------------------------------------------------*)

Lemma rule_2 : forall M1 M2 : lambda, standard_red M1 M2 → forall N1 N2 : lambda,
standard_red N1 N2 → forall i:nat, standard_red (subst_rec M1 N1 i) (subst_rec M2 N2 i).
Proof.
intro M1. intro M2. intro H.
induction H.

(* Var case: *)
intros N1 N2 H i0.
unfold subst_rec.
unfold insert_Ref.
destruct (compare i0 i) eqn:H0.
destruct s.
apply rule_1.
trivial.
apply rule_1.

(* ABS case: *)
intros N1 N2 H0 i.
simpl.
assert (H2: standard_red (lift_rec N1 0) (lift_rec N2 0)).
apply lift_i. trivial.
assert (H3 : forall i: nat,
standard_red (subst_rec M (lift_rec N1 0) i) (subst_rec N (lift_rec N2 0) i)).
pose proof IHstandard_red as pp.
specialize pp with (1:= H2). trivial.
assert (H4: standard_red (subst_rec M (lift_rec N1 0) (S i))
```

```
(subst_rec N (lift_rec N2 0) (S i))).
apply H3.
pose proof ABS as pp.
specialize pp with (1:= H4). trivial.

(*APL case: *)
intros N0 N3 H1 i. simpl.
assert (H2: standard_red (subst_rec M1 N0 i) (subst_rec M2 N3 i)).
pose proof IHstandard_red1 as pp. specialize pp with (1:= H1). trivial.
assert (H3: standard_red (subst_rec N1 N0 i) (subst_rec N2 N3 i)).
pose proof IHstandard_red2 as pp. specialize pp with (1:= H1). trivial.
pose proof APL as pp. specialize pp with (1:= H2) (2:= H3). trivial.

(*RDX case: *)
intros N1 N2 H1 i. simpl. unfold subst in H0.
assert (H2: subst_rec (subst_rec M2 N 0) N1 i =
subst_rec (subst_rec M2 (lift_rec N1 0) (S i)) (subst_rec N N1 i) 0 ).
apply substitution_lemma. lia.
unfold subst in IHstandard_red.
assert (H3: standard_red (subst_rec (subst_rec M2 N 0) N1 i) (subst_rec P N2 i)).
pose proof IHstandard_red as pp. specialize pp with (1:= H1). trivial.
assert (H4: standard_red (subst_rec (subst_rec M2 (lift_rec N1 0) (S i))
(subst_rec N N1 i) 0) (subst_rec P N2 i)).
rewrite ← H2. trivial.
assert (H5: name_eval (subst_rec M1 N1 i) (subst_rec (Abs M2) N1 i)).
apply subs_name_eval. trivial.
simpl in H5.
pose proof RDX as pp. specialize pp with (1:= H5) (2:= H4). trivial.

Qed.

Lemma rule_3 : forall M N : lambda, name_eval_1 M N → forall P : lambda, standard_red N P →
standard_red M P.
```

```
Proof.
intro M. intro N. intro H. induction H.

(* beta_n case: *)
intros P H.
assert (H1: name_eval (Abs M) (Abs M)). apply refl_name_eval.
pose proof RDX as pp.
specialize pp with (1 := H1) (2 := H); trivial.

(* mu case: *)
intros P H0.
inversion H0.
    (* APL subcase: *)
assert (H6: standard_red M1 M3).
pose proof IHname_eval_1 as pp.
specialize pp with (1:= H3). trivial.
pose proof APL as pp.
specialize pp with (1 := H6) (2 := H5); trivial.

    (* RDX subcase: *)
assert (H6: name_eval M1 (Abs M3)).
apply trans_name_eval with (N1); trivial.
apply one_step_name_eval; trivial.
pose proof RDX as pp.
specialize pp with (1 := H6) (2 := H5); trivial.

Qed.

Lemma rule_4 : forall M N P : lambda, name_eval M N → standard_red N P → standard_red M P.
Proof.
intros M N P H H0. induction H.
(*Base case: *)
pose proof rule_3 as pp.
```

```
specialize pp with (1 := H) (2 := H0); trivial.

(*Reflexive case: *)
trivial.

(*Transitive case: *)
apply IHname_eval1.
apply IHname_eval2.
trivial.

Qed.

Lemma rule_5_linha : forall M1 M3 N1 N2 : lambda, standard_red M1 M3 → forall M2 : lambda,
M3 = Abs M2 → standard_red N1 N2 → standard_red (App M1 N1) (subst N2 M2).
Proof.
intros. induction H.
inversion H0.
inversion H0.
assert (H4: name_eval (App (Abs M) N1) (subst N1 M)).
apply one_step_name_eval.
apply beta_name_eval.
assert (H5: standard_red (subst N1 M) (subst N2 M2)).
pose proof rule_2 as pp.
unfold subst.
specialize pp with (1 := H) (2 := H1); auto.
rewrite ← H3. trivial.
pose proof rule_4 as pp.
specialize pp with (1 := H4) (2 := H5); trivial.
inversion H0.
assert (H5: name_eval (App M1 N) (App (Abs M0) N)).
apply right_apl_n; trivial.
assert (H6: name_eval (App (Abs M0) N) (subst N M0)).
apply one_step_name_eval.
```

```
apply beta_name_eval.
assert (H7: name_eval (App M1 N) (subst N M0) ).
apply trans_name_eval with (App (Abs M0) N); trivial.
assert (H8: name_eval (App (App M1 N) N1) (App (subst N M0) N1) ).
apply right_apl_n; trivial.
rewrite → H0 in H2.
assert (H9: standard_red (App (subst N M0) N1) (subst N2 M2)).
apply IHstandard_red. trivial.
pose proof rule_4 as pp.
specialize pp with (1:= H8) (2:= H9). trivial.
Qed.

Lemma rule_5 : forall M1 M2 N1 N2 : lambda, standard_red M1 (Abs M2) → standard_red N1 N2 →
standard_red (App M1 N1) (subst N2 M2).
Proof.
intros.
pose proof rule_5_linha as pp.
specialize pp with (1:= H).
apply pp. trivial. trivial.
Qed.

Lemma rule_6_linha : forall M1 M2 : lambda, standard_red M1 M2 → forall M3 N : lambda,
M2 = App (Abs M3) N → standard_red M1 (subst N M3).
Proof.
intros. induction H.
inversion H0.
inversion H0.
inversion H0.
rewrite → H3 in H.
rewrite → H4 in H1.
pose proof rule_5 as pp.
specialize pp with (1 := H) (2 := H1). trivial.
assert (H5: standard_red (subst N0 M2) (subst N M3)).
```

```
apply IHstandard_red. trivial.
pose proof RDX as pp.
specialize pp with (1 := H) (2 := H5).  trivial.
Qed.



Lemma rule_6 : forall M1 M3 N0 : lambda, standard_red M1 (App (Abs M3) (N0)) →
standard_red M1 (subst N0 M3).
Proof.
intros.
pose proof rule_6_linha as pp.
specialize pp with (1:= H).
apply pp. trivial.

Qed.


Lemma rule_7 : forall M N : lambda, standard_red M N → forall P : lambda, red1 N P →
standard_red M P.
Proof.
intro M. intro N. intro H. induction H.

(*VAR case:
impossible case: *)
intros P H. inversion H.


(*ABS case: *)
intros P H0.
inversion H0.
assert (H4: standard_red M N0).
pose proof IHstandard_red as pp.
```

```
specialize pp with (1:= H2). trivial.
pose proof ABS as pp.
specialize pp with (1:= H4). trivial.
(*APL case: *)
intros P H1.
inversion H1.
    (*beta_n subcase: *)
rewrite ← H3 in H.
assert (H5: standard_red (App M1 N1) (App (Abs M) N2)).
pose proof APL as pp.
specialize pp with (1:= H) (2:= H0). trivial.
pose proof rule_6 as pp.
specialize pp with (1:= H5). trivial.
   (*mu subcase: *)
assert (H6: standard_red M1 N0).
pose proof IHstandard_red1 as pp.
specialize pp with (1:= H5). trivial.
pose proof APL as pp.
specialize pp with (1:= H6) (2:= H0). trivial.
   (*V subcase: *)
assert (H6: standard_red N1 N0).
pose proof IHstandard_red2 as pp.
specialize pp with (1:= H5). trivial.
pose proof APL as pp.
specialize pp with (1:= H) (2:= H6). trivial.
(*RDX case: *)
intros P0 H1.
assert (H2: standard_red (subst N M2) P0).
pose proof IHstandard_red as pp.
specialize pp with (1:= H1). trivial.
pose proof RDX as pp.
```

```
specialize pp with (1:= H) (2:= H2). trivial.

Qed.

Lemma rule_8 : forall M N P : lambda, standard_red M N → red N P → standard_red M P.
Proof.
intros M N P H H0. induction H0.
(*Base case: *)
pose proof rule_7 as pp.
specialize pp with (1 := H) (2 := H0); trivial.
(*Reflexive case: *)
trivial.
(*Transitive case: *)
apply IHred2.
apply IHred1.
trivial.
Qed.

(*-------------------------------------------------- ---------------------------*)

(*--------------------------- Standardization Theorem ---------------------------*)

Theorem standardization : forall M N : lambda, red M N ↔ standard_red M N.
Proof.
split.

(*"Only if" direction: *)
intro H. induction H.
(*Base case: *)
assert (H1: standard_red M M).
apply rule_1.
pose proof rule_7 as pp.
specialize pp with (1 := H1) (2 := H); trivial.
```

```
(*Reflexice case: *)
apply rule_1.
(*Transitive case: *)
pose proof rule_8 as pp.
specialize pp with (1 := IHred1) (2 := H0); trivial.

(*"If" direction: *)
intro H. induction H.
(* VAR case: M = Ref i and N = Ref i *)
apply refl_red.
(* ABS case: M = Abs M' and N = Abs N' *)
apply red_abs. trivial.
(* APL case: M = App M1 N1 and N = M2 N2 *)
assert (H1: red (App M1 N1) (App M2 N1)).
apply red_appl. trivial.
assert (H2: red (App M2 N1) (App M2 N2)).
apply red_appr. trivial.
apply trans_red with (App M2 N1). trivial. trivial.
(* RDX case: M = App M1 N*)
assert (H1: red M1 (Abs M2)).
induction H.
apply one_step_red.
induction H.
apply beta.
apply app_red_l. trivial.
apply refl_red.
apply trans_red with (N0); trivial.
assert (H2: red (App M1 N) (App (Abs M2) N)).
apply red_appl. trivial.
assert (H3: red1 (App (Abs M2) N) (subst N M2)).
apply beta.
assert (H4: red (subst N M2) P). trivial. apply trans_red with (App (Abs M2) N).
trivial. apply trans_red with (subst N M2).
```

```
apply one_step_red in H3.
 trivial. trivial.

Qed.

(*------------------------------------------------------------------------------*)

(*------------------------- Corollary: Transitivity of ⇒ n -------------------------*)

Theorem rule_9 : forall M N P : lambda,
standard_red M N → standard_red N P → standard_red M P.
Proof.
intros M N P H1 H2.
assert (H3: red N P).
apply standardization. trivial.
pose proof rule_8 as pp.
apply pp with N. trivial. trivial.
Qed.
```

Appendix F

This appendix contains the full Coq code for the theory of λ-calculus with the De Bruijn indices, introduces the definition of standard reduction sequence, proves the equivalence between the standard reduction relation and the standard reduction sequences approaches, i.e., formalizes all the results corresponding to Chapter 5. The code below was developed under version 8.12.2 of the Coq proof assistant.

```

(*------------------------ Standard Reduction Sequence ------------------------*)

(*------------------------- Lists of lambda terms -------------------------*)

Inductive term_list : Set :=
  | nil
  | cons (M : lambda) (L : term_list).

Notation "M :: L" := (cons M L).
Notation "[ ]" := nil.

(*------------------- Append: concatenates (appends) two lists ------------------- *)

Fixpoint app (L1 L2 : term_list) : term_list :=
  match L1 with
  | nil => L2
  | h :: t => h :: (app t L2)
  end.
```

```
Notation "L1 · L2" := (app L1 L2) (at level 50) : type_scope.

Lemma concatenate_assoc : forall L1 L2 L3 : term_list, (L1 · L2) · L3 = L1 · (L2 · L3).
Proof.
intros L1 L2 L3.
induction L1.
simpl. trivial.
simpl.
rewrite ← IHL1.
trivial.
Qed.

(*------------------------------ Auxiliar functions ------------------------------*)

Fixpoint Abs_list (L : term_list) : term_list :=
  match L with
  | nil ⇒ nil
  | M :: L1 ⇒ Abs M :: Abs_list (L1)
  end.

Fixpoint Apl_arg (L : term_list) : lambda → term_list :=
  fun N : lambda ⇒
  match L with
  | nil ⇒ nil
  | M :: L1 ⇒ (App M N) :: (Apl_arg L1 N)
  end.

Fixpoint Apl_fun (L : term_list) : lambda → term_list :=
  fun M : lambda ⇒
  match L with
  | nil ⇒ nil
  | N :: L1 ⇒ (App M N) :: (Apl_fun L1 M)
```

```
  end.

(*------------------- Standard Reduction Sequences (s.r.s.) -------------------*)

Inductive standard_red_seq : term_list → Prop :=
  | VAR' : forall i : nat, standard_red_seq ((Ref i) :: [])
  | ABS' : forall L : term_list, standard_red_seq L → standard_red_seq (Abs_list L)
  | APL' : forall L1 L2 : term_list, forall M N : lambda, standard_red_seq (L1 · (M :: []))
  → standard_red_seq (N :: L2) →
  standard_red_seq (Apl_arg L1 N · (( App M N) :: []) · Apl_fun L2 M )
  | RDX' : forall N1 N2 : lambda, forall L : term_list, name_eval_1 N1 N2 →
  standard_red_seq (N2 :: L) → standard_red_seq (N1 :: (N2 :: L)).


(*---------------------------------- Lemmas ----------------------------------*)

Lemma abs_lists : forall L1 L2 : term_list, Abs_list (L1 · L2) = Abs_list L1 · Abs_list L2.
Proof.
intros.
induction L1.
simpl. trivial.
simpl.
rewrite ← IHL1.
trivial.
Qed.

Lemma apl_fun_lists : forall L1 L2 : term_list, forall N : lambda, Apl_fun (L1 · L2) N =
(Apl_fun L1 N) · (Apl_fun L2 N).
Proof.
intros.
induction L1.
simpl. trivial.
simpl.
```

```
rewrite ← IHL1.
trivial.
Qed.

(*Lemma arg_fun_lists : forall L1 L2 : term_list, forall N : lambda,
Apl_arg (L1 · L2) N = (Apl_arg L1 N) ·(Apl_arg L2 N).
Proof.
intros.
induction L1.
simpl. trivial.
simpl.
rewrite ← IHL1.
trivial.
Qed.*)

Lemma single_list_srs : forall M : lambda, standard_red_seq (M :: [ ]).
Proof.
intro M.
induction M.
(* VAR case: *)
apply VAR'.
(* ABS case: *)
assert (H0: standard_red_seq (Abs_list (M :: [])) ).
pose proof ABS' as pp.
apply pp. trivial.
simpl in H0. trivial.
(* APL case: *)
assert (H0: standard_red_seq ((Apl_arg [] M2 · ( (App M1 M2) :: [ ])) · Apl_fun [] M1) ).
pose proof APL' as pp.
apply pp. simpl. trivial. trivial.
simpl in H0. trivial.
Qed.
```

```
(*-- Alternative characterization of cbn-evaluation --*)

Inductive name_eval_t : lambda → lambda → Prop :=
  | refl_name_eval_t : forall M : lambda, name_eval_t M M
  | trans_name_eval_t : forall M N P : lambda, name_eval_1 M N → name_eval_t N P →
  name_eval_t M P.

Lemma admissible_trans : forall M N P : lambda, name_eval_t M N → name_eval_t N P →
name_eval_t M P.
Proof.
intros.
induction H.
trivial.
assert (H2: name_eval_t N P).
apply IHname_eval_t. trivial.
apply trans_name_eval_t with (N). trivial. trivial.
Qed.

Lemma equiv_name_eval: forall M N : lambda, name_eval M N ↔ name_eval_t M N.
Proof.
intros.
split.
intro.
induction H.
apply trans_name_eval_t with (N). trivial.
apply refl_name_eval_t.
apply refl_name_eval_t.
apply admissible_trans with (N). trivial.
trivial.
intros.
induction H.
apply refl_name_eval.
apply trans_name_eval with (N).
```

```
apply one_step_name_eval. trivial. trivial.
Qed.


(*-----------------------------------------------------------------------*)

(*Equivalence between s.r.s. and ⇒ n *)

(*--------------------- Theorem 1: ⇒ n implies s.r.s. ---------------------*)


Require Import Coq.Program.Equality.

Lemma standard_red_1 : forall M N : lambda, standard_red M N → M = N ∨
(exists L : term_list, standard_red_seq (M :: L · (N :: []))).
Proof.
intros M N H.
induction H.

(* VAR case: *)
auto.

(* ABS case: *)
destruct IHstandard_red.
      (* H0: M = N *)
rewrite ← H0.
auto.
      (* H0 : exists L : term_list, standard_red_seq (M :: L ·(N :: [ ])) *)
destruct H0 as [L].
assert (H1: standard_red_seq (Abs M :: Abs_list (L · ( N :: [ ])))).
pose proof ABS' as pp.
specialize pp with (1:= H0).
simpl in pp. trivial.
```

```
assert (H2: Abs_list (L · (N :: [ ])) = Abs_list L · Abs_list (N :: [ ]) ).
apply abs_lists.
rewrite → H2 in H1.
simpl in H1.
right.
exists (Abs_list L). trivial.

(* APL case: *)
destruct IHstandard_red1.
destruct IHstandard_red2.
    (* M1 = M2 ∧ N1 = N2 *)
rewrite ← H1.
rewrite ← H2.
auto.
    (* M1 = M2 ∧ exists L : term_list, standard_red_seq (N1 :: L ·(N2 :: [ ])) *)
destruct H2 as [L2].
right.
exists (Apl_fun L2 M1).
assert (H3: standard_red_seq ((Apl_arg [] N1 · (( App M1 N1) :: []) ·
Apl_fun (L2 · (N2 :: [ ])) M1)) .
pose proof APL' as pp.
apply pp.
simpl.
apply single_list_srs.
trivial.
simpl in H3.
rewrite ← H1.
assert (H4: Apl_fun (L2 · (N2 :: [ ])) M1 = (Apl_fun L2 M1) · (Apl_fun (N2 :: [ ]) M1)).
apply apl_fun_lists.
rewrite → H4 in H3.
simpl in H3.
trivial.
destruct IHstandard_red2.
```

```

(*exists L : term_list, standard_red_seq (M1 :: L ·(M2 :: [ ])) ∧ N1 = N2*)
destruct H1 as [L1].
right.
exists (Apl_arg L1 N1).
assert (H3: standard_red_seq ((( Apl_arg (M1 :: L1)) N1 · (( App M2 N1) :: [])) ·
Apl_fun [] M2)) .
pose proof APL' as pp.
apply pp.
trivial.
apply single_list_srs.
simpl in H3.
rewrite ← H2.
assert (H4: (( Apl_arg L1 N1 · (App M2 N1 :: [ ])) · [ ]) = (Apl_arg L1 N1 ·
(( App M2 N1 :: [ ]) · [ ]))).
apply concatenate_assoc.
rewrite → H4 in H3.
simpl in H3. trivial.

(* exists L : term_list, standard_red_seq (M1 :: L ·(M2 :: [ ])) ∧
exists L : term_list, standard_red_seq (N1 :: L ·(N2 :: [ ])) *)
destruct H1 as [L1].
destruct H2 as [L2].
pose proof APL' as pp.
assert (H3: standard_red_seq (Apl_arg (M1 :: L1) N1 · ( (App M2 N1) :: [ ]) ·
Apl_fun (L2 · (N2 :: [ ])) M2) ).
apply pp. trivial. trivial.
simpl in H3.
right.
assert (H4: Apl_fun (L2 · (N2 :: [ ])) M2 = Apl_fun L2 M2 · Apl_fun (N2 :: [ ]) M2 ).
apply apl_fun_lists.
rewrite → H4 in H3.
simpl in H3.
```

```
exists (Apl_arg L1 N1 · ((App M2 N1) :: [ ]) · Apl_fun L2 M2).
assert (H5: ((Apl_arg L1 N1 · (App M2 N1 :: [ ])) · Apl_fun L2 M2) · (App M2 N2 :: [ ]) =
(Apl_arg L1 N1 · (App M2 N1 :: [ ])) · (Apl_fun L2 M2 · (App M2 N2 :: [ ])) ).
apply concatenate_assoc.
rewrite → H5. trivial.

(*RDX case: *)
assert (H10: name_eval_t M1 (Abs M2)).
apply equiv_name_eval. trivial.
destruct IHstandard_red.

    (* H1 : subst N M2 = P *)
dependent induction H10.
          (* Reflexive case: *)

right.
exists ([]).   simpl.
pose proof RDX' as pp.
apply pp.
apply beta_name_eval.
apply single_list_srs.

          (* Base/transitive case: *)
right.
assert (H2: App N0 N = subst N M2 ∨ (exists L : term_list, standard_red_seq(App N0 N :: L ·
(subst N M2 :: [ ]))) ).
specialize IHname_eval_t with (M2).
apply IHname_eval_t.
apply equiv_name_eval. trivial. trivial. trivial. trivial.
destruct H2.

        (* H2 : App N0 N = subst N M2 *)
rewrite ← H2.
```

```
exists ([]).    simpl.
pose proof RDX'  as pp.
apply pp.
apply app_red_name_eval_1. trivial.
apply single_list_srs.

(* H2 : exists L : term_list, standard_red_seq (App N0 N :: L ·(subst N M2 :: [ ])) *)
destruct H2 as [L1].
exists (App N0 N ::  L1).  simpl.
pose proof RDX'  as pp.
apply pp.
apply app_red_name_eval_1. trivial. trivial.


    (* H1 : standard_red_seq (subst N M2 :: L1 ·(P :: [ ])) *)
destruct H1 as [L1].
dependent induction H10.
          (* Reflexive case: *)
right.
exists (subst N M2 ::  L1).
simpl.
pose proof RDX'  as pp.
apply pp.
apply beta_name_eval. trivial.

          (* Base/transitive case:*)
right.
assert (H3:  App N0 N = P ∨
(exists L :  term_list, standard_red_seq (App N0 N ::  L ·  (P ::  [ ])))   ).
specialize IHname_eval_t with (M2).
apply IHname_eval_t.
apply equiv_name_eval. trivial. trivial. trivial. trivial.
destruct H3.
```

```
                (* H3: App N0 N = P *)
exists ([]).
simpl.
pose proof RDX' as pp.
apply pp.
rewrite ← H3.
apply app_red_name_eval_1. trivial.
apply single_list_srs.

(* H3 : exists L : term_list, standard_red_seq (App N0 N :: L ·(P :: [ ])) *)
destruct H3 as [L2].
exists (App N0 N :: L2).
simpl.
pose proof RDX' as pp.
apply pp.
apply app_red_name_eval_1. trivial. trivial.

Qed.

(*---------------------------------------------------------------------------*)

(*------------- Auxiliar Lemmas to prove s.r.s. implies ⇒ n -------------*)


Lemma aux_1 : forall M N : lambda, forall L0 L : term_list, Abs_list L0 = M :: N :: L →
exists M0 : lambda, M = Abs M0 ∧ (exists N0 : lambda, N = Abs N0 ∧ (exists L2 : term_list, L
= Abs_list L2 ∧ L0 = M0 :: N0 :: L2)).
Proof.
dependent induction L.
intros.
dependent induction L0.
inversion H.
inversion H.
```

```
exists (M0). split. trivial.
dependent induction L0.
inversion H2.
inversion H2.
exists (M1). split. trivial.
exists (L0). split. trivial. trivial.
intros.
dependent induction L0.
inversion H.
inversion H.
exists (M0).
split. trivial.
dependent induction L0.
inversion H2.
inversion H2.
exists (M1). split. trivial.
exists (L0). split. trivial. trivial.

Qed.

Lemma aux_2 : forall M N : lambda, forall L : term_list, standard_red_seq (M :: N :: L) →
standard_red_seq (N :: L).
Proof.
intros.
dependent induction H.

(* VAR' case: *)
(* impossible *)

(* ABS' case: *)
pose proof aux_1 as pp.
assert (H1: exists M0 : lambda, M = Abs M0 ∧ (exists N0 : lambda, N = Abs N0 ∧
(exists L2 : term_list, L = Abs_list L2 ∧ L0 = M0 :: N0 :: L2)) ).
```

```
apply pp. trivial.
destruct H1 as [M0].
destruct H0.
destruct H1 as [N0].
destruct H1.
destruct H2 as [L2].
destruct H2.
rewrite → H1.
rewrite → H2.
assert (H4: Abs_list (N0 :: L2) = Abs N0 :: Abs_list L2).
simpl. trivial.
rewrite ← H4.
pose proof ABS' as ABS'.
apply ABS'.
apply IHstandard_red_seq with (M0). trivial.

(* APL' case: *)
dependent induction L1.
simpl in x.
dependent induction L2. simpl in x.
     (* L1 = [] ∧ L2 = [] *)
inversion x. (* impossible*)

     (* L1 = [] ∧ L2 = N' :: L2' *)
simpl in x.
inversion x.
pose proof APL' as pp.
assert (H5: standard_red_seq ((Apl_arg [] M · (App M0 M :: [ ])) · Apl_fun L2 M0)).
apply pp. trivial.
apply IHstandard_red_seq2 with (N0). trivial.
simpl in H5. trivial.

     (* L1 = M :: L1' ∧ L2 = L2*)
```

```
simpl in x.
dependent induction L1.
          (* L1' = []*)
simpl in x.
inversion x.
pose proof APL' as pp.
assert (H5: standard_red_seq ((Apl_arg [] N0 · (App M0 N0 :: [ ])) · Apl_fun L2 M0)).
apply pp. simpl. apply single_list_srs. trivial.
simpl in H5. trivial.

          (* L1' = M0 :: L1 ∧ L2 = L2*)
simpl in x.
inversion x.
pose proof APL' as pp.
assert (H5: standard_red_seq ((Apl_arg (M0 :: L1) N0 · (App M1 N0 :: [ ])) · Apl_fun L2 M1)).
apply pp. apply IHstandard_red_seq1 with (M). trivial. trivial.
simpl in H5. trivial.

(* RDX ' case: *)
trivial.

Qed.

Lemma aux_3 : forall L : term_list, forall M : lambda, Abs_list L = M :: [ ] →
exists N : lambda, M = Abs N ∧ L = N :: [].
Proof.
dependent induction L.
intros.
simpl in H. inversion H.
intros.
simpl in H.
inversion H.
destruct L.
```

```
exists (M).
split. trivial. trivial.
rewrite → H2.
exists (M).
split. trivial.
inversion H2.
Qed.

Lemma aux_4 : forall M N : lambda, name_eval M N → red M N.
Proof.
intros M N H.
induction H.
apply one_step_red.
induction H.
apply beta.
apply app_red_l. trivial.
apply refl_red.
apply trans_red with (N); trivial.
Qed.

Lemma aux_5 : forall L1 L2 : term_list, forall M N P : lambda, L1 · (M :: []) · (N :: · L2)
= P :: [] → False.
Proof.
dependent induction L1.
intros. simpl in H.
inversion H.
intros.
simpl in H. inversion H.

dependent induction L1. simpl in H2. inversion H2. inversion H2.
Qed.

Lemma aux_6 : forall M N : lambda, standard_red_seq (M :: (N :: [])) → standard_red M N.
```

```
Proof.
intros.
dependent induction H.
(* VAR' case: *)
(* impossible *)

(* ABS' case: *)
dependent induction L. simpl in x.  inversion x.
simpl in x.
inversion x.
assert (H3: exists N1, N = Abs N1 ∧ L = N1 :: []).
pose proof aux_3 as pp.
apply pp.
trivial.
destruct H3 as [N1].
destruct H0.
rewrite → H0.
pose proof ABS as pp.
apply pp.
apply IHstandard_red_seq.
rewrite → H3. trivial.

(* APL' case: *)
(* Problema com as çõfunes! *)

(* RDX' case: *)
Focus 2.
apply one_step_name_eval in H.
apply aux_4 in H.
apply standardization. trivial.
```

```
dependent induction L1. simpl in x.
     (* L1 = [] *)
dependent induction L2. simpl in x.
          (* L2 = [] *)
inversion x.
          (* L2 = M :: L2' *)
dependent induction L2. simpl in x.
               (* L2' = [] *)
inversion x.
(**)
pose proof APL as pp.
apply pp.
apply rule_1.
apply IHstandard_red_seq2. trivial.
               (* L2' = M0 :: L2'' *)
simpl in x.
inversion x.
     (* L1 = M :: L1' *)
dependent induction L2.
          (* L2 = [] *)
dependent induction L1.
               (* L1' = [] *)
simpl in x.
inversion x.
pose proof APL as pp.
apply pp.
apply IHstandard_red_seq1. simpl. trivial.
apply rule_1.
               (* L1' = M0 :: L1'' *)
simpl in x.
dependent induction L1.

simpl in x.  inversion x.
```

```
inversion x.
simpl in x.
inversion x.
pose proof aux_5 as pp.
assert (H4: (Apl_arg L1 N0 · (App M1 N0 :: [ ])) · (App M1 M0 :: Apl_fun L2 M1) = N :: [ ] →
False).
apply pp.
assert (H5: False).
apply H4. trivial. contradiction.
Qed.

Lemma aux_7 : forall M : lambda, forall L1 L2 : term_list, Abs_list L1 = M :: L2 →
exists M0 : lambda, M = Abs M0 ∧ exists L3 : term_list, L2 = Abs_list L3.
Proof.
dependent induction L1.
intros.
inversion H.
intros.
inversion H.
exists (M0).
split. trivial.
exists (L1). trivial.
Qed.

Lemma aux_8 : forall L1 L2 : term_list, Abs_list L1 = Abs_list L2 → L1 = L2.
Proof.
dependent induction L1.

(* L1 = [] *)
intros.
dependent induction L2.
     (* L2 = [] *)
trivial.
```

```
    (* L2 = M :: L2' *)
inversion H.

(* L1 = M :: L1' *)
dependent induction L2.
    (* L2 = [] *)
intros.
inversion H.
    (* L2 = M0 :: L2' *)
intros.
inversion H.
assert (H3: L1 = L2).
apply IHL1. trivial.
rewrite ← H3. trivial.

Qed.

Lemma aux_9 : forall L1 L2 : term_list, forall M : lambda, Abs_list L1 =
Abs M :: Abs_list L2 → L1 = M :: L2.
Proof.
dependent induction L1.
intros.
inversion H.
intros.
simpl in H.
inversion H.
assert (H3: L1 = L2).
apply aux_8. trivial.
rewrite ← H3. trivial.
Qed.

Lemma aux_10 : forall M N : lambda, forall L : term_list, standard_red_seq (M :: N :: L) →
standard_red_seq (M :: N :: []).
```

```
Proof.
intros.
dependent induction H.

(* VAR' case: *)
(* impossible *)

(* ABS' case: *)
dependent induction L0. simpl in x. inversion x.
simpl in x.
inversion x.
assert (H3: exists N0 : lambda, N = Abs N0 ∧ (exists L3 : term_list, L = Abs_list L3)).
apply aux_7 with (L0). trivial.
destruct H3 as [N0].
destruct H0.
destruct H3 as [L3].
rewrite → H0.
assert (H4: Abs_list (M :: N0 :: []) = Abs M :: Abs N0 :: [ ]).
simpl. trivial.
rewrite ← H4.
pose proof ABS' as pp.
apply pp.
rewrite ← H1 in x. rewrite → H0 in x.
apply IHstandard_red_seq with (L3).
rewrite → H3 in x.
rewrite → H0 in H2.
rewrite → H3 in H2.
apply aux_9.
simpl. trivial.

(* APL' case: *)
dependent induction L1. dependent induction L2.
    (* L1 = [] ∧ L2 = [] *)
```

```
(* This subcase is impossible. *)
inversion x.
    (* L1 = [] ∧ L2 = M :: L2' *)
simpl in x.
inversion x.
pose proof APL' as pp.
assert (H5: standard_red_seq ((Apl_arg [] N0 · (App M0 N0 :: [ ])) · Apl_fun (M :: []) M0) ).
apply pp. trivial. apply IHstandard_red_seq2 with (L2). trivial.
simpl in H5. trivial.
dependent induction L2.
    (* L1 = M :: L1' ∧ L2 = [] *)
dependent induction L1.
            (* L1' = [] *)
simpl in x.
inversion x.
pose proof APL' as pp.
assert (H5: standard_red_seq ((Apl_arg (M :: []) N0 · (App M0 N0 :: [ ])) · Apl_fun [] M0) ).
apply pp. trivial. trivial. simpl in H5. trivial.
            (* L1' = M :: M0 :: L1'' *)
simpl in x.
inversion x.
pose proof APL' as pp.
assert (H5: standard_red_seq ((Apl_arg (M :: []) N0 · (App M0 N0 :: [ ])) · Apl_fun [] M0)).
apply pp. simpl. apply IHstandard_red_seq1 with (L1 · (M1 :: [])).  simpl. trivial.
trivial. simpl in H5. trivial.
    (* L1 = M :: L1' ∧ L2 = M0 :: L2' *)
dependent induction L1.
        (* L1' = [] *)
simpl in x. inversion x.
pose proof APL' as pp.
assert (H5: standard_red_seq ((Apl_arg (M :: []) N0 · (App M1 N0 :: [ ])) · Apl_fun [] M1) ).
apply pp. trivial. apply single_list_srs.
simpl in H5. trivial.
```

```
          (* L1' = M :: M0 :: L1'' *)
simpl in x.  inversion x.
pose proof APL'  as pp.
assert (H5:  standard_red_seq ((Apl_arg (M ::  [])  N0 · (App M0 N0 ::  [ ]))  · Apl_fun []  M0) ).
apply pp.  apply IHstandard_red_seq1 with (L1 ·  (M2 ::  [])).    simpl.  trivial.
apply single_list_srs. simpl in H5.  trivial.


(* RDX ' case: *)
pose proof RDX'  as pp.
apply pp.  trivial.
apply single_list_srs.

Qed.

(*-------------------------------------------------------------------------*)

(*--------------------- Theorem 2: s.r.s. implies ⇒ n --------------------*)

Lemma standard_red_2 : forall L :  term_list, forall M :  lambda, standard_red_seq (M ::  L)
→ (L=[]  ∨
(exists N :  lambda, exists L'  :  term_list, L = L'  ·  (N ::  [])     ∧ standard_red M N)).
Proof.
intros.
dependent induction L.
(* L = [] *)
auto.
(* L = M :: L' *)
assert (H1:  L = [ ]  ∨ (exists (N :  lambda) (L'  :  term_list),
L = L'  ·  (N ::  [ ])  ∧ standard_red M N)).
apply IHL.
apply aux_2 with (M0).  trivial.
destruct H1.
```

```
  (* L = [] *)
right.
exists (M). exists ([]).  simpl.
rewrite ← H0.
split. trivial.
rewrite → H0 in H.
apply aux_6. trivial.
  (* L = L' · (N :: [ ]) ∧ standard_red M N) *)
destruct H0 as [N].
destruct H0 as [L'].
destruct H0.
right.
exists (N). exists (M ::  L' ). simpl.
rewrite → H0.
split. trivial.
apply rule_9 with (M).
apply aux_6.
apply aux_10 with (L). trivial.
trivial.

Qed.

(*---------------------------------------------------------------------------*)

(*----------- Auxiliar Lemmas to prove the equivalence ⇒ n and s.r.s. ----------*)

Lemma aux_11 : forall M : lambda, forall L : term_list, [ ] = L · (M ::  [ ])  → False.
Proof.
simple induction L.
simpl.
intro.
inversion H.
intros.
```

```
inversion H0.
Qed.


Lemma aux_12 : forall M M' : lambda, forall L L' : term_list, L · (M :: []) =
L' · (M' :: []) → L = L' ∧ M = M'.
Proof.
intros.
dependent induction L.
dependent induction L'.
(* L = [] ∧ L' = [] *)
simpl in H. inversion H.
split.
trivial. trivial.
(* L = [] ∧ L' != [] *)
simpl in H.
inversion H.
pose proof aux_11 as pp.
assert (H3: [ ] = L' · (M' :: [ ]) → False).
apply pp.
contradiction.
dependent induction L'.
(* L != [] ∧ L' = [] *)
simpl in H.
inversion H.
pose proof aux_11 as pp.
assert (H3: [ ] = L · (M :: [ ]) → False).
apply pp.
assert (H4: False).
apply H3.
rewrite → H2. trivial.
contradiction.

```

```
(* L != [] ∧ L' != [] *)
inversion H.
assert (H3: L = L' ∧ M = M'). apply IHL. trivial.
destruct H3. rewrite → H0.
split. trivial. trivial.

Qed.


(*----------------- Corollary: ⇒ n equivalent to s.r.s.------------------*)

Lemma s_r_s_equiv: forall M N : lambda, standard_red M N ↔
(M = N ∨ exists L : term_list, standard_red_seq (M :: L · (N :: []) )).
Proof.
intros.
split.
intro.
apply standard_red_1. trivial.
intro.
destruct H.
rewrite ← H. apply rule_1.
destruct H as [L].
pose proof standard_red_2 as pp.
assert (H1: L · (N :: [ ]) = [ ] ∨ (exists (N' : lambda) (L' : term_list), L · (N :: [ ]) =
L' · (N' :: [ ]) ∧ standard_red M N')).
apply pp. trivial.
destruct H1.
dependent induction L.
simpl in H0. inversion H0.
simpl in H0.
inversion H0.
destruct H0 as [N'].
destruct H0 as [L'].
```

```
destruct H0.
pose proof aux_12 as aux_12.
assert (H2: L = L' ∧ N = N').
apply aux_12. trivial.
destruct H2.
rewrite ⟶ H3. trivial.
Qed.
```

Bibliography

[1] T. Altenkirch. “A formalization of the strong normalization proof for System F in LEGO.” In: *TLCA 1993: Typed Lambda Calculi and Applications*. Ed. by M. Bezem and J.F.Groote. Vol. 664. Lecture Notes in Computer Science. Springer, 1993, pp. 13–28.

[2] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, and B. C. Pierce. “Mechanized Metatheory for the Masses: The POPLMARK Challenge.” In: *Lecture Notes in Computer Science* 3603 (2005), pp. 50–65.

[3] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. “Engineering Formal Metatheory.” In: *ACM SIGPLAN Notices* 43 (2008), pp. 3–15.

[4] H. Barendgregt, W. Dekkers, and R. Statman. *Lambda Calculi with types*. Cambridge University Press, 2013.

[5] H. P. Barendregt. *The Lambda Calculus Its syntax and Semantics*. NORTH-HOLLAND, 1981.

[6] H. Barendregt. “The impact of the lambda calculus in logic and computer science.” In: *Bulletin of Symbolic Logic* 3 (1997), pp. 181–215.

[7] H. Barendregt and E. Barendsen. “Introduction to Lambda Calculus.” In: (1994). url: `https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf`.

[8] S. Berghofer and C. Urban. “A Head-to-Head Comparison of de Bruijn Indices and Names.” In: *Electronic Notes in Theoretical Computer Science* 174(5) (2007), pp. 53–67.

[9] N. 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.

[10] C. Clack, C. Myers, and E. Poon. *Programming with Miranda*. Prentice Hall, 1995.

[11] E. Copello, N. Szasz, and Á. Tasistro. "Formal metatheory of the Lambda calculus using Stoughton's substitution." In: *Theoretical Computer Science* 685 (2017), pp. 65–82.

[12] M. Copes. *A machine-checked proof of the Standardization Theorem in Lambda Calculus using multiple substitution*. MSc Thesis, Universidad ORT Uruguay, 2018.

[13] T. Coquand. *An algorithm for testing conversion in Type Theory*. Cambridge University Press, 1991.

[14] B. Goldberg. "Functional Programming Languages." In: *ACM Computing Surveys* 28 (1996), pp. 249–251.

[15] G. Gonthier. "Formal Proof—The Four- Color Theorem." In: *Notices of the AMS* 5 (2008).

[16] F. Guidi. "Standardization and Confluence in Pure Lambda-Calculus Formalized for the Matita Theorem Prover." In: *Journal of Formalized Reasoning* 5(1) (2012), pp. 1–25.

[17] P. G. Harrison and A. J. Field. *Functional programming*. Addison-Wesley, 1988.

[18] G. Huet. "Residual Theory in lambda-Calculus: A Formal Development." In: *Journal of Functional Programming* 4(3) (1994), pp. 371–394.

[19] F. Joachimski and R. Matthes. "Standardization and Confluence for a Lambda Calculus with Generalized Applications." In: *RTA 2000: Rewriting Techniques and Applications*. Ed. by L. Bachmair. Vol. 1833. Lecture Notes in Computer Science. Springer, 2000, pp. 141–155.

[20] R. Kashima. "A Proof of the Standardization Theorem in λ-Calculus." In: *RIMS Kokyuoroku* 1217 (2001), pp. 37–44.

[21] A. Laretto. *Formalizations of the Church-Rosser Theorem in Agda*. BSc Thesis, Università degli Studi di Torino, 2020.

[22] R. P. N. Lazarom. "Strong normalization in a typed lambda calculus with lambda structures types." In: *Studies in Logic and the Foundations of Mathematics* 133 (1973), pp. 389–468.

[23] R. Loader. *Notes on simply typed lambda calculus*. Technical Report ECS-LFCS-98-381, LFCS, Univ. of Edinburgh, 1998.

[24] J. Maraist, M. Odersky, D. N.Turner, and P. Wadler. "Call-by-name, call-by-value, call-by-need and the linear lambda calculus." In: *Electronic Notes in Theoretical Computer Science* (1995), pp. 370–392.

[25] J. McKinna and R. Pollack. “Pure Type Systems Formalized.” In: *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA ’93, Utrecht, The Netherlands, March 16-18, 1993*. Ed. by M. Bezem and J. Groote. Springer Berlin, 1993, pp. 289–305.

[26] J. McKinna and R. Pollack. “Some Lambda Calculus and Type Theory Formalized.” In: *Journal of Automated Reasoning* 23 (1999), pp. 373–409.

[27] T. Nipkow. “More Church–Rosser Proofs.” In: *Journal of Automated Reasoning* 26 (2001), pp. 51–66.

[28] M. Pereira and S. M. de Sousa. *Introdução à Programação Funcional em OCaml*. Universidade da Beira Interior, 2012.

[29] F. Pfenning and C. Elliot. “Higher-order abstract syntax.” In: *ACM SIGPLAN Notices* 23(5) (1988), pp. 199–208.

[30] G. D. Plotkin. “Call-by-name, call-by-value and the λ-calculus.” In: *Theoretical Computer Science* 1(2) (1975), pp. 125–159.

[31] E. Post. “Formal Reductions of the General Combinatorial Decision Problem.” In: *American Journal of Mathematics* 65(2) (1943), pp. 197–215.

[32] R. Rojas. “A Tutorial Introduction to the Lambda Calculus.” In: *arXiv:1503.09060* (2015).

[33] J. E. Santo, L. Pinto, and T. Uustalu. “Modal Embeddings and Calling Paradigms.” In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by H. Geuvers. Vol. 131. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 18:1–18:20.

[34] J. E. Santo, L. Pinto, and T. Uustalu. “Plotkin’s call-by-value λ-calculus as a modal calculus.” In: *Journal of Logical and Algebraic Methods in Programming* 127 (2022), p. 100775.

[35] P. Sestoft. “Demonstrating Lambda Calculus Reduction.” In: *The Essence of Computation*. Ed. by T. Mogensen, D. A. Schmidt, and I. H. Sudborough. Vol. 2566. Lecture Notes in Computer Science. Springer, 2002, pp. 420–435.

[36] N. Shankar. “A mechanical proof of the Church-Rosser theorem.” In: *Journal of the ACM* 35(3) (1988), pp. 475–522.

[37] M. H. B. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. ELSEVIER, 2006.

[38] A. Stoughton. "Substitution Revisited." In: *Theoretical Computer Science* 59(3) (1988), pp. 317–325.

[39] T. C. D. Team. "The Coq Proof Assistant Reference Manual." In: (2022). url: `https://github.com/coq/coq/releases/tag/V8.16.0`.

[40] R. Viehof. *call-by-name, call-by-value and abstract machines*. Radboud University Nijmegen, 2012.

[41] F. Wiedijk. "Formal proof – getting started." In: *Notices of the American Mathematical Society* 55(11) (2008).