PEDECIBA Informática
Uruguay

On the Formalisation of the Metatheory of the Lambda Calculus and Languages with Binders

Ernesto Copello
copello@ort.edu.uy

Tesis presentada en cumplimiento parcial de los requisitos para el grado de **Doctor en Informática**
Pedeciba Informática

Montevideo, Uruguay, Agosto de 2017

Directores de Tesis: Dra. Nora Szasz
Dr. Álvaro Tasistro
Universidad ORT Uruguay

On the Formalisation of
the Metatheory of the Lambda Calculus
and Languages with Binders

Ernesto Copello

ISSN 0797–6410
Tesis de Doctorado en Informática
Reporte Técnico RT 17-XX
PEDECIBA
Montevideo, Uruguay, Agosto de 2017

Resumen

Este trabajo trata sobre el razonamiento formal verificado por computadora involucrando lenguajes con operadores de ligadura.

Comenzamos presentando el Cálculo Lambda, para el cual utilizamos la sintaxis histórica, esto es, sintaxis de primer orden con sólo un tipo de nombres para las variables ligadas y libres. Primeramente trabajamos con términos concretos, utilizando la operación de sustitución múltiple definida por Stoughton [61] como la operación fundamental sobre la cual se definen las conversiones alfa y beta. Utilizando esta sintaxis desarrollamos los principales resultados metateóricos del cálculo: los lemas de sustitución, el teorema de Church-Rosser y el teorema de preservación de tipo (Subject Reduction) para el sistema de asignación de tipos simples. En una segunda formalización reproducimos los mismos resultados, esta vez basando la conversion alfa sobre una operación más sencilla, que es la de permutación de nombres. Utilizando este mecanismo, derivamos principios de inducción y recursión que permiten trabajar identificando términos alfa equivalentes, de modo tal de reproducir la llamada *convención de variables de Barendregt* [4]. De este modo, podemos imitar las demostraciones al estilo "lápiz y papel" dentro del riguroso entorno formal de un asistente de demostración.

Como una generalización de este último enfoque, concluimos utilizando técnicas de programación genérica para definir una base para razonar sobre estructuras genéricas con operadores de ligadura. Definimos un universo de tipos de datos regulares con información de variables y operadores de ligadura, y sobre éstos definimos operadores genéricos de formación, eliminación e inducción. También introducimos una relación de alfa equivalencia basada en la operación de permutación y derivamos un principio de iteración/inducción que captura la convención de variables anteriormente mencionada. A modo de ejemplo, mostramos cómo definir el Cálculo Lambda y el sistema F en nuestro universo, ilustrando no sólo la reutilización de las pruebas genéricas, sino también cuán sencillo es el desarrollo de nuevas pruebas en estos casos.

Todas las formalizaciones de esta tesis fueron realizadas en Teoría Constructiva de Tipos y verificadas utilizando el asistente de pruebas Agda [46].

Palabras Clave: Lenguajes de Programación, Metateoría Formal, Cálculo Lambda, Programación Genérica, Teoría de Tipos

Abstract

This work is about formal, machine-checked reasoning on languages with name binders.

We start by considering the λ-calculus using the historical (first order) syntax with only one sort of names for both bound and free variables. We first work on the concrete terms taking Stoughton's multiple substitution operation [61] as the fundamental operation upon which the α- and β-conversion are defined. Using this syntax we reach well-known meta-theoretical results, namely the Substitution lemmas, the Church-Rosser theorem and the Subject Reduction theorem for the system of assignment of simple types. In a second formalisation we reproduce the same results, this time using an approach in which α-conversion is defined using the simpler operation of name permutation. Using this we derive induction and recursion principles that allow us to work by identifying terms up to α-conversion and to reproduce the so-called *Barendregt's variable convention* [4]. Thus, we are able to mimic "pencil and paper" proofs inside the rigorous formal setting of a proof assistant.

As a generalisation of the latter, we conclude by using generic programming techniques to define a framework for reasoning over generic structures with binders. We define a universe of regular datatypes with variables and binders information, and over these we define generic formation, elimination, and induction operations. We also introduce an α-equivalence relation based on the swapping operation, and are able to derive an α-iteration/induction principle that captures Barendregt's variable convention. As an example, we show how to define the λ-calculus and System F in our universe, and thereby we are able to illustrate not only the reuse of the generic proofs but also how simple the development of new proofs becomes in these instances.

All formalisations in this thesis have been made in Constructive Type Theory and completely checked using the Agda proof assistant [46].

Keywords: Programming Languages, Formal Metatheory, Lambda Calculus, Generic Programming, Type Theory

Acknowledgments

First of all, I wish to thank my supervisors Nora Szasz and Álvaro Tasistro for their invaluable contribution which made possible this work, and for the patience shown while reviewing the many drafts of this thesis, when I had lost mine.

I want to thank Daniel Fridlender and Miguel Pagano for the key discussions and many contributions throughout the work done in the formalisation of the Church-Rosser theorem in the article presented in chapter 3.

I would also like to thank Ana Bove and Maribel Fernandez for correcting many of my mistakes, and their feedback on the Agda code and drafts of the paper presented in chapter 4.

This work was supported by a PhD fellowship from Agencia Nacional de Investigación e Innovación (ANII), and a research assistant position at Universidad ORT: thanks for both institutions. I also want to thank the Programa de Desarrollo de las Ciencias Básicas (PEDECIBA) within which this PhD was held.

Finally, I want to extend my gratitude to my family.

Contents

CHAPTER 1

Introduction

1.1 Context

Writing a precise semantic definition of a full-scale programming language is a daunting task. However, considering the potential benefits for both language users and implementers, it is surprising how few of such efforts can be found reported in the literature. Computer programs are increasingly being required to withstand rigorous analysis of their properties, and the specification of programming languages constitutes the basis upon which the understanding of particular programs and their properties may rest. In particular, the properties we would like our programs to warrant intrinsically depend on the proper working of compilers. Such proper working can only be granted with certainty by employing a precise definition of the compiled language. Therefore, we need to carefully address the way we specify and reason about programming languages.

After 27 years of its publication, Standard ML [42] remains one of the few examples of a programming language that is precisely defined while still being designed for large scale industrial projects. Most of its specification can be found in several articles, typically written in a combination of natural language and specific mathematical notation. Although papers of such kind are, of course, written by experts, they often do contain errors. Even when considering small programming languages there exist examples of formalisation errors. For example, in the original presentation of Session Types [31], the typing relation is not preserved under α-conversion. Only after 10 years was this flaw discovered and fixed by Yoshida and Vasconcelos in [69]. This issue in particular was the main motivation of my Master's thesis [12, 63], which consisted on the partial certification of a type checker for this language.

To prevent issues as the former from arising, while still allowing languages to scale up, some degree of automated checking is required. Proof assistants like Coq [39], Isabelle [44] and Agda [47] make such checking possible, providing the user with a logical framework where soundness can be verified, and in some cases also with a programming language. The programming language embedded in the proof assistant enables the user to reuse the formalised semantics to certify the language's compiler. Examples of this can be found in [9] for the simply-typed lambda calculus, and of course in the impressive case of *CompCert* [35], a certified C compiler intended to be used in life-critical and mission-critical software written in C, meeting high levels of assurance by using the Coq proof assistant for both programming the compiler, and proving its correctness.

Proof assistants help with the burden of proving the soundness of language formalisations, but they introduce encoding problems. In particular, the problem of representing and reasoning about structures with binders and α-equivalent terms is central to formailising the meta-theory of some relevant programming languages. The formalisation of this problem in a proof assistant is a complex task. This becomes clear just by checking out how many techniques and choices in the formulation of the basic definitions have been developed in the past decades, while at the same time none of them has been widely adopted. This complexity arises even in the simplest language with the aforementioned characteristics, i.e. the λ-calculus.

Indeed, in order to compare the so many different approaches to the formalisation of the meta-theory of programming languages and as a way of measuring the adoption of proof assistants by the research community, the POPLmark challenge [3] proposes a set of well-defined problems based on a call-by-value variant of System F [26], enriched with records, record subtyping, and record patterns. Central aspects in this challenge are variable bindings and complex inductions over the abstract syntax with binders. They also propose that the successful solutions should be evaluated against the following criteria:

- *Reasonable overhead:* the overhead cost in the formalisation should not be prohibitive, and intrinsic to the formalised problem, not to the chosen technique.
- *Transparency:* the formalised proofs should be human-readable, and not too radically different from classic pen-and-paper proofs. In this way a reasonable entry cost is paid by a reader not familiar with the proof assistant to understand the proof details.
- *Cost of entry:* the infrastructure should be usable by non theorem prover experts.

In addition to the previous features, the challenge structure implicitly adds an *escalation* factor, as its problems have several sub-parts, each one adding more features to the target language, by checking how the proposed solution can be re-used to solve them.

In this line of work, we study techniques to reason over abstract syntax trees with binders. In the first chapters of this thesis we mainly use λ-calculus as a minimal primigenial language over which to experiment and compare formalisations. In spite of the simplicity of the language, subtle issues arise in its formalisation, and it is precisely simplicity that enables us to study them in isolation. In a later chapter we extend the obtained results to general languages with binders.

1.2 The Problem

The λ-calculus was introduced by Church [10] to formalise the concept of *function* as rule of computation. Given a denumerable set of names (variables) V, the set of the λ-terms is defined inductively as follows:

Definition 1 (λ-calculus terms).

$M, N ::= \; x \mid MN \mid \lambda x.M$

That is: there are variables, application, and abstraction of a variable name over a term. An occurrence of a variable x in a term M is called *bound* if it is in the scope of an abstraction λx in M, *binding* (or *binder*) if it is the x in a λx, and *free* otherwise. We denote $\mathrm{fv}(M)$ the set of free variables of a λ-term M. Although later many other formulations appeared, in this work we stick to the historical one, with *one* sort of names to serve both as free and bound variables. It is precisely one contention of this thesis that it is feasible and reasonable to develop the metatheory of the λ-calculus using this syntax.

Such formulation brings about two different issues:

- On the one hand side, bound names denote parameters for which other terms are substituted in computations, i. e. $(\lambda x.M)\ N$ computes to $M[x := N]$ (β-contraction rule). The right-hand side term corresponds to the substitution of the term N for the free occurrences of the abstracted variable x in the abstraction body M. Because bound names in M may coincide with free names in N, we have what we shall call the *substitution* issue, i.e. that substitution of terms for real variables has to be performed in some sophisticated way so as to avoid capture of names by binders.
- On the other hand, because of the existence of bound names, there is what we shall call the *α-conversion* issue, i.e. that terms differing only in the choice of their bound names should be functionally indistinguishable.

The Substitution Issue

Although the aforementioned syntax is the one used from the very beginning of the λ-calculus in the '30s [10], it was not until 1958 that a correct and detailed definition of substitution was given by Curry and Feys [16]:

Definition 2 (Curry and Feys's Substitution Operation)**.**

$$
\begin{aligned}
y[x := N] &= \begin{cases} N & \text{if } x = y \\ y & \text{otherwise} \end{cases} \\
(MP)[x := N] &= M[x := N]P[x := N] \\
(\lambda y.M)[x := N] &= \begin{cases} \lambda y.M & \text{if } x \notin \mathrm{fv}(\lambda y.M) \\ \lambda y.(M[x := N]) & \text{if } x \in \mathrm{fv}(\lambda y.M) \wedge \\ & \quad y \notin \mathrm{fv}(N) \\ \lambda z.(M[y := z][x := N]) & \text{otherwise, with the first } z \\ & \text{s.t.} z \notin \{x\} \cup \mathrm{fv}(M) \cup \mathrm{fv}(N) \end{cases}
\end{aligned}
$$

The final clause performs a binder renaming to prevent any free occurrences of y in N from becoming bound by the considered abstraction, in other words, to prevent any variable capture.

This definition is not structurally recursive; rather, the recursion is on the *length* of the term wherein the substitution is performed. But, to ascertain that the length of the recursive calls always decreases, a proof has to be given, which must therefore be simultaneous to the justification of the well-foundedness of the definition. Also consequently, proofs of properties of the substitution have generally to be conducted on the length of terms. This may explain why this definition has not been extensively used in formalisations.

The α-Conversion Issue

We can observe in the β-contraction rule $(\lambda x.M)\ N\ \rhd_\beta\ M[x := N]$ that any particular variable name selection –x in this case– is irrelevant to the final substitution result. That is, bound names merely identify places for which terms are to be substituted. For example, we can note that the terms $\lambda x.x$ and $\lambda y.y$ represent the same (identity) function, as applying a β-contraction to any λ-term, returns it inaltered in both cases.

Indeed, the syntax of the λ-calculus, presented at the beginning of this section has distinct terms representing the same function objects, even at the syntactic level. This problem is addressed by the definition of an α-equivalence relation which relates terms representing the same function.

Curry and Feys in [16] properly define the α-conversion relation as the reflexive, transitive and compatible with the syntactic constructors closure of the following α-contraction relation:

Definition 3 (Curry and Feys' α-contraction)**.**

$$\frac{y \notin \mathrm{fv}(M)}{\lambda y.M[x := y]\ \rhd_\alpha \lambda x.M}$$

The free variable premise grants that y is fresh enough not to bind any preexisting free variable in the term M.

Which Comes First?

Curry and Feys' constitute what we may call the *historical approach* to treating the two aforementioned issues of the metatheory of the λ-calculus. It namely consists in defining the α-conversion relation in terms of substitution. Reasoning with substitution becomes of course ubiquitous in the development of the meta-theory. Using Curry and Feys' definition forces to employ induction on the length of terms instead of the more natural induction on the syntax, and it also happens that a mismatch arises between natural inductive definitions of fundamental relations and the non-structurally inductive structure of the substitution operation. These inconveniences led Stoughton [61] in 1988 to propose a simultaneous multiple variable substitution operation defined by simple structural recursion.

In chapter 3 we carry out a formalisation of Stoughton's substitution definition, which allows us to develop the fundamental results of the metatheory of the λ-calculus, namely the Church-Rosser theorem and the Subject Reduction theorem for the system of assignment of simple types in a completely formal way within reasonable cost boundaries. This is the first contribution of this thesis.

On the other hand, a whole family of alternatives is associated to the idea that, contrary to the historical standpoint, the issue of α-conversion is actually more fundamental than that of the definition of substitution. Or, more concretely, that the issue of how to reason and compute modulo α-conversion should be solved first so as to yield, among other things, a sufficiently abstract definition of substitution.

A prominent illustration of this practice is given in Barendregt's book [4]. For instance, the following are excerpts from this work:

"2.1.12 CONVENTION. Terms that are α-congruent are identified [on a syntactic level]"

"2.1.13 VARIABLE CONVENTION. If $M_1, \dots, M_n$ occur in a certain mathematical context (e.g., definition, proof), then in these terms all bound variables are chosen to be different from the free variables."

"2.1.14. MORAL. Using conventions 2.1.12 and 2.1.13 one can work with λ-terms in the naive way."

Definition 4 (Barendregt's Substitution)**.**

$$\begin{array}{llcll}
x & [x := N] & \equiv & N & \\
y & [x := N] & \equiv & y & \textit{if } x \not\equiv y \\
(M_1 M_2) & [x := N] & \equiv & (M_1[x := N])(M_2[x := N]) & \\
(\lambda y.M_1) & [x := N] & \equiv & \lambda y.(M_1[x := N]) &
\end{array}$$

"In the fourth clause it is not needed to say "provided that $y \not\equiv x$ and $y \notin fv(N)$". By the variable convention this is the case."

This convention has received a standard name, namely the *Barendregt Variable Convention*, which we will denote by BVC. It conveniently establishes that all bound variables can be chosen to be different from the free variables in certain mathematical context (e.g., definition, proof), and assumes α-convertible terms are syntactically equal. When working under this convention, Barendregt considers λ-terms as representatives of their equivalence classes, interpreting substitution as an operation on the α-equivalence classes of terms. Substitution is performed on representatives, provided that the bound variables are named properly, as the previous variable convention dictates. If we call *real calculus* the one that identifies α-convertible terms, and *raw calculus* the one working on concrete terms, then we can say that Barendregt works at the real calculus level, but using convenient raw representatives.

Now, in Barendregt's book two ways are proposed to formally justify the BVC. One is to work with de Bruijn indices, which eliminates bound names and therefore α-conversion altogether, as we will see in the next chapter. As we stated before, we are interested in pursuing a nominal approach to syntax, so this alternative is not satisfactory to us. The second justification given by Barendregt is based on Curry and Feys' definition of substitution, and their proof that substitution is compatible with α-conversion. It follows that substitution can be interpreted as an operation on α-classes of terms, i.e., on terms of the real calculus. This path is not amenable to full formalisation due to the intricate formulation of the substitution definition, as already shown.

However, if α-conversion is to be granted a more fundamental status than that of substitution, it should not be defined in terms of the latter —and indeed it need not be, since we are only replacing a name by another name, and not by a whole term. One possibility is to use *name permutations*, which in turn are defined as finite sequences of *name swappings.* These were introduced in [24, 53] following ideas of Fraenkel and Mostowski's set theory, and allow for an implementation of α-conversion with convenient properties.

The second contribution of this thesis is a justification of the BVC in these terms. Specifically, we are able to provide principles of induction and recursion for the real calculus implementing the BVC in Constructive Type Theory and deriving them from just simple structural induction on raw terms. Equality remains the simple definitional one, and we do not perform any kind of quotient on terms. This result is presented in chapter 4. The principles have to be extended in order to cope with induction on relations which allows us to show, in chapter 5, how it is possible within this approach to reach the fundamental theorems of Church-Rosser and Subject Reduction.

1.3 Structure of this Thesis

In the next chapter we survey the state of the art in the mechanisation of the meta-theory of formal languages with binders, especially the λ-calculus. We classify the main approaches developed to represent and reason about these languages, i.e., nominal, nameless and higher-order abstract syntax, and study the most relevant works within each of them. As we are interested in the many proof subtleties, we pay special attention to works that are mechanised using proof assistants.

In chapter 3 we present the formalisation of the λ-calculus in Constructive Type Theory based on Stoughton's multiple substitution [61]. We prove that this substitution is α-compatible, the Church-Rosser theorem, and the Subject Reduction theorem for the system of assignment of simple types. The whole development was formally checked using the Agda proof assistant [47]. This chapter consists in the article "Formal Metatheory of the Lambda Calculus Using Stoughton's Substitution", published in Theoretical Computer Science [14].

In chapters 4 and 5 we present the formalisation of the λ-calculus in Constructive Type Theory based on the variable swapping operation. With this formalisation we are able to derive induction and recursion principles at the real calculus level. These principles aim to reproduce the use of the BVC in pen-and-paper works, while trying to hide all its complexity in the rigorous context of the Agda proof assistant. Using the derived principles we were able to reproduce the same fundamental meta-theoretical results for the λ-calculus as we did in the preceding chapter. Chapter 4 consists in the article "Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory", published in Electronic Notes in Theoretical Computer Science [13]. Part of Chapter 5 has been accepted for publication in the 12th Workshop on Logical and Semantic Frameworks with Applications (LSFA 2017).

In chapter 6, we present the third contribution of the thesis: We extend the previous chapters results, applying generic programming techniques to address the formalisation of generic structures with binders. We define a universe of regular data types with variable binding information, and over these we define generic formation, elimination, and induction operators. We also introduce an α-equivalence relation based on the swapping operation, and are able to derive α-iteration/induction principles that capture the BVC. As an example, we show how to define the λ-calculus and System F in our universe, exhibiting not only the reuse of the generic proofs but also how simple the development of new proofs becomes in these instances.

Finally, in chapter 7 we present our overall conclusions, final remarks and further work.

CHAPTER 2

State of the Art

In this chapter we present the state of the art in the formalisation of the meta-theory of languages with binders, especially within the context of proof assistants. We first classify the main approaches developed to represent and reason about such languages and then we study the most relevant works written in each of these lines:

Nominal Syntax

Within this approach variables are named with identifiers. An α-conversion relation is defined to introduce the equivalence of expressions via renaming of their bound variables. We can identify the following variations:

- **Historical:** All the variables belong to the same domain of variables.
- **â la Frege:** Free and bound variables belong to different domains.
- **Nominal Techniques:** The swapping operation is used to define the α-conversion

Nameless Syntax

Within this approach abstractions are nameless. A bound variable occurrence is represented by the number of binders that are in the scope between its occurrence and its corresponding binder. Because of this, no α-conversion notion is needed in this formulation, and α-equivalent terms are syntactically identical. Free variables are represented as indices denoting binders out of the scope. A variation of this syntax combines nominal syntax for free variables and indices for bound variables.

Higher-Order Abstract Syntax

In this approach, meta-language abstractions are used to represent abstractions in the introduced language structure. In this way, the substitution operation is handled at the meta-level.

In the following sections we will introduce the key aspects of each one of these approaches.

2.1 Nominal Syntax

In this section we study first order formalisations of the λ-calculus using named variables as done in most pen-and-paper classical works.

Several solutions were proposed to overcome the problems enumerated in the previous chapter for the nominal syntax. In particular, addressing substitution, Stoughton [61] presented in 1988 a simultaneous multiple variable substitution operation. It is defined by simple structural recursion on raw terms, following the standard syntax, and an interesting theory arises concerning α-conversion as we will explain in the next chapter. Moreover, further simplification is achieved by not distinguishing so many cases when considering the substitution in the abstraction case, that is, by merging the second and third sub-cases of the abstraction case in Curry's definition (definition 2), performing a uniform renaming of the bound variables, even though this renaming is not necessary when no variable capture occurs. Possibly the most interesting consequence of this method of uniform renaming is that the identity substitution normalizes terms with respect to α-conversion. This approach has been formalised by Lee [33] in the Coq proof assistant. In his work, Lee represents the multiple substitutions as total functions from variables to terms (the same is done in Stoughton's work), and defines the propositional identity of substitutions as their extensional equivalence. In the next chapter we present a formalisation of Stoughton's syntax in Constructive Type Theory which has been fully machine-checked using the system Agda [46]. In this formalisation we use the propositional identity to reflect the definitional, and thus decidable, equality. We also represent the multiple substitutions as functions, but avoid using such fully extensional equivalence.

McKinna and Pollack [40] syntactically distinguish free from bound variables, calling them parameters and variables respectively. This approach was suggested by several authors: Frege [22] in 1879, Prawitz [56] and Gentzen [25] in 1965, and Coquand [15] in 1991. McKinna and Pollack developed a substantial work in the Lego proof assistant, including the Church-Rosser and standardization theorems for β-reduction, and the basic theory of Pure Type Systems, leading to the strengthening theorem and type checking algorithms for this system. This modified syntax comes with some overhead: two substitution operations must be defined, one for parameters and another one for variables. These definitions do not prevent variable capture, so some preconditions are required to use them in a safe way. In order to restrict the sort of bound variables to occur in bound positions only, they introduce a well-formedness predicate over terms prescribing when this is the case. In fact, it is this predicate, which is defined recursively over the first order abstract syntax terms, that gives rise to the proof principles this work employs. In the spirit of Stoughton's multiple substitution, they introduce a multiple renaming operation over parameters. They notice that bijective renamings are useful but surprisingly difficult to construct, so they finally introduce the swapping operation over parameters. Proofs with statements about change of names are performed by induction on the length of the terms, as expected.

In 2001, Peter Homeier [30] tried to recreate the BVC in the Isabelle/HOL proof assistant, using a multiple substitution to prove the Church-Rooser theorem. His work defines α-conversion without using substitution, in an inductive definition with only three syntax directed rules. Homeier uses the Isabelle's HOL package to obtain the type of α-equivalence classes. This module introduces a mapping function between raw terms and α-equivalence classes, and the property that if two raw terms are α-convertible then the corresponding α-equivalence classes are structurally equal. We suspect that to be able to prove the previous property, this package hides some kind of de Bruijn indices representation, but this translation is far from being transparent. Moreover, for every function defined at the raw terms level, a quite complex adequacy proof must be given in order to extend the function to the type of α-equivalence classes, proving that the function applied to α-convertible raw terms returns identical or α-convertible expressions.

Ford and Mason [21] in 2001 evaluated the PVS proof assistant proving the Church-Rooser theorem in a call by value λ-calculus formulation. Their substitution uses the renaming operation on bound variables to prevent capture in the abstraction case, and as renaming preserves terms size, this definition is easily shown to be well-founded. This work presents an interesting technique in the inductive definition of the α-conversion relation $\overset{\alpha}{\equiv}$, which is usually called *cofinite quantification* in the literature. We show the abstraction case in figure 2.1. The premise states that there exists a finite set of names T such that for all names x not belonging to this set or to the free variables of the abstraction's bodies, it should hold that both bodies are α-convertible after renaming their bound variables with the name x.

$$\frac{\exists T \in Fin(V),\ \forall x \notin T \cup \mathrm{fv}(e_0) \cup \mathrm{fv}(e_1),\ e_0[x_0 := x] \overset{\alpha}{\equiv} e_0[x_0 := x]}{\lambda x_0.e_0 \overset{\alpha}{\equiv} \lambda x_1.e_1}$$

Figure 2.1: Ford and Mason's abstraction case of the α-conversion definition.

When proving an α-convertible property, they are able to conveniently select a finite set of variables T, and prove the premise for the abstraction bodies without considering those variables in the proof context. So this allows them to reason in the same way as when using the BVC, wisely selecting the name of the bound variables out of some finite context. Furthermore, it is frequently the case in proofs that one chooses some freshness context, and later on, in a wider context, we have to review our choice to accommodate to finitely many more freshness constraints. Thus, it is convenient to introduce a notation that tells us we require some property to hold over some infinite range of variables, without mentioning it explicitly. However, proving the simplest properties of this $\overset{\alpha}{\equiv}$ relation is quite challenging in Ford and Mason's work, requiring several mutual inductions over the length of the terms, because the substitution is defined as presented by Curry and Feys in [16].

In chapter 4 we use a similar introduction rule for the abstraction case of the α-equivalence relation. However, our definition is inspired on the one presented by Krivine in [32], but using swapping insted of substitution. Krivine also defines a multiple substitution operation in his work, similar to Stoughton's definition, but with no renaming clause as he works over the real calculus.

In 2002 Pitts and Gabbay introduced the *Nominal Logic* [24], a first-order many-sorted logic with equality, containing primitives for renaming via name-swapping, for freshness of names, and for name-binding. The swapping operation has much nicer logical properties than the more general, non-bijective forms of renaming. This operation provides a sufficient foundation for a theory of structural induction/recursion for the syntax modulo α-equivalence classes for *equivariant* assertions/functions. The equivariant property says that the truth/return value of an assertion/function is invariant under name-swapping of the names. Since 2005, several works using this technique in the Isabelle/HOL proof assistant have been developed by Urban et. al [65, 66], achieving a representation for α-equivalent lambda terms that is based on names, is inductive, and comes with a structural induction principle on equivariant properties, where the inductive lambda-case needs to be proved only for terms with fresh binders. In the next section we study this formulation in deep, and in section 2.3 we survey mechanisations of Nominal Logic for the λ-calculus using the Higher order abstract syntax (HOAS) technique.

In 2003 Vestergaard and Brotherston [68] present a syntax with two related layers: the raw terms and the equivalence classes of these terms under the α relation. Substitution, α-conversion and β-reduction are defined at the raw level; their substitution does not prevent from variable capture, so preconditions should hold to safely apply it. They present conditions under which results about β confluence at the quotient level can be derived from the same results at the raw level. Then they present a confluence proof conducted exclusively by the primitive proof principles of the raw syntax, and the considered recursive relations, and it follows that this result can be extended to the quotiented level. However, the last proof involves arbitrary inter-mixtures of α- and β-reductions, bringing great complexity to the final proof of confluence. We believe that the main contributions of this work are:

- The study of the conditions under which confluence at the raw and the real calculus are equivalent.
- Although using a standard induction principle, the BVC takes a central role in the proof of the diamond property of the β-reduction proof at the raw level. They use a variation of the BVC, named the BCF, and its associated properties.

Nominal Techniques

Nominal logic provides a mathematical model for names in mathematics. It is based on Fraenkel and Mostowski's set theory (FM) introduced in the first half of the 20th century. They introduced *atoms* to model names, which have specific operations and properties that traverse all the components of the theory. One of the basic operations is the permutation of atoms, that is, a bijective mapping between atoms. Based on permutation, this theory reflects the fact that atoms have no internal structure, and that “one atom will do as well as any other atom”. Therefore, every property should be preserved under a permutation of atoms, which is called *equivariance* property in nominal literature.

Another key concept in this theory is the *finite support*: Given a set X, a set S of atoms *supports an element* $x \in X$, if x is invariant under every permutation of atoms which is the identity on S. The finite support axiom in the FM set theory claims that for any element in the theory there exists a finite set S supporting it. This definition aims to formalise that there exists a finite set of atoms S occurring in any element x, such any permutation that does not change these finite atoms, when applied to x will have no effect on it.

Gabbay and Pitts [23, 24] use this general theory to deal with bound names and α-equivalence classes in any abstract syntax. They provide inductive types up to α-equivalence, in other words, datatypes which admit structural induction and recursion principles while at the same time constitute an α-equivalence quotient of the syntax. For this, they consider constructions and properties that are invariant with respect to the permutation of names, that is, equivariant constructions. They are thereby able to derive principles of recursion/induction over the α-equivalence classes. This equivariance enables them to permute names, usually to avoid some form of accidental name-capture, while preserving the inductive hypotheses. Indeed, name permutation has much nicer logical properties than more general, non-bijective forms of renaming. Even though name permutation is less general than renaming, it supports the construction of a theory of names based upon it. The class of equivariant predicates has excellent logical properties, as it contains the equality, and is closed under negation, conjunction, disjunction, existential and universal quantifications. The same is not true for renaming, for example the inequality is not preserved by renaming.

The swapping of the occurrences of an atom a with another b in an object x is usually denoted as $(a\ b)x$. For example, in next figure we show the abstraction introduction rule for the α-equivalence relation based on the swapping operation:

$$\frac{(a\ b)x \sim_\alpha (a'\ b)y}{\lambda a.x \sim_\alpha \lambda a'.y}\ b \neq a \text{ and } b \text{ does not occur in } x, y$$

At the base of their theory Gabbay and Pitts adopt the notion of finite supported mathematical objects, which gives a well-behaved way, in terms of name-permutations, of expressing the fact that atoms are fresh for mathematical objects if they do not belong to some finite set supporting them. This notion allows them to extend the concept of *fresh names* from finite objects (as abstract syntax trees) to infinite ones (as infinite sets and functions). In fact, they need that the functions used in the instantiation of their iteration principle are finitely supported.

In [60] an ML extension to declare and manipulate algebraic data types with variable binding constructs is presented. These constructions with binders can be deconstructed in a safe way: In the case of an abstraction inspection, a variable swapping with a freshly generated binder is computed in the abstraction body, giving the user a fresh binder and the correspondly renamed body of the opened abstraction. This mechanism grants that values with binders are operationally equivalent if and only if they represent α-equivalent objects. This result is proved giving a denotational semantics into the universe of FM-sets, and then proving that this denotational semantics matches the operational one. In this way they are able to prove that values of the introduced data types with binders represent α-equivalence classes of object-level syntax.

In the next section we will discuss several mechanised works in the Isabelle/HOL proof assistant based on these techniques, and consequently using a HOAS representation of the λ-terms abstract syntax.

As previously mentioned, this theory allows for a nice formalisation of the main concepts concerning bindings, such as α-equivalence and freshness, in terms of the swapping operation on names. We further develop these ideas in chapters 4 and 5. However, our development diverges from the nominal theory in the choice function $\chi : 2^{\mathrm{V}} \to \mathrm{V}$, which selects a fresh variable not belonging to some finite set of variables. We represent variables by an infinite enumeration, and the returned fresh variable is determined by choosing the first variable in the enumeration not in the given set. This choice function is not equivariant neither finitely supported. However, nominal techniques are compatible with the existence of non finitely supported elements. Indeed, we successfully apply some of these techniques to achieve induction/recursion principles over the real calculus, and in chapter 6 we generalise the previous results to any abstract syntax with binders. To implement this iterators over α-equivalence classes we apply a similar technique to the one developed in [60], described above. Moreover, we are also able to prove that our iteration principles are operationally equivalent for α-equivalent objects, what we will call the *strong α-compatibility* property. As we embed proofs of properties within our iteration principle, our framework enables the user not only to program α-compatible functions, but also gives facilities to prove properties about these functions using α-induction principles, that is, at the quotient level.

2.2 Nameless Syntax

This syntax was introduced by de Bruijn [17]. It introduces some overhead involving "shifted" terms and contexts, which require extra statements in theorems, making it difficult to keep track of the correspondence between informal and formal versions of the same result. Besides, the object language medium-size terms of this representation are not human readable, so we believe this approach fails in passing the transparency property. One advantage of this syntax is that the overhead due to reasoning over α-equivalence classes is eliminated, since α-equivalent terms are syntactically equal. It may be argued that the balance is positive, as many formalisations chose this approach, but in some sense the overhead introduced is by the nature of the codification, and not by the problem itself. De Bruijn's representation is an encoding of the informal notion of binding and does not address the relationship between free and bound variable names: a free variable actually occurs, but its identity is lost when it becomes bound. We next introduce interesting works that use this codification, while being able to hide in great manner the previous considerations.

In [27], Gordon uses the Cambridge HOL proof assistant to define an induction principle equivalent to the one developed in chapter 4, and shown figure 2.2.

$$\begin{array}{l}
(\forall a)\ P(a) \\
(\forall M\ N)\ P(M) \wedge P(N) \Rightarrow P(MN) \\
(\forall M\ a)\ a \notin X \wedge P(M) \Rightarrow P(\lambda a.M) \\
\hline
(\forall M)\ P(M)
\end{array}$$

Figure 2.2: α-induction principle

This principle, as the BVC, enables to choose the abstraction variable not in some given finite set of variables X, for the abstraction case of a proof by induction over terms. That is, we are able to choose a fresh enough representative from an arbitrary α-equivalence class of terms. Gordon uses the locally nameless variation of de Bruijn's syntax to represent λ-terms. This syntax was already suggested by de Bruijn [17], in which "free variables have names but the bound variables are nameless". The main property of this syntax is that α-convertible terms are syntactically equal. However, invalid terms appear in this representation, and a well-formedness predicate is needed to exclude ill-formed terms from the formalisation. Because of this last issue, well-formedness hypothesis should be added to the premises of all proofs. On the other hand, the main advantage of this mixed strategy is that theorems can be expressed in the conventional form, without the de Bruijn encoding, and in spite of this, the renaming of bound variables for fresh ones is still supported in proofs, because syntactical equality is up to α-conversion. However, when a renaming has to be done to pick another witness of an α-equivalence class, the classical primitive inductive hypothesis does not have any information about the new renamed sub-term, becoming necessary in general to perform an induction over the length of terms. In this way, they are able to apply the inductive hypothesis to the renamed sub-term, because its length is strictly decreasing. To overcame this overhead, Gordon introduces the previous induction principle for decidable predicates, which, as expected, is proved by induction on the length of de Bruijn's terms.

As an example of this methodology, some substitution lemmas from Hindley and Seldin's book [29] are directly derived using this induction principle, without recourse to theorems about the underlying locally nameless representation, neither exposing the internal renaming done to select fresh variables. However, new operations and relations are not easily introduced, as they must be defined at the locally nameless level, and also proved to be closed under a well-formedness predicate. Besides, an axiomatisation of the behavior and properties of new operations must be introduced at the raw level, to be able to use them without exposing any internal notation.

2.3 Higher Order Abstract Syntax

Higher Order Abstract Syntax (HOAS) was introduced by Church [11], and mainly developed by Martin-Löf [45], Miller [41] and Pflenning [50, 58]. It uses the functions of the meta-logic to encode bindings in an object language. In this way, the capture-free substitution operation is handled at the meta-logic level, allowing its reuse in object languages. However, λ-terms with free variables are more difficult to represent. HOAS presents two variations for possible implementations:

- HOAS: the abstract syntax abstractions are represented by meta-level functions with type (λ-term $\rightarrow$ λ-term).
- weak HOAS: the abstractions are represented by meta-level functions with type (names $\rightarrow$ λ-term).

In its first form, the HOAS binding model is quite different form the informal one. However, it introduces several simplifications: substitution is free and variable freshness is irrelevant. But it also introduces several complications: because meta-level abstractions are a component of the object language abstract syntax, structural induction is lost. Besides, as HOAS uses the full meta-level function space, it causes the introduction of ill-formed terms in the model. While some people find HOAS convenient, its significant departure from standard mathematical practice makes it quite difficult to handle by readers not familiar with the HOAS technique.

On the other hand, in weak HOAS it is usually necessary to introduce an α-compatibility relation, and it is not possible to re-use the meta-level substitution operation. However, this representation has no exotic terms, and an induction principle can be automatically derived.

There exist several works, developed during approximately a decade, on the formalisation of induction/recursion principles over α-equivalence classes of λ-terms, all of them developed in the Isabelle/HOL proof assistant and the HOAS encoding. In the following paragraphs we present some of these works.

Gordon and Melham [28], based on Gordon's work [27], present five axioms characterising λ-terms in a nominal syntax, but syntactically identifying α-equivalent terms. They formalise this work in the Cambridge HOL proof assistant. One of their axioms allows the introduction of new functions over λ-terms at the conventional named syntax level, and without any overhead. They do so by introducing an iteration principle over lambda terms. To define a function from terms to some type β using this principle, for the variable and application case, as usual, one must provide functions of types $vnames \rightarrow \beta$ and $\beta \times \beta \rightarrow \beta$ correspondingly. For the abstraction case $\lambda a.M$, a function $f : (vnames \rightarrow \beta) \rightarrow \beta$ should be provided. By doing so, we can obtain the recursive result $(f\ b)$ for some renamed sub-term $M[a{:=}b]$, for any variable b. So this iteration principle does not return the recursive result for some arbitrary fixed abstraction name, and thus it does not leak any information about the original binder a. By doing so, it cannot distinguish between α-convertible terms. Hence, it can be proved that this iteration principle defines α-compatible functions. This work begins to explore a HOAS approach, as they introduce a constructor $Abs : (vnames \rightarrow terms) \rightarrow terms$, that is, any meta-level function from variables to terms represents a lambda abstraction in the embedded language via the *Abs* function. The crux of their development is the proof of the existence of a model for the *Abs* function, which is based on an infinitary intersection over the variable names space. So they prove the existence of the model, but they do not give explicitly a computable one. We do not know in deep the Isabelle/HOL proof assistant capabilities, and the paper does not give much more information about the encoding of the function *Abs* in their formalisation. Thus, is not easy to evaluate nor to deduce how feasible is to transfer the *Abs* function to a Constructive Type Theory setting. As a direct consequence, their iteration principle can be used to define functions at a logical level, and allows to prove properties about them, but it is not possible to compute them. From the proposed iteration principle they derive an induction principle that, taking $\beta = Bool$, in a similar way to the iteration one, gives the induction hypothesis for any renaming of the bound variable in the abstraction case. Then, using this induction principle they derive Gordons's induction principle for decidable predicates.

Continuing Gordon and Melham's work, Norrish [48] introduced a method to define functions in a much more familiar way, close to the classical principles of primitive recursion. For this, he uses some ideas of Gabbay-Pitts nominal techniques approach, introducing the name swapping operation as a basics for the abstract syntax with binders. He presents a set of increasingly complex functions in order to measure the expressive power of his recursion principle. His iteration principle passes the test, but has some side conditions to be proven about the functions used to instantiate it. For instance, he must prove that each function used to define the cases in the iteration is finited supported –that is, it does not create too many fresh variables–, and that it behaves in a linear way through swapping –that is, it is equivariant.

Urban and Tasson [66] use Gabbay-Pitts' theory more deeply to construct an induction principle similar to the introduced by Gordon, as shown in figure 2.3. They use the concept of finite support of nominal sets, instead of the free-variables function over terms to state the freshness conditions. They base this work on a weak HOAS [18] syntax, providing a nominal layer above it. They prove the substitution composition lemma as an example of the use of their induction principle. Our work in chapters 4 and 5 is in the line of this one, but we keep using a name-carrying syntax and not a variation of a HOAS as they do.

Given some predicate P over λ-terms and a set A supporting the swapping operation, then:

$$\frac{\begin{array}{l}(\forall a)\ P(a,A)\\ (\forall M,N)\ P(M,A)\wedge P(N,A)\Rightarrow P(MN,A)\\ (\forall a,M)\ a \text{ fresh in } A\wedge P(M,A)\Rightarrow P(\lambda a.M,A)\end{array}}{(\forall M)\ P(M,A)}$$

Figure 2.3: α-induction principle

In [65], Urban and Norrish study under which conditions the BVC can be safely reproduced on inductions over relations on λ-terms. For this, they introduce a *variable convention compatibility* condition for each introduction rule in a relation definition, in order to support proofs using the BVC: Firstly, all functions and side conditions should be equivariant. Secondly, the side conditions should imply that all bound variables do not occur free in the conclusions. Finally, all bound variables should be distinct. If the previous conditions hold, they are able to derive a strengthened relation induction principle, where all binders can be chosen distinct from some given context. They identify the lemma of substitution preserving parallel reduction and also the weakening lemma for the simply typed λ-calculus as examples of problematic uses of the BVC. For the typing relation they are able to derive the new induction principle. However, for the parallel reduction relation they need to modify the classical definition in order to satisfy their variable convention compatibility. In chapter 5 we also address these issues, and with suitable α-induction principles on terms we do not need to derive ad-hoc induction principles for the involved relations.

2.4 Conclusions

As seen before, the different approaches to address abstract syntax issues can be mainly divided into: first-order and higher-order. In the first-order approaches variables are encoded using names or numbers, whereas higher-order approaches use the meta-level function space to encode the bindings in the object language level. Indeed, the higher-order approach is appealing because capture-avoidance and α-equivalence can be handled at the meta-level. However, we consider that this is exactly the reason to avoid it in the present work, as we want to address these specific issues in detail, and not to push them to other level.

If we do not consider de Brujin's nameless representation for binders, we can say that the first-order approach is the most similar to standard informal presentations. We exclude de Brujin's representation from our work because its statements require the introduction of operations that are exotic to the addressed problem, and have to deal with encoding issues. In the next chapters we will present our work based on two standard first-order approaches. First, in the more classical setting, we will work using Stoughton's multiple substitution operation, and as in classical formalisations, all the meta-theory will be based on this substitution. Later, we will use nominal techniques, where the name swapping operation takes a central role in the developed theory.

CHAPTER 3

Stoughton's Multiple Substitution

This chapter consists in the article *"Formal Metatheory of the Lambda Calculus Using Stoughton's Substitution"*, published in Theoretical Computer Science [14]. We present a formalisation of the λ-calculus in Constructive Type Theory using a nominal approach with one sort of names for both free and bound variables, and without identifying α-convertible terms. We use a multiple substitution operation based on Stougthon's [61]. We prove that this substitution is α-compatible, the Church-Rosser theorem, and the Subject Reduction theorem for the simply typed λ-calculus à la Curry. The whole development has been formally checked using the Agda proof assistant [46]. The conception of this development was done in tight collaboration with the co-authors, and the coding was completely developed by myself.

3.1 Introduction

The Lambda calculus was originally formulated with *one* sort of names to serve both as real (free) and apparent (bound) variables [10]. Such design brought about two different issues:

- On the one hand side, because of the existence of bound names, there is what we shall call the α*-conversion* issue, i.e. that terms differing only in the choice of their bound names should be functionally indistinguishable.
- And, on the other, because bound names in one term may coincide with free names in another, we have what we shall call the *substitution* issue, i.e. that substitution of terms for real variables has to be performed in some sophisticated way so as to avoid capture of names by binders.

Historically too, substitution was granted the more fundamental place within the couple above, since it was used in the definition of α-conversion. Substitution itself was just left undefined by Church [10] in the original formulation of the calculus but later its complexity became a prime motivation for Curry and Feys [16], which provided the first definition, somewhat as follows:

$$x[y := P] = \begin{cases} P & \mathit{if}\ x = y \\ x & \mathit{if}\ x \neq y \end{cases}$$

$$(MN)[y := P] = M[y := P]\ N[y := P]$$

$$(\lambda x.M)[y := P] = \begin{cases} \lambda x.M\ \textit{if } y \textit{ not free in } \lambda x.M \\ \lambda x.M[y := P]\ \textit{if } y \textit{ free in } \lambda x.M \textit{ and } x \textit{ not free in } P \\ \lambda z.(M[x := z])[y := P]\ \textit{if } y \textit{ free in } \lambda x.M \textit{ and } x \textit{ free in } P, \\ \quad \textit{where } z \textit{ is the first variable not free in } M, P. \end{cases}$$

The complexity lies in the last case, i.e. the one requiring to rename the bound variable of the abstraction wherein the substitution is performed. The recursion proceeds, evidently, on the *size* of the term; but, to ascertain that $M[x := z]$ is of a size smaller than that of $\lambda x.M$, a proof has to be given and, since the renaming is effected by the very same operation of substitution that is being defined, such a proof must be simultaneous to the justification of the well-foundedness of the whole definition. The procedure is indeed intricate and, incidentally, hardly ever mentioned as such in the several texts introducing the Lambda calculus along this line.

Also as a consequence of the definition above, there is the inconvenience that proofs of properties of the substitution operation have often to be conducted on the size of terms and have generally three subcases corresponding to abstractions, with possibly two invocations to the induction hypothesis in the subcase considered above. For instance consider the following proposition, which can be taken as stating that substitution is free from name capture:

$x \in FV(M) \Rightarrow FV(M[x := P]) = (FV(M) \setminus \{x\}) \cup FV(P),$

where $FV(M)$ stands for the set of free variables of M. In the case of abstractions we have to consider $FV((\lambda x.M)[y := P])$ which in the complex subcase becomes $FV(\lambda z.M[x := z][y := P])$. In order to compute $FV(M[x := z][y := P])$ from the outside we need first a (size) induction hypothesis on $M[x := z]$, and then, in a second step, the induction hypothesis has to be applied further to M.

Since the fundamental relations of α-conversion and β-computation and conversion are defined in terms of substitution, reasoning with this operation becomes ubiquitous in the metatheory of the Lambda calculus. Within the present approach, it also happens that all too often a mismatch arises between natural inductive definitions of those fundamental relations and the inductive structure of the substitution operation, which forces to employ induction on terms instead of induction on the relations in question —with the inconveniences just pointed out. For instance, we naturally have a rule

$$\frac{M \twoheadrightarrow N}{\lambda x.M \twoheadrightarrow \lambda x.N}$$

as part of the definition of β-computation. But, in proving for instance that this relation is compatible with substitution, i.e. that

$$M \twoheadrightarrow N \Rightarrow M[y := P] \twoheadrightarrow N[y := P]$$

the induction hypothesis on M and N cannot be used in the complicated case of abstraction in which the thesis is

$$\lambda z.(M[x := z])[y := P] \twoheadrightarrow \lambda z'.(N[x := z'])[y := P]$$

Some alternative approach to treating substitution is therefore necessary, especially if one is, like we are, interested in actually carrying out the completely formal metatheory to a substantial extent, e.g. by employing some of the several proof assistants available.

A whole family of alternatives is associated to the idea that, contrary to the historical standpoint, the issue of α-conversion is actually more fundamental than that of the definition of non-capturing substitution. Or, more concretely, that the issue of how to reason and compute *modulo α-conversion* should be solved first so as to yield, among other things, a sufficiently abstract definition of substitution.

One alternative along this view is to apply the principles of what has become known as *nominal abstract syntax* [23, 24, 52–54]: α-conversion is defined in terms of a basic operation of *name permutation* which acts uniformly on free and bound names. Then principles of induction and recursion are formulated that allow, under natural conditions, to work on abstract α-classes of terms, thus formalising informal practice as carried out in most textbooks. With this method it is possible, for instance, to formally define substitution in the simple way given place to by the application of the so-called *Barendregt variable convention*. The approach has been implemented on machine in [1, 13, 65, 66].

The former is *nominal* syntax as opposed to the "nameless" terms of de Bruijn [17] or its more up-to-date version, *locally nameless syntax* [2, 7]. These works take the radical approach to eliminate bound names and therefore α-conversion altogether. The result is a more machine-like presentation of the calculus for which a certain overhead necessary to ensure soundness (particularly in handling substitution) cannot be entirely avoided.

Finally, probably the first proposal of formalisation of reasoning modulo α-conversion is [28]. This work gives an axiomatisation of the syntax of the Lambda calculus in which equality embodies α-conversion and for whose resulting (abstract) terms a method of definition by recursion is provided. It ultimately rests upon the use of higher-order abstract syntax within the system HOL.

In this paper we are, however, interested in further pursuing the historical approach. This means, to begin with, to agree to the priority of substitution over α-conversion and to search for tractable definitions thereof.

A first way out within this perspective consists in employing for the local or bound names a type of symbol different from the one of the real variables: that was, to begin with, Frege's choice in the first fully fledged formal language [22], which featured universal quantification as a binder, and was later made again by at least Gentzen [25], Prawitz [56] and Coquand [15]. Within the field of machine-checked meta-theory, McKinna and Pollack [40] used the approach to develop substantial work in the proof assistant Lego, concerning both the pure Lambda calculus and Pure Type Systems. Now, the method is not without some overhead: there must be one substitution operation for each kind of name and a well-formedness predicate to ensure that bound names do not occur unbound –so that induction on terms becomes in fact induction on this predicate.

If, still, one persists in sticking to the original syntax, there first appears the idea, employed in [57], of introducing an operation of *renaming* consisting in the replacement of a bound name by another. Renaming is simpler than substitution, as it proceeds just naïvely, and can be used to implement α-conversion provided the new name is chosen so that it does not at all occur in the term in question. It is also sufficient, under the same proviso, to implement the complex case of substitution. This latter use of renaming provides a way out of the complexity of the justification of the definition of substitution as exposed above. But, on the other hand, it cannot avoid the need of induction on the size of terms, both for justifying the recursion employed in substitution and in the subsequent proofs involving this operation.

We maintain that the genuinely historical approach to the metatheory of the lambda calculus is the one initiated by Curry and Feys [16] and later pursued at least partly in [29]. And that, when trying to fully formalize this theory and give it an implementation on machine, there appears a thesis worth testing, namely that it was Stoughton [61] who provided the right formulation of substitution. The prime insight is simple: In the difficult case where renaming of a bound variable is necessary, structural recursion is recovered if one lets substitutions become *multiple* (*simultaneous*) instead of keeping them just unary. Moreover, further simplification is achieved if one does not bother in distinguishing so many cases when considering the substitution in an abstraction and just performs uniformly the renaming of the bound variable: indeed, given that equivalence under renaming of bound variables is natural and necessary, it makes no point to try to preserve as much as possible the identity of the concrete terms, as in the Curry-Feys definition. As pointed out by Stoughton, the idea of using multiple substitution comes from [20], whereas the one of uniform renaming is originally presented in [57].

In this paper we present proofs of fundamental results of the metatheory of the Lambda calculus employing Stoughton's definition of substitution. Specifically, concerning β-reduction we prove the Church-Rosser theorem and the Subject Reduction theorem for the system of assignment of simple types. The definitions and proofs have been fully formalised in Constructive Type Theory [37] and machine-checked employing the system Agda [46]. The corresponding code is available at `https://github.com/ernius/formalmetatheory-stoughton`. In the subsequent text we give the proofs in English with a considerable level of detail so that they serve for clarifying their formalisation. We hope thereby to show that what we have called the historical approach to the metatheory of the calculus is indeed formally and machine-tractable, thanks to the adequate reformulation of the substitution operation. Indeed, the proofs of both principal theorems follow standard strategies, and are formalised in a gentle manner.

The structure of the paper is as follows: in section 2 we begin by introducing the syntax of the Lambda calculus and the definition of substitution, together with some basic propositions. We also present a little theory concerning the composition of substitutions which is not indispensable for establishing our main results but allows for a more elegant presentation thereof and bears some interest in itself. Section 3 is about α-conversion, which is given an inductive definition directed by the structure of terms. We then establish two results, namely that the so defined α-conversion is a congruence and that it is compatible with substitution (the substitution lemma). Thereby the two first sections constitute themselves into a reformulation of Stoughton's original paper [61] in which we give alternative definitions and proofs obtaining what we believe is a simpler structure of the whole. Section 4 introduces the notion of β-reduction and proves the Church-Rosser theorem by using the standard method due to Tait and Martin-Löf which involves the formulation and study of the parallel β-reduction. In section 5 we formulate the system of assignment of simple types to terms, and then also prove that it is compatible with substitution. From this result, the closure of typing under α-conversion and the subject reduction property of β-reduction are also proven. The overall conclusions are exposed in section 6.

This paper is an extended and revised version of [64], which only included the treatment of α-conversion up to the Substitution Lemma. Additional material includes Section 4 on β-reduction and the Church-Rosser theorem, Section 5 on the simple typed system and the Subject Reduction theorem, and the part of Section 2 on composition of substitutions. Also, as already mentioned, we use natural mathematical syntax instead of Agda's in order to improve readability.

3.2 Substitution

Syntax We start with a denumerable type V of *names*, also to be called *variables* — i.e. for concreteness we can put $\mathsf{V} =_{def} \mathbb{N}$ or $\mathsf{V} =_{def} \mathsf{String}$. Letters x, y, z with primes or subindices shall stand for variables. The type Λ of *terms* is defined inductively as usual —here below we show it as a grammar for abstract syntax:

$M, N ::= x \mid MN \mid \lambda x.M.$

In concrete syntax we assume the usual convention according to which application binds tighter than abstraction.

Freedom and freshness We now define inductively what it is for a variable x to occur *free* in a term M, which we write $x * M$:

$$\frac{}{x * x} \qquad \frac{x * M}{x * MN} \qquad \frac{x * N}{x * MN} \qquad \frac{x * M}{x * \lambda y.M}\, x \neq y$$

The negation of this relation is also given an inductive definition: we write it $x \mathrel{\#} M$ and read it *x fresh in M*, borrowing notation and terminology from the theory of nominal abstract syntax:

$$\frac{}{x \mathrel{\#} y}\, x \neq y \qquad \frac{x \mathrel{\#} M \qquad x \mathrel{\#} N}{x \mathrel{\#} MN} \qquad \frac{}{x \mathrel{\#} \lambda x.M} \qquad \frac{x \mathrel{\#} M}{x \mathrel{\#} \lambda y.M}$$

An important relation between terms is that of *sameness of free variables*:

$M \sim_* N =_{def} (\forall x{\in}\mathsf{V})\ (x * M \Leftrightarrow x * N).$

Substitutions We shall work with multiple (simultaneous) substitutions associating terms to variables. Therefore a type Σ of substitutions is very naturally introduced as:

$\Sigma =_{def} \mathsf{V} \to \Lambda.$

Now this definition could be deemed much too wide, because the only substitutions arising in computing or comparing terms are identity almost everywhere and, for instance, the latter admit a decidable extensional equality whereas our general functions do not. The point is, however, sorted out by the following observations.

Let us first use ι for the identity substitution, i.e. the function mapping each variable to itself as term, and introduce the following operation of *update*. If σ is a substitution, x a variable and M a term, then $\sigma, x{:=}M$ is another substitution, defined by:

$(\sigma, x{:=}M)\, x =_{def} M$

$(\sigma, x{:=}M)\, y =_{def} \sigma\, y$ if $x \neq y$.

Then, starting up from ι, the operation of update generates every concrete substitution to be ever encountered. We could have axiomatised a type of *(finite) tables* for the substitutions, but actually at no point at which we reason generally over substitutions do we need to constrain them into those generated by the operations above. Instead it turns out that our main results concern the operation of substitutions on the free variables of given terms, i.e. their restrictions to such variables. Therefore it is convenient to introduce a type of *restrictions*:

$\mathsf{P} =_{def} \Sigma \times \Lambda$.

The restriction of substitution σ to term M is to be written $\sigma \downharpoonleft M$. Now several useful relations are defined on restrictions. The first one is *extensional equality*:

$\sigma \downharpoonleft M \cong \sigma' \downharpoonleft M' =_{def} M \sim_* M' \wedge (x * M \Rightarrow \sigma\, x = \sigma'\, x)$.

A noticeable particular case of this is when M and M' are one and the same term. In that case we write $\sigma \cong \sigma' \downharpoonleft M$ and get the following characterisation directed by the structure of M. We use all variables universally quantified, unless otherwise stated.

Proposition 1.

1. $\sigma \cong \sigma' \downharpoonleft x \;\Leftrightarrow\; \sigma\, x = \sigma'\, x$.
2. $\sigma \cong \sigma' \downharpoonleft MN \;\Leftrightarrow\; \sigma \cong \sigma' \downharpoonleft M \wedge \sigma \cong \sigma' \downharpoonleft N$.
3. $\sigma \cong \sigma' \downharpoonleft \lambda x.M \;\Leftrightarrow\; (\sigma, x{:=}N) \cong (\sigma', x{:=}N) \downharpoonleft M$, *for some* N.

The proof can be carried out by straightforward logical calculations, using that $y * MN \Leftrightarrow y * M \vee y * N$ as well as $y * \lambda x.M \Leftrightarrow y * M \wedge y \neq x$. These, in turn, are immediate from the inductive definition of $\_ * \_$. $\square$

We next extend freedom and freshness of variables to restrictions:

$x * (\sigma \downharpoonleft M) =_{def} (\exists y * M)\; x * \sigma\, y$.

$x \,\#\, (\sigma \downharpoonleft M) =_{def} (\forall y * M)\; x \,\#\, \sigma\, y$.

We can also characterise these relations following the structure of the term:

Proposition 2.

1. $y * (\sigma \downharpoonleft x) \;\Leftrightarrow\; y * \sigma\, x$.
2. $y * (\sigma \downharpoonleft MN) \;\Leftrightarrow\; y * (\sigma \downharpoonleft M) \vee y * (\sigma \downharpoonleft N)$.
3. $y * (\sigma \downharpoonleft \lambda x.M) \;\Leftrightarrow\; y * (\sigma, x{:=}z \downharpoonleft M)$ *with* $z \neq y$.
4. $y \,\#\, (\sigma \downharpoonleft x) \;\Leftrightarrow\; y \,\#\, \sigma\, x$.
5. $y \,\#\, (\sigma \downharpoonleft MN) \;\Leftrightarrow\; y \,\#\, (\sigma \downharpoonleft M) \wedge y \,\#\, (\sigma \downharpoonleft N)$.
6. $y \,\#\, (\sigma \downharpoonleft \lambda x.M) \;\Leftrightarrow\; y \,\#\, (\sigma, x{:=}z \downharpoonleft M)$ *with* $z \neq y$.

The proof is similar to the one of preceding proposition. $\square$

We likewise extend sameness of free variables to restrictions:

$(\sigma \downharpoonleft M) \sim_* (\sigma' \downharpoonleft M') =_{def} x * (\sigma \downharpoonleft M) \Leftrightarrow x * (\sigma' \downharpoonleft M')$.

And then the following is proven by simple calculation from the definitions:

Proposition 3. $\sigma \downharpoonleft M \cong \sigma' \downharpoonleft M' \;\Rightarrow\; (\sigma \downharpoonleft M) \sim_* (\sigma' \downharpoonleft M')$. $\square$

We shall also use the abbreviated notation $\sigma \sim_* \sigma' \downharpoonleft M$ for $(\sigma \downharpoonleft M) \sim_* (\sigma' \downharpoonleft M)$.

Action of substitutions on terms The effect of multiple substitutions on terms can be defined by simple structural recursion on the latter. The crucial insight in this respect is the observation that if one lets substitutions become multiple (i.e. simultaneous) then the renamings of bound variables that will eventually be necessary can be recorded *together* with the originally acting substitution so that the resulting (enlarged) substitution just passes on to act on the (unmodified) body of the abstraction. Indeed, no collision can arise, since the original substitution is destined for the free variables of the term, and therefore not for the bound name to be modified.

A complementary insight is that when a binder is crossed the collection of free variables to be affected is increased by one, and therefore the originally acting substitution should be appropriately instructed on what to do with that name. It is just pointless to enter a distinction of cases destined to avoid the modification of this name whenever possible, as in the Curry-Feys definition. The reason is that renaming should be non-harmful: terms differing only in the choice of bound names should have identical behavior and we are forced to take care of this principle as soon as some renaming is ever allowed. This observation leads to treating substitutions on abstractions uniformly: we search for an appropriate name to replace the bound variable, record the renaming into the current substitution and go ahead into the body. The new name should not capture any of the names introduced into its scope by effect of the substitution. It therefore suffices that it be fresh in the restriction of the original substitution to the abstraction on which it is acting. Also, the action of the substitution on the term must be a function of these two, and therefore the choice of the new name should be determined by the restriction in question. We thus arrive at the definition below. The action of substitution σ on term M is written $M\sigma$. In concrete syntax it will bind tighter than application:

$$\begin{array}{lll} x\sigma & =_{def} & \sigma x \\ (MN)\sigma & =_{def} & M\sigma\ N\sigma \\ (\lambda x.M)\sigma & =_{def} & \lambda y.M(\sigma, x{:=}y) \text{ where } y = \chi\,(\sigma \downharpoonright \lambda x.M). \end{array}$$

The function χ acts on restrictions and performs the choice of the new bound name as explained above. Its result should actually depend only on the collection of free names in the restriction in question, so we can specify it by the following requirements, to be called the *choice axioms*:

1. $\chi\,\rho \# \rho$.
2. $\rho \sim_* \rho' \;\Rightarrow\; \chi\,\rho = \chi\,\rho'$.

A choice function can be readily implemented by just returning e.g. the *first* variable not free in the given restriction —thus resembling the Curry-Feys definition. Our Agda code for the present development provides such implementation together with its correctness proof. Here we omit further details.

Now there are two first basic results concerning the action of substitutions on terms. We begin by showing that extensional equality of restrictions to a term is equivalent to yielding equal results when acting on that term:

Lemma 1. *$\sigma \cong \sigma' \downharpoonright M \;\Leftrightarrow\; M\sigma = M\sigma'$.*

The proof is by structural induction on M. We spell it in detail.

For the case of a variable x, we have:

$\sigma \cong \sigma' \downharpoonright x$
$\Leftrightarrow$ (Proposition 1)

$\sigma\, x = \sigma'\, x$
$\Leftrightarrow$ (Action of substitutions)

$x\sigma = x\sigma'$.

For applications MN we observe:

$\sigma \cong \sigma' \downharpoonleft MN$
$\Leftrightarrow$ (Proposition 1)

$\sigma \cong \sigma' \downharpoonleft M \wedge \sigma \cong \sigma' \downharpoonleft N$
$\Leftrightarrow$ (Induction)

$M\sigma = M\sigma' \wedge N\sigma = N\sigma'$
$\Leftrightarrow$ (Application formation)

$M\sigma\ N\sigma = M\sigma'\ N\sigma'$
$\Leftrightarrow$ (Action of substitutions)

$(MN)\sigma = (MN)\sigma'$.

Finally for abstractions $\lambda x.M$ we first observe $\chi\,(\sigma \downharpoonleft \lambda x.M) = \chi\,(\sigma' \downharpoonleft \lambda x.M)$, in the following way:

$\sigma \cong \sigma' \downharpoonleft \lambda x.M$
$\Rightarrow$ (Proposition 3)

$\sigma \sim_* \sigma' \downharpoonleft \lambda x.M$
$\Rightarrow$ (Choice axiom 2)

$\chi\,(\sigma \downharpoonleft \lambda x.M) = \chi\,(\sigma' \downharpoonleft \lambda x.M)$.

Let now $y = \chi\,(\sigma \downharpoonleft \lambda x.M) = \chi\,(\sigma' \downharpoonleft \lambda x.M)$. We then have:

$\sigma \cong \sigma' \downharpoonleft \lambda x.M$
$\Leftrightarrow$ (Proposition 1)

$(\sigma, x{:=}y) \cong (\sigma', x{:=}y) \downharpoonleft M$
$\Leftrightarrow$ (Induction)

$M(\sigma, x{:=}y) = M(\sigma', x{:=}y)$
$\Leftrightarrow$ (Abstraction formation)

$\lambda y.M(\sigma, x{:=}y) = \lambda y.M(\sigma', x{:=}y)$
$\Leftrightarrow$ (Action of substitutions)

$(\lambda x.M)\sigma = (\lambda x.M)\sigma'$.

$\square$

Secondly, we prove the following lemma establishing that no capture of free names occurs by effect of the substitution as defined:

Lemma 2 (No Capture). $y * M\sigma \ \Leftrightarrow\ y * (\sigma \downharpoonleft M)$.

The proof is by structural induction on M. The cases of a variable and of applications are straightforward using Proposition 2. We show the case of abstractions:

$y * (\lambda x.M)\sigma$
$\Leftrightarrow$ (Action of substitutions)

$y * \lambda z.M(\sigma, x{:=}z)$
$\Leftrightarrow$ (Definition of $*$)

$y * M(\sigma, x{:=}z) \wedge z \neq y$
$\Leftrightarrow$ (Induction)

$y * (\sigma, x{:=}z \downharpoonleft M) \wedge z \neq y$
$\Leftrightarrow$ (Proposition 2)

$y * (\sigma \downharpoonleft \lambda x.M)$. □

The conciseness of both statement and proof of each of these two preceding lemmas is quite remarkable, especially in comparison with the corresponding versions using unary substitution.

On the composition of substitutions We now proceed to considering a little theory concerning the sequential composition of substitutions. The theory consists of a series of definitions and results given all along the original article by Stoughton which we here factor out expecting to clarify how they too can be formalised. Some of the ultimate results are helpful in the rest of the development.

The composition of substitutions is of course conceivable because the action on terms as defined above extends each substitution to a function from terms to terms. It is then natural to define $\sigma' \circ \sigma$ as the substitution satisfying

$(\sigma' \circ \sigma)\, x =_{def} (\sigma\, x)\sigma'$.

The first important result turns now out to be that the action of this composite substitution is equivalent to the sequence of the actions of the composed ones. Before getting to that it is useful to state:

Proposition 4. $x \# M \;\Rightarrow\; \sigma, x{:=}N \cong \sigma \downharpoonleft M$.

The proof is immediate from the definition of the conclusion. □

Proposition 5. $(\sigma' \circ \sigma) \downharpoonleft M \sim_* \sigma' \downharpoonleft (M\sigma)$.

The proof is by some calculation from the definitions. □

Further, in the present context it is convenient to speak of the extensional equality of substitutions. This can be defined in terms of the corresponding equality of restrictions introduced above, in the following way:

$\sigma \cong \sigma' =_{def} (\forall x : \mathsf{V})\, \sigma \cong \sigma' \downharpoonleft x$.

It then follows easily from the definition of extensional equality of restrictions that:

Proposition 6. $\sigma \cong \sigma' \;\Leftrightarrow\; \sigma \cong \sigma' \downharpoonleft M$. □

Now we can prove the following, by simple calculation from the definitions:

Proposition 7 (Distributivity of composition over update)**.** $\sigma' \circ (\sigma, x{:=}M) \cong (\sigma' \circ \sigma), x{:=}M\sigma'$.
□

And now the main result alluded to above is the following:

Proposition 8. *$M(\sigma' \circ \sigma) = (M\sigma)\sigma'$.*

The proof is by structural induction on M. We detail the case of abstractions: First we observe that, by Proposition 5, $(\sigma' \circ \sigma) \downharpoonright \lambda x.M \sim_* \sigma' \downharpoonright (\lambda x.M)\sigma$. Hence we can put $y = \chi((\sigma' \circ \sigma) \downharpoonright \lambda x.M) = \chi(\sigma' \downharpoonright (\lambda x.M)\sigma)$ by axiom 2 of the choice function. Now,

$(\lambda x.M)(\sigma' \circ \sigma)$
= (Action of substitutions)

$\lambda y.M((\sigma' \circ \sigma), x{:=}y)$,

and, on the other hand,

$((\lambda x.M)\sigma)\sigma'$
= (Action of substitutions, with $x' \# \sigma \downharpoonright \lambda x.M$)

$\lambda y.(M(\sigma, x{:=}x'))(\sigma', x'{:=}y)$
= (Induction)

$\lambda y.M(\sigma', x'{:=}y \circ \sigma, x{:=}x')$
= (Propositions 7 and 6 and Lemma 1)

$\lambda y.M((\sigma', x'{:=}y \circ \sigma), x{:=}y)$.

It is therefore sufficient to show $(\sigma' \circ \sigma), x{:=}y \cong (\sigma', x'{:=}y \circ \sigma), x{:=}y \downharpoonright M$. Let then $z * M$. If $z = x$ then the two substitutions in question coincide at z (yielding the term y). If otherwise $z \neq x$, we observe first that it follows $z * \lambda x.M$ and therefore $x' \# \sigma\, z$. Let us now calculate the right hand side substitution on z:

$((\sigma', x'{:=}y \circ \sigma), x{:=}y)\, z$
= ($z \neq x$)

$(\sigma', x'{:=}y \circ \sigma)\, z$
= (Composition of substitutions)

$(\sigma\, z)(\sigma', x'{:=}y)$
= (Proposition 4, using $x' \# \sigma\, z$)

$(\sigma\, z)\sigma'$
= (Composition of substitutions)

$(\sigma' \circ \sigma)\, z$
= ($z \neq x$)

$((\sigma' \circ \sigma), x{:=}y)\, z$.

□

The following is obtained by direct calculation:

Proposition 9.

1. *$(\sigma_1 \circ \sigma_2) \circ \sigma_3 \cong \sigma_1 \circ (\sigma_2 \circ \sigma_3)$.*
2. *$\sigma \circ \iota \cong \sigma$.* □

However, we do not have ι as left identity to composition. Due to the uniform renaming of the bound variables performed in the action of substitution we get the result only up to α-conversion, which we shall define in the next section. We end up with a result that will be useful below:

Proposition 10. $z \,\#\, \lambda x.M \Rightarrow \sigma, x{:=}y \cong (\sigma, z{:=}y) \circ (\iota, x{:=}z) \downharpoonleft M$.

The proof involves simple calculations and distinction of cases $\square$

3.3 Alpha-conversion

Alpha-conversion is now defined inductively and in a syntax directed manner as follows:

$$\frac{}{x \sim_\alpha x} \qquad \frac{M \sim_\alpha M' \qquad N \sim_\alpha N'}{MN \sim_\alpha M'N'}$$

$$\frac{M(\iota, x{:=}z) \sim_\alpha M'(\iota, x'{:=}z)}{\lambda x.M \sim_\alpha \lambda x'.M'}\; z \,\#\, \lambda x.M, \lambda x'.M'$$

This definition is inspired in one given in [53] and it is also similar to the one in [55]. The symmetry of the abstraction rule favours certain proofs as we shall comment shortly. It is not present in Stoughton's definition, which renames the bound variable of one of the abstractions into the one of the other under appropriate circumstances. Stoughton's definition includes rules for reflexivity, symmetry and transitivity of the relation right from the beginning and then one of the final results of the work is an almost syntax directed characterisation —namely with two different rules for abstractions. We instead shall show that the present syntax-directed definition gives an equivalence relation, which is the first important result to be reached below. The other one is that α-conversion is compatible with substitution. As we comment in detail in the final section, our development is considerably simpler, although at the price of employing size induction at one point, which we shall indicate. Stoughton's development, meanwhile, is entirely free from size induction.

Our first result is:

Lemma 3. $M \sim_\alpha M' \;\Rightarrow\; M \sim_* M'$.

The proof is by induction on $\sim_\alpha$, with some calculation needed in the abstraction case. $\square$

The next result is that α-equivalent terms submitted to one and the same substitution get *equalized*. This is due to the uniform renaming of abstractions and the fact that the new name chosen is determined by the restriction of the substitution to the free variables of the abstraction. Since α-equivalent terms (in particular, abstractions) have the same free variables, then the name chosen when effecting a substitution on any two α-equivalent abstractions will be the same. Formally, we need:

Proposition 11. $M \sim_* M' \;\Rightarrow\; \sigma \downharpoonleft M \sim_* \sigma \downharpoonleft M'$,

which follows by just logical calculations. $\square$

We then have:

Lemma 4. *$M \sim_\alpha M' \;\Rightarrow\; M\sigma = M'\sigma$.*

The proof is by induction on $\sim_\alpha$. In the case of abstractions $\lambda x.M$ and $\lambda x'.M'$ we first notice as above that we can put $y = \chi\,(\sigma \downharpoonright \lambda x.M) = \chi\,(\sigma \downharpoonright \lambda x'.M')$. Then,

$(\lambda x.M)\sigma$
$=$ (Action of substitutions)

$\lambda y.M(\sigma, x{:=}y)$,

and

$(\lambda x'.M')\sigma$
$=$ (Action of substitutions)

$\lambda y.M'(\sigma, x'{:=}y)$.

Now consider $z \# \lambda x.M, \lambda x'.M'$ as in the premise of the abstraction rule of $\sim_\alpha$. Then we can show the bodies of the two abstractions equal, as follows:

$M(\sigma, x{:=}y)$
$=$ (Proposition 10, using $z \# \lambda x.M$)

$M((\sigma, z{:=}y) \circ (\iota, x{:=}z))$
$=$ (Action of the composition of substitutions)

$(M(\iota, x{:=}z))(\sigma, z{:=}y)$
$=$ (Induction)

$(M'(\iota, x'{:=}z))(\sigma, z{:=}y)$
$=$ (Action of the composition of substitutions)

$M'((\sigma, z{:=}y) \circ (\iota, x'{:=}z))$
$=$ (Proposition 10, using $z \# \lambda x'.M'$)

$M'(\sigma, x'{:=}y)$.
□

We can now get to the following important result:

Lemma 5. *$M\iota = M'\iota \;\Rightarrow\; M \sim_\alpha M'$.*

The proof is by complete induction on the size of M with a subordinated induction of the same kind on M'. Let us look at the case in which both are abstractions. We have, just by definition of the action of substitutions, that $(\lambda x.M)\iota = \lambda y.M(\iota, x{:=}y)$ and, similarly, $(\lambda x'.M')\iota = \lambda y'.M'(\iota, x'{:=}y')$. But then by hypothesis these two are equal and therefore $y' = y$ and $M(\iota, x{:=}y) = M'(\iota, x'{:=}y)$. Moreover, $y \# \lambda x.M$, which is the same as $y \# \iota \downharpoonright \lambda x.M$, and similarly for $\lambda x'.M'$. Now we reason as follows:

$M(\iota, x{:=}y) = M'(\iota, x'{:=}y)$
$\Rightarrow$ (Congruence of substitution action)

$(M(\iota, x{:=}y))\iota = (M'(\iota, x'{:=}y))\iota$
$\Rightarrow$ (Induction)

$M(\iota, x{:=}y) \sim_\alpha M'(\iota, x'{:=}y)$
$\Rightarrow$ ($\sim_\alpha$, using $y \# \lambda x.M, \lambda x'.M'$)

$\lambda x.M \sim_\alpha \lambda x'.M'$.

□

We are certainly leaving implicit the calculations of sizes involved in the articulation of this induction. In the completely formal version in Agda we use a standard library Induction.Nat which provides a well founded recursion operator. The proof is 30 lines long and uses mainly lemmas about the order relation on natural numbers. We could alternatively have used Agda's sized types.
The following is now immediate:

Corollary 1. $M \sim_\alpha M' \;\Leftrightarrow\; M\iota = M'\iota$. □

Notice that the corollary provides a method of normalisation with respect to $\sim_\alpha$. Actually, after Lemma 5 the following could also be said immediate:

Lemma 6. $\sim_\alpha$ *is an equivalence relation.*

We illustrate the proof with just the case of reflexivity: evidently $M\iota = M\iota$, hence $M \sim_\alpha M$. In the same way, symmetry and transitivity of equality are transferred to $\sim_\alpha$. □

We hereby achieve a syntax directed characterisation of $\sim_\alpha$, plus the result that it is a congruence, in a much more direct way than the one in Stoughton's paper or its formalisation in [33]. The conciseness achieved justifies, to our mind, the use we have made of size induction; we do, however, remark that this is not essential, in the sense that we could have formalised the whole development as originally by Stouhgton in [61], only that in a much lengthier way.

We now arrive to the so-called Substitution Lemma for $\sim_\alpha$, which establishes that this relation is compatible with substitutions. To begin with, the α-equivalence of substitutions is also properly defined on restrictions:

$(\sigma \downharpoonright M) \sim_\alpha (\sigma' \downharpoonright M') =_{def} M \sim_* M' \wedge (x * M \Rightarrow \sigma\, x \sim_\alpha \sigma'\, x)$.

As in other cases, we write $\sigma \sim_\alpha \sigma' \downharpoonright M$ when M and M' above coincide. We now first have a straightforward generalisation of the Corollary 1:

Corollary 2. $\sigma \sim_\alpha \sigma' \downharpoonright M \;\Leftrightarrow\; \iota \circ \sigma \cong \iota \circ \sigma' \downharpoonright M$. □

Now we state:

Lemma 7. $\sigma \sim_\alpha \sigma' \downharpoonright M \;\Rightarrow\; M\sigma \sim_\alpha M\sigma'$.

The proof is the following calculation showing $(M\sigma)\iota = (M\sigma')\iota$, whence the thesis:

$(M\sigma)\iota$
= (Action of composition of substitutions)

$M(\iota \circ \sigma)$
= (Since $\sigma \sim_\alpha \sigma' \downharpoonright M$, using Corollary 2 and Lemma 1)

$M(\iota \circ \sigma')$
= (Action of composition of substitutions)

$(M\sigma')\iota$. □

This lemma also admits a proof by structural induction on M. The induction-free proof is a consequence of the normalisation procedure embodied in the corollaries 1 and 2. We finally arrive at:

Lemma 8 (Substitution Lemma for $\sim_\alpha$)**.** *$M \sim_\alpha M'$ and $\sigma \sim_\alpha \sigma' \downharpoonright M \;\Rightarrow M\sigma \sim_\alpha M'\sigma'$.*

The proof is now a very short calculation:

$M\sigma$
$\sim_\alpha$ (Lemma 7)

$M\sigma'$
$\sim_\alpha$ (Lemma 4 and reflexivity of $\sim_\alpha$)

$M'\sigma'$. □

Also simple calculations from Lemma 5 conduct to the two rules for abstractions in Stoughton´s definition of $\sim_\alpha$:

Corollary 3.

1. $y \# M \;\Rightarrow\; \lambda x.M \sim_\alpha \lambda y.M(\iota, x{:=}y)$.
2. $M \sim_\alpha M' \;\Rightarrow\; \lambda x.M \sim_\alpha \lambda x.M'$. □

The following are also immediate from Lemma 5, concerning the effect of the identity substitution.

Corollary 4.

1. $M\iota \sim_\alpha M$.
2. $(\iota \circ \sigma) \sim_\alpha \sigma$. □

As with equality before, the second line above could as well be just $(\iota \circ \sigma) \sim_\alpha \sigma \downharpoonright x$.

Finally, we give three results to be employed in the next section:

Corollary 5.

1. $y \# \sigma \downharpoonright \lambda x.M \;\Rightarrow\; (M(\sigma, x{:=}y))(\iota, y{:=}N) \sim_\alpha M(\sigma, x{:=}N)$.
2. $y \# \sigma \downharpoonright \lambda x.M \;\Rightarrow\; (\lambda x.M)\sigma \sim_\alpha \lambda y.M(\sigma, x{:=}y)$.
3. $\lambda x M \sim_\alpha \lambda y N \;\Rightarrow\; M \sim_\alpha N(\iota, y{:=}x)$. □

3.4 Beta-Reduction and the Church-Rosser Theorem

Beta-contraction is the simple relation given as $(\lambda x.M)N \;\triangleright\; M(\iota, x{:=}N)$. We obtain beta-reduction by considering the reflexive, transitive, and compatible with the syntactic constructors, closure of the former, augmented with $\sim_\alpha$. Beta-reduction is confluent, which has a classical proof by Tait and Martin-Löf, whose formalisation in Type Theory we depict presently.

The proof rests upon the property of confluence of the so-called *parallel* reduction, which we present here below as an inductive definition. The rules are essentially the same as in [4] or [62] but we have to add a rule allowing explicit α-conversion, since we are working with concrete terms, i.e. not identified under such relation. The definition below can also be regarded as an inductive formalisation of the version in [29]:

$$\frac{}{x \rightrightarrows x} \qquad \frac{M \rightrightarrows M' \qquad N \rightrightarrows N'}{MN \rightrightarrows M'N'} \qquad \frac{M \rightrightarrows M'}{\lambda x.M \rightrightarrows \lambda x.M'}$$

$$\frac{M \rightrightarrows M' \qquad N \rightrightarrows N'}{(\lambda x.M)N \rightrightarrows M'(\iota, x{:=}N')} \qquad \frac{M \rightrightarrows M' \qquad M' \sim_\alpha M''}{M \rightrightarrows M''}$$

It will show convenient to make use of the relation $\rightrightarrows_0$ that obtains by omitting the last rule above, i.e. by not performing steps of $\sim_\alpha$ conversion. Parallel reduction can be extended to (restrictions of) substitutions in a direct way, i.e.

$\sigma \rightrightarrows \sigma' \downharpoonleft M =_{def} x * M \Rightarrow \sigma\, x \rightrightarrows \sigma'\, x.$

We have, to begin with, the following:

Lemma 9.

1. $M \rightrightarrows M' \;\Rightarrow\; x * M' \Rightarrow x * M.$
2. $M \rightrightarrows M' \;\Rightarrow\; x \# M \Rightarrow x \# M'.$

The proofs are simple inductions on $\rightrightarrows$. □

It easily follows that

Corollary 6. *$M \rightrightarrows M'$ and $\sigma \rightrightarrows \sigma' \downharpoonleft M \;\Rightarrow\; y \# \sigma \downharpoonleft M \;\Rightarrow\; y \# \sigma' \downharpoonleft M'$.* □

Now we prove that parallel reduction is compatible with substitution:

Lemma 10 (Substitution lemma for parallel reduction)**.**
$M \rightrightarrows M'$ and$\sigma \rightrightarrows \sigma' \downharpoonleft M \quad\Rightarrow\quad M\sigma \rightrightarrows M'\sigma'$.

The proof is by induction on $\rightrightarrows$. We spell it in full.

For a variable x, we have:

$x\sigma$
$=$ (Action of substitutions)

$\sigma\, x$
$\rightrightarrows$ ($\sigma \rightrightarrows \sigma' \downharpoonleft x$)

$\sigma'\, x$
$=$ (Action of substitutions)

$x\sigma'$.

For application $MN \rightrightarrows M'N'$:

$(MN)\sigma$
$=$ (Action of substitutions)

$M\sigma\ N\sigma$
$\rightrightarrows$ (Induction and $\rightrightarrows$ of applications)

$M'\sigma\ N'\sigma$
$=$ (Action of substitutions)

$(M'N')\sigma.$

For abstraction $\lambda x.M \rightrightarrows \lambda x.M'$:

$(\lambda x.M)\sigma$
$=$ (Action of substitutions, with $y \mathrel{\#} \sigma \downharpoonright \lambda x.M$)

$\lambda y.M(\sigma, x{:=}y)$
$\rightrightarrows$ (Induction, with $\sigma, x{:=}y \rightrightarrows \sigma', x{:=}y \downharpoonright M$, and $\rightrightarrows$ of abstractions)

$\lambda y.M'(\sigma', x{:=}y)$
$\sim_\alpha$ (Corollary 5.2, using $y \mathrel{\#} \sigma' \downharpoonright \lambda x.M'$ which follows from $y \mathrel{\#} \sigma \downharpoonright \lambda x.M$ by Corollary 6)

$(\lambda x.M')\sigma'.$

Notice that the two last steps above amount to *one* application of the rule of $\rightrightarrows$ involving α-conversion (i.e. the last rule of the definition of $\rightrightarrows$) and therefore yield the result required.

For the case of a β-parallel reduction, i.e. $(\lambda x.M)N \rightrightarrows M'(\iota, x{:=}N')$:

$((\lambda x.M)N)\sigma$
$=$ (Action of substitutions, with $y \mathrel{\#} \sigma \downharpoonright \lambda x.M$)

$(\lambda y.M(\sigma, x{:=}y))N\sigma$
$\rightrightarrows$ (Induction, with $\sigma, x{:=}y \rightrightarrows \sigma', x{:=}y \downharpoonright M$, and outermost application of the β-rule of $\rightrightarrows$)

$(M'(\sigma', x{:=}y))(\iota, y{:=}N'\sigma')$
$\sim_\alpha$ (Corollary 5.1, using $y \mathrel{\#} \sigma' \downharpoonright \lambda x.M'$ which follows from $y \mathrel{\#} \sigma \downharpoonright \lambda x.M$ by Corollary 6)

$M'(\sigma', x{:=}N'\sigma')$
$=$ (Composition of substitutions and distributivity over update)

$(M'(\iota, x{:=}N'))\sigma'.$

Finally, for the last case, where $M \rightrightarrows M'$ and $M' \sim_\alpha M''$:

$M\sigma$
$\rightrightarrows$ (Induction)

$M'\sigma'$
$=$ (Lemma 4)

$M''\sigma'.$

$\square$

Notice that this rather straightforward induction on $\rightrightarrows$ is possible because of the structural definition of the action of substitutions. A definition of substitutions by recursion on the size of terms, be it the Curry-Feys one or one using renaming, would oblige to use size induction on terms in this proof, with considerable disadvantage.

Now we stand quite close to establishing the confluence of $\rightrightarrows$, i.e. that $M \rightrightarrows M'$ and $M \rightrightarrows M''$ imply the existence of P such that $M' \rightrightarrows P$ and $M'' \rightrightarrows P$. Actually, a proof by induction on $M \rightrightarrows M'$ with subordinate case analysis of $M \rightrightarrows M''$ can be found in [4] and could easily be adapted —except for the fact that it does not consider steps of $\sim_\alpha$ conversion, since it identifies terms up to such relation. However, the mentioned proof may actually be considered as proceeding by induction on our $\rightrightarrows_0$ relation of reduction and used as a basis for formalisation within our framework. We therefore adopt the following strategy, very similar to the one employed in the presentation in [29]:

1. We prove an $\sim_\alpha$*-postponement* lemma, to the effect that $M \rightrightarrows N \Rightarrow M \rightrightarrows_0 P \wedge P \sim_\alpha N$ for some P.
2. We prove that it also holds that $M \sim_\alpha M' \wedge M' \rightrightarrows N \Rightarrow M \rightrightarrows N$, i.e. a symmetric version of the rule of $\sim_\alpha$ conversion step.
3. We prove the following confluence property: $M \rightrightarrows_0 M'$ and $M \rightrightarrows_0 M''$ imply the existence of P such that $M' \rightrightarrows P$ and $M'' \rightrightarrows P$.

From these results it is rather direct to show the desired:

Lemma 11 (Confluence of parallel reduction)**.** $M \rightrightarrows M'$ *and* $M \rightrightarrows M'' \;\Rightarrow\; (\exists P)(M' \rightrightarrows P$ *and* $M'' \rightrightarrows P)$.

$M \rightrightarrows M' \wedge M \rightrightarrows M''$
$\Rightarrow$ ($\sim_\alpha$ postponement)

$M \rightrightarrows_0 M_0' \wedge M \rightrightarrows M_0''$ with $M_0' \sim_\alpha M' \wedge M_0'' \sim_\alpha M''$
$\Rightarrow$ (confluence, i.e. property 3 above)

$(\exists P)(M_0' \rightrightarrows P \wedge M_0'' \rightrightarrows P)$ with $M' \sim_\alpha M_0' \wedge M'' \sim_\alpha M_0''$
$\Rightarrow$ (property 2 above)

$(\exists P)(M' \rightrightarrows P \wedge M'' \rightrightarrows P)$.

□

The lemmas of the enumeration above are as follows:

Lemma 12 (Postponement of $\sim_\alpha$ steps)**.** $M \rightrightarrows N \;\Leftrightarrow\; M \rightrightarrows_0 P \wedge P \sim_\alpha N$ *for some* P.

The direction from right to left obtains directly by observing that $M \rightrightarrows_0 P$ implies $M \rightrightarrows P$ and application of the last rule of the definition of $\rightrightarrows$, i.e. that of $\sim_\alpha$ conversion step. The direction from left to right is an easy induction on $\rightrightarrows$. □

Actually, a symmetric form of the converse of this result also holds, i.e. one that could be called of "prepending" of $\sim_\alpha$ conversion steps:

Lemma 13. $M \sim_\alpha P \wedge P \rightrightarrows_0 N \;\Rightarrow\; M \rightrightarrows N$.

The proof is by induction on M. In the case of applications, two subcases have to be considered depending on the rule employed for performing $\rightrightarrows_0$. In the case of abstractions, part 3 of Corollary 5 is applied. □

Using the latter it is direct to show the symmetric of the rule of $\sim_\alpha$ steps of the definition of $\rightrightarrows$:

Lemma 14. $M \sim_\alpha P \wedge P \rightrightarrows N \;\Rightarrow\; M \rightrightarrows N$.

$M \sim_\alpha P \wedge P \rightrightarrows N$
$\Rightarrow$ ($\sim_\alpha$ postponement)

$M \sim_\alpha P \wedge P \rightrightarrows_0 P_0 \wedge P_0 \sim_\alpha N$
$\Rightarrow$ (Lemma 13)

$M \rightrightarrows P_0 \wedge P_0 \sim_\alpha N$
$\Rightarrow$ (Rule of $\sim_\alpha$ step)

$M \rightrightarrows N.$

□

It only remains to establish the following confluence result:

Lemma 15. *$M \rightrightarrows_0 M'$ and $M \rightrightarrows_0 M'' \;\Rightarrow\; (\exists P)(M' \rightrightarrows P$ and $M'' \rightrightarrows P)$.*

The proof is essentially the standard induction on $M \rightrightarrows_0 M'$ with subordinate case analysis of $M \rightrightarrows_0 M''$, as developed in [4]. Working on $\rightrightarrows_0$, as established by our strategy above, alleviates significantly the number of combinations to be considered. Let us illustrate some of the detail involved by analyzing the case of the rule of parallel reduction of applications. We therefore put ourselves in the position that $MN \rightrightarrows_0 M'N'$, with both $M \rightrightarrows_0 M'$ and $N \rightrightarrows_0 N'$. If now in addition $MN \rightrightarrows_0 Q$, it turns out that there are two cases by way of which this may come about. The first one is that $Q = M''N''$ with both $M \rightrightarrows_0 M''$ and $N \rightrightarrows_0 N''$, i.e. by use of the rule of parallel reduction of applications. But then, just by the induction hypotheses, we get that there exist P_1 and P_2 such that, on the one hand side, $M' \rightrightarrows P_1$ and $M'' \rightrightarrows P_1$ and, on the other $N' \rightrightarrows P_2$ and $N'' \rightrightarrows P_2$. Therefore, $M'N' \rightrightarrows P_1P_2$ and $M''N'' \rightrightarrows P_1P_2$, as desired. The second case is when $MN \rightrightarrows_0 Q$ is arrived at by use of the β-parallel reduction rule. Then it must be $M = \lambda x M_0$ and $Q = M_0''(\iota, x{:=}N'')$ with $M_0 \rightrightarrows_0 M_0''$ and $N \rightrightarrows_0 N''$. Now, in the first place, we get by induction hypothesis that, from $N \rightrightarrows_0 N'$ and $N \rightrightarrows_0 N''$ it follows that there exists P_1 such that both $N' \rightrightarrows P_1$ and $N'' \rightrightarrows P_1$. On the other hand, by analysis of the rules of parallel reduction of abstractions, we must have $M' = \lambda x M_0'$ and, since $M_0 \rightrightarrows_0 M_0''$, also $\lambda x M_0 \rightrightarrows_0 \lambda x M_0''$. Hence, by induction hypothesis, there exists P such that both $\lambda x M_0' \rightrightarrows P$ and $\lambda x M_0'' \rightrightarrows P$. But, by the lemma of postponement of $\sim_\alpha$-conversion steps, we now that there must exist also $P' \sim_\alpha P$ such that both $\lambda x M_0' \rightrightarrows_0 P'$ and $\lambda x M_0'' \rightrightarrows_0 P'$. Further, by analysis of the rules of parallel reduction of abstractions, it must be $P' = \lambda x P_0$ with both $M_0' \rightrightarrows_0 P_0$ and $M_0'' \rightrightarrows_0 P_0$. Therefore what we had at the beginning was that $(\lambda x M_0)N \rightrightarrows_0 (\lambda x M_0')N'$ and $(\lambda x M_0)N \rightrightarrows_0 M_0''(\iota, x{:=}N'')$. But now $(\lambda x M_0')N' \rightrightarrows P_0(\iota, x{:=}P_1)$ by use of the β-parallel reduction rule and, on the other hand, $M_0''(\iota, x{:=}N'') \rightrightarrows P_0(\iota, x{:=}P_1)$ by the substitution lemma of $\rightrightarrows$. This gives us the desired confluence.
□

Now we proceed to the following:

Lemma 16. *If a reduction relation R is confluent, then so is its reflexive and transitive closure R^*.*

The proof is standard, by a double induction. □

Corollary 7. *$\rightrightarrows^*$ is confluent.* □

If we now write $\twoheadrightarrow$ for the relation of β reduction, we have:

Lemma 17. *$\twoheadrightarrow \; = \; \rightrightarrows^*$,*

from which we finally arrive at:

Theorem 1 (Church-Rosser)**.** *Beta-reduction is confluent.* □

It only remains to discuss the proof of Lemma 17, which is actually completely standard. We namely prove that:

1. One-step β-reduction $\rightarrow$ is included in $\rightrightarrows$. The relation $\rightarrow$ is the contextual, i.e. compatible with the syntactic constructors, closure of the simple relation of β-contraction, augmented with $\sim_\alpha$. The proof proceeds by a corresponding simple induction. It follows that β-reduction $\twoheadrightarrow\ =\ \rightarrow^*$ is included in $\rightrightarrows^*$.

2. $\rightrightarrows$ is included in $\twoheadrightarrow$. It then follows that $\rightrightarrows^*$ is included in $\twoheadrightarrow^* = \twoheadrightarrow$. This proof is also by a direct induction on $\rightrightarrows$. Let us show the interesting case, corresponding to the β-parallel reduction rule:

   We have $M \rightrightarrows M'$ and $N \rightrightarrows N'$ and need to prove $(\lambda x M)N \twoheadrightarrow M'(\iota, x{:=}N')$. Now, by induction hypotheses, we get both $M \twoheadrightarrow M'$ and $N \twoheadrightarrow N'$, and reason as follows:

   $M \twoheadrightarrow M'$
   $\Rightarrow$ (By compatibility of $\twoheadrightarrow$ with the syntactic constructors)

   $\lambda x M \twoheadrightarrow \lambda x M'$
   $\Rightarrow$ (Idem, using $N \twoheadrightarrow N'$)

   $(\lambda x M)N \twoheadrightarrow (\lambda x M')N'$
   $\Rightarrow$ (Transitivity of $\twoheadrightarrow$, using $(\lambda x M')N' \twoheadrightarrow M'(\iota, x{:=}N')$)

   $(\lambda x M)N \twoheadrightarrow M'(\iota, x{:=}N')$.
   $\square$

3.5 Assignment of Simple Types

Let now ν be a syntactic category of ground types. Then the category of *simple types* is given by the following grammar:

$\alpha, \beta ::=\ \nu \ \mid\ \alpha \rightarrow \beta.$

We consider *contexts* of *variable declarations* as lists thereof. These lists are actually the implementation of finite tables, i.e. we will have the following operations and relations on contexts:

- The *empty* context $\cdot$.
- The *update* of a context Γ with a declaration $x : \alpha$, to be written $\Gamma, x : \alpha$. As explained below, adding a declaration of a variable overrides any prior declaration for the same variable.
- We will write $x \in \mathsf{dom}\,\Gamma$ the fact that x is declared in Γ.
- A *lookup* operation returning the type of any variable declared in a context. For context Γ and variable x this is to be written $\Gamma\, x$. This operation therefore satisfies:

  $$\begin{array}{lll}(\Gamma, x : \alpha)\, x & =_{def} & \alpha \\ (\Gamma, y : \alpha)\, x & =_{def} & \Gamma\, x \text{ if } y \neq x.\end{array}$$

- A relation of *inclusion* (or *extension*) between contexts defined as follows:

  $\Gamma \preccurlyeq \Delta =_{def} x \in \mathsf{dom}\,\Gamma \Rightarrow (x \in \mathsf{dom}\,\Delta \ \wedge\ \Delta\, x = \Gamma\, x).$

The system of assignment of simple types to pure terms is defined inductively by the following rules:

$$\frac{}{\Gamma \vdash x : \Gamma x}\; x \in \mathsf{dom}\,\Gamma \qquad \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}$$

$$\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.M : \alpha \to \beta}$$

The system satisfies the following *weakening* lemmas:

Proposition 12 (Weakening).

1. $\Gamma \preccurlyeq \Delta \;\Rightarrow\; \Gamma \vdash M : \alpha \;\Rightarrow\; \Delta \vdash M : \alpha.$
2. $x \# M \;\Rightarrow\; \Gamma \vdash M : \alpha \;\Rightarrow\; \Gamma, x : \beta \vdash M : \alpha.$

The proofs proceed by induction on the type system. The second result uses the first one for the case of functional abstractions □

The assignment of types extends itself to substitutions, as follows:

$\sigma : \Gamma \to \Delta =_{def} x \in \mathsf{dom}\,\Gamma \Rightarrow \Delta \vdash \sigma\, x : \Gamma\, x.$

That σ is assigned the "type" $\Gamma \to \Delta$ amounts therefore to consider its restriction to the variables declared in Γ. Then the terms assigned to those variables in σ must respect the declarations in Γ, depending in turn on the declarations in Δ. It shows convenient to consider as well restrictions of these typed substitutions to (the free variables of) terms, in the following way:

$(\sigma : \Gamma \to \Delta) \downharpoonleft M =_{def} x * M \Rightarrow x \in \mathsf{dom}\,\Gamma \Rightarrow \Delta \vdash \sigma\, x : \Gamma\, x.$

Most often, but not necessarily, the term M in these restrictions will be such that $\Gamma \vdash M : \alpha$ and therefore its free variables shall all be declared in Γ. It could be argued that it is preferable from a theoretical point of view to introduce instead restrictions of *contexts* relative to given terms and leave unmodified the general notion of typed substitution. We have, however, found that the present formulation leads to a more direct formalisation. Some useful properties concerning typing and substitutions are put together in the following proposition:

Proposition 13.

1. $\iota : \Gamma \to \Gamma.$
2. $\Gamma \vdash M : \alpha \;\Rightarrow\; (\iota, x{:=}M) : \Gamma, x : \alpha \to \Gamma.$
3. $\sigma : \Gamma \to \Delta \;\Rightarrow\; (\sigma : \Gamma \to \Delta) \downharpoonleft M.$
4. $(\iota, y{:=}x : \Gamma, y : \alpha \to \Gamma, x : \alpha) \downharpoonleft M(\iota, x{:=}y).$
5. $x \# \sigma \downharpoonleft \lambda y M \wedge (\sigma : \Gamma \to \Delta) \downharpoonleft \lambda y M \Rightarrow (\sigma, y{:=}x : \Gamma, y : \alpha \to \Delta, x : \alpha) \downharpoonleft M.$

The first four items are direct. For the last one, use that, given z such that $z * M$ and $z \in \mathsf{dom}\,(\Gamma, y : \alpha)$, either $z = y$ or $z \neq y$. In the first case the result is immediate. In the second, we have both $(\sigma, y{:=}x)\; z = \sigma\; z$ and $(\Gamma, y : \alpha)\; z = \Gamma\; z$; now $\Delta \vdash \sigma\; z : \Gamma\; z$ follows from $z * M$ and $z \neq y$, hence $z * \lambda y M$, and $(\sigma : \Gamma \to \Delta) \downharpoonleft \lambda y M$. Finally, apply part 2 of the weakening lemma using that $x \# \sigma \downharpoonleft \lambda y M$ implies $x \# \sigma\; z$. □

We finally arrive at the crucial substitution lemma:

Lemma 18 (Substitution lemma for type assignment)**.** $\Gamma \vdash M : \alpha \Rightarrow \sigma : \Gamma \to \Delta \;\Rightarrow\; \Delta \vdash M\sigma : \alpha.$

This is obtained from the following lemma using part 3 of proposition 13:

Lemma 19. $\Gamma \vdash M : \alpha \;\Rightarrow\; (\sigma : \Gamma \to \Delta) \downharpoonleft M \;\Rightarrow\; \Delta \vdash M\sigma : \alpha.$

The proof is by induction on $\Gamma \vdash M : \alpha$. For the interesting case of abstractions, we have the premise $\Gamma, x : \alpha \vdash M : \beta$ and $(\sigma : \Gamma \to \Delta) \downharpoonleft \lambda xM$. Now it obtains

$\Delta \vdash (\lambda xM)\sigma : \alpha \to \beta$
$\Leftarrow$ (action of substitution, with $z = \chi(\sigma \downharpoonleft \lambda xM)$)

$\Delta \vdash \lambda zM(\sigma, x{:=}z) : \alpha \to \beta$
$\Leftarrow$ (typing rule for abstractions)

$\Delta, z : \alpha \vdash M(\sigma, x{:=}z) : \beta$
$\Leftarrow$ (induction hypothesis)

$(\sigma, x{:=}z : \Gamma, x : \alpha \to \Delta, z : \alpha) \downharpoonleft M$
$\Leftarrow$ (Proposition 13, part 5)

$z = \chi(\sigma \downharpoonleft \lambda xM) \;\#\, \sigma \downharpoonleft \lambda xM$ and $(\sigma : \Gamma \to \Delta) \downharpoonleft \lambda xM.$

□

Again, the former induction on the type system is possible because of the structural definition of the action of substitutions.

It is now direct to get:

Lemma 20. *Typing is preserved by β-contraction.*

$\Gamma \vdash (\lambda xM)N : \beta$
$\Rightarrow$ (typing rules)

$\Gamma, x : \alpha \vdash M : \beta$ and $\Gamma \vdash N : \alpha$
$\Rightarrow$ (Proposition 13, part 2)

$\iota, x{:=}N : \Gamma, x : \alpha \to \Gamma$
$\Rightarrow$ (Substitution Lemma)

$\Gamma \vdash M(\iota, x{:=}N) : \beta.$
□

Now, by simple induction on the contextual closure it follows that:

Corollary 8. *One step β-reduction preserves typing.* □

We now turn to showing that typing is compatible with $\sim_\alpha$ too. Firstly we obtain:

Lemma 21. $\Gamma \vdash M\iota : \alpha \;\Leftrightarrow\; \Gamma \vdash M : \alpha.$

The direction from right to left follows immediately from the substitution lemma and part 1 of Proposition 13. The converse goes by induction on the typing relation. The non-trivial case of abstractions is as follows:

$\Gamma \vdash (\lambda x M)\iota : \alpha \to \beta$
$\Rightarrow$ (action of substitution, with $z = \chi(\iota \downharpoonleft \lambda x M)$)

$\Gamma \vdash \lambda z M(\iota, x{:=}z) : \alpha \to \beta$
$\Rightarrow$ (typing rules)

$\Gamma, z : \alpha \vdash M(\iota, x{:=}z) : \beta$
$\Rightarrow$ (Proposition 13 part 4 and Lemma 20)

$\Gamma, x : \alpha \vdash M(\iota, x{:=}z)(\iota, z{:=}x) : \beta$
$\Rightarrow$ ($z = \chi(\iota \downharpoonleft \lambda x M) \,\#\, \iota \downharpoonleft \lambda x M$)

$\Gamma, x : \alpha \vdash M\iota : \beta$
$\Rightarrow$ (induction hypothesis)

$\Gamma, x : \alpha \vdash M : \beta$
$\Rightarrow$ (typing rule for abstractions)

$\Gamma \vdash \lambda x M : \alpha \to \beta.$
$\square$

Thence we arrive at:

Lemma 22. $\Gamma \vdash M : \alpha$ *and* $M \sim_\alpha N \;\Rightarrow\; \Gamma \vdash N : \alpha$,

using normalisation by ι as follows:

$\Gamma \vdash M : \alpha$
$\Rightarrow$ (Lemma 21)

$\Gamma \vdash M\iota : \alpha$
$\Rightarrow$ (Corollary 1, i.e. $M \sim_\alpha N \Rightarrow M\iota = N\iota$)

$\Gamma \vdash N\iota : \alpha$
$\Rightarrow$ (Lemma 21)

$\Gamma \vdash N : \alpha.$
$\square$

We now are able to conclude with

Theorem 2 (Subject Reduction). $\Gamma \vdash M : \alpha$ *and* $M \twoheadrightarrow N \;\Rightarrow\; \Gamma \vdash N : \alpha$

which follows by an easy induction using Corollary 8 and Lemma 22 for the base cases. $\square$

3.6 Conclusions

To our mind, the present work contributes two things:
Firstly, it shows that what we have called the "historical" approach to the meta-theory of the Lambda calculus can be carried out in a completely formal manner so as to scale up to the principal results of the theory. By "historical approach" we mean the one initiated by Curry-Feys [16] and continued at least partly by Hindley and Seldin [29], which consists in treating the calculus in its original syntax —with one sort of names for both free and bound variables—, granting substitution a more basic status than that of α-conversion, and working all the time

with concrete terms, i.e. without identifying terms up to α-conversion. The formal treatment is made feasible because of the use of Stoughton's substitution, which has shown therefore to be the appropriate one for this kind of syntax.
Within the general approach to syntax chosen, the main work to compare is the one by Vestergaard and Brotherston [68] which uses modified rules of α-conversion and β-reduction based on unary substitution to formally prove the Church-Rosser theorem in Isabelle-HOL. Substitution does not proceed in cases of capture and they use explicit α-conversion to perform the renaming achieved by our substitution. As a consequence, the Church-Rosser theorem requires an administrative layer of reasoning for showing that α-conversion and β-reduction interact correctly. This consists in a rather complex definition of a new auxiliary relation for α-conversion, which we do not need. On the other hand, they only use structural principles of induction, either on terms or on relations. As already indicated and commented again below, our use of size induction is not essential, but only convenient for simplifying the presentation of the theory of α-conversion with respect to Stoughton's [61]. Besides, thanks to the use of the multiple form of substitution defined uniformly by structural recursion, we have been able to employ standard strategies for achieving the two principal results, namely confluence and preservation of typing by β-reduction.

Our second contribution consists in presenting Stoughton's theory of substitutions in a new way. First of all, it is based on inductive types and relations, instead of on ordinary set theory. Besides, it presents the following features: (1) It bases itself upon the notion of *restriction* of a substitution to (the free variables of) a term. As a consequence we have used the corresponding finite notions of equality and α-equivalence, whereas Stoughton and the formalisation by Lee [33] use extensional, and thus generally undecidable, equality —in the case of the formalisation, via an ad-hoc postulate in type theory. The extensional equality could have also been avoided by keeping track of the finite domain of each substitution, given that these are identity almost everywhere. But it actually turns out that the relevant relations concerning substitution in this theory are most conveniently formulated as concerning restrictions, which is due to the fact that the behavior of substitutions manifests itself in interaction with terms.
(2) Alpha-equivalence is given as a strictly syntax-directed inductive definition, which is easily proven to be an equivalence relation and therefore a congruence. This stands in contrast to Stoughton's work, which starts with a definition of α-conversion as the least congruence generated by a simple renaming of bound variable –a definition comprising six rules, whereas ours consists of three. Stoughton's whole development is then directed towards characterising α-conversion in the form of a syntax-based definition that contains nevertheless two rules corresponding to abstractions. This therefore gives a neat result standing in correspondence with ours; but the proof is surprisingly dilatory, requiring among others the substitution lemma for α-conversion. The issue manifests itself also in a rather involved character of Lee's formalisation, as witnessed by his own comments in [33]. Two lemmas are crucial in the whole development, whatever strategy is taken: The first is the one stating that substitutions *equalize* α-equivalent terms, i.e. $M \sim_\alpha N \;\Rightarrow M\,\sigma = N\,\sigma$. This is very directly proven in our case by induction on $\sim_\alpha$ due to the symmetric character of the rule for abstractions, which is not the case for Stoughton's version of $\sim_\alpha$ and gives rise to the difficulties pointed out above. The second important lemma is the one stating that equality under the identity substitution implies α-equivalence, i.e. $M\,\iota = N\,\iota \;\Rightarrow M \sim_\alpha N$. This one is very easily proven by Stoughton using symmetry and transitivity of $\sim_\alpha$, since these properties are available from the beginning, whereas we need to proceed by induction on the length of M. The latter might be argued to depart from Stoughton's original goals to simplify the methods of reasoning generally employed. Now, as a matter of fact, it has been the only one point in which a principle of induction other than just structural has been used in our proofs and, to our mind, the overall cost of the development pays off such

expenditure. Specifically, our proof that $\sim_\alpha$ is a congruence is finally quite concise and down to the point, not needing in particular the substitution lemma. The induction on the size of terms could be straightforwardly encoded in Agda using library functions.

As further work, concerning the formalisation of the metatheory of the Lambda calculus, we could complete the presentation with proofs of the Standarisation Theorem and of Strong Normalisation of the system with simple types. We also believe it interesting to investigate the generalisation of the approach to systems of languages with binders as e.g. the one presented in [54].

CHAPTER 4

Alpha-Structural Induction and Recursion Part I - Substitution Lemmas

In this chapter we reproduce the article *"Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory"*, published in Electronic Notes in Computer science [13]. In this work we present another formalisation of the λ-calculus in Constructive Type Theory, this time basing α-conversion upon the name swapping operation. We formulate principles of induction and recursion, which allow to work modulo α-conversion. These principles are all derived from the simple structural induction principle on concrete terms, and aim at reproducing the use of the BVC in pen-and-paper works, while trying to hide all its complexity in the rigorous context of a proof assistant. We work out applications to some fundamental meta-theoretical results, such as the substitution lemma for α-conversion and the substitution composition lemma. The whole work is implemented in the Agda proof assistant [46]. In this work my contribution was the conception of the right α-induction and recursion principles, as well as the whole codification in Agda.

4.1 Introduction

We are interested in methods for formalising in constructive type theory the meta-theory of the lambda-calculus. The main reason for this is that the lambda calculus is both a primigenial programming language and a prime test bed for formal reasoning on tree structures that feature (name) binding.

Specifically concerning the latter, the informal procedure consists to begin with in "identifying terms up to α-conversion". However, this is not simply carried out when functions are defined by recursion and properties proven by induction. The problem has to do with the fact that the consideration of the α-equivalence classes is actually conducted through the use of convenient representatives thereof. These are chosen by the so-called Barendregt Variable Convention (BVC): each term representing its α-class is assumed to have bound names all different and different from all names free in the current context. Now, a general validity criterion determines that this procedure ought to be accompanied in all cases by the verification that the proofs and results of functions depend only on the α-class and do not vary with the particular choice of the representative in question. Such verification is seldom accomplished but yet it is not the main difficulty concerning the validity of the constructions so performed. The crucial point is that e.g. inductive proofs are often carried out employing the structural principle for concrete terms —and then it may well happen that an induction step corresponding to functional abstractions can be carried out for a conveniently chosen bound name but not for an arbitrary one as the principle requires.

The problem can be avoided by the use of de Bruijn's nameless syntax [17] or its more up-to-date version *locally nameless* syntax [2, 7], which uses names for the free or global variables and the indices counting up to the binding abstractor for the occurrences of local parameters. But these methods are not without overhead in the form of several operations or well-formedness predicates. As a result, there certainly is a relief in not having to consider α-conversion; but, at the same time, the nameless syntax seriously affects the connection between actual formal procedures and what could be considered the natural features of syntax. The same has to be said of the map representation introduced in [38].

A different alternative is to replace the (as explained above, problematic) use of structural induction and recursion principles on concrete terms by that of so-called *alpha*-structural principles working directly on the α-equivalence classes. This means providing principles that allow to prove properties by induction and to define functions by recursion by direct use of the BVC, so as to ease the burden associated to the verification of the validity of the procedure.

A first attempt in this direction is [28], which gives an axiomatic description of lambda terms in which equality embodies α-conversion and that provides a method of definition of functions by recursion on such type of objects. This work ultimately rests upon the use of higher-order abstract syntax within the HOL system, and a theoretical model using de Bruijn's nameless syntax is sketched to show the soundness of the system of axioms. In [24, 53, 54], models of syntax with binders are introduced which formulate the basic concepts of abstraction, α-equivalence and a name being "sufficiently fresh" in a mathematical object, on the basis of the simple operation of name swapping. This theory —which has become known as *nominal abstract syntax*— provides a framework of (first-order) languages with binding with associated principles of α-structural recursion and induction that are based on the verification of the non-dependence of the mathematical objects in the current context, as well as of the results of step functions used in recursive definitions, on the bound names chosen for the representatives of the α-classes involved. Implementations of this approach have been tried in Isabelle/HOL [66] and Coq [1]. In the first case the solution rests upon a weak version of higher-order abstract syntax, whereas the second one is an axiomatisation in which —similarly to [28] cited above— equality is postulated as embodying α-conversion and a model of the system based on locally nameless syntax has been constructed.

Yet another approach to the formulation of the alpha-structural principles originates in the observation that, if the property to be tried is α-compatible —i.e., it is actually a property of the α-classes and not just of the concrete terms— then (complete) induction on the *size* of terms can be used to bridge over the possible gap pointed out above in proofs by induction that confine themselves to convenient choice of bound names. Indeed, suppose you need to prove $\mathcal{P}(\lambda x.M)$; now, if what you have is a step from $\mathcal{P}(M^*)$ to $\mathcal{P}(\lambda x^*.M^*)$ for a convenient renaming of the term, then you will be able to use your strong size-induction hypothesis on M^*, since this is still of a size lesser than that of $\mathcal{P}(\lambda x.M)$. Hence you will arrive at $\mathcal{P}(\lambda x^*.M^*)$ and from there to the desired $\mathcal{P}(\lambda x.M)$ because of the α-compatibility of $\mathcal{P}$. This motivates trying to provide a mechanism of this kind to formalise the use of the BVC, and that is what we attempt in this paper. The result is that we are able to provide principles of alpha-structural induction and recursion, implementing the BVC in constructive type theory, using just the ordinary first-order, name-carrying syntax and actually *without* using the strong induction on the size of the terms —i.e. we are able to derive the principles in question from just simple structural induction on concrete terms. To such effect we define α-equivalence by using the basic concepts of nominal abstract syntax, namely freshness and swapping of names. Equality remains the simple definitional one and we do not either perform any kind of quotient construction. The whole development is implemented in the Agda system [46].

The rest of the paper goes as follows: in section 2 we present the infrastructure just mentioned. Section 3 presents the principles, starting from the simple structural induction on terms and ending up with the recursion principle on α-classes. In section 4 we show several applications that bring about certain feeling for the usefulness of the method. Finally, section 5 compares with related work and points out conclusions and further work.

The present is actually a literate Agda document, where we hide some code for reasons of conciseness. The entire code is available at:

https://github.com/ernius/formalmetatheory-nominal

and has been compiled with the last Agda version 2.4.2.2 and 0.9 standard library.

4.2 Infrastructure

Agda

Agda implements Constructive Type Theory [37] (*type theory* for short). It is actually a functional programming language in which:

1. Inductive types can be introduced as usual, i.e. by enumeration of their constructors, but they can be parameterised in objects of other types. Because of the latter it is said that type theory features *families* of types (indexed by a base type) or *dependent* types.

2. Functions on families of types respect the dependence on the base object, which is to say that they are generally of the form $(x : \alpha) \to \beta_x$ where β_x is the type parameterised on x of type α. Therefore the type of the output of a function depends on the *value* of the input.

3. Functions on inductive types are defined by *pattern-maching* equations.

4. Every function of the language must be terminating. The standard form of recursion that forces such condition is *structural* recursion and is, of course, syntactically checked.

5. Because of the preceding feature, type theory can be interpreted as a constructive logic. Specifically, this is achieved by representing propositions as inductive types whose constructors are the introduction rules, i.e. methods of direct proof, of the propositions in question.

Therefore we can say in summary that sets of data, predicates and relations are defined inductively, i.e. by enumeration of their constructors.

Syntax

The set Λ of terms is as usual. It is built up from a denumerable set of names, which we shall call *atoms*, borrowing terminology from nominal abstract syntax.

```
data Λ : Set where
  v     : Atom → Λ
  _·_   : Λ → Λ → Λ
  ƛ     : Atom → Λ → Λ
```

The following is called the *freshness* relation. It holds when a variable does not occur free in a term. Parameters to a function written between curly brackets can be omitted when invoking the function.

```
data _#_ (a : Atom) :  Λ → Set where
  #v    : {b : Atom}            → b ≢ a           → a # v b
  #·    : {M N : Λ }            → a # M → a # N   → a # M · N
  #ƛ≡   : {M : Λ}                                 → a # ƛ a M
  #ƛ    : {b : Atom}{M : Λ}     → a # M           → a # ƛ b M
```

Next comes the fundamental operation of *swapping* of atoms. A finite sequence (composition) of atom swaps constitutes a (finite) atom *permutation* which is the renaming mechanism to be used on terms. The action of atom swaps is first defined on atoms themselves:

```
(_•_)ₐ_ : Atom → Atom → Atom → Atom
( a • b )ₐ c  with c ≟ₐ a
... | yes  _            = b
... | no   _  with c ≟ₐ b
...           | yes _   = a
...           | no  _   = c
```

Here it extends to terms:

```
(_•_)_ : Atom → Atom → Λ → Λ
( a • b ) v c      = v (( a • b )ₐ c)
( a • b ) M · N  = (( a • b ) M) ·  (( a • b ) N)
( a • b ) ƛ c M  = ƛ (( a • b )ₐ c) (( a • b ) M)
```

And the same goes for permutations, which are *lists* of swaps:

```
_•ₐ_ : Π → Atom → Atom
π •ₐ a = foldr (λ s b → ( proj₁ s • proj₂ s )ₐ b) a π

_•_ : Π → Λ → Λ
π • M = foldr (λ s M → ( proj₁ s • proj₂ s ) M) M π
```

We now introduce α-conversion, denoted by ∼α. We use a syntax-directed definition that uses co-finite quantification in the case of the lambda abstractions:

```
data _∼α_ : Λ → Λ → Set where
  ∼αv  : {a : Atom} → v a ∼α v a
```

```
∼α·  : {M M' N N' : Λ} → M ∼α M' → N ∼α N'
       → M · N ∼α M' · N'
∼αƛ  : {M N : Λ}{a b : Atom}(xs : List Atom)
       → ((c : Atom) → c ∉ xs → ( a • c ) M ∼α ( b • c ) N)
       → ƛ a M ∼α ƛ b N
```

The idea is that for proving two abstractions α-equivalent you should be able to prove the respective bodies α-equivalent when you rename the bound names to any name not free in both abstractions. The condition on the new name can be generalised to "any name not in a given list", yielding an equivalent relation. The latter condition is harder to prove, but more convenient to use when you assume ∼α to hold, which is more often the case in the forthcoming proofs.

4.3 Alpha-Structural Induction and Recursion Principles

We start with the simple structural induction over the concrete Λ terms:

```
TermPrimInd : {l : Level}(P : Λ → Set l)
  → (∀ a → P (v a))
  → (∀ M N → P M → P N → P (M · N))
  → (∀ M b → P M → P (ƛ b M))
  → ∀ M → P M
```

Figure 4.1: Concrete Structural Induction Principle

The next induction principle provides a strong hypothesis for the lambda abstraction case: it namely allows to assume the property for all renamings (given by finite permutations of names) of the body of the abstraction:

```
TermIndPerm : {l : Level}(P : Λ → Set l)
  → (∀ a → P (v a))
  → (∀ M N → P M → P N →  P (M · N))
  → (∀ M b → (∀ π → P (π • M)) → P (ƛ b M))
  → ∀ M → P M
```

Figure 4.2: Strong Permutation Induction Principle

Notice that the hypothesis provided for the case of abstractions is akin to the corresponding one of the principle of strong or complete induction on the size of terms, only that expressed in terms of name permutations. This principle can be derived from the former, i.e. from simple structural induction, in very much the same way as complete induction on natural numbers is derived from ordinary mathematical induction. That is to say, we can use structural induction to prove $(\forall \pi)P(\pi \bullet M)$ given the hypotheses of the new principle, from which $P\,M$ follows. For the interesting case of abstractions, we have to prove $(\forall \pi)P(\pi \bullet ƛ\, a\, M)$, which is equal to $(\forall \pi)P(ƛ\,(\pi \bullet_{\mathsf{A}} a)\,(\pi \bullet M))$. The hypothesis of the new principle give us in this case $(\forall M', b)((\forall \pi')P(\pi' \bullet M') \to P(ƛ\, b\, M'))$. Now, instantiating M' as $\pi \bullet M$ and b as $\pi \bullet_{\mathsf{A}} a$, we obtain the desired result if we know that $(\forall \pi')P(\pi' \bullet \pi \bullet M)$, which holds by induction hypothesis of the structural principle.

We call a predicate α-compatible if it is preserved by α-conversion:

$$\begin{array}{l}
\mathsf{\alpha CompatiblePred} : \{l : \mathsf{Level}\} \to (\Lambda \to \mathsf{Set}\ l) \to \mathsf{Set}\ l \\
\mathsf{\alpha CompatiblePred}\ P = \{M\ N : \Lambda\} \to M \sim\alpha\ N \to P\ M \to P\ N
\end{array}$$

For α-compatible predicates we can use the preceding principle to derive the following:

$$\begin{array}{l}
\mathsf{Term\alpha PrimInd} : \{l : \mathsf{Level}\}(P : \Lambda \to \mathsf{Set}\ l) \\
\quad \to \mathsf{\alpha CompatiblePred}\ P \\
\quad \to (\forall\ a \to P\ (\mathsf{v}\ a)) \\
\quad \to (\forall\ M\ N \to P\ M \to P\ N \to P\ (M \cdot N)) \\
\quad \to \exists\ (\lambda\ vs \to (\forall\ M\ b \to b \notin vs \to P\ M \to P\ (ƛ\ b\ M))) \\
\quad \to \forall\ M \to P\ M
\end{array}$$

This new principle enables us to carry out the proof of the abstraction case by choosing a bound name different from the names in a given list vs. It gives a way to emulate the Barendregt Variable Convention (BVC) since, indeed, the names to be avoided will always be finitely many; in using the principle we must provide a list that includes them. This same principle is provided in [1], only that we here give it a proof in terms of the ones previously introduced, instead of just postulating it. Our aim is to employ this principle whenever possible, thereby hiding the use of the swap operation which is confined to the previous principles exposed. The interesting case in the implementation of the principle is of course that of the functional abstraction. We must put ourselves in the position in which we are using the former strong principle and are given an abstraction $ƛ\, b\, M$ for which we have to prove P. We have to employ to this effect the clause of our new principle corresponding to the functional abstractions, which forces us to employ a name b^* out of the given list vs. Therefore we can aspire at proving P for a renaming of the original term, say $ƛ\, b^*\, M^*$. The required result will then follow from the α-compatibility of the predicate P provided $ƛ\, b^*\, M^* \sim\alpha\ ƛ\, b\, M$. This imposes the condition that the name b^* be chosen fresh in the original term $ƛ\, b\, M$ —and that $M^* = (b^* \bullet b)\, M$. We know PM^* and therefore $P(ƛ\, b^*\, M^*)$ because we know P for any renaming of M, by the hypothesis of the strong principle from which we start.

A very important point in this implementation is that, given the list of names to be avoided, we can and do choose b^* deterministically for each class of α-equivalent terms. Indeed, if we determine b^* as e.g. the first name out of the given list that is fresh (i.e. not free) in the originally given term, then the result will be one and the same for every term of each α-class, since α-equivalent terms have the same free variables. Hence the representative of each α-class chosen by this method will be fixed for each list of names to be avoided, which constitutes a basis for using the method for defining *functions* on the α-classes. This will work by associating to (each term of) the class the result of the corresponding computation on the canonically chosen representative.

More precisely, let us say that a function $f : \Lambda \to A$ is *strongly* α-compatible iff $M \sim\alpha\ N \Rightarrow f\, M = f\, N$. We can now define an iteration principle over raw terms which always produces strongly α-compatible functions. For the abstraction case, this principle also allows us to give a list of variables from where the abstractions variables are not to be chosen. This iteration principle is derived from the BVC induction principle ($\mathsf{Term\alpha PrimInd}$) in a direct manner, just using a trivial constant predicate equivalent to the type A. We exhibit the type and code of the iterator:

```
ΛIt  : {l : Level}(A : Set l)
→ (Atom → A)
→ (A → A → A)
→ List Atom × (Atom → A → A)
→ Λ → A
```

To repeat the idea, the iterator works as a function on α-classes because for each given abstraction, it will yield the result obtained by working on a canonically chosen representative that is determined by the list of names to be avoided and the (free names of the) α-class in question.

Strong compatibility would not obtain if we tried directly to formulate a recursion instead of an iteration principle, but we can recover the more general form by the standard procedure of computing pairs one of whose components is a term. Thereby we arrive at the next recursion principle over terms, which also generates strong α-compatible functions.

```
ΛRec  : {l : Level}(A : Set l)
      → (Atom → A)
      → (A → A → Λ → Λ → A)
      → List Atom × (Atom → A → Λ → A)
      → Λ → A
```

4.4 Applications in Meta-Theory

We present several applications of the iteration/recursion principle defined in the preceding section. In the following two sub-sections we implement two classic examples of λ-calculus theory. In the section 4.6 we also apply our iteration/recursion principle to the examples of functions over terms presented in [48]. This work presents a sequence of increasing complexity functions, with the purpose of testing the applicability of recursion principles over λ-calculus terms.

Free Variables

We implement the function that returns the free variables of a term.

```
fv : Λ → List Atom
fv = ΛIt (List Atom) [_] _++_ ([] , ƛ v r → r - v)
```

As a direct consequence of strong α-compatibility of the iteration principle we have that α-equivalent terms have the same free variables.

The relation _*_ holds when a variable occurs free in a term.

```
data _*_ : Atom → Λ → Set where
   *v  :  {x : Atom}                                → x * v x
   *·l  :  {x : Atom}{M N : Λ}  → x * M            → x * (M · N)
   *·r  :  {x : Atom}{M N : Λ}  → x * N            → x * (M · N)
   *ƛ  :  {x y : Atom}{M : Λ}   → x * M → y ≢ x   → x * (ƛ y M)
```

We can use our BVC-like induction principle to prove the following proposition:

```
Pfv* : Atom → Λ → Set
Pfv* a M = a ∈ fv M → a * M
```

In the case of lambda abstractions we are able to simplify the proof by choosing the bound name different from a. This flexibility comes at a cost, i.e. we need to prove that the predicate Pfv*a is α-compatible in order to use the chosen induction principle. This α-compatibility proof is direct once we prove that * is an α-compatible relation and the fvfunction is strong α-compatible. The last property is direct because we implemented fvwith the iteration principle, so the extra cost is just the proof that * is α-compatible. This in turn could be directly obtained if we defined the relation establishing that a variable a is free in a term as a recursive function, as follows:

```
_free_ : Atom → Λ → Set
(_free_) a = ΛIt Set (λ b → a ≡ b) _⊎_ ([ a ] , λ _ → id)
```

For the variable case we return the propositional equality of the searched variable to the term variable. The application case is the disjoint union of the types returned by the recursive calls. Finally, in the abstraction case we can choose the abstraction variable to be different from the searched one. In this way we can ignore the abstraction variable and return just the recursive call containing the evidence of any free occurrence of the searched variable in the abstraction body. This implementation is strong compatible by construction because we have built it from our iterator principle, so it is also immediate from this definition that α-equivalent terms have the same free variables.

Substitution

We implement capture avoiding substitution in the following way:

```
hvar : Atom → Λ → Atom → Λ
hvar x N y with x ≟ₐ y
... | yes _ = N
... | no  _ = v y
-
_[_:=_] : Λ → Atom → Λ → Λ
M [ a := N ] = ΛIt Λ (hvar a N) _·_ (a :: fv N , ƛ) M
```

It shows to be quite close to the simple pencil-and-paper version assuming the BVC. Notice that we explicitly indicate that the bound name of the canonical representative to be chosen must be different from the replaced variable and not occur free in the substituted term. Again because of the strong α-compatibility of the iteration principle we obtain the following result for free:

```
lemmaSubst1 : {M N : Λ}(P : Λ)(a : Atom)
   → M ∼α N
   → M [ a := P ] ≡ N [ a := P ]
lemmaSubst1 {M} {N} P a
   = lemmaΛItStrongαCompatible
              Λ (hvar a P) _·_ (a :: fv P) ƛ M N
```

Using the induction principle in figure 4.2 we prove:

```
lemmaSubst2 : ∀ {N} {P} M x
  → N ∼α P → M [ x ≔ N ] ∼α M [ x ≔ P ]
```

From the two previous results we directly obtain the α-substitution lemma:

```
lemmaSubst : {M N P Q : Λ}(a : Atom)
  → M ∼α N → P ∼α Q
  → M [ a ≔ P ] ∼α N [ a ≔ Q ]
lemmaSubst {M} {N} {P} {Q} a M∼N P∼Q
  = begin
      M [ a ≔ P ]
    ≈⟨ lemmaSubst1 P a M∼N ⟩
      N [ a ≔ P ]
    ∼⟨ lemmaSubst2 N a P∼Q ⟩
      N [ a ≔ Q ]
    ■
```

In turn, with the preceding result we can derive that our substitution operation is α-equivalent with a naïve one for fresh enough bound names:

```
lemmaƛ∼[] : ∀ {a b P} M → b ∉ a :: fv P
  → ƛ b M [ a ≔ P ] ∼α ƛ b (M [ a ≔ P ])
```

We can combine this last result with the TermαPrimInd principle which emulates BVC convention, and mimic in this way pencil-and-paper inductive proofs over α-equivalence classes of terms about substitution operation. As an example we show next the substitution composition lemma:

```
PSC : ∀ {x y L} N → Λ → Set
PSC {x} {y} {L} N M = x ≢ y → x ∉ fv L
  → (M [ x ≔ N ]) [ y ≔ L ] ∼α (M [ y ≔ L ])[ x ≔ N [ y ≔ L ] ]
```

We first give a direct equational proof that PSC predicate is α-compatible:

```
αCompatiblePSC : ∀ {x y L} N → αCompatiblePred (PSC {x} {y} {L} N)
αCompatiblePSC {x} {y} {L} N {M} {P} M∼P PM x≢y x∉fvL
  = begin
      (P [ x ≔ N ]) [ y ≔ L ]
    - Strong α compatibility of inner substitution operation
    ≈⟨ cong (λ z → z [ y ≔ L ]) (lemmaSubst1 N x (σ M∼P)) ⟩
      (M [ x ≔ N ]) [ y ≔ L ]
    - We apply that we know the predicate holds for M
    ∼⟨ PM x≢y x∉fvL ⟩
      (M [ y ≔ L ]) [ x ≔ N [ y ≔ L ] ]
    - Strong α compatibility of inner substitution operation
    ≈⟨ cong (λ z → z [ x ≔ N [ y ≔ L ] ]) (lemmaSubst1 L y (M∼P)) ⟩
      (P [ y ≔ L ]) [ x ≔ N [ y ≔ L ] ]
    ■
```

For the interesting abstraction case of the α-structural induction over the lambda term, we assume the abstraction variables in the term are not among the replaced variables or free in the substituted terms. In this way the substitution operations become α-compatible to naïve substitutions, and the induction hypothesis allows us to complete the inductive proof in a direct maner. The code fragment becomes:

```
begin
  (ƛ b M [ x := N ])   [ y := L ]
- Inner substitution is α equivalent
- to a naive one because b ∉ x :: fv N
≈⟨ lemmaSubst1 L y (lemmaƛ∼[] M b∉x::fvN)  ⟩
  (ƛ b (M [ x := N ])) [ y := L ]
- Outer substitution is α equivalent
- to a naive one because b ∉ y :: fv L
∼⟨ lemmaƛ∼[] (M [ x := N ]) b∉y::fvL ⟩
  ƛ b ((M [ x := N ])  [ y := L ])
- We can now apply our inductive hypothesis
∼⟨ lemma∼αƛ (IndHip x≢y x∉fvL) ⟩
  ƛ b ((M [ y := L ])  [ x := N [ y := L ] ])
- Outer substitution is α equivalent
- to a naive one because b ∉ x :: fv N [y := L]
∼⟨ σ (lemmaƛ∼[] (M [ y := L ]) b∉x::fvN[y:=L]) ⟩
  (ƛ b (M [ y := L ])) [ x := N [ y := L ] ]
- Inner substitution is α equivalent
- to a naive one because b ∉ y :: fv L
≈⟨ sym (lemmaSubst1 (N [ y := L ])  x (lemmaƛ∼[] M b∉y::fvL))  ⟩
  (ƛ b M [ y := L ])   [ x := N [ y := L ] ]
■
```

Remarkably these results are directly derived from the first primitive induction principle, and no induction on the length of terms or accessible predicates were needed in all of this formalization.

4.5 Conclusions

The main contribution of this work is a full implementation in Constructive Type Theory of principles of induction and recursion allowing to work on α-classes of terms of the lambda calculus. The crucial component seems to be what we called a BVC-like induction principle allowing to choose the bound name in the case of the abstractions so that it does not belong to a given list of names. This principle is, on the one hand, derived (for α-compatible predicates) from ordinary structural induction on concrete terms, thus avoiding any form of induction on the size of terms or other more complex forms of induction. And, on the other hand, it gives rise to principles of recursion that allow to define functions on α-classes, specifically, functions giving identical results for α-equivalent terms. We have also shown by way of a number of examples that the principles provide a flexible framework quite able to pleasantly mimic pencil-and-paper practice.

Our work departs from e.g. [54] in that we do fix the choice of representatives for implementing the alpha-structural recursion thereby forcing this principle to yield identical results for α-equivalent terms. This might be a little too concrete but, on the other hand, it gives us the possibility of completing a simple full implementation on an existing system, as different from other works which base themselves on postulates or more sophisticated systems of syntax or methods of implementation.

We wish to continue exploring the capabilities of this method of formalisation by studying its application to the meta-theory of type systems. We also wish to deepen its comparison to the method based on Stoughton's substitutions [61], which we started to investigate in [64] and which we believe can give rise to formulations similar to the one exposed here.

4.6 Appendix: Iteration/Recursion Applications

In the following sections we successfully apply our iteration/recursion principle to all the examples from [48]. This work presents a sequence of functions whose definitions are increasing in complexity to provide a test for any principle of function definition, where each of the given functions respects the α-equivalence relation.

Case Analysis and Examining Constructor Arguments

The following family of functions distinguishes between constructors returning the constructor components, giving in a sense a kind of *pattern-matching*.

$$\begin{array}{lll}
isVar & : \Lambda \to & Maybe\ (Variable) \\
isVar & (v\ x) & = Just \\
isVar & (M \cdot N) & = Nothing \\
isVar & (\lambda x M) & = Nothing
\end{array}
\qquad
\begin{array}{lll}
isApp & : \Lambda \to & Maybe\ (\Lambda \times \Lambda) \\
isApp & (v\ x) & = Nothing \\
isApp & (M \cdot N) & = Just(M, N) \\
isApp & (\lambda x M) & = Nothing
\end{array}$$

$$\begin{array}{lll}
isAbs & : \Lambda \to & Maybe\ (Variable \times \Lambda) \\
isAbs & (v\ x) & = Nothing \\
isAbs & (M \cdot N) & = Nothing \\
isAbs & (\lambda x M) & = Just(x, M)
\end{array}$$

Next we present the corresponding encodings into our iteration/recursion principle:

```
isVar : Λ → Maybe Atom
isVar = ΛIt  (Maybe Atom)
             just
             (λ _ _ → nothing)
             ([] , λ _ _ → nothing)
-
isApp : Λ → Maybe (Λ × Λ)
isApp = ΛRec  (Maybe (Λ × Λ))
              (λ _ → nothing)
              (λ _ _ M N → just (M , N))
              ([] , λ _ _ _ → nothing)
-
isAbs : Λ → Maybe (Atom × Λ)
isAbs = ΛRec  (Maybe (Atom × Λ))
              (λ _ → nothing) (λ _ _ _ _ → nothing)
              ([] , λ a _ M → just (a , M))
```

Simple recursion

The size function returns a numeric measurement of the size of a term.

$$\begin{array}{lll} size & : \Lambda \rightarrow & \mathbb{N} \\ size & (v\ x) & = 1 \\ size & (M \cdot N) & = size(M) + size(N) + 1 \\ size & (\lambda x M) & = size(M) + 1 \end{array}$$

```
size : Λ → ℕ
size = ΛIt ℕ (const 1) (λ n m → suc n + m) ( [] , λ _ n → suc n)
```

Alpha Equality

This function decides the α-equality relation between two terms.

```
equal : Λ → Λ → Bool
equal = ΛIt (Λ → Bool) vareq appeq ([] , abseq)
  where
  vareq : Atom → Λ → Bool
  vareq a M with isVar M
  ... | nothing  = false
  ... | just b     = ⌊ a ≟ₐ b ⌋
  appeq : (Λ → Bool) → (Λ → Bool) → Λ → Bool
  appeq fM fN P with isApp P
  ... | nothing            = false
  ... | just (M' , N')  = fM M' ∧ fN N'
  abseq : Atom → (Λ → Bool) → Λ → Bool
  abseq a fM N with isAbs N
  ... | nothing = false
  ... | just (b , P) = ⌊ a ≟ₐ b ⌋ ∧ fM P
```

Observe that isAbs function also normalises N, so it is correct in the last line to ask if the two bound names are the same.

Recursion Mentioning a Bound Variable

The enf function is true of a term if it is in η-normal form. It invokes the fv function, which returns the set of a term's free variables and was previously defined.

$$\begin{array}{lll} enf & : \Lambda \rightarrow & Bool \\ enf & (v\ x) & = True \\ enf & (M \cdot N) & = enf(M) \wedge enf(N) + 1 \\ enf & (\lambda x M) & = enf(M) \wedge (\exists N, x / isApp(M) == Just(N, v\ x) \Rightarrow x \in fv(N)) \end{array}$$

```
_⇒_ : Bool → Bool → Bool
false  ⇒ b = true
true   ⇒ b = b
-
```

```
enf : Λ → Bool
enf = ΛRec Bool (const true) (λ b1 b2 _ _ → b1 ∧ b2) ([] , absenf)
  where
  absenf : Atom → Bool → Λ → Bool
  absenf a b M with isApp M
  ... | nothing = b
  ... | just (P , Q) = b ∧ (equal Q (v a) ⇒ a ∈b (fv P))
```

Recursion with an Additional Parameter

Given the ternary type of possible directions to follow when passing through a term (Lt, Rt, In), corresponding to the two sub-terms of an application constructor and the body of an abstraction, return the set of paths (lists of directions) to the occurrences of the given free variable in a term. Assume *cons* insert an element in front of a list.

$$\begin{array}{rlll}
 & vposns & : Variable \times \Lambda \to & List\ (List\ Direction) \\
 & vposns & (x, v\ y) & = if\ (x == y)\ then\ [[]]\ else\ [] \\
 & vposns & (x, M \cdot N) & = map\ (cons\ Lt)\ (vposns\ x\ M) +\!\!+ \\
 & & & \quad map\ (cons\ Rt)\ (vposns\ x\ N) \\
x \neq y \Rightarrow & vposns & (x, \lambda y M) & = map\ (cons\ In)\ (vposns\ x\ M)
\end{array}$$

Notice how the condition guard of the abstraction case is translated to the list of variables from where not to choose the abstraction variable.

```
data Direction : Set where
  Lt Rt In : Direction
-
vposns : Atom → Λ → List (List Direction)
vposns a = ΛIt (List (List Direction)) varvposns appvposns ([ a ] , absvposns)
  where
  varvposns : Atom → List (List Direction)
  varvposns b with a ≟ₐ b
  ... | yes _ = [ [] ]
  ... | no  _ = []
  appvposns : List (List Direction) → List (List Direction)
    → List (List Direction)
  appvposns l r = map (_::_ Lt) l ++ map (_::_ Rt) r
  absvposns : Atom → List (List Direction) → List (List Direction)
  absvposns a r = map (_::_ In) r
```

Recursion with Varying Parameters and Terms as Range

A variant of the substitution function, which substitutes a term for a variable, but further adjusts the term being substituted by wrapping it in one application of the variable named "0" per traversed binder.

$$\begin{array}{rlll}
& sub' & : \Lambda \times Variable \times \Lambda & \rightarrow \Lambda \\
& sub' & (P, x, v\ y) & = if\ (x == y)\ then\ P\ else\ (v\ y) \\
& sub' & (P, x, M \cdot N) & = (sub'(P, x, M)) \cdot (sub'(P, x, N)) \\
\left.\begin{array}{r} y \neq x \ \wedge \\ y \neq 0 \ \ \wedge \\ y \notin fv(P) \end{array}\right\} \Rightarrow & sub' & (P, x, \lambda y M) & = \lambda y (sub'((v\ 0) \cdot M, x, M))
\end{array}$$

To implement this function with our iterator principle we must change the order of the parameters, so our iterator principle now returns a function that is waiting for the term to be substituted. In this way we manage to vary the parameter through the iteration.

```
hvar : Atom → Atom → Λ → Λ
hvar x y with x ≟ₐ y
... | yes _ = id
... | no  _ = λ _ → (v y)
-
sub' : Atom → Λ → Λ → Λ
sub' x M P = ΛIt  (Λ → Λ)
                  (hvar x)
                  (λ f g N →  f N · g N)
                  (x :: 0 :: fv P , λ a f N → ƛ a (f ((v 0) · N)))
                  M P
```

CHAPTER 5

Alpha-Structural Induction and Recursion Part II - Church-Rosser and Subject Reduction

In this chapter we continue the work started in the previous chapter, formulating new induction principles for the λ-calculus with α-conversion based upon name swapping. We successfully apply those induction principles to obtain some fundamental meta-theoretical results, such as the Church-Rosser theorem and the Subject Reduction theorem for the simply typed λ-calculus à la Curry. The whole development has been machine-checked using the system Agda. Part of this work has been accepted for publication in the 12th Workshop on Logical and Semantic Frameworks with Applications (LSFA 2017).

5.1 Introduction

In the work presented in the preceding chapter we presented an α-induction principle (which we called TermαPrimInd) that enables us to carry out the proof of the lambda-abstraction case of the induction by choosing a bound name different from the names in a given list of names, thus emulating the BVC. Our induction principle requires the property being proved to be α-*compatible*, that is, it must be preserved by the α-conversion relation. From our induction principle we also directly derived an alpha-iteration principle. This principle has the property of defining *strong alpha compatible* functions, that is, the results of applying the function to alpha convertible terms are syntactically equal. In contrast to similar works, our iteration principle does not have any side conditions to be verified and allows us to perform computations, not just logical proofs.

However, the aforementioned α-induction principle fails when we want to use it to prove the Church-Rosser theorem, specifically in the proof of the crucial substitution lemma of the parallel reduction relation, which was presented as lemma 10 in chapter 3. When we try to prove this result by an induction on terms, the application subcase involves the substitution $((\lambda y.P)Q)[x := N]$. The usual informal proof requires y to be fresh in the term N, in order to push the substitution inside the abstraction without capturing any free occurrence of y in N. However, our α-induction principle gives no freshness premises in the application subcase. Because of this, a variable renaming would be required, bringing complexity to the proof, and departing from usual pen-and-paper developments. We will later explain in detail this situation in the proof of lemma 1.

In this chapter we present a strengthened α-induction principle on λ-terms that can be used to overcome this problem. In this way we can scale the metatheory of the λ-calculus up to the Church-Rosser theorem, formally resembling the BVC in the formalisation. Specifically, concerning β-reduction, we prove three *substitution lemmas* showing that substitution is compatible with α-conversion, parallel β-reduction, and typing in the simply typed λ-calculus à la Curry. To the best of our knowledge, this is the first detailed publication of a formalisation of the Church-Rosser and Subject Reduction theorems based upon name swapping, and using an α-induction principle on terms.

The work that most closely resembles this one is [65], where Urban and Norrish show how to emulate the BVC when performing induction on relations over λ-terms. They illustrate the use of the induction principles by proving the substitution lemma for the parallel reduction relation, and the weakening lemma of the typing relation. They present two induction principles on relations, one for the parallel reduction relation, and another one for the typing relation. When carrying out a proof by induction on these relations they are able to avoid some finite set of variable names as binders. To prove these strengthened induction principles they require that the relation definition rules satisfy the following preconditions: all functions and side conditions should be equivariant, the side conditions must imply that all bound variables do not occur free in the conclusions, and all bound variables must be distinct. If the previous conditions hold, they are able to derive a strengthened relation induction principle, where all binders can be chosen distinct from some given context. Hence, they have to modify the definition of the original relations to satisfy these preconditions in order to be able to prove the soundness of their induction principles.

All the definitions and proofs presented in this chapter have been fully formalised in Constructive Type Theory [37] and machine-checked employing the system Agda [46]. The corresponding code is public, and it is available at:

https://github.com/ernius/formalmetatheory-nominal-Church-Rosser

In the subsequent text we give the proofs in English with a considerable level of detail so that they serve for clarifying their formalisation.

In this section we will recall some definitions and results from the previous chapter that are necessary for a better understanding of the material presented in this one. In section 5.3 we introduce two new strengthened α-induction principles on λ-terms that will be useful to prove the main results of this chapter. Then, in section 5.4 we present the notion of β-reduction and prove the Church-Rosser theorem by using the standard method due to Tait and Martin-Löf which involves the formulation and study of the parallel β-reduction. In section 5.5 we present the Subject Reduction theorem of the simply typed λ-calculus.

5.2 Preliminaries

Variables belong to a denumerable set of names, and terms are inductively defined as usual:

$$M, N ::= x \mid MN \mid \lambda x.M$$

The *freshness* relation states that a variable does not occur free in a term:

$$\frac{x \neq y}{x\#y} \qquad \frac{x\#M \qquad x\#N}{x\#MN} \qquad \frac{x\#M}{x\#\lambda y.M} \qquad x\#\lambda x.M$$

$x \notin_b M$ denotes that the variable x does not occur in a binding position in the term M:

$$x \notin_b y \qquad \frac{x \notin_b M \qquad x \notin_b N}{x \notin_b MN} \qquad \frac{x \neq y \qquad x \notin_b M}{x \notin_b \lambda y.M}$$

Next comes the operation of *swapping* of names. A finite sequence (composition) of name swaps constitutes a finite name *permutation* which is the renaming mechanism to be used on terms. The action of swapping is first defined on names themselves:

$$(x\ y)\ z\ = \begin{cases} y & \text{if}\ \ z = x \\ x & \text{if}\ \ z = y \\ z & \text{if}\ \ x \neq z \wedge y \neq z, \end{cases}$$

The swapping operation is directly extended to terms by swapping all names occurring in a term, including abstraction positions. The *permutation* operation is just defined as the sequential application of a list of swaps. We usually use π to denote permutations, and the application of a permutation π to a term M is written πM. $(x\ y)\pi$ denotes the permutation consisting of the swap $(x\ y)$ followed by the permutation π.

In figure 5.1 we give a syntax-directed definition of α-conversion ($\sim_\alpha$) based on the swapping operation.

$$(\sim_\alpha\text{v})\ \frac{}{x \sim_\alpha x} \qquad (\sim_\alpha\text{a})\ \frac{M \sim_\alpha M' \qquad N \sim_\alpha N'}{MN \sim_\alpha M'N'}$$

$$(\sim_\alpha\lambda)\ \frac{\exists xs, \forall z \notin xs, (x\ z)M \sim_\alpha (y\ z)N}{\lambda x.M \sim_\alpha \lambda y.N}$$

Figure 5.1: Alpha equivalence relation

In the previous chapter we proved that this is an equivalence relation, preserved by the permutation operation (equivariant), and equivalent to the original definition.

We also derived induction and iteration principles with strengthened hypotheses for the lambda abstraction case, namely enabling us to choose a bound name different from the names in a given list xs (TermαPrimInd). One important point in our implementation is that functions defined using this iteration principle yield the same result for α-equivalent terms. We called these functions *strongly* α-compatible.

The substitution operation is defined using this iteration principle, and as a direct consequence of this, the following lemma is automatically derived.

Lemma 1 (α-compatibility of substitution)**.**

$$M \sim_\alpha M' \Rightarrow M[x{:=}N] = M'[x{:=}N]$$

The following results are successfully proved using our induction principles.

Lemma 2 (Substitution preserves α-conversion).

$$N \sim_\alpha N' \Rightarrow M[x{:=}N] \sim_\alpha M[x{:=}N']$$

Lemma 3 (Substitution under permutation).

$$\pi\ (M[x{:=}N]) \sim_\alpha (\pi\ M)[(\pi\ x){:=}(\pi\ N)]$$

The next lemma shows that substitution commutes with abstraction up to α-conversion. This is because hypotheses ensure a fresh enough binder.

Lemma 4 (Substitution commutes with abstraction).

$$x \neq y \wedge x\#N \ \Rightarrow\ (\lambda x.M)[y{:=}N] \sim_\alpha \lambda x.(M[y{:=}N])$$

Lemma 5 (Substitution composition).

$$x \neq y \wedge x\#P \ \Rightarrow\ M[x{:=}N][y{:=}P] \sim_\alpha M[y{:=}P][x{:=}N[y{:=}P]]$$

The following results were not proved in the preceding chapter, so we show their proofs in detail.

Lemma 6 (Swapping substitution variable).

$$x\#M \Rightarrow ((x\ y)M)[x{:=}N] \sim_\alpha M[y{:=}N]$$

Proof. We use our α-induction principle.
First, for arbitrary names x, y and term N we define the following predicate over terms:

$$\Pi(M) \equiv x\#M \Rightarrow (x\ y)M[x{:=}N] \sim_\alpha M[y{:=}N]$$

We prove Π is α-compatible, that is, if $M{\sim_\alpha}P$ and $\Pi(M)$, then $\Pi(P)$. Assume $\Pi(M)$ and $x\#P$, then as freshness is preserved through $\sim_\alpha$, we have that $x\#M$. Then we proceed as follows:

$$\begin{array}{ccl}
((x\ y)P)[x{:=}N] & = & \{\sim_\alpha \text{ equivariance and lemma 1}\} \\
((x\ y)M)[x{:=}N] & \sim_\alpha & \{\Pi(M) \text{ and } x\#M\} \\
M[y{:=}N] & = & \{\text{lemma 1}\} \\
P[y{:=}N] & &
\end{array}$$

Next, we show the proof of the interesting abstraction case of the induction.

We have $x\#\lambda z.M$ and we can choose $z \notin \{x, y\} \cup \mathrm{fv}(N)$.

We need to prove $((x\ y)(\lambda zM))[x{:=}N] \sim_\alpha (\lambda zM)[y{:=}N]$. As $x\#\lambda z.M$ and $z \neq x$ we get $x\#M$, then we can reason as follows:

$$\begin{array}{ccl}
((x\ y)(\lambda z.M))[x{:=}N] & = & \{\text{def. of swap}\} \\
(\lambda((x\ y)z).((x\ y)M))[x{:=}N] & = & \{z \notin \{x, y\}\} \\
(\lambda z.((x\ y)M))[x{:=}N] & \sim_\alpha & \{\text{lemma 4}\} \\
\lambda z.(((x\ y)M)[x{:=}N]) & \sim_\alpha & \{\text{i.h.}\} \\
\lambda z.(M[y{:=}N]) & \sim_\alpha & \{\text{lemma 4}\} \\
(\lambda z.M)[y{:=}N] & &
\end{array}$$

□

The preceding proof illustrates the usual pen-and-paper informal practice, which basically uses the BVC to assume binders fresh enough in some defined context, allowing us to apply substitution in a naive way without need of renaming.

The next result is a direct consequence of the previous lemma.

Lemma 7.

$$x\#\lambda y.M \Rightarrow ((x\ y)M)[x{:=}N] \sim_\alpha M[y{:=}N]$$

5.3 Alpha Induction Principles

In this section we introduce two new α-induction principles. The first one is presented in figure 5.2. It is a strengthened version of the one defined in the preceding chapter, where the induction hypothesis of the abstraction allows to assume the property for all permutations of the body. This principle is useful to deal with relations which make use of the permutation operation in their definitions. We will show an example of this situation in the proof of lemma 13 in next section.

$$\frac{\begin{array}{l} P\ \alpha\text{-compatible} \\ (\forall x)\ P(x) \\ (\forall M,N)\ (P(M) \wedge P(N) \Rightarrow P(MN)) \\ (\exists xs, \forall M, \forall x {\notin} xs)\ ((\forall \pi)\ P(\pi\ M) \Rightarrow P(\lambda x.M)) \end{array}}{(\forall M)\ P(M)}$$

Figure 5.2: Alpha induction principle with permutations

Proof. The α-permutation induction principle shown in figure 5.2 is proved using the one in figure 5.3 (which was presented in figure 2 of the previous chapter).

$$\frac{\begin{array}{l} (\forall x)\ P(x) \\ (\forall M,N)\ (P(M) \wedge P(N) \Rightarrow P(MN)) \\ (\forall M,x)\ ((\forall \pi)\ P(\pi\ M)\ \Rightarrow P(\lambda x.M)) \end{array}}{(\forall M)\ P(M)}$$

Figure 5.3: Strong permutation induction principle

The variable and application cases are direct. For the abstraction case, given any term M and variable x, we must prove $P(\lambda x.M)$ knowing:

$$(\forall \pi)\ P(\pi\ M) \tag{5.1a}$$

$$(\exists xs, \forall M', \forall y {\notin} xs)\ ((\forall \pi')\ P(\pi'\ M') \Rightarrow P(\lambda y.M')) \tag{5.1b}$$

Let xs be a list of names as in (5.1b). Let us further pick y not in xs and also fresh in $\lambda x.M$. Then for all M', π', $P(\pi'\ M') \Rightarrow P(\lambda y.M')$ holds. So taking $M' = (x\ y)M$ we have that $(\forall \pi')P(\pi'\ ((x\ y)M)) \Rightarrow P(\lambda y.(x\ y)M)$. Now, as $\pi'((x\ y)M) = ((x\ y)\pi')M$, we can use (5.1a) to get $P(\lambda y.(x\ y)M)$ from (5.1b), and finally $P(\lambda x.M)$ because P is α-compatible and $\lambda x.M \sim_\alpha \lambda y.(x\ y)M$. This last α-equivalence holds because we have chosen y fresh in $\lambda x.M$. □

The next induction principle (figure 5.4) enables us to assume bound variables not in a given finite list of names xs through the entire induction, and not only for the abstraction case.

$$\frac{\begin{array}{l} P\ \alpha\text{-compatible} \\ (\forall x)\ P(x) \\ (\forall M, N)\ ((\forall y \in xs, y \notin_b MN) \wedge P(M) \wedge P(N) \Rightarrow P(MN)) \\ (\forall M, x)\ ((\forall y \in xs, y \notin_b \lambda x.M) \wedge P(M) \Rightarrow P(\lambda x.M)) \end{array}}{(\forall M)\ P(M)}$$

Figure 5.4: Strengthened α-induction principle

To derive this principle we introduce the *rewrite* function that, given a list of names and a term, returns an α-convertible term that does not contain any element of the given list as binder. To prove $P(M)$ for any term M, we proceed as follows: given a list of names xs, an alpha compatible predicate P, and the following proofs:

$$\begin{array}{l} (\forall x)\ P(x) \\ (\forall M, N)\ ((\forall y \in xs, y \notin_b MN) \wedge P(M) \wedge P(N) \Rightarrow P(MN)) \\ (\forall M, x)\ ((\forall y \in xs, y \notin_b \lambda x.M) \wedge P(M) \Rightarrow P(\lambda x.M)) \end{array} \tag{5.2}$$

we prove the following predicate Π by primitive induction on terms:

$$\Pi(M) = ((\forall x \in xs)\ \Rightarrow x \notin_b M) \Rightarrow P(M) \tag{5.3}$$

Then, we use this predicate Π on the term *rewrite*(xs, M), which has no bound variables in xs to obtain $P($*rewrite*$(xs, M))$. Finally, as P is α-compatible and *rewrite*$(xs, M) \sim_\alpha M$ we get that $P(M)$ holds for any M.

In turn, the proof of $\Pi(M)$ by structural induction on M is straightforward because of the syntax directed definition of $\notin_b$:

Proof.

- Variable case: Direct.
- Application case: We need to prove $\Pi(MN)$ for any M, N, such that $\Pi(M)$ and $\Pi(N)$ hold. That is, we have to prove $P(MN)$, given that any variable x in xs satisfies that $x \notin_b MN$. Then, by the syntax directed definition of $\notin_b$, we directly have that $x \notin_b M$ and $x \notin_b N$, and so we are able to use the induction hypothesis on M and N to get $P(M)$ and $P(N)$. So, we have all the premises in the second assertion in (5.2) hold, and hence its conclusion $P(MN)$.
- Abstraction case: We must prove $\Pi(\lambda y.M)$, that is, we need to prove $P(\lambda y.M)$ knowing that every variable x in xs satisfies that $x \notin_b \lambda y.M$. By the definition of $\notin_b$, we have that $x \neq y$ and $x \notin_b M$. We can apply the last result to the inductive hypothesis $\Pi(M)$ to get $P(M)$. Finally, we get the desired result using the third assertion in (5.2).

□

5.4 Parallel Beta Reduction

The β-reduction relation ($\rightarrow_\beta$) can be defined as the reflexive, transitive and compatible with the syntactic constructors, closure of the β-contraction $(\lambda x.M)N \rhd_\beta M[x{:=}N]$. The classical proof of confluence of β-reduction by Tait and Martin-Löf rests upon the property of confluence of the so-called parallel reduction, which can apply several β-contractions "in parallel" in one single step. We present our definition in figure 5.5.

$$(\rightrightarrows\text{v})\ \frac{}{x \rightrightarrows x} \qquad\qquad (\rightrightarrows\text{a})\ \frac{M \rightrightarrows M' \qquad N \rightrightarrows N'}{MN \rightrightarrows M'N'}$$

$$(\rightrightarrows\lambda)\ \frac{\exists xs, \forall z \notin xs, (x\ z)M \rightrightarrows (y\ z)N}{\lambda x.M \rightrightarrows \lambda y.N}$$

$$(\rightrightarrows\beta)\ \frac{\lambda x.M \rightrightarrows \lambda y.P' \quad N \rightrightarrows P'' \quad P'[y{:=}P''] \sim_\alpha P}{(\lambda x.M)N \rightrightarrows P}$$

Figure 5.5: Parallel reduction relation

The first three rules have the same form as the ones defining the α-conversion relation presented in fig. 5.1, which evidences that we want this parallel reduction to be a congruence over the α-conversion, that is, if $M \rightrightarrows N$, $M \sim_\alpha M'$ and $N \sim_\alpha N'$ then $M' \rightrightarrows N'$ holds. We will prove this property in lemmas 11 and 12. Finally, note that the β-rule has an extra premise involving α-conversion. This is because our substitution operation modifies the bound names in terms as a consequence of being defined with our alpha iteration principle. Without this premise we would not be able to prove that $\rightrightarrows$ is α-compatible on its right hand side.

Note also that we have as base case $x \rightrightarrows x$, instead of the usual $M \rightrightarrows M$ that appears in, for instance, Barendregt's book [4]. This alternative base case rule is more convenient to prove results about this relation by induction on terms. The next lemma shows that this change does not modify the defined relation.

Lemma 8 (Reflexivity of $\rightrightarrows$).

Proof. Direct application of the permutation induction principle shown in figure 5.3. □

Next we prove that $\rightrightarrows$ is equivariant. This is a basic property in a nominal setting, establishing that names are interchangeable.

Lemma 9 (Equivariance of $\rightrightarrows$).

$$M \rightrightarrows N \ \Rightarrow\ \pi M \rightrightarrows \pi N$$

Proof. By induction on the definition of $\rightrightarrows$. The variable and application cases are direct.

In the abstraction case, we have to prove $\lambda(\pi\ x).(\pi\ M) \rightrightarrows \lambda(\pi\ y).(\pi\ N)$ with $\lambda x.M \rightrightarrows \lambda y.N$ as hypothesis. The only rule with an abstraction as an outermost constructor in its conclusions is $(\rightrightarrows\lambda)$, so we can assume that its premise holds for any z not in some list of variables xs. We can also exclude the domain of the permutation π from the range of variable z, and reason as follows:

$$\begin{array}{ll}
(x\ z)M \rightrightarrows (y\ z)N & \Rightarrow \ \{\text{i.h.}\} \\
\pi((x\ z)M) \rightrightarrows \pi((y\ z)N) & \Rightarrow \ \{\text{def. of perm.}\} \\
((\pi\, x)\ (\pi\, z))(\pi\, M) \rightrightarrows ((\pi\, y)\ (\pi\, z))(\pi\, N) & \Rightarrow \ \{\text{as } z \notin \text{dom}(\pi) \text{ then } (\pi\, z) = z\} \\
((\pi\ x)\ z)(\pi\ M) \rightrightarrows ((\pi\ y)\ z)(\pi\ N) & \Rightarrow \ \{(\rightrightarrows \lambda)\ \text{ rule}\} \\
\lambda(\pi\ x).(\pi\ M) \rightrightarrows \lambda(\pi\ y).(\pi\ N). &
\end{array}$$

In the $(\rightrightarrows\beta)$ case we must prove:

$$(\lambda(\pi\ x).(\pi\ M))(\pi\ N) \rightrightarrows \pi\ P$$

The hypotheses are: $\lambda x.M \rightrightarrows \lambda y.P'$, $N \rightrightarrows P''$ and $P \sim_\alpha P'[y{:=}P'']$. By direct application of the induction hypothesis to the first two premises we get:

$$\begin{array}{r}
\lambda(\pi\ x).(\pi\ M) \rightrightarrows \lambda(\pi\ y).(\pi\ P') \\
\text{and } \pi\ N \rightrightarrows \pi\ P''
\end{array} \tag{5.4}$$

Then, using the third premise we can reason as follows:

$$\begin{array}{ll}
P \sim_\alpha P'[y{:=}P''] & \Rightarrow \ \{\sim_\alpha \text{equivariance}\} \\
\pi\ P \sim_\alpha \pi\ (P'[y{:=}P'']) & \Rightarrow \ \{\text{lemma 3}\} \\
\pi\ P \sim_\alpha (\pi\ P')[(\pi\ y){:=}(\pi\ P'')] & \Rightarrow \ \{\sim_\alpha \text{symmetric}\} \\
(\pi\ P')[(\pi\ y){:=}(\pi\ P'')] \sim_\alpha \pi\ P &
\end{array}$$

We obtain the desired result using the $(\rightrightarrows\beta)$ rule with (5.4) and the last result as premises. □

As a direct consequence of the previous lemma we derive the following result:

Corollary 1 (Preservation of $\rightrightarrows$ under abstraction)**.**

$$M \rightrightarrows N \Rightarrow \lambda x.M \rightrightarrows \lambda x.N$$

The following lemmas state that our parallel reduction relation is preserved by α-equivalence, both results are proved by easy inductions on the parallel reduction relation.

Lemma 10 (Right α-compatibility of $\rightrightarrows$)**.**

$$M \rightrightarrows N \wedge N \sim_\alpha P \Rightarrow M \rightrightarrows P$$

Lemma 11 (Left α-compatibility of $\rightrightarrows$)**.**

$$M \sim_\alpha N \wedge N \rightrightarrows P \Rightarrow M \rightrightarrows P$$

As $\rightrightarrows$ is reflexive, we can now prove in a direct manner that α-conversion is included in the parallel reduction.

Lemma 12 ($\sim_\alpha \subseteq \rightrightarrows$).

Proof. Given $M \sim_\alpha N$, as $\rightrightarrows$ is reflexive by lemma 8, we also know $M \rightrightarrows M$. Then using lemma 10 we obtain the desired result. □

As $\rightrightarrows$ basically applies β-contractions, no free variable should be introduced at any relation step, therefore freshness is preserved.

Lemma 13 ($\rightrightarrows$ preserves freshness).

$$x\#M \wedge M \rightrightarrows N \Rightarrow x\#N$$

Proof. We use the α-induction principle with permutations (fig. 5.2) on the term M. In order to apply this principle we must prove, for any variable x, that the following predicate is α-compatible.

$$\Pi(M) = (\forall N)(x\#M \wedge M \rightrightarrows N \Rightarrow x\#N)$$

As the freshness and parallel reduction relations are α-compatible, Π is also α-compatible. Now, for the main result we proceed by induction on M, only showing the interesting abstraction case. We have that $x\#\lambda y.M$ and $\lambda y.M \rightrightarrows \lambda z.N$, and we must prove $x\#\lambda z.N$. The parallel reduction hypothesis, by its syntax directed definition, must be the result of an application of the ($\rightrightarrows\lambda$) rule, so we get its premise $\forall w, w \notin xs, (y\ w)M \rightrightarrows (z\ w)N$. The α-induction principle allows us to exclude some variables for the abstraction case, so we can also assume $x \neq y$. Using this inequality and the hypothesis $x\#\lambda y.M$ we get by definition that $x\#M$ holds. Now let u be a variable such that $u\#N, u \notin xs$ and $u \neq x$, then $x\#(y\ u)M$ because $x \neq y, u$ and $x\#M$. We can apply the premise of the ($\rightrightarrows\lambda$) rule with u, as $u \notin xs$, and we get $(y\ u)M \rightrightarrows (z\ u)N$. We use the induction hypothesis on M and permutation $(y\ u)$ with the previous two results to get $x\ \#(z\ u)N$. We also have that $\lambda u.(z\ u)N \sim_\alpha \lambda z.N$ because $u\ \#N$. Then as $\sim_\alpha$ preserves freshness, we get the desired result. □

We can now prove the following inversion lemmas, which state that the original definition of parallel reduction by Takahashi [62] (which we note $\rightrightarrows_T$ in the next definition) can be derived from ours. These lemmas will be useful in the proof of the diamond property of $\rightrightarrows$.

$$\frac{}{x \rightrightarrows_T x} \qquad \frac{M \rightrightarrows_T M' \qquad N \rightrightarrows_T N'}{MN \rightrightarrows_T M'N'} \qquad \frac{M \rightrightarrows_T M'}{\lambda x.M \rightrightarrows_T \lambda x.M'}$$

$$\frac{M \rightrightarrows_T M' \qquad N \rightrightarrows_T N'}{(\lambda x.M)N \rightrightarrows_T M'[x{:=}N']}$$

Figure 5.6: Takahashi's parallel reduction relation.

Lemma 14 ($\rightrightarrows$ λ-inversion).

$$\lambda x.M \rightrightarrows M' \Rightarrow (\exists M'')(M \rightrightarrows M'' \ \wedge\ M' \sim_\alpha \lambda x.M'' \ \wedge \lambda x.M \rightrightarrows \lambda x.M'')$$

Proof. By definition of $\rightrightarrows$ it must be the case that $\lambda x.M \rightrightarrows M'$ is a result of an application of $(\rightrightarrows \lambda)$ rule, then we have that M' is in an abstraction $\lambda y.N$, and there exists a list of variables xs such that $\forall z \notin xs, (x\ z)M \rightrightarrows (y\ z)N$. We take $M'' = (x\ y)N$, and prove that M'' satisfies the three properties of the thesis.

- Let z be a variable such that $z \notin xs$ and $z \# \lambda y.M'$. By definition of #, $x \# \lambda x.M$, and then, as parallel reduction preserves freshness, $x \# \lambda y.N$ should also hold. So:

$$\begin{array}{lll} (x\ z)M \rightrightarrows (y\ z)N & \Rightarrow & \{\rightrightarrows \text{equivariance}\} \\ (x\ z)(x\ z)M \rightrightarrows (x\ z)(y\ z)N & \Rightarrow & \{\text{swap self inverse}\} \\ M \rightrightarrows (x\ z)(y\ z)N & \Rightarrow & \{(*)\} \\ M \rightrightarrows (x\ y)N & & \end{array}$$

  (*) In the last step of the previous deduction we applied lemma 10 with the premise that $(x\ z)(y\ z)N \sim_\alpha (x\ y)N$. This swapping cancellation property requires z and x to be fresh enough, as it is in this case.

- To prove $\lambda y.N \sim_\alpha \lambda x.(x\ y)N$, as x is fresh in $\lambda y.N$, we can swap y with x in this term to get the alpha equivalent term $\lambda x.(x\ y)N$.

- We can apply lemma 10 with $\lambda x.M \rightrightarrows \lambda y.N$ and the α-equivalence obtained above to prove $\lambda x.M \rightrightarrows \lambda x.(x\ y)N$.

□

Lemma 15 ($\rightrightarrows$ β-inversion)**.**

If $(\lambda x M)N \rightrightarrows P$ *is obtained by application of the* $(\rightrightarrows \beta)$ *rule in the following way:*

$$(\rightrightarrows\beta)\ \frac{\lambda x.M \rightrightarrows \lambda y.M' \quad N \rightrightarrows N' \quad M'[y{:=}N'] \sim_\alpha P}{(\lambda x.M)N \rightrightarrows P}$$

then

$$(\exists M'')\,(\lambda x.M \rightrightarrows \lambda x M'' \wedge\ M''[x{:=}N'] \sim_\alpha P)$$

Proof. We prove $M'' = (y\ x)M'$ satisfies the thesis.

- $x \# \lambda x.M$ and $\lambda x M \rightrightarrows \lambda y M'$ so by lemma 13 $x \# \lambda y M'$. We can then swap y with x in the last term and obtain the alpha equivalent term $\lambda x.(y\ x)M'$, using lemma 9. Then, by lemma 10 we get $\lambda x.M \rightrightarrows \lambda x.(y\ x)M'$.

- For the second condition we reason as follows:

$$\begin{array}{lll} ((y\ x)M')[x{:=}N'] & = & \{\text{swap commutativity}\} \\ ((x\ y)M'))[x{:=}N'] & \sim_\alpha & \{\text{corollary 7}\} \\ M'[y{:=}N'] & \sim_\alpha & \{\text{hypothesis}\} \\ P & & \end{array}$$

□

Theorem 1 ($\rightrightarrows$ substitution lemma)**.**

$$M \rightrightarrows M' \wedge N \rightrightarrows N' \Rightarrow M[x{:=}N] \rightrightarrows M'[x{:=}N']$$

The substitution lemma for parallel reduction is the crux of the Church-Rosser theorem, and the place in which our original α-induction principles fail in capturing the BVC. If we perform induction on the term M, the problem appears in the beta application case. In this case $M = (\lambda y.P)Q$, hence we need to prove $((\lambda y.P)Q)[x{:=}N] \rightrightarrows R[x{:=}N']$. But as we are in the application case of the induction, the original α-induction principle gives no freshness information about the binder y. The use of the BVC would allow us to choose y different from x and fresh in N, and with those freshness conditions we could push the substitution inside the abstraction without any variable capture by the use of lemma 4.

Therefore, we next use our strengthened α-induction principle presented in figure 5.4 to prove this result.

Proof. Given terms N, N' such that $N \rightrightarrows N'$, and a variable x, we define the following predicate over terms:

$$\Pi(M) \equiv (\forall M')(M \rightrightarrows M' \Rightarrow M[x{:=}N] \rightrightarrows M'[x{:=}N'])$$

We first prove that Π is α-compatible, that is: $\Pi(M) \wedge M{\sim_\alpha}N \Rightarrow \Pi(N)$. This is a direct consequence of both substitution and $\rightrightarrows$ being α-compatible (lemmas 1,10,11).
Then we can use our strengthened α-induction principle to prove Π by induction on the term M, excluding the variable x, and the free variables in terms N and N' from the binders in M.

We show the proof of the interesting application and abstraction cases.

- Application case: we have to prove $(\forall P, Q)$ $((\forall z \in \{x\}{\cup}\mathrm{fv}(N){\cup}\mathrm{fv}(N'),\ z \notin_b P\ Q)\ \Pi(P) \wedge \Pi(Q) \Rightarrow \Pi(P\ Q))$. We have two subcases according to which rule of the parallel reduction is used to obtain the parallel reduction in the conclusion of $\Pi(P\ Q)$.
  - ($\rightrightarrows$a) rule subcase: we have that $P \rightrightarrows P'$ and $Q \rightrightarrows Q'$ and we need to prove that $(P\ Q)[x{:=}N] \rightrightarrows (P'\,Q')[x{:=}N']$. The proof is a direct application of the ($\rightrightarrows a$) rule to the induction hypotheses.
  - ($\rightrightarrows\beta$) rule subcase: given $(\lambda y.P)Q \rightrightarrows R$ we must prove:

    $$((\lambda y.P)Q)[x{:=}N] \rightrightarrows R[x{:=}N']$$

    We use the inversion lemma 15 to obtain that there exists P'' such that $\lambda y.P \rightrightarrows \lambda y.P'' \wedge P''[y{:=}Q'] \sim_\alpha R$. Next, as we have assumed the binder y different from x and also fresh in N and N', we can reason as follows:

    $$\begin{array}{ll} \lambda y.P \rightrightarrows \lambda y.P'' & \Rightarrow\ \{\text{i.h.}\} \\ (\lambda y.P)[x{:=}N] \rightrightarrows (\lambda y.P'')[x{:=}N'] & \Rightarrow\ \{\text{lemma } 4\} \\ \lambda y.(P[x{:=}N]) \rightrightarrows \lambda y.(P''[x{:=}N']) & \end{array}$$

    By the induction hypothesis we also know $Q[x{:=}N] \rightrightarrows Q'[x{:=}N']$. So if we prove:

    $$P''[x{:=}N'][y{:=}Q'[x{:=}N']] \sim_\alpha R[x{:=}N'] \tag{5.5}$$

    we will be able to apply the ($\rightrightarrows\ \beta$) rule and get that:

    $$(\lambda y.(P[x{:=}N]))(Q[x{:=}N]) \rightrightarrows R[x{:=}N']$$

    Then, using the freshness premises, we can pull out the substitution operation on the left side of this parallel reduction, and using the lemma 11, of α-compatibility of $\rightrightarrows$, we finally get the desired result.

It just remains to prove (5.5) to end the proof of this subcase. Again, here the classical informal proofs use the BVC. We can also mimic this practice in this case since our induction principle gives us a binder y distinct form x and fresh in N'. Then, we have the freshness premises to successfully apply the substitution composition lemma 5 and conclude this proof in the following steps:

$$\begin{array}{ll}
P''[x{:=}N'][y{:=}Q'[x{:=}N']] & \sim_\alpha \{\text{lemma 5}\} \\
P''[y{:=}Q'][x{:=}N'] & = \ \{\text{lemma 1 and } P''[y{:=}Q'] \sim_\alpha R\ \} \\
R[x{:=}N'] &
\end{array}$$

- Abstraction case: we must prove $(\forall P, y)\ (\forall z \in \{x\} \cup \text{fv}(N) \cup \text{fv}(N'), z \notin_b \lambda y.P) \wedge \Pi(P) \Rightarrow \Pi(\lambda y.P)$. We apply the inversion lemma 15 to the hypothesis $\lambda y.P \rightrightarrows Q$ to get that there exists Q' such that: $P \rightrightarrows Q'$, $\lambda y.P \rightrightarrows \lambda y.Q'$ and $Q \sim_\alpha \lambda y.Q'$. Then, we can conclude the proof in the following way:

$$\begin{array}{ll}
P \rightrightarrows Q' & \Rightarrow \{\text{ind. hyp.}\} \\
P[x{:=}N] \rightrightarrows Q'[x{:=}N'] & \Rightarrow \{\rightrightarrows \text{equivariance}\} \\
(y\ z)(P[x{:=}N]) \rightrightarrows (y\ z)(Q'[x{:=}N']) & \Rightarrow \{(\rightrightarrows\lambda)\ \text{rule}\} \\
\lambda y.P[x{:=}N] \rightrightarrows \lambda y.Q'[x{:=}N'] & \Rightarrow \{\text{lemma 4}\} \\
(\lambda y.P)[x{:=}N] \rightrightarrows (\lambda y.Q')[x{:=}N'] & \Rightarrow \{\text{lemma 1 and } Q \sim_\alpha \lambda y.Q'\} \\
(\lambda y.P)[x{:=}N] \rightrightarrows Q[x{:=}N'] &
\end{array}$$

□

In [65], the authors proceed by induction on the relation, so x, N, N' are universally quantified over the β-contraction rule definition, and they are forced to add the same freshness premises that we were able to assume in this proof –by the use of our strengthened α-induction principle– directly into the premises of their modified beta rule of the parallel relation. In contrast, we are performing induction on the term M to prove the predicate Π, and hence we are able to leave those variables as a fixed context outside the definition of Π. Then by the use of our strengthened α-induction principle we are able to mimic the BVC also in the application case of the proof, specifically in the previously exposed $(\rightrightarrows\beta)$ rule subcase.

Finally, we can prove the diamond property of the parallel reduction. Instead of directly proving it by induction on terms (which can easily be done), we will follow the shorter method by Takahashi [62]. For this we first define the "star" operation (figure 5.7), such that for any λ-term M, M^* is the result of contracting all the β-redexes existing in M simultaneously. Then we prove that for any terms M, N, if $M \rightrightarrows N$, then $N \rightrightarrows M^*$ (lemma 16). Finally, the diamond property of $\rightrightarrows$ follows directly as a corollary of this result.

$$\begin{array}{llcl}
x^* & & = & x \\
(\lambda x.M)^* & & = & \lambda x.M^* \\
(x & M)* & = & xM^* \\
((M_1 M_2) & M_3)^* & = & (M_1 M_2)^* M_3^* \\
((\lambda x.M_1) & M_2)^* & = & M_1^*[x := M_2^*]
\end{array}$$

Figure 5.7: Takahashi's star function

Lemma 16 (Star property).

$$M \rightrightarrows N \Rightarrow N \rightrightarrows M^*$$

Proof. By induction on M. We prove the interesting application and abstraction cases.

- Abstraction case: we have to prove that $N \rightrightarrows (\lambda x.M)^* = \lambda x.M^*$, knowing that $\lambda x.M \rightrightarrows N$ holds. We can use the inversion lemma 14 on the latter to obtain the existence of the term N' such that: $N \sim_\alpha \lambda x.N'$ and $M \rightrightarrows N'$. We can now apply the inductive hypothesis, and then corollary 1 to $M \rightrightarrows N'$, and obtain $\lambda x.N' \rightrightarrows \lambda x.(M^*)$. This last result directly gives us the desired result by lemma 11 since we know that $N \sim_\alpha \lambda x.N'$.
- Application case: we have three subcases. The first two are when the term is not a redex (third and fourth lines of star operation) and are directly derived from the inductive hypotheses.
  Finally, the redex case can be subdivided accordingly to which rule, ($\rightrightarrows$a) or ($\rightrightarrows\beta$), is used in the last step of its parallel reduction hypotheses.
  - ($\rightrightarrows$a) subcase: we have that $\lambda x.M \rightrightarrows N$ and $M' \rightrightarrows N'$, and we need to prove that $NN' \rightrightarrows ((\lambda x.M)M')^* = M^*[x := M'^*]$. We begin applying the inversion lem. 14 to the hypothesis $\lambda x.M \rightrightarrows N$ to get that there exists N'' such that $N \sim_\alpha \lambda x.N''$ and $M \rightrightarrows N''$. We can now apply the inductive hypothesis to the latter, and then the corollary 1 to conclude $\lambda x.N'' \rightrightarrows \lambda x.M^*$. Besides, we can also apply the induction hypothesis to the premise $M' \rightrightarrows N'$ to get $N' \rightrightarrows M'^*$. We can combine the last two infered parallel reductions, using the ($\rightrightarrows\beta$) rule, and derive that $(\lambda x.N'')N' \rightrightarrows M^*[x := M'^*]$ holds. From this result we directly get the desired result just noticing that $N\,N' \sim_\alpha (\lambda x.N'')N'$, because $N \sim_\alpha \lambda x.N''$ and $N' \sim_\alpha N'$. Hence, by left α-compatibility of the parallel relation (lemma 11) we finish this subproof case.
  - $\rightrightarrows\beta$ subcase: we have the following hypotheses: $\lambda xM \rightrightarrows \lambda yN$, $M' \rightrightarrows N'$ and $N[y := N'] \sim_\alpha P$, and we need to prove that $P \rightrightarrows M^*[x := M'^*]$ holds. We proceed analogously to the previous subcase and derive that there exists N'' such that $\lambda y.N \sim_\alpha \lambda x.N''$, $N'' \rightrightarrows M^*$ and $N' \rightrightarrows M'^*$. Then we apply the substitution lemma for $\rightrightarrows$ (lemma 1) to obtain $N''[x := N'] \rightrightarrows M^*[x := M'^*]$. Finally, we can use left α-compatibility lemma 11 to finish the proof if we prove that $P \sim_\alpha N''[x := N']$. Next we prove that this last alpha equivalence holds:

    $$\begin{array}{ll} P & \sim_\alpha \{\text{hypothesis}\} \\ N[y := N'] & \sim_\alpha \{\text{by lemma 7 as } x\#\lambda y.N\} \\ ((x\ y)N)[x := N'] & = \ \ \{\text{by lemma 1 as } (x\ y)N \sim_\alpha N''\} \\ N''[x := N'] & \end{array}$$

    In the previous derivation we used the freshness condition $x\#\lambda y.N$, which follows from $\lambda y.N \sim_\alpha \lambda x.N''$, $x\#\lambda x.N''$, and that freshness is preserved under α-conversion.

□

As a direct consequence of the previous lemma, we have the following result:

Lemma 17 (Diamond property of $\rightrightarrows$).

$$M \rightrightarrows N \wedge M \rightrightarrows P \Rightarrow \exists Q, N \rightrightarrows Q \wedge P \rightrightarrows Q$$

Proof. We use the previous lemma twice, one for each hypothesis, directly getting that the term M^* satisfies the thesis. □

We omit the proof details of the next results because they do no deal with λ-terms. They are proved in a direct way as in the classical literature.

Definition 1 (Confluence). *A relation is* confluent *if its reflexive and transitive closure has the diamond property.*

Lemma 18. *If a relation R has the diamond property then it is confluent*

Lemma 19. *If a reduction relation R is confluent, then so is its reflexive and transitive closure R^*.*

As a direct application of the preceding two lemmas, we obtain:

Lemma 20. $\rightrightarrows^*$ *is confluent.*

If we now consider the β-reduction $\rightarrow_\beta$, we have:

Lemma 21. $(\rightarrow_\beta \cup \sim_\alpha)^* = \rightrightarrows^*$

Proof. As usually done, we prove the double inclusion. To prove $(\rightarrow_\beta \cup \sim_\alpha)^* \subseteq \rightrightarrows^*$, it is enough to prove $(\rightarrow_\beta \cup \sim_\alpha) \subseteq \rightrightarrows$. By lemma 12 we know $\sim_\alpha \subseteq \rightrightarrows$, and $\rightarrow_\beta \subseteq \rightrightarrows$ can be proved by a direct induction on the $\rightarrow_\beta$ reduction relation.

Finally, to prove $\rightrightarrows^* \subseteq (\rightarrow_\beta \cup \sim_\alpha)^*$ we prove $\rightrightarrows \subseteq (\rightarrow_\beta \cup \sim_\alpha)^*$ by a direct induction on $\rightrightarrows$. Then, by the monotonicity of * over $\subseteq$, we get the desired result $\rightrightarrows^* \subseteq ((\rightarrow_\beta \cup \sim_\alpha)^*)^*$ by the idempotence of *. □

Using the last two lemmas we finally arrive at the Church-Rosser theorem.

Theorem 2 (Church-Rosser). *The relation $\rightarrow_\beta \cup \sim_\alpha$ is confluent.*

5.5 Assignment of Simple Types

Let now τ be a syntactic category of ground types. Then the category of *simple types* is given by the following grammar:

$$\alpha, \beta ::= \tau \mid \alpha \rightarrow \beta$$

A context is a list of pairs of names and types representing a finite mapping between them. This mapping is defined associating to each name the *first type* occurring paired to it in the list. Because of this, our typing contexts may contain repeated variables, and for each variable its first occurrence will define its type.

We will write $x \in \Gamma$ when x is declared in the a context Γ. As usual, we denote the mapping application Γx. In all applications of this mapping we will require $x \in \Gamma$ in the premises. We say a context Γ is included in another context Δ, denoted as $\Gamma \subseteq \Delta$, if $\forall x \in \Gamma, x \in \Delta \wedge \Gamma x = \Delta x$. Extending or overriding a context Γ with a new association (x, α) will be denoted $\Gamma, x : \alpha$.

We can now define the simple type assignment relation in figure 5.8.

$$\vdash\text{v}\ \frac{x \in \Gamma}{\Gamma \vdash x : \Gamma x} \qquad \vdash\text{a}\ \frac{\Gamma \vdash M : \alpha \rightarrow \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}$$

$$\vdash\lambda\ \frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x M : \alpha \rightarrow \beta}$$

Figure 5.8: Typing assignment

Lemma 22 (Weakening Lemma).

$$\Gamma \subseteq \Delta \wedge \Gamma \vdash M : \alpha \Rightarrow \Delta \vdash M : \alpha$$

Proof. By a direct induction on the typing relation. As an example we show the $(\vdash\lambda)$ rule case. We have to prove $\Delta \vdash \lambda x.M : \alpha \rightarrow \beta$, which by the $(\vdash \lambda)$ rule can be derived from $\Delta, x : \alpha \vdash M : \beta$. We can finally apply the inductive hypothesis to prove the latter as $\Gamma, x : \alpha \subseteq \Delta, x : \alpha$ can be easily proved from the hypothesis $\Gamma \subseteq \Delta$. $\square$

In [65] the typing context Γ cannot have repeated variables. So the abstraction introduction rule of their typing relation requires the extra premise $x\#\Gamma$. This extra premise is problematic in the proof of the weakening lemma, since for the abstraction case to hold it is also necessary that $x\#\Delta$. Although their induction principle allows them to assume this freshness condition, the proof becomes more complex.

Lemma 23 (Weakening with Fresh Variables).

$$x\#M \wedge \Gamma \vdash M : \alpha \Rightarrow \Gamma, x : \beta \vdash M : \alpha$$

Proof. By induction on the typing relation. In the abstraction case we need to prove $\Gamma, x : \beta \vdash \lambda y.M : \gamma \rightarrow \alpha$, which by the $(\vdash\lambda)$ rule can proved if we have $\Gamma, x : \beta, y : \gamma \vdash M : \alpha$. If $x = y$, as $\Gamma, y : \gamma \subseteq \Gamma, x : \beta, y : \gamma$, we can obtain the desired result by the previous lemma and the hypothesis $\Gamma, y : \gamma \vdash M : \alpha$. If $x \neq y$, as $x\#\lambda y.M$, then x must be fresh in M. We have $\Gamma, y : \gamma, x : \beta \vdash M : \alpha$ by induction hypothesis, hence as $x \neq y$ we can interchange the tuples $y : \gamma$ and $x : \beta$ in the typing context of this hypothesis by the previous lemma, and obtain $\Gamma, x : \beta, y : \gamma \vdash M : \alpha$. Finally, applying rule $(\vdash\lambda)$ we conclude the proof. $\square$

Reasoning in a similar way we can prove the next strengthening result:

Lemma 24 (Strengthening with Fresh Variables).

$$x\#M \wedge \Gamma, x : \beta \vdash M : \alpha \Rightarrow \Gamma \vdash M : \alpha$$

We extend the swapping operation to contexts by swapping the names in them as expected, and denoting the operation in the usual way. By doing so we can now formalise the equivariance property of the typing relation.

Lemma 25 (Equivariance of the Typing Relation)**.**

$$\Gamma \vdash M : \alpha \Rightarrow (x\ y)\Gamma \vdash (x\ y)M : \alpha$$

Proof. Direct induction on the typing relation. □

We can now prove that the typing relation is preserved under α-conversion:

Lemma 26 (Typing relation is α-compatible)**.**

$$\Gamma \vdash M : \alpha \wedge M \sim_\alpha N \Rightarrow \Gamma \vdash N : \alpha$$

Proof. The proof uses the strong permutation induction principle presented in 5.3 on the term M. The variable and application cases are direct. In the abstraction case we have to prove $\Gamma \vdash \lambda y.N : \alpha \to \beta$ with $\Gamma \vdash \lambda x.M : \alpha \to \beta$ and $\lambda x.M \sim_\alpha \lambda y.N$ as hypotheses. From the second hypothesis we have that, by the definition of $\sim_\alpha$, there exists some list of variables xs such that $\forall z \notin xs, (x\ z)M \sim_\alpha (y\ z)N$. Now we can pick a variable $z \neq y$, fresh in N, and neither in the domain of Γ nor in the list xs. Then we reason from the first hypothesis as follows:

$$\begin{array}{ll}
\Gamma \vdash \lambda x.M : \alpha \to \beta & \Rightarrow \{\text{type assignment definition}\} \\
\Gamma, x : \alpha \vdash M : \beta & \Rightarrow \{\text{lemma 25}\} \\
(x\ z)(\Gamma, x : \alpha) \vdash (x\ z)M : \beta & \Rightarrow \\
(x\ z)\Gamma, z : \alpha \vdash (x\ z)M : \beta &
\end{array}$$

As $z \notin dom(\Gamma)$ we have that $(x\ z)\Gamma, z : \alpha \subseteq \Gamma, z : \alpha$, then by the weakening lemma 22, we obtain $\Gamma, z : \alpha \vdash (x\ z)M : \beta$. Then, from the second hypothesis, as we chose $z \notin xs$, we can derive that $(x\ z)M \sim_\alpha (y\ z)N$, and continue in the following way:

$$\begin{array}{ll}
\Gamma, z : \alpha \vdash (x\ z)M : \beta & \Rightarrow \{\text{by i.h.}\} \\
\Gamma, z : \alpha \vdash (y\ z)N : \beta & \Rightarrow \{\text{lemma 25}\} \\
(y\ z)(\Gamma, z : \alpha) \vdash (y\ z)(y\ z)N : \beta & \Rightarrow \{\text{swap application and idempotency of swapping}\} \\
(y\ z)\Gamma, y : \alpha \vdash N : \beta & \Rightarrow \{\text{by weakening lemma 23 as } z\#N\} \\
(y\ z)\Gamma, y : \alpha, z : \alpha \vdash N : \beta & \Rightarrow \{\text{by lemma 22 as } (y\ z)\Gamma, y : \alpha, z : \alpha \subseteq \Gamma, y : \alpha, z : \alpha\} \\
\Gamma, y : \alpha, z : \alpha \vdash N : \beta & \Rightarrow \{\text{by strengthening lemma 24 as } z\#N\} \\
\Gamma, y : \alpha \vdash N : \beta & \Rightarrow \{by(\vdash \lambda) \text{ rule}\} \\
\Gamma \vdash \lambda y.N : \alpha \to \beta &
\end{array}$$

□

We can now use the original α-induction principle to prove the crucial substitution lemma for the typing relation.

Lemma 27 (Substitution Lemma for the Typing Relation)**.**

$$\Gamma, x : \beta \vdash M : \alpha \wedge \Gamma \vdash N : \beta \Rightarrow \Gamma \vdash M[x := N] : \alpha$$

Proof. We use the α-induction principle (fig. 2.2) on the term M, and avoid x and the free variables in N as binding names in the abstraction case of the induction. The predicate to be proved is α-compatible on M as a direct consequence of the substitution lemma 1 and the previous result.

In the interesting abstraction case we need to prove $\Gamma \vdash (\lambda y.M)[x := N] : \gamma \to \alpha$ with hypotheses $\Gamma, x : \beta \vdash \lambda y.M : \gamma \to \alpha$ and $\Gamma \vdash N : \beta$. By the syntax directed rules of the typing relation, the term $\lambda y.M$ must be typed using the $(\vdash\lambda)$ rule with the premise $\Gamma, x : \beta, y : \gamma \vdash M : \alpha$. We assumed $y \neq x$, so using the weakening lemma 22 we can interchange the tuples $x : \beta$ and $y : \gamma$ in the typing context without modifying the mapping to get that $\Gamma, y : \gamma, x : \beta \vdash M : \alpha$. Again, the α-induction freshness hypothesis enables us to pick $y \# N$ so we can weaken the $\Gamma \vdash N : \beta$ hypothesis, extending the typing context with the tuple $y : \gamma$. We can now apply the induction hypothesis with the previous results to get $\Gamma, y : \gamma \vdash M[x := N] : \alpha$, and then by the $(\vdash\lambda)$ rule we get $\Gamma \vdash \lambda y(M[x := N]) : \gamma \to \alpha$. As $y \notin \{x\} \cup \mathrm{fv}(N)$, we can safely pull out the substitution operation in the latter typed term, in a naive way, getting the α-equivalent term $\lambda y.(M[x := N])$. Then, as the typing relation is α-compatible by the previous lemma, we finally get the desired result. □

A direct corollary of the previous result is that the typing judgement is preserved by β-contraction

Corollary 2 (Typing judgement is preserved by β-contraction)**.**

$$\Gamma \vdash M : \alpha, M \rhd N \Rightarrow \Gamma \vdash N : \alpha$$

Then, we can easily prove the following:

Corollary 3. $\Gamma \vdash M : \alpha, M \to_\beta N \Rightarrow \Gamma \vdash N : \alpha$

Proof. As the β-reduction relation is the contextual closure of the β-contraction we do a direct induction on the closure relation rules, where for the β-contraction case we directly use last corollary. □

Finally, the subject reduction theorem is proved by induction on the reflexive-transitive closure of $\to_\beta \cup \sim_\alpha$, using directly the last corollary and the α-compatibility of the typing relation (lemma 26) for the $\to_\beta$ and $\sim_\alpha$ cases respectively.

Theorem 3 (Subject Reduction)**.** $\Gamma \vdash M : \alpha, M \to_\alpha^* N \Rightarrow \Gamma \vdash N : \alpha$

5.6 Conclusions

The original α-induction principle on terms defined in the preceding chapter fails in emulating BVC usually used in the proof of the substitution lemma for the parallel reduction relation. To overcome this, we propose a novel strengthened α-induction principle that allows us to successfully deal with this problem. We use this principle to develop some metatheory of the λ-calculus, where our formalisation goes up to the confluence theorem for the β-reduction. Furthermore, we formalise the simply typed λ-calculus up to the Subject Reduction theorem. Our development successfully uses the original α-induction principle in the crucial substitution lemma for the typing relation. The latter is possible because we allow the occurrence of repeated variables in our typing contexts.

The definition of the parallel reduction relation has to be done in such a way as to ensure that the relation is α-compatible. Because of this, it looks more concrete than the classical one, as presented by Barendregt [4] or Takahasi [62]. However, we are able to prove inversion lemmas that allow us to recover the original parallel reduction definition, and from them we are able to reproduce Takahashi's proof of the diamond property.

In a similar work, Urban and Norrish [65] also have to modify the parallel reduction relation in order to derive an ad-hoc induction principle on the parallel reduction to successfully prove the substitution lemma for this relation. We believe our approach is more direct and general, since we derive an induction principle on simple terms, and not on the more complex relations over them. As shown in the beta subcase of the proof of the substitution theorem, we are able to derive the freshness conditions for the binders directly from our α-induction principle, as in the BVC, and not explicitly imposing them in the definition of the parallel reduction relation.

CHAPTER 6

Generic Binding Framework

In this chapter we generalise the techniques developed in the preceding two chapters, applying generic programming methods to address the formalisation of generic structures with binders. We define a universe of regular datatypes with variable binding information, and on these we define generic formation, elimination, and induction operators. We also introduce an α-equivalence relation based on the swapping operation, and we derive α-iteration/induction principles that capture the BVC.

6.1 Introduction

The definition of generic functions by recursion on the description of datatypes is the basic idea of generic programming. This method works by defining a datatype, introduced as the *universe* in for instance [37], which contains datatype descriptions, such as "a list is either empty or a pair consisting of a parameter and a sublist" or "a tree is either a leaf with a parameter or a pair with two subtrees". Indeed, the universe constructors correspond to these commons: "either", "pair", "parameter" and "substructure" abstracted out of the previous informal descriptions. Then, a decoding function is introduced, which interprets instances of the universe, usually called universe *codes*, into actual datatype instances.

In a dependently typed setting we can define generic functions over the universe of codes and the associated decoded datatypes. In other words, the codes give us enough information to properly traverse the structure of their corresponding datatypes. The following is an outline of such technique, showing the generic signature of an equality test over some universe instance `code`.

```
equal :  (code :  Universe) → decode code → decode code → Bool
```

This would be a simple example of a generic function embedded in several programming languages. In Haskell, for instance, the `deriving Eq` primitive instructs the compiler to automatically derive the syntactical equality for any datatype definition. More interesting examples include generic iterations and recursion principles directly derived from inductive types. Indeed, the traversal of a recursive structure is a classical operation that can be described for any recursive structure. Therefore, generic programming avoids code duplication by allowing us to abstract out common operations in datatypes. One can wonder how many other practical behaviors can be generalised from such general structures. In this chapter we enrich a universe of datatypes with variables and scoping information, introducing nominal techniques over any abstract syntax with binders.

The particular universe election has a direct impact on the datatypes and on the set of supported generic functions it can express. In this chapter we present a universe of *Regular Trees* [43], extended with variables and binding information. We define generic formation, elimination, and induction operators over this universe. Moreover, our universe extension also allows us to introduce an α-equivalence relation based on the swapping operation. Then, as done in the previous chapters for the λ-calculus, we derive an α-iteration/induction principle that captures the BVC. We develop the λ-calculus and System F as examples of languages defined in our universe, and show how to derive the naive substitution operation, and also the capture-avoiding substitution, plus several other results proved at a generic level, that is, results we will be able to reuse in the λ-calculus and System F instances. We also prove specific lemmas using the α-induction principle for our universe construction, showing in this way practical usages of the proposed framework. These examples show the flexibility of our approach, as we are able to prove particular language results as it is usually done using pen-and-paper. Some notation from the codes of our universe permeates our proofs, but this only involves notation overhead, and does not modify the basic structure of the proofs and their underlying general ideas.

6.2 Related work

The implementation of meta-programs, that is, programs that manipulate syntax, requires operations that are not specific to a single language, but are common to several ones. As meta-programs are usual pieces of software, there exist several examples of works supporting general syntax based manipulations. These developments are based on different representations.

The first use of generic programming in a dependently typed setting is presented by Pfeifer and Ruess in [49], and it uses the LEGO programming language. The universe presented describes datatypes that are sums of products, which corresponds to an "either of certain tuples" informal description. In [6, 19] Dybjer and Bove present several universes with more expressive power, enabling the description of families of datatypes and introducing parameters on their descriptions. In [43] Morris presents a universe construction allowing mutual inductive datatypes.

Programming languages supporting native constructions to declare and manipulate abstract syntax with binders are presented by Shinwell, et. al in [59,60], where an ML extension *FreshML*, and an O'Caml extension *Fresh O'Caml* are correspondly developed. These languages allow to deconstruct datatypes with binders in a safe way, that is, in the case of an abstraction inspection, a variable swapping operation with a freshly generated binder is computed in the abstraction body. In this way, the language user has access only to a fresh binder, and the correspondly renamed body of the opened abstraction. This mechanism grants that values with binders are operationally equivalent if they represent α-equivalent objects. This result is proved in [60] by introducing a denotational semantics of the object language FreshML into FM-sets (Fraenkel and Mostowski sets). They prove that this denotational semantics matches the operational one. In this way, they are able to prove that values of the introduced abstract syntax with binders properly represent α-equivalence classes of the object-level syntax. In [8] Cheney carries out a similar work, but instead of developing a language extension, he implements a Haskell library called *FreshLib*. As he does not implement a language from scratch, this work introduces generic programming techniques in its implementation in order to support the required level of genericity.

All previous works address common operations dealing with general structures with binders. Although some of these developments give proofs about the soundness of their approaches, their main concern is the implementation of meta-programs. In [34], Lee et al. use generic programming techniques to develop mechanisations of formal meta-theory in the Coq proof assistant. This work allows the user to choose between nominal, locally nameless or de Bruijn first-order syntax. For each of these representations, they offer several infrastructure operations and their associated lemmas. For instance, for the locally nameless setting, two different substitutions are needed for bound and free variables correspondingly. In the case of System F, where terms and type variables have binding constructions, this representation involves six different substitution operations. Hence, as the number of syntactic sorts supporting binding constructions increases in the object language, there is a combinatorial explosion of the number of operations and lemmas involved in its formalisation. They manage to address these issues defining these operations and associated lemmas in a generic re-usable way. Moreover, they provide a small annotation language to describe the binding structure of the object language, from which they can automatically derive an isomorphism between the object language and their generic universe syntax. However, introducing inductive relations in this framework requires the user to provide a mapping between the concrete relation, defined at the object language level, and the generic relation. They claim to be able to successfully instantiate some cases of the POPLmark challenge [3] in their framework, validating their approach both for the locally nameless and the de Bruijn first-order syntax, and comparing some metrics of their approach against other solutions. However, their particular choice of universe makes it impossible to have more than one sort of binder per datatype, so they cannot represent in their setting a language such as Session Types [69], where there exist three distinct sort of binders: parameters, channels and ports within a concurrent calculus, as we will see in detail in next section. We believe their work addresses reusing and usability in great manner, but lacks in extensibility and abstraction. By using this framework the user can reuse several operations and lemmas that hide some of the work required by the underlying chosen binders representation. However, in order to introduce new operations and prove results, the user may have to deal with the underlying generic abstract syntax language. Their work seems to support the nominal syntax, although no α-conversion relation, neither any other relation over terms is presented. Indeed, they do not further develop the nominal syntax, beyond the basic definitions of a nominal abstract syntax.

In [36], Licarta and Harper codify a universe that mixes binding and computation constructions in Agda, where computations are represented as meta-level functions injected in the universe constructions, i.e., they embed a HOL syntax in their development. Their representation is based on a well-scoped de Bruijn representation, that is, de Bruijn terms associated with a context indexing the free variables. For this universe, they provide a generic substitution operation, and prove context weakening and strengthening lemmas.

Our work uses generic programming techniques to develop the meta-theory of abstract syntax with binders in a generic way, as in the previously described works. But we chose to maintain names for binders like as usually done in informal practice.

6.3 Regular Tree Universe with Binders

As in Lee et al. [34], we choose a simplification of the universe of regular tree datatypes presented by Morris in [43]. In Morris' original work, the universe of regular trees can represent recursive types using μ-types (from [51]). However, instead of the nominal approach traditionally used with recursive type binders, this universe uses a well-scoped de Brujin representation. Therefore, in order to properly interpret the full universe, a definition indexed by a context with the multiple μ-recursive positions definitions is required. Our representation in Agda (figure 6.1), simplifies this burden at the expense of not being able to represent mutually recursive datatypes. In other words, our universe construction admits only a single top-level μ-recursive type binder. However, it is rich enough in structure to address languages without mutually recursive datatypes, such as λ-calculus, System F and Session Types processes.

In this chapter we will show the Agda code of our development. The code is available at:

https://github.com/ernius/genericBindingFramework

```
data Functor : Set₁ where
  |1|    :                       Functor
  |R|    :                       Functor
  |E|    : Set               →  Functor
  |Ef|   : Functor           →  Functor
  _|+|_  : Functor → Functor →  Functor
  _|x|_  : Functor → Functor →  Functor
  |v|    : Sort              →  Functor
  |B|    : Sort    → Functor →  Functor
```

Figure 6.1: Regular tree universe with binders.

The first three base constructors in figure 6.1 represent the embedding of: the unity type, a recursive position, and an arbitrary datatype. The fourth constructor embeds a datatype representable in our universe, while the next two constructors represent correspondly the sum and product types. Finally, the last two introduction rules are specific to our desired domain of abstract syntaxes with binders. As our universe supports different sorts of variables, the variables and binders constructors receive as parameters the sort of variables that they introduce or bind. The binder constructor also receives the descriptor of the bound subterm.

For example, the type representations of natural numbers and lists of natural numbers can be defined in our universe as follows:

$$\begin{aligned} \text{FNat} &= |1|\ |{+}|\ |\text{R}| \\ \text{FListNat} &= |1|\ |{+}|\ (|\text{Ef}|\ \text{FNat})\ |{\times}|\ |\text{R}| \end{aligned}$$

Next, we extract from [51] the more familiar definitions of the natural numbers and lists of natural numbers. The analogies between both definitions can be easily recognised.

$$\begin{aligned} \text{Nat} &= \mu R.(1 + R) \\ \text{ListNat} &= \mu R.(1 + \text{Nat} \times R) \end{aligned}$$

In figure 6.2 we introduce the missing μ-operator in our definition. For this, we mutually define the decoding or interpretation function ⟦_⟧, and the fixed point μ-operator for our universe.

```
mutual
  ⟦_⟧ : Functor → Set → Set
  ⟦ |1|        ⟧ _  = ⊤
  ⟦ |E|     B  ⟧ _  = B
  ⟦ |Ef|    F  ⟧ _  = μ F
  ⟦ |R|        ⟧ A  = A
  ⟦ F |+|   G  ⟧ A  = ⟦ F ⟧ A  ⊎ ⟦ G ⟧ A
  ⟦ F |x|   G  ⟧ A  = ⟦ F ⟧ A  × ⟦ G ⟧ A
  ⟦ |v|  S     ⟧ _  = V
  ⟦ |B|  S  G  ⟧ A  = V         × ⟦ G ⟧ A

  data μ (F : Functor) : Set where
    ⟨_⟩ : ⟦ F ⟧ (μ F) → μ F
```

Figure 6.2: Regular tree universe interpretation.

The unit case |1| just returns the unit type in Agda. The next two cases correspond to embeddings, and they respectively return the injected set and the fixed point of the embedded universe descriptor. In the recursive case |R|, we impose the fixed point semantics by returning the Set interpretation argument. For the sum and product cases we return Agda's disjoint sum and product datatype constructions respectively applied to the recursive calls. Finally, in the last two rules we can observe how the variable and binder constructions inject the fixed set of variables V in the interpreted datatype. We assume a denumerable set V of variables with a decidable equality. Note that in this last two cases the sort argument S has no impact on the interpreted set. Indeed, we have only one kind of variables V. This sort identifier will be relevant in next sections to implement generic operations related to binding issues.

In fact our universe construction describes a function from Set to Set. Indeed, the encoded set must be interpreted as the fixed point of the decoding function ⟦_⟧. This function can be viewed as a *functor* in category theory, and as we will see in the next section, it consequently supports the classical map and fold operations.

Next, we properly define the natural numbers and the list of natural numbers as the fixed point of the previously introduced functors, resembling Pierce's constructions. Note that no name is needed in the outer μ-constructor as our universe admits only one top-level μ-recursive type binder.

$$\begin{aligned} \text{Nat} &= \mu\ \text{FNat} \\ \text{ListNat} &= \mu\ \text{FListNat} \end{aligned}$$

In the following example, we illustrate the use of our universe, and in particular of the variables and binders constructions, by encoding the λ-calculus in it (figure 6.3). We show the corresponding classical concrete syntax definition using comments, that are written following a "-" to the right of each line. This definition has only one sort of variables identified with the sort SortλTermVars.

We next introduce infix constructors, resembling the concrete syntax of the λ-calculus, and hiding away our universe code constructions.

```
λF : Functor                        - M,N :-
λF =   |v| SortλTermVars            - x
   |+|  |R| |x| |R|                 - | M N
   |+|  |B| SortλTermVars |R|       - | λ x M

λTerm : Set
λTerm = μ λF
```

Figure 6.3: λ-calculus.

```
v : V → λTerm
v = ⟨_⟩ ∘ inj₁

_·_ : λTerm → λTerm → λTerm
M · N = ⟨ inj₂ (inj₁ (M , N)) ⟩

ƛ : V → λTerm → λTerm
ƛ n M = ⟨ inj₂ (inj₂ (n , M)) ⟩
```

In figure 6.3 we present the codification of the System F. As this language also needs bindings at the type level, this encoding illustrates the use of two distinct sorts of binders: SortFTypeVars and SortλTermVars.

```
tyF : Functor                                  - t,r :-
tyF =  |v| SortFTypeVars                       - α
   |+|  |R| |x| |R|                            - | t → r
   |+|  |B| SortFTypeVars |R|                  - | ∀ α .  t
                                               -
tF : Functor                                   - M,N :-
tF =   |v| SortFTermVars                       - x
   |+|  |R| |x| |R|                            - | M N
   |+|  |Ef| tyF |x| |B| SortFTermVars |R|     - | λ x :  t .  M
   |+|  |R| |x| |Ef| tyF                       - | M t
   |+|  |B| SortFTypeVars |R|                  - | Λ α .  M

FType : Set
FType = μ tyF

FTerm : Set
FTerm = μ tF
```

Figure 6.4: System F.

Our universe construction departs from the one presented by Lee et al. in several aspects. Firstly, in their work the variable construction is defined at the top-level, that is, in the μ datatype. Hence, in their universe, for any described datatype, its elements are either a variable or an element following the structure given by some functor. By doing so they are able to define the substitution operation in a generic way. However, this election makes their work depart from the category theory approach. In contrast, we maintain our framework compatible to a categorical setting, showing latter how easily we can recover the substitution as an instance of the more general fold operation. Secondly, our constructions for binders and variables have a sort argument to distinguish between distinct sorts of variables. Lee et al. can only have one sort of binder for each datatype. This makes it impossible to model more than one sort of binders for the same datatype, as it is the case of Session Types processes [69]. In figure 6.5 we show the syntax of the datatype of processes in this language, which has three kinds of bindable names: parameters x , channels k and ports a.

$$
\begin{array}{lcll}
P & :- & \text{request } a(k) \text{ in } P & \text{(binds channel } k \text{ in } P) \\
 & | & \text{accept } a(k) \text{ in } P & \text{(binds channel } k \text{ in } P) \\
 & | & k!(x) \text{ in } P & \text{(binds parameter } x \text{ in } P) \\
 & | & (\nu k)P & \text{(binds channel } k \text{ in } P) \\
 & | & (\nu a)P & \text{(binds port } a \text{ in } P) \\
 & | & \ldots &
\end{array}
$$

Figure 6.5: Session Types processes.

However, we cannot model the syntax of the types of Session Types, shown in figure 6.6, as it involves a mutual recursive datatype.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
T & :- & S & S & :- & end \\
 & | & Nat & & | & ?T.S \\
 & | & \ldots & & | & !T.S \\
 & | & \ldots & & | & \ldots
\end{array}
$$

Figure 6.6: Session Types.

6.3.1 Map and Fold

In figures 6.7 and 6.8 we present the classical map and fold operations as they are usually introduced in category theory. Our definition only adds two extra rules for the variables and binders constructions.

```
mapF : {A B : Set}(F : Functor) → (A → B) → ⟦ F ⟧ A → ⟦ F ⟧ B
mapF (|v|  S)         f x          = x
mapF |1|              f tt         = tt
mapF (|E|  A)         f e          = e
mapF (|Ef|  F)        f e          = e
mapF |R|              f e          = f e
mapF (G₁ |+|  G₂)     f (inj₁ e)   = inj₁  (mapF G₁ f e)
mapF (G₁ |+|  G₂)     f (inj₂ e)   = inj₂  (mapF G₂ f e)
mapF (G₁ |x|  G₂)     f (e₁ , e₂)  = mapF G₁ f e₁   , mapF G₂  f e₂
mapF (|B|  S   G)     f (x , e)    = x              , mapF G   f e
```

Figure 6.7: Classical map operation.

```
{-# TERMINATING #-}
foldT : {A : Set}(F : Functor) → (⟦ F ⟧ A → A) → μ F → A
foldT F f ⟨ e ⟩ = f (mapF F (foldT F f) e)
```

Figure 6.8: Classical fold operation.

Unfortunately this definition of fold does not pass Agda's termination checker. The recursive call to foldT is passed to the higher order function mapF, and because of this reason the termination checker cannot see how mapF is using it.

To make the fold operation pass the termination checker we have to fuse map and fold into a single function, as done in [47] for a similar regular tree universe. In figure 6.9 we show our implementation of the function foldmap. It needs to keep two functors, since the fold (recursive) part works always over the same functor argument F, while, for the map part, the auxiliary functor argument G gives the position during the traversal of the structure of functor F. Therefore, this function only uses the functor F in the recursive case rule (the |R| case) in which the right hand side expression basically begins a new traversal of the functor F, in a way similar to the original definition of fold. It does so by providing with a fresh copy of F in the position of the auxiliary argument G. The rest of the rules are equivalent to a map over the functor G. Note that this definition terminates because the argument of type ⟦ G ⟧ (μ F) decreases with each call.

Finally, the new fold operation is defined as a recursive instance of foldmap, as shown in figure 6.10.

```
foldmap :  {A : Set}(F G  : Functor) →  (⟦ F ⟧ A → A)
   →        ⟦ G ⟧ (μ F) → ⟦ G ⟧ A
foldmap F (|v|   S)       f x          =  x
foldmap F |1|             f tt         =  tt
foldmap F (|E|   A)       f e          =  e
foldmap F (|Ef|  G)       f e          =  e
foldmap F |R|             f ⟨ e ⟩      =  f     (foldmap F F   f e)
foldmap F (G₁ |+|  G₂)    f (inj₁ e)   =  inj₁  (foldmap F G₁  f e)
foldmap F (G₁ |+|  G₂)    f (inj₂ e)   =  inj₂  (foldmap F G₂  f e)
foldmap F (G₁ |x|  G₂)    f (e₁ , e₂)  =  foldmap F G₁ f e₁  ,
                                          foldmap F G₂   f e₂
foldmap F (|B| S    G)    f (x , e)    =  x                  ,
                                          foldmap F G    f e
```

Figure 6.9: Terminating fold-map fusion operation.

```
fold : {A : Set}(F : Functor) → (⟦ F ⟧ A → A) → μ F → A
fold F f e = foldmap F |R| f e
```

Figure 6.10: Terminating fold operation.

In figure 6.11 we continue with the λ-calculus example introduced in the previous section. As an example, we define a function vars that counts the number of variable occurrences in a term. We do so by instantiating it as a case of the fold operation. This function could also be defined equivalently by an explicit recursion on λ-terms.

```
varsaux : ⟦ λF ⟧ ℕ → ℕ
varsaux (inj₁ _)              = 1
varsaux (inj₂ (inj₁ (m , n))) = m + n
varsaux (inj₂ (inj₂ (_ , m))) = m

vars : μ λF → ℕ
vars = fold λF varsaux
```

Figure 6.11: Fold application example.

6.3.2 Primitive Induction

In this section we develop an elimination rule that is more generic than the fold operation defined above. This elimination rule captures proof by induction, and is based on the recursion rule given by Benke et al. in [5]. However, our development departs from their work in the following points: Firstly, they derive an elimination rule for a simpler universe construction, based on one-sorted term algebras, and defined through the more simpler signatures, instead of over functors as we do. For instance, their universe does not allow the injection of previously defined datatypes. This is the case for the list of natural numbers datatype, where natural numbers are injected into lists constructions. Secondly, their induction principle does not pass the termination checker due to reasons similar to the ones we discussed for the first version of the fold operation.

To define the foldInd function, first we introduce the auxiliary function fih (figure 6.12). This function receives a predicate P over the fixpoint of a functor F and an auxiliary functor G (with a similar role as the one for the previous foldmap function), and constructs a corresponding predicate of type ⟦ G ⟧ (μ F) → Set. This predicate represents the predicate P holding for every recursive position μ F in an element of type ⟦ G ⟧ (μ F).

```
fih  :  {F : Functor}(G : Functor)(P : μ F → Set)
     →  ⟦ G ⟧ (μ F) → Set
fih (|v|  S)       P x          = ⊤
fih |1|            P tt         = ⊤
fih (|E|  B)       P e          = ⊤
fih (|Ef| G)       P e          = ⊤
fih |R|            P e          = P e
fih (G₁ |+| G₂)    P (inj₁ e)   = fih G₁  P e
fih (G₁ |+| G₂)    P (inj₂ e)   = fih G₂  P e
fih (G₁ |x| G₂)    P (e₁ , e₂)  = fih G₁  P e₁ × fih G₂ P e₂
fih (|B| S  G)     P (x , e)    = fih G   P e
```

Figure 6.12: fih function.

We can now present our induction principle. We will do it in a similar way as we did above for the fold function. First, we introduce the fold-map fusion function foldmapFh (figure 6.13). Then, we use this function to directly derive the induction principle as a recursive instance of the fold-map fusion (figure 6.14).

We next give an example of the use of this induction principle. We prove that the application of the function vars to any lambda term is grater than zero. We introduce the predicate Pvars (figure 6.15) representing the property to be proved, and then we introduce an auxiliary lemma

```
foldmapFh : {F : Functor}(G : Functor)(P : μ F → Set)
            → ((e : ⟦ F ⟧ (μ F)) → fih F P e →  P ⟨ e ⟩)
            → (x : ⟦ G ⟧ (μ F)) → fih G P x
foldmapFh (|v|  S)         P hi n         = tt
foldmapFh |1|              P hi tt        = tt
foldmapFh (|E|  B)         P hi b         = tt
foldmapFh (|Ef|  F)        P hi b         = tt
foldmapFh {F} |R|          P hi ⟨ e ⟩     = hi e (foldmapFh {F} F P hi e)
foldmapFh (G₁ |+|  G₂)     P hi (inj₁  e) = foldmapFh G₁  P hi e
foldmapFh (G₁ |+|  G₂)     P hi (inj₂  e) = foldmapFh G₂  P hi e
foldmapFh (G₁ |x|  G₂)     P hi (e₁  , e₂) = foldmapFh G₁  P hi e₁ ,
                                             foldmapFh G₂  P hi e₂
foldmapFh (|B| S    G)     P hi (x   , e) = foldmapFh G   P hi e
```

Figure 6.13: Fold-map fusion.

```
foldInd : (F : Functor)(P : μ F → Set)
          → ((e : ⟦ F ⟧ (μ F)) → fih F P e →  P ⟨ e ⟩)
          → (e : μ F) → P e
foldInd F P hi e = foldmapFh {F} |R| P hi e
```

Figure 6.14: Induction principle.

plus>0, stating that the sum of two positive numbers is also positive.

```
PVars : μ λF → Set
PVars M = vars M > 0

plus>0 : {m n : ℕ} → m > 0 → n > 0 → m + n > 0
plus>0 {m} {n} m>0 n>0 = ≤-steps m n>0
```

Figure 6.15: Pvars and plus>0.

The proof that Pvars holds for every term M is a direct application of the induction principle as shown in figure 6.16. The variable case is direct, while the application case is the application of the lemma plus>0 to the inductive hypotheses. Finally, the abstraction case is a direct application of the inductive hypothesis.

```
provePVars : (M : μ λF) → PVars M
provePVars = foldInd λF PVars proof
  where
  proof : (e : ⟦ λF ⟧ (μ λF)) → fih λF PVars e → PVars ⟨ e ⟩
  proof (inj₁ x)                 tt          = s≤s z≤n
  proof (inj₂ (inj₁ (M  , N)))   (PM , PN)   = plus>0 PM PN
  proof (inj₂ (inj₂ (_  , M)))   PM          = PM
```

Figure 6.16: Proof of Pvars.

As in the definition of vars above (figure 6.11), this proof could also have been done by a direct induction on terms. However, these elimination principles will be the base upon which we will develop more interesting proofs in further sections.

6.3.3 Fold with Context Information

In this section we present a particular useful instantiation of the previously introduced fold operator. This instantiation aims at reproducing nominal techniques in our work.

We present its definition in figure 6.17. We introduce an extra argument with type μ C, which is used by the folded function f. This function is partially applied to this context argument, and then passed to the previously defined fold function. Hence, this argument acts as an explicit invariant context for the function f through the entire fold operation. Another difference with the original fold operation is that the result of this instance is a datatype μ H encoded in our universe instead of an arbitrary Set.

```
foldCtx :  {C H : Functor}(F : Functor)
   →       (μ C → ⟦ F ⟧ (μ H) → μ H)
   →       μ C → μ F → μ H
foldCtx F f c = fold F (f c)
```

Figure 6.17: Fold with Context.

From this function we can directly derive the naive substitution operation. In order to do this, we first give the functor descriptor cF for the context argument. It represents the pair containing the substituted term and the variable for which it is substituted.

```
cF =  |v| SortλTermVars |x| |Ef| λF
```

Then, we define the auxiliary function substaux (figure 6.18) which, given a term structure with the results of the recursive calls in its recursive positions, constructs the result of the substitution. The application and abstraction cases directly reconstruct the corresponding terms from the recursive call results. Note that we hide the universe codes in the right side of this definition by using the previously introduced constructors of the λ-calculus.

```
substaux : μ cF → ⟦ λF ⟧ (μ λF) → μ λF
substaux _         (inj₂ (inj₁ (t₁ , t₂)))  = t₁ · t₂
substaux _         (inj₂ (inj₂ (y , t)))    = ƛ y t
substaux ⟨ x , N ⟩ (inj₁ y) with x ≟v y
... | yes _                                 = N
... | no _                                  = v y
```

Figure 6.18: Naive substitution auxiliary function.

For the variable case, we compare the variables and apply the substitution if they are equal, otherwise we return the variable unchanged, as usually done. The application case is immediate. In the abstraction case we do not check whether the abstracted variable is different from the substituted one, as in Barendregt's substitution definition. In fact, this comparison would be pointless because, as we are using an iteration principle, we do not have the original abstraction body subterm.

Finally, we instantiate the foldCtx function with substaux, and its appropriate context pair to get the naive substitution operation. This definition is equivalent to the one presented by Barendregt in [4].

```
_[_:=_]ₙ : λTerm → V → λTerm → λTerm
M [ x := N ]ₙ  = foldCtx λF substaux (⟨ x , N ⟩) M
```

Figure 6.19: Naive substitution.

6.4 Name Swapping

In this section we develop nominal techniques over the introduced universe. We begin with the basic swapping operation. This operation completely traverses a functorial structure, swapping occurrences of variables (either free, bound or binding) of some given sort of variables.

Its implementation is similar to the implementation of the function fold. We use an auxiliary function swapF, presented in figure 6.20, that takes a function F and an extra functor argument G, and traverses its structure until recursive or embedded positions are reached, from where we restart this auxiliary argument with either the original recursive functor F or the embedded functor correspondingly. Note that this last case differs from the definition of fold, where this case is a base case. However, in this definition we must also traverse the embedded functor instance, as we are swapping all the variables in the structure, including the variables present in any embedded structure. Because of this, we can not derive the swaping operation as an instance of fold. In the variable and abstractions cases we use the swapping operation over variables, introduced in chapter 4.

```
swapF  :    {F : Functor}(G : Functor)
         →  Sort → V → V → ⟦ G ⟧ (μ F) → ⟦ G ⟧ (μ F)
swapF (|v|  S')       S a b c with S' ≟S S
... | yes  _                      =  ( a • b )ₐ c
... | no   _                      =  c
swapF |1|             S a b tt    =  tt
swapF (|E|  _)        S a b e     =  e
swapF (|Ef| G)        S a b ⟨ e ⟩  =  ⟨ swapF G S a b e ⟩
swapF {F} |R|         S a b ⟨ e ⟩  =  ⟨ swapF F S a b e ⟩
swapF (G₁ |+|  G₂)  S a b (inj₁ e)  =  inj₁ (swapF G₁ S a b e)
swapF (G₁ |+|  G₂)  S a b (inj₂ e)  =  inj₂ (swapF G₂ S a b e)
swapF (G₁ |x|  G₂)  S a b (e₁ , e₂)  =  swapF G₁ S a b e₁  ,
                                      swapF G₂ S a b e₂
swapF (|B| S'   G)   S a b (c , e) with S' ≟S S
... | yes  _  =                       ( a • b )ₐ  c      ,
                                      swapF G S a b e
... | no   _  =                       c                  ,
                                      swapF G S a b e
```

Figure 6.20: Auxiliary function swapF.

Finally, in figure 6.21 we present the swapping operation as a recursive case instance of the swapF function.

```
swap : {F : Functor} → Sort → V → V → μ F → μ F
swap S a b e = swapF |R| S a b e
```

Figure 6.21: Swapping operation.

Once the swapping operation is defined, we are able to generically prove the lemma in figure 6.22. This lemma states that the fold operation is well-behaved with respect to swapping, given that the correspondly folded operation is also well-behaved. The proof is a direct induction on terms.

```
lemmaSwapFoldEquiv  :  {F H : Functor}{S : Sort}
        {x y : V}{e : μ F}{f  :  ⟦ F ⟧ (μ H) → μ H}
    →  ({x y : V}  {e  :  ⟦ F ⟧ (μ H)}
      →            f (swapF F S x y e) ≡ swap {H} S x y (f e))
    →  fold F f (swap S x y e) ≡ swap S x y (fold F f e)
```

Figure 6.22: Fold is well-behaved with respect to swapping.

We also introduce a similar lemma in figure 6.23 for the fold instance with context information presented in the previous section. Its proof (fig. 6.24) follows by a direct equational derivation using the previous lemma, and the proof that f is well-behaved with respect to swapping.

```
lemmaSwapFoldCtxEquiv : {C H F  : Functor}{S : Sort}{x y : V}
        {e : μ F}{f  : μ C → ⟦ F ⟧ (μ H) → μ H}{c : μ C}
    →  ({c : μ C}{S : Sort}{x y : V}{e : ⟦ F ⟧ (μ H)}
      →  f (swap S x y c) (swapF F S x y e) ≡ swap S x y (f c e))
    →  foldCtx F f (swap {C} S x y c) (swap {F} S x y e)
       ≡
       swap {H} S x y (foldCtx F f c e)
```

Figure 6.23: Fold with context is well-behaved with respect to swapping.

```
lemmaSwapFoldCtxEquiv {C} {H} {F} {S} {x} {y} {⟨ e ⟩} {f} {c} prf =
   begin≡
     foldCtx F f (swap {C} S x y c) (swap {F} S x y ⟨ e ⟩)
   ≡⟨ refl                                                   ⟩
     f  (swap {C} S x y c)
        (foldmap F F (f (swapF |R| S x y c)) (swapF F S x y e))
   ≡⟨ cong  (f (swap {C} S x y c))
            (lemmaSwapFoldEquivCtxF {F} {F} {H} {S} {x} {y} {e}
               {λ x y → f (swap {C} S x y c)} {f c} prf)  ⟩
     f (swap {C} S x y c) (swapF F S x y (foldmap F F (f c) e))
   ≡⟨ prf                                                    ⟩
     swap {H} S x y (foldCtx F f c ⟨ e ⟩)
   ■
```

Figure 6.24: Proof of lemmaSwapFoldCtxEquiv.

With these examples we show how to develop generic functions in our universe with binders, and how to prove generic lemmas about their interaction with the previously introduced iteration principles.

We now illustrate the application of this lemma to the λ-calculus terms defined in figure 6.3, directly deriving the lemma in figure 6.25. This lemma states that swapping commutes with substitution, which is a result that was particularly useful in the developments presented in the preceding chapters. In the proof we can see the use of lemma-substauxSwap which states that the auxiliary function substaux (fig. 6.19) used to define the substitution is well-behaved with respect to swapping.

```
lemma-[]Swap : {x y z : V}{M N : λTerm}
  → (( y • z ) M) [ ( y • z )_a x ≔ ( y • z ) N ]_n
    ≡
    ( y • z ) (M [ x ≔ N ]_n)
lemma-[]Swap {x} {y} {z} {M} {⟨ N ⟩}
  = lemmaSwapFoldCtxEquiv  {cF} {λF} {λF} {SortλTermVars}
    {y} {z} {M} {substaux} {⟨ x , ⟨ N ⟩ ⟩}
    (λ {c} {S} {x} {y} {e} →
      lemma-substauxSwap {c} {S} {x} {y} {e})
```

Figure 6.25: Substitution is well-behaved with respect to swapping.

The previous example shows how feasible is to instantiate generic proofs, and to derive useful lemmas in particular instances of our generic universe.

6.5 Alpha Equivalence Relation.

In figure 6.26 we introduce the generic definition of the α-equivalence relation over our universe, named ∼α.

Its definition follows a process similar to the one used before to implement generic functions over our universe. First, we define an auxiliary relation ∼αF, which is inductively defined introducing an auxiliary functor G, used to traverse the functor F structure. For the interesting binder case, we follow an idea similar to the one used in chapter 4, that is, we define that two abstractions are α-equivalent if there exists some list of variables xs, such that for any given variable z not in xs, the result of swapping the corresponding binders with z in the abstraction bodies is α-equivalent. Note that the swapping is performed only over the sort of variables bound by this binder position, leaving any other sort of variables unchanged.

We are able to prove that this is an equivalence relation, and also that it is preserved under swapping in a similar way as previously done in chapter 4.

As we did before with the swapping operation, we now proceed to study how the iteration principles behave under the introduced α-conversion relation. We begin by proving that the fold operation is α-compatible if applied to α-compatible functions. In figure 6.27 we prove this lemma for the fold-map fusion with a direct induction on terms. In the functor embedding case (|Ef| G) we use the reflexive property of α-conversion. The only interesting case is the last binder case, which is direct by using that α-conversion is preserved by swapping.

```
data ∼αF {F : Functor} : (G : Functor)
   → ⟦ G ⟧ (μ F) →  ⟦ G ⟧ (μ F) → Set where
      ∼αV    :  {x : V}{S : Sort}  →  ∼αF (|v| S)          x          x
      ∼α1    :                        ∼αF |1|               tt         tt
      ∼αE    :  {B : Set}{b : B}   →  ∼αF (|E| B)          b          b
      ∼αEf   :  {G : Functor}{e e' : ⟦ G ⟧ (μ G)}
             →  ∼αF G e e'         →  ∼αF (|Ef| G)         ⟨ e ⟩      ⟨ e' ⟩
      ∼αR    :  {e e' : ⟦ F ⟧ (μ F)}
             →  ∼αF F e e'         →  ∼αF |R|              ⟨ e ⟩      ⟨ e' ⟩
      ∼α+₁   :  {F₁ F₂ : Functor}{e e' : ⟦ F₁ ⟧ (μ F)}
             →  ∼αF F₁ e e'        →  ∼αF (F₁ |+| F₂)  (inj₁ e)   (inj₁ e')
      ∼α+₂   :  {F₁ F₂ : Functor}{e e' : ⟦ F₂ ⟧ (μ F)}
             →  ∼αF F₂ e e'        →  ∼αF (F₁ |+| F₂)  (inj₂ e)   (inj₂ e')
      ∼αx    :  {F₁ F₂ : Functor}{e₁ e₁' : ⟦ F₁ ⟧ (μ F)}
                {e₂ e₂' : ⟦ F₂ ⟧ (μ F)}
             →  ∼αF F₁ e₁ e₁' → ∼αF F₂ e₂ e₂'
                                   →  ∼αF (F₁ |x|  F₂)  (e₁ , e₂)  (e₁' , e₂')
      ∼αB    :  (xs : List V){S : Sort}{x y : V}
                {G : Functor}{e e' : ⟦ G ⟧ (μ F)}
             →  ((z : V) → z ∉ xs  → ∼αF G  (swapF G S x z e)
                                            (swapF G S y z e'))
                                   →  ∼αF (|B| S G)      (x , e)    (y  , e')

_∼α_  : {F : Functor} → μ F → μ F → Set
_∼α_  = ∼αF |R|
```

Figure 6.26: Alpha equivalence relation.

Finally, in figure 6.28 we prove this lemma for the fold operation as a direct recursive case instance of the previous fold-map lemma.

As a corollary we can directly derive the result in figure 6.29. This lemma states that the fold with contexts operator (figure 6.17) is α-compatible on its context, provided the folded function is α-compatible on its arguments.

We define other relations over our universe in a similar way as we have done previously for the α-equivalence relation. For instance the notOccurBind relation (fig. 6.30) holds if some given variable does not occur in any binder position within a term. Note in this relation we discard the variables sort information. We do so to simplify our next development as we will explain latter.

We finally find useful the following extension of the previous relation to lists of variables, which we call ListNotOccurBind, and holds if all the variables in a given list do not occur in any binder position (associated with any sort) in a term.

```
ListNotOccurBindF  :   {F : Functor}(G : Functor)
                    →  List V → ⟦ G ⟧ (μ F) → Set
ListNotOccurBindF G xs e = All (λ x → notOccurBindF x G e) xs

ListNotOccurBind : {F : Functor} → List V → μ F → Set
ListNotOccurBind {F} xs e = ListNotOccurBindF |R| xs e
```

```
lemma-foldmapfα : {F H : Functor}(G : Functor)
      {f f' : ⟦ F ⟧ (μ H) → μ H}
   → ({e e' : ⟦ F ⟧ (μ H)} → ∼αF F e e' → f e ∼α f' e')
   → (e : ⟦ G ⟧ (μ F)) → ∼αF G (foldmap F G f e) (foldmap F G f' e)
lemma-foldmapfα (|v| S)          p e       = ∼αV
lemma-foldmapfα |1|              p e       = ∼α1
lemma-foldmapfα {F} |R|          p ⟨ e ⟩
   = p        (lemma-foldmapfα F   p e)
lemma-foldmapfα (|E| x)          p e       = ∼αE
lemma-foldmapfα (|Ef| G)         p e       = ρF
lemma-foldmapfα (G₁ |+| G₂) p (inj₁ e)
   = ∼α+₁  (lemma-foldmapfα G₁  p e)
lemma-foldmapfα (G₁ |+| G₂) p (inj₂ e)
   = ∼α+₂  (lemma-foldmapfα G₂  p e)
lemma-foldmapfα (G₁ |x| G₂) p (e₁ , e₂)
   = ∼αx    (lemma-foldmapfα G₁  p e₁)
            (lemma-foldmapfα G₂  p e₂)
lemma-foldmapfα (|B| S    G)   p (x , e)
   = ∼αB [] (λ y _ → lemma∼swapEquivF (lemma-foldmapfα G p e) x y)
```

Figure 6.27: Fold-map function α-compatibility property.

```
lemma-foldfα : {F H : Functor}{f f' : ⟦ F ⟧ (μ H) → μ H}
               → ({e e' : ⟦ F ⟧ (μ H)} → ∼αF F e e' → f e ∼α f' e')
               → (e : μ F) → fold F f e ∼α fold F f' e
lemma-foldfα {F} p e = lemma-foldmapfα |R| p e
```

Figure 6.28: Fold function α-compatibility property.

```
lemma-foldCtxαCtx : {F H C : Functor}
      {f : μ C → ⟦ F ⟧ (μ H) → μ H}{c c′ : μ C}
   → ({e e′ : ⟦ F ⟧ (μ H)}{c c′ : μ C}
      → c ∼α c′ → ∼αF F e e′ → f c e ∼α f c′ e′)
   → c ∼α c′
   → (e : μ F) → foldCtx F f c e ∼α foldCtx F f c′ e
lemma-foldCtxαCtx {F} {f = f} {c} {c′} p c∼c′ e
   = lemma-foldfα (p c∼c′) e
```

Figure 6.29: Fold context function α-compatibility corollary.

Using this relation we are able to prove the lemma in figure 6.31. This lemma states that the fold with context principle is α-compatible on its two arguments if the provided function is α-compatible and well-behaved with respect to swapping. Note that this lemma extends the one given before in figure 6.28, although it requires extra freshness premises, and that the folded function is preserved under swapping.

```
data notOccurBindF (x : V){F : Functor} :
  (G : Functor) → ⟦ G ⟧ (μ F) → Set where
  notOccurBv    :   {m : V}{S : Sort}
                 →  notOccurBindF x (|v| S)          m
  notOccurB1    :   notOccurBindF x |1|              tt
  notOccurBE    :   {B : Set}{b : B}
                 →  notOccurBindF x (|E| B)          b
  notOccurBEf   :   {G : Functor}{e : ⟦ G ⟧ (μ G)}
                 →  notOccurBindF x G e
                 →  notOccurBindF x (|Ef| G )        ⟨ e ⟩
  notOccurBR    :   {e : ⟦ F ⟧ (μ F)}
                 →  notOccurBindF x F e
                 →  notOccurBindF x |R|              ⟨ e ⟩
  notOccurBinj₁ :   {G₁ G₂ : Functor}{e : ⟦ G₁ ⟧ (μ F)}
                 →  notOccurBindF x G₁ e
                 →  notOccurBindF x (G₁ |+| G₂)  (inj₁ e)
  notOccurBinj₂ :   {G₁ G₂ : Functor}{e : ⟦ G₂ ⟧ (μ F)}
                 →  notOccurBindF x G₂ e
                 →  notOccurBindF x (G₁ |+| G₂)  (inj₂ e)
  notOccurBx    :   {G₁ G₂ : Functor}
                    {e₁ : ⟦ G₁ ⟧ (μ F)}{e₂ : ⟦ G₂ ⟧ (μ F)}
                 →  notOccurBindF x G₁ e₁
                 →  notOccurBindF x G₂ e₂
                 →  notOccurBindF x (G₁ |x| G₂)   (e₁ , e₂)
  notOccurBB≢  :  {G : Functor}{e : ⟦ G ⟧ (μ F)}{y : V}{S : Sort}
                 →  x ≢ y → notOccurBindF x G e
                 →  notOccurBindF x (|B| S G)       (y , e)

_notOccurBind_ : {F : Functor}(n : V) → (μ F) → Set
_notOccurBind_ {F} x = notOccurBindF x {F} |R|
```

Figure 6.30: OccurBind relation.

```
lemma-foldCtxα  : {F H C : Functor}{f : μ C → ⟦ F ⟧ (μ H) → μ H}
        {c c' : μ C}{e e' : μ F}
  →  ({e e′ :  ⟦ F ⟧ (μ H)}{c c′ : μ C}
    → c ∼α c′ → ∼αF F e e′ → f c e ∼α f c′ e′)
  → ({c : μ C}{S : Sort}{x y : V}{e : ⟦ F ⟧ (μ H)}
    → f (swap S x y c) (swapF F S x y e) ≡ swap S x y (f c e))
  → ListNotOccurBind (fv c)   e
  → ListNotOccurBind (fv c')  e'
  → c ∼α c' → e ∼α e'
  → foldCtx F f c e ∼α foldCtx F f c' e'
```

Figure 6.31: Fold context α-compatibility property.

6.5.1 Alpha Fold

In this section we present a fold operation that works at the level of α-equivalence classes of terms, that is, it defines α-compatible functions.

First, we introduce the function bindersFreeαElem that takes a list of variables xs and an element e, and returns an α-compatible fresh element, with its binders not in the given list. This freshness function will be useful to reproduce the BVC, which basically states that we can always pick a term with its binders fresh from a given context.

```
bindersFreeαElem : {F : Functor}(xs : List V)(e : μ F)
                   → ∃ (λ e' → ListNotOccurBind {F} xs e')
```

This function has the important property of being strong α-compatible, i.e. it returns the same result for α-convertible terms.

Now, based on this freshness function, we directly implement the α-fold principle in figure 6.32.

```
foldCtxα : {C H : Functor}(F : Functor)
           → (μ C → ⟦ F ⟧ (μ H) → μ H)
           → μ C → μ F → μ H
foldCtxα F f c e = foldCtx F f c (proj₁ (bindersFreeαElem (fv c) e))
```

Figure 6.32: Fold alpha.

This iteration principle first finds a fresh term for a given context c, and then directly applies the fold operation over it. We developed this iteration principle following a different approach from the one taken in chapter 4, where we renamed the binders during the fold traverse. In this definition we chose to separate these two stages in order to reuse the previously defined fold operation and its properties.

We can now properly justify the name "alpha" given to the introduced iteration principle. Firstly, as bindersFreeαElem returns syntactical equal terms when applied to α-convertible terms, we have that our function is trivially strong α-compatible on its last term argument. Secondly, as a direct consequence the lemma already proved for our iteration principle foldCtx in figure 6.29, this new principle inherits its α-compatibility in its context argument from foldCtx, given that the function received is also α-compatible on its arguments. Thus, the presented iteration principle works at the real calculus level when the given function works at the same level of α-equivalence classes.

Now we are able to derive the proper capture avoiding substitution operation for the lambda calculus example by a direct application of the introduced α-fold principle. In fact this definition is exactly the same as the one given before for the naive substitution in figure 6.19, but using now the α-fold operation instead of the fold one.

```
_[_:=_] : λTerm → V → λTerm → λTerm
M [ x := N ]  = foldCtxα λF substaux (⟨ x , N ⟩) M
```

Figure 6.33: Substitution operation.

The classical substitution lemmas stating that substitution is well-behaved with respect to α-conversion (figure 6.34) are directly inherited from the α-compatibility of this iteration principle both on the term and the context to which it is applied . The preceding proof about the α-compatibility in the context argument requires a direct lemma, named lemma-substaux, about the substaux function (figure 6.18), stating that this function is α-compatible.

```
lemma-substα  :   {M M′ N : λTerm}{x : V}
                → M ∼α M′ → M [ x := N ] ≡ M′ [ x := N ]
lemma-substα {M} {M′} M∼M′
   = lemma-foldCtxα-StrongαCompatible {cF} {λF} {λF}
       {substaux} M′ M∼M′

lemma-substα′  :   {x : V}{M N N′ : λTerm}
                 → N ∼α N′ → M [ x := N ] ∼α M [ x := N′ ]
lemma-substα′ {x} {M} (∼αR N∼N′)
   = lemma-foldCtxα-cxtα
       lemma-substaux (∼αR (∼αx ∼αV (∼αEf N∼N′))) M
```

Figure 6.34: Substitution lemmas.

Next lemma in figure 6.35 relates the presented α-fold principle with the previously defined one, giving sufficient conditions under which the two principles return α-convertible terms. First, the folded function must be α-compatible on its two arguments, and also well-behaved with respect to swapping. Secondly, we need a freshness premise stating that the free variables in the context do not occur bound in the applied term.

```
lemma-foldCtxα-foldCtx : {C H : Functor}(F : Functor)
       {f : μ C → ⟦ F ⟧ (μ H) → μ H}{c : μ C}{e : μ F}
   → ({e e′  : ⟦ F ⟧ (μ H)}{c c′ : μ C} → c ∼α c′ → ∼αF F e e′
               → f c e ∼α f c′ e′)
   → ({c     : μ C}{S : Sort} {x y : V}{e : ⟦ F ⟧ (μ H)}
               → f (swap S x y c) (swapF F S x y e) ≡ swap S x y (f c e))
   → ListNotOccurBind (fv c) e
   → foldCtxα F f c e ∼α foldCtx F f c e
```

Figure 6.35: Fold and α-fold relations.

We instantiate the previous lemma to the λ-calculus example in figure 6.36 to obtain a corresponding lemma stating under which conditions the two substitution operations defined (figures 6.19 and 6.33) are α-convertible. Its proof requires two direct lemmas about the substaux function (figure 6.18), the first one stating that it is α-compatible, and the second one stating that it is well-behaved under swapping (which was already used to prove the substitution lemma in figure 6.25).

6.5.2 Alpha Induction Principle

As we did before in chapter 5, in this section we develop an α-induction principle for α-compatible predicates, allowing the user to prove properties for terms with binders fresh from a given context of variables.

We derive this principle following a procedure similar to the one used to infer the principle in section 6.3.2 . Hence, we define the auxiliary function fihα extending a given predicate P over

```
lemmaSubsts : {z : V}{M N : λTerm}
              → ListNotOccurBind (z :: fv N) M
              → M [ z := N ] ∼α M [ z := N ]ₙ
lemmaSubsts {z} {M} {N} nb
= lemma-foldCtxα-foldCtx
   {cF} {λF} λF {substaux} {⟨ z , N ⟩} {M}
   lemma-substaux
   (λ {c} {S} {x} {y} {e} → lemma-substauxSwap {c} {S} {x} {y} {e})
   (fv2ctx {z} {M} {N} nb)
```

Figure 6.36: Substitution operations α-compatibility.

a datatype μ F to a predicate over the datatype ⟦ G ⟧ (μ F) (figure 6.37). The meaning of this predicate is that P holds for every recursive position μ F in a datatype ⟦ G ⟧ (μ F). Besides, this function adds freshness assertions with respect to some given list xs in the recursive and binder cases of its definition: In the binder case, it states that the binder is not in the given list xs, while, in the recursive case, it states that no variable in the list xs occurs in a binder position in its recursive term e.

```
fihα : {F : Functor}(G : Functor)(P : μ F → Set) → List V
   → ⟦ G ⟧ (μ F) → Set
fihα (|v|  S)       P xs n         = ⊤
fihα |1|            P xs tt        = ⊤
fihα (|E|  B)       P xs b         = ⊤
fihα (|Ef| G)       P xs e         = ⊤
fihα |R|            P xs e         = P e       ×
                                     (∀ a → a ∈ xs → a notOccurBind e)
fihα (G₁ |+| G₂)    P xs (inj₁ e)  = fihα G₁  P xs e
fihα (G₁ |+| G₂)    P xs (inj₂ e)  = fihα G₂  P xs e
fihα (G₁ |x| G₂)    P xs (e₁ , e₂) = fihα G₁  P xs e₁  × fihα G₂ P xs e₂
fihα (|B| S  G)     P xs (x , e)   = x ∉ xs    × fihα G  P xs e
```

Figure 6.37: Alpha induction auxiliary function.

We state and prove this principle in figure 6.38.

In order to prove it we proceed in a similar way as done in the proof of the α-fold principle. That is, we firstly use the function bindersFreeαElem (from section 6.5.1) over the parameter e and the freshness context xs to get an α-equivalent term e' with binders not occurring in the list xs. Then we apply the primitive induction principle (fig. 6.14) over the fresh term e' to prove the following predicate P':

$$P'(x) \equiv (\forall c \in xs \Rightarrow c\ notOccurrBind\ x) \Rightarrow P(x)$$

Finally, we apply the proof of predicate P' to the term e' and its freshness hypothesis to obtain that $P\ e'$ must hold. Hence, as the predicate P is α-compatible, and $e \sim_\alpha e'$, we finally get that $P\ e$ should also hold.

The proof of P' is done using the auxiliary lemma lemma-fih∧notOccurBind⇒fihα. This lemma recursively reconstructs a proof of fihα $P\ xs\ e$ given that fih $P\ xs\ e$ holds and that the binders of e do not occur in the context xs. This proof is just a generalisation of the already presented

```
αPrimInd : {F : Functor}(P : μ F → Set)(xs : List V)
   → αCompatiblePred P
   → ((e : ⟦ F ⟧ (μ F)) → fihα F P xs e → P ⟨ e ⟩)
   → ∀ e → P e
αPrimInd {F} P xs aP p e
   with bindersFreeαElem xs e
      | lemma-bindersFreeαAlpha xs e
...   | e' , notBind | e'∼e
   = aP  e'∼e
         (foldInd     F
            (λ e → (∀ c → c ∈ xs → c notOccurBind e) → P e)
            (λ e hi notBind
               → (p  e
                     (lemma-fih∧notOccurBind⇒fihα  {F} F P e xs hi
                        (λ c c∈xs → notOccurBRinv (notBind c c∈xs)))))
            e'
            (get notBind))
```

Figure 6.38: Alpha induction principle.

in chapter 5 for the strengthened α-induction principle for the lambda calculus in figure 5.4. In that chapter we were also able to prove the Church-Rosser theorem for the λ-calculus using an equivalent induction principle. Therefore, we conjecture that following the same procedure we would be able to achieve the confluence of β-reduction result within our generic framework.

6.6 Codification of a BVC proof technique.

In figure 6.39 we show the proof of a result that validates the BVC and usual practices in pen-and-paper proofs within our generic framework. It states that for any α-compatible predicate P, we can prove $P\ e$ for any term e by just proving it for terms whose binders are all different from their own free variables and from the variables in an arbitrary list xs.

```
αProof : {F : Functor}(P : μ F → Set)(xs : List V)
   → αCompatiblePred P
   → ((e : μ F)  → ListNotOccurBind xs       e
                 → ListNotOccurBind (fv e)  e → P e )
   → ∀ e → P e
```

Figure 6.39: BVC proof principle.

Our presentation introduces an explicit premise about the α-compatibility of the predicate being proved, which in general is not explicitly mentioned in informal developments, but is required.

To prove $P\ e$ for arbitrary e we proceed as follows: We first find a fresh enough term e' such that $e' \sim_\alpha e$ using the function bindersFreeαElem. Then, we can use the hypothesis for the fresh term e' to derive that $P\ e'$ holds. Finally, $P\ e$ must also hold, as P is α-compatible. We do not show the code of the proof, since it is similar to others previously presented.

Next we illustrate the use of this result to prove the substitution composition lemma for the System F. First, we prove this lemma for the naive substitution operation. In figure 6.40 we introduce the property to be proved. An extra freshness premise stating that x does not occur bound in the term L is required, since we use the naive substitution.

```
PSCn : {x y : V}{L : FTerm} → FTerm → FTerm → Set
PSCn {x} {y} {L} N M
  =  x ∉ y :: fv L
  →  x notOccurBind L
  →  (M [ x := N ]ₙ) [ y := L ]ₙ ∼α (M [ y := L ]ₙ)[ x := N [ y := L ]ₙ ]ₙ
```

Figure 6.40: Substitution composition property.

We show the proof of this property in figure 6.41.

```
lemma-substCompositionN :  {x y : V}{M N L : FTerm}
                           → PSCn {x} {y} {L} N M
lemma-substCompositionN {x} {y} {M} {N} {L}
  = foldInd tF (PSCn {x} {y} {L} N) lemma-substCompositionNAux M
```

Figure 6.41: Naive substitution composition proof.

The proof is done using the structural induction principle given in figure 6.14. All the inductive cases are proved inside the auxiliary lemma lemma-substCompositionNAux. We show the interesting abstraction case of this auxiliary lemma in figure 6.42.

```
lemma-substCompositionNAux  (inj₂ (inj₂ (inj₁ (t , z , M)))) (_ , hiM)
                            x∉yfvL  xnbL  =
begin
  (ƛ z t M) [ x := N ]ₙ [ y := L ]ₙ
≈⟨ refl                                         ⟩
  ƛ z t (M [ x := N ]ₙ [ y := L ]ₙ)
∼⟨ ∼αR (∼α+₂ (∼α+₂ (∼α+₁
  (∼αx  ρF
        (lemma∼+B (hiM x∉yfvL xnbL)))))) ⟩
  ƛ z t (M [ y := L ]ₙ [ x := N [ y := L ]ₙ ]ₙ)
≈⟨ refl                                         ⟩
  (ƛ z t M) [ y := L ]ₙ [ x := N [ y := L ]ₙ ]ₙ
■
```

Figure 6.42: Abstraction case of the naive substitution composition lemma.

This equational proof is constructed following the usual pen-and-paper practice: First we push the substitution inside the abstraction, then, by the inductive hypothesis we know that the composition of substitutions in the abstraction bodies are α-convertible, and hence we are able to prove that the entire abstraction is α-convertible too, using the auxiliary lemma lemma∼+B. Finally, we push back the substitutions outside the abstraction to conclude the proof.

Now we prove the substitution composition lemma for the capture-avoiding substitution operation using the introduced α-proof technique. We begin by defining the functor describing a triplet of terms TreeTermF. Then, we introduce the predicate PSComp over triplets, stating the composition lemma for the substitution.

```
TreeFTermF =  |Ef| tF |x| |Ef| tF |x| |Ef| tF
TreeFTerm = μ TreeFTermF

PSComp : {x y : V} → TreeFTerm → Set
PSComp {x} {y} ⟨ M , N , L ⟩ = x ∉ y :: fv L
   → (M [ x ≔ N ]) [ y ≔ L ] ∼α (M [ y ≔ L ])[ x ≔ N [ y ≔ L ] ]
```

Figure 6.43: Substitution composition property.

We need to prove that PSComp is α-compatible with respect to triplets of terms. In figure 6.44 we show its proof. It uses pattern-matching on the α-equivalence premise for the triplet in order to obtain the α-equivalence of each of its terms. Then, we carry out an equational proof using basically the previous substitution lemmas and the inductive hypotheses. We have added comments to the proof derivation, explaining each of its steps.

In figure 6.45 we show the core of the proof. It uses the preceding substitution lemmas to replace the classical substitution operations with the naive ones. This can be done because we have freshness premises stating that in the introduced triplet context all binders are different from the free variables in the involved terms, and also from variables x and y. Finally, we work in a very much the same way as we did at the beginning to recover the classical substitutions from the naive ones.

There are many auxiliary lemmas and boilerplate code concerning the freshness premises involved in the last proof which we do not show in this presentation. These are hidden inside auxiliary lemmas as: y:fvL∉bM[x:=N]$_n$ and y:fvL∉bN occurring in the previous proof. The first of these lemmas, for instance, proves that neither the variable y nor the free variable binding in L occur bound in $M[x{:=}N]_n$, which is easy to informally verify from the freshness premises. However, we believe further work is necessary to automatise some of these proofs, or even rewriting the freshness relations in order to alleviate its handling.

Finally, in figure 6.46 we use the introduced α-proof principle with the previous proof obligations to finish the proof.

Note how, by applying the α-proof technique to a triplet of terms, we were able to get sufficient freshness premises to develop a proof similar in structure to pen-and-paper ones in a direct manner. This is possible because in our generic framework we can state the α-equivalence of any structure (triplets in this case), and not just language terms.

```
αCompatiblePSComp : ∀ {x y : V}
   → αCompatiblePred {TreeFTermF} (PSComp {x} {y})
αCompatiblePSComp {x} {y} {⟨ M , N , L ⟩} {⟨ M′ , N′ , L′ ⟩}
                   (∼αR (∼αx M∼M′ (∼αx N∼N′ L∼L′))) PMs x∉y:fvL′
   = begin
       (M′ [ x := N′ ]) [ y := L′ ]
     - Strong α compability of inner substitution operation
     ≈⟨ cong (λ z → z [ y := L′ ]) (lemma-substα (σ M∼M’))    ⟩
       (M  [ x := N′ ]) [ y := L′ ]
     - Strong α compability of outter substitution operation
     ≈⟨ lemma-substα  {M [ x := N′ ]} {M [ x := N ]}
                      (lemma-substα′ {x} {M} (σ N∼N’))        ⟩
       (M  [ x := N  ]) [ y := L′ ]
     - Outter substitution is alpha-compatible in its substituted argument
     ∼⟨ lemma-substα′ {y} {M [ x := N ]} (σ L∼L’)              ⟩
       (M  [ x := N  ]) [ y := L  ]
     - Application of the inductive hypothesis
     ∼⟨ PMs x∉y:fvL                                            ⟩
       (M  [ y := L  ]) [ x := N [ y := L ] ]
     - Strong α compability of inner substitution operation
     ≈⟨ cong (λ P → P [ x := N [ y := L ] ])
             (lemma-substα M∼M’)                               ⟩
       (M′ [ y := L  ]) [ x := N [ y := L ] ]
     - Inner substitution is alpha-compatible in its substituted argument
     ≈⟨ lemma-substα  {M′ [ y := L ]} {M′ [ y := L′ ]}
                      {N [ y := L ]} {x}
                      (lemma-substα′ {y} {M′} L∼L’)            ⟩
       (M′ [ y := L′ ]) [ x := N [ y := L ] ]
     - Strong α compability of substitution operation in subsituted term
     ≈⟨ cong (λ P → (M′ [ y := L′ ]) [ x := P ])
             (lemma-substα N∼N’)                               ⟩
       (M′ [ y := L′ ]) [ x := N′ [ y := L ] ]
     - Outter substitution is alpha-compatible in its substituted argument
     ∼⟨ lemma-substα′ {x} {M′  [ y := L′ ]} {N′ [ y := L ]}
                      (lemma-substα′ {y} {N′} L∼L’)            ⟩
       (M′ [ y := L′ ]) [ x := N′ [ y := L′ ] ]
     ■
```

Figure 6.44: PSComp predicate α-compatibility.

```
αproof : {x y : V}(Ms : μ TreeFTermF)
   → ListNotOccurBind (x :: y :: []) Ms
   → ListNotOccurBind (fv Ms) Ms
   → PSComp {x} {y} Ms
αproof {x} {y} ⟨ M , N , L ⟩ nOcc nOcc2 x∉y:fvL
   =  begin
        (M  [ x := N ])  [ y := L ]
      ≈⟨ lemma-substα  {M [ x := N ]}
                       (lemmaSubsts {x} {M} {N} x:fvN∉bM)      ⟩
        M   [ x := N ]_n   [ y := L ]
      ∼⟨ lemmaSubsts {y} {M [ x := N ]_n} {L} y:fvL∉bM[x:=N]_n   ⟩
        M   [ x := N ]_n   [ y := L ]_n
      ∼⟨ lemma-substCompositionN  {x} {y} {M} {N} {L}
                                  x∉y:fvL x∉bL                 ⟩
        M   [ y := L ]_n   [ x := N [ y := L ]_n  ]_n
      ∼⟨ lemma-substnα′  {x} {M [ y := L ]_n}
                         (σ (lemmaSubsts {y} {N} y:fvL∉bN))     ⟩
        M   [ y := L ]_n   [ x := N [ y := L ]   ]_n
      ∼⟨ σ (lemmaSubsts  {x} {M [ y := L ]_n} {N [ y := L ]}
                         x:fvN[y:=L]∉bM[y:=L]_n)               ⟩
        M   [ y := L ]_n   [ x := N [ y := L ]   ]
      ≈⟨ lemma-substα (σ (lemmaSubsts {y} {M} {L} y:fvL∉bM))  ⟩
        (M  [ y := L  ])  [ x := N [ y := L ]   ]
   ■
```

Figure 6.45: Proof of Substitution composition lemma.

```
lemma-substComposition2  :    {x y : V}{Ms : TreeFTerm}
                           →  PSComp {x} {y} Ms
lemma-substComposition2 {x} {y} {⟨ M , N , L ⟩}
= αProof  (PSComp {x} {y})
          (x :: y :: [])
          (αCompatiblePSComp {x} {y})
          αproof
          ⟨ M , N , L ⟩
```

Figure 6.46: Proof of Substitution composition lemma using α-proof principle.

CHAPTER 7

Conclusions

There exists a great interest in the use of proof assistants for the formalisation of programming languages. However, in spite of the amount of work carried out in this area, the community remains fragmented on the central issue of how to represent the abstract syntax, and in particular, how to represent bound variables. There exists a collection of different name binding representation techniques, with no one coming up as a clear optimal solution. Acknowledging this fact, Aydemir et al. in [3] proposed the POPLmark challenge, a set of tasks designed to evaluate the many proposals, pointing out critical issues that usually arise in the formalisation of programming languages.

Our work goes after those criteria, looking for a representation as close to pencil-and-paper informal practice as possible, while remaining formal. The usual informal procedure consists in identifying terms up to α-conversion. However, this is not easily carried out when functions are defined by recursion and properties are proven by induction over abstract syntax terms. The problem has to do with the fact that the consideration of the α-equivalence classes is in general actually conducted through the use of convenient representatives. These are chosen by the so-called Barendregt variable convention (BVC [4]), which basically states that we can consider each term to have bound names different from names free in the context of a proof. This usual informal practice collides with the primitive induction principles automatically derived from the inductive definition of the abstract syntax by nowadays proof assistants, which require proof obligations quantified over arbitrary terms, and not over α-equivalence classes, or as in the informal practice, over convenient representatives.

In chapter 3 we pursued the traditional approach to the λ-calculus metatheory, treating the calculus in its original syntax with one sort of names (for both free and bound variables), with an α-conversion definition based on the substitution operation, and working all the time with concrete terms, i.e. without identifying terms up to α-conversion. We have corroborated that this approach can be carried out in a completely formal way, so as to scale up to the principal results of the theory. This is possible by the use of Stoughton's multiple substitution operation, introduced in [61]. We believe our main contribution consists in presenting Stoughton's theory of substitutions in a new way, based upon the notion of restriction of a substitution to the free variables of a term. Because of this we can use the corresponding finite notions of equality and α-equivalence, whereas Stoughton and other formalisations (Lee, [33]) use extensional, and thus generally undecidable, equality. Moreover, we define a strictly syntax-directed inductive definition of the α-equivalence relation, which is easily proven to be an equivalence relation, and therefore a congruence. This stands in contrast to Stoughton's work, which starts with a definition of α-conversion as the least congruence generated by a simple renaming of the bound variable –a definition comprising six rules, whereas ours consists of three. Stoughton's whole development is directed towards characterising α-conversion in the form of a syntax-based definition that contains nevertheless two rules corresponding to abstractions. This therefore gives a neat result standing in correspondence with ours; but the proof is surprisingly dilatory, requiring among others the substitution lemma for α-conversion. The issue manifests itself also in a rather involved character of Lee's formalisation, as witnessed by his own comments.

Within the general approach to syntax chosen, the main work to compare with ours is Vestergaard and Brotherston's [68], which is indeed quite successful in using modified rules of α-conversion and β-reduction, with added freshness premises. They work with the naive substitution operation, i.e. at the raw calculus level, to formally prove the Church-Rosser theorem at the real calculus level in Isabelle-HOL. However, this mixture of levels results in a rather complicated proof of the β-confluence property. We have completed a development equivalent to theirs in scope, but based on a capture-avoiding multiple substitution operation, working always at the raw calculus level.

In chapter 4 we have explored an approach based on the simpler swapping operation, from which we have derived principles of induction and recursion allowing us to work on α-classes of λ-terms. The crucial component is what we have called a BVC-like induction principle. This principle allows us to choose the bound name in the case of the abstractions so that it does not belong to a given list of names. This principle (for α-compatible predicates) is derived from ordinary structural induction on concrete terms, thus avoiding any form of induction on the size of terms, or other more complex forms of induction. On the other hand, it gives rise to principles of recursion that allow to define functions on α-classes, specifically, functions giving identical results for α-equivalent terms.

Our work departs from Pitts and Gabbay's [24, 53] because we do fix the choice of representatives for implementing the alpha-structural recursion thereby forcing this principle to yield identical results for α-equivalent terms. This might be a little too concrete but, on the other hand, it gives us the possibility of completing a simple full implementation on an existing system, as different from other works which base themselves on postulates or more sophisticated systems of syntax, or methods of implementation.

There exist a continuous line of work studying name based formalisations of the λ-calculus including structural induction principles, where the abstraction case needs to be proved for only fresh binders, that is, trying to resemble the BVC. However, we depart from these works as we stick to a first order syntax while these works are based in variations of the HOL technique.

In chapter 5 we have proposed a novel strengthened alpha-induction principle on λ-terms that allows us to emulate the BVC in the proof of the substitution lemma for the parallel β-reduction, which was not possible with the original formulation. Using this principle, our formalisation scales up to the β-confluence theorem. We have also proved the preservation of typing under α-equivalence and β-reduction for the simply typed λ-calculus à la Curry using our induction principle. Besides, for the crucial substitution lemma for the typing relation, we are able to carry out the subject reduction theorem using the original alpha induction principle, since our typing contexts can contain repeated variables.

In summary, we have proved the β-reduction confluence and the subject reduction theorems for the λ-calculus within two completely different formalisations in Agda. In spite of the fact that the first one sticks more closely to the usual practice of using the substitution operation in the definition of the crucial α-conversion relation, the development required a very subtle notion of restriction of a substitution to the free variables of a term. Indeed, it was a nice novel introduction, although it departs from usual pen-and-paper proofs. On the other hand, our second experiment using an α-conversion definition is based on the simpler swapping operation, and in this way it departs from the classical metatheory. But at the same time, it reaches a great level of abstraction by being able to mimic the BVC technique through the use of the introduced α-induction principles. Those principles allow us to hide most of the complexity introduced by the named syntax.

With the gained experience after applying nominal techniques within the λ-calculus framework, in the last chapter we have addressed the formalisation of a general first order named syntax with multi-sorted binders. We have done so by applying a combination of generic programming and nominal techniques to derive fold operations, variable swapping, the α-conversion relation, and α-induction/iteration principles for any language abstract syntax with binders. We have successfully derived the λ-calculus and System F as instances of the introduced general framework. For these examples we were able to derive both the naive and the capture-avoiding substitutions as direct instances of the corresponding fold and α-fold principles. We directly inherited the classical substitution lemmas for the α-conversion, and the good behavior of substitution under the swapping operation from fold properties already proved. We also proved at the general level a lemma stating sufficient conditions under which the fold and α-fold functions are α-equivalent. Therefore, as substitution operations are direct instances of these iteration principles, we get in an almost free manner a result about the relation between the naive and the capture-avoiding substitution operations for the λ-calculus and System F. This result is particularly useful in the proof of the crucial substitution composition lemma, which is conducted using an introduced α-proof technique, that enables use to mimic the BVC in a more general way than the one explored in previous chapters through the use of term induction.

Further Work

We believe a more systematic comparison of the many works addressing the binding issue in the context of proof assistants is required. However, such work constitutes in itself a methodological challenge, among other things requiring the mastering of several proof assistants. Because of this we have not pursued this goal in the present work.

It would be interesting to extend our presented formalisations with more results of the metatheory of λ-calculus, in order to study how well our formalisations scale up with these extensions. In particular, the Standarisation Theorems, and the Normalisation and Strong Normalisation Theorems for the system of assignment of simple types would be important extensions to be addressed in future work.

Another possible extension is developing alpha induction and iteration principles, as we have done using nominal techniques, based on Stoughton's multiple substitution. Although our principles were based on swapping operation, we think it is feasible to interchange this operation with Sthoughton's multiple substitution, and recover the same principles. In this way, we believe some final proofs can be carried out in a simpler manner, in particular, avoiding some explicit renaming substitutions within them.

The presented definitions of parallel reduction relation in chapters 3 and 5 differ from pen-and-paper original ones. Indeed, we had to make them α-compatible by definition, that is, they must be preserved by α-conversion, whereas in pen-and-paper works this is not necessary because it is usually accepted to equate α-convertible terms, in spite of the fact of being working with raw terms. Because of this, our parallel reduction definition had to explicitly introduce the independence from any particular choice of binder names in its definition. This is reasonable as we are ultimately trying to prove properties at the real calculus level while working at the raw one. However, the modifications in the definitions of the relations introduce noise in the rest of the formalisation, departing from the usual informal practice. For example, in chapter 3 we must introduce the concept of the postponement of the α-conversion introduction rule in the presented parallel reduction, whereas, in chapter 5, we have introduced an alternative parallel relation definition that includes the α-conversion definition rules in it. However, in this last case, after proving that the relation is α-compatible, we have been able to recover the usual definition of the parallel reduction through the introduction of suitable inversion lemmas, which allowed us to continue the formalisation as if our definition were Takahasi's one.

At the end of the last chapter we continue working at the raw level, but under BVC premises, as in the proof of the substitution composition lemma. We have continued exploring this line of work within our generic framework, studying alternative solutions to the previously described problems in the definition of the parallel relation, which be believe could emulate the elegant development of Barendregt, scaling it up to the Church-Rosser confluence result. As it can derived from Hindley's lemma 1.20 in [29], we can work under BVC preconditions at the raw level, using the naive substitution, and then lift the results to the real level replacing term's identity by α-equivalence relation, and naive substitution for the capture avoiding one, and all theorems at will stay true. The explored line of work follows this idea, working as far as possible under BVC premises, at the raw level but under enough freshness premises. This freshness premises allows us to work with the simpler naive substitution operation and syntactical equality, keeping the formalisation simpler, as no renamings are done by the naive substitution, while the freshness premises prevents from any variable capture occurrence.

We would like to continue developing the presented binding generic framework and developing the previously discussed line of work. We are particular interested in studying the possibility of automating certain proofs using the Agda's *reflection mechanism.* In [67] it is explored recent additions to the Agda proof assistant enabling reflection, in the style of Lisp and Template Haskell's compile time meta-programming. Basically, Agda's reflection mechanisms make it possible to convert a program fragment into its corresponding abstract syntax tree and vice versa. The idea behind proof by reflection is: given that type theory is both a programming language and a proof system, it is possible to define functions that compute proofs. Therefore, reflection introduces the possibility of mechanically constructing a proof of a theorem by inspecting its shape. First, we can directly use this reflection mechanism to provide an automatic isomorphism between an user given datatype with binders annotations, to a codification in our generic programming universe. In this way we would be able to avoid the boilerplate code introduced by the codes of our generic universe. This solution is provided for a similar universe within the Coq proof assistant in [34], whereas in [67] an isomorphism is given for some fixed Agda's datatype, showing this is feasible nowadays in Agda. However, reflection mechanism seems not to be still a stable feature in Agda, and it has suffered several modifications in the lasts Agda releases, which has restrained us from its use in our development. Secondly, we believe that the overhead introduced by proof obligations about freshness premises can be automatised using reflection. By doing so our framework would be able to mimic common practice without any overhead. It also remains studying the possible rephrasing of the various freshness relations introduced in the generic framework, in order to alleviate its burden, and by the way facilitate the desired level of automatisation.

Bibliography

[1] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. *Electronic Notes in Theoretical Computer Science*, 174(5):69–77, June 2007.

[2] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering Formal Metatheory. *ACM SIGPLAN Notices*, 43(1):3–15, January 2008.

[3] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pages 50–65, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] Hendrik Barendregt. *The λ-calculus Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, revised edition, 1984.

[5] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, December 2003.

[6] Ana Bove and Peter Dybjer. Language engineering and rigorous software development. chapter Dependent Types at Work, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009.

[7] Arthur Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.

[8] James Cheney. Scrap your Nameplate (Functional Pearl). In *ICFP 2005*, pages 180–191. ACM, 2005.

[9] Adam Chlipala. A certified type-preserving compiler from λ-calculus to assembly language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 54–65, New York, NY, USA, 2007. ACM.

[10] Alonzo Church. A set of postulates for the foundation of logic part I. *Annals of Mathematics*, 33(2):346–366, 1932.

[11] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[12] Ernesto Copello. Inferencia de tipos de sesión. Master's thesis, Universidad ORT Uruguay, 2012.

[13] Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove, and Maribel Fernández. Alpha-structural induction and recursion for the λ-calculus in constructive type theory. *Electronic Notes in Theoretical Computer Science*, 323:109 – 124, 2016.

[14] Ernesto Copello, Nora Szasz, and Álvaro Tasistro. Formal metatheory of the lambda calculus using Stoughton's substitution. *Theoretical Computer Science*, 685:65 – 82, 2017.

[15] T. Coquand. An Algorithm for Testing Conversion in Type Theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, Cambridge, 1991.

[16] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, 1958. Second printing 1968.

[17] N. G. de Bruijn. λ-calculus Notation with Nameless Dummies, A Tool for Automatic Formula Manipulation, with Applications to the Church-Rosser Theorem. *Indagationes Mathematicae (Koninglijke Nederlandse Akademie van Wetenschappen)*, 34(5):381–392, 1972.

[18] Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-Order Abstract Syntax in Coq. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *TLCA*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 1995.

[19] Peter Dybjer. Inductive sets and families in martin-löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[20] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical logic (2. ed.).* Undergraduate texts in mathematics. Springer, 1994.

[21] Jonathan M. Ford and Ian A. Mason. Operational Techniques in PVS - A Preliminary Evaluation. *Electronic Notes in Theoretical Computer Science*, 42:124–142, 2001.

[22] G. Frege. *Begriffsschrift, eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens.* Halle, 1879. English translation in From Frege to Gödel, a Source Book in Mathematical Logic (J. van Heijenoort, Editor), Harvard University Press, Cambridge, 1967, pp. 1–82.

[23] Murdoch J. Gabbay. Foundations of nominal techniques: Logic and semantics of variables in abstract syntax. *The Bulletin of Symbolic Logic*, 17(2):161–229, 2011.

[24] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3—5):341—363, July 2001.

[25] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1969. Edited by M. E. Szabo.

[26] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[27] Andrew D. Gordon. A Mechanisation of Name Carrying Syntax up to Alpha Conversion. In *Proceedings of Higher Order Logic Theorem Proving and its Applications*, Lecture Notes in Computer Science, pages 414–426, 1993.

[28] Andrew D. Gordon and Thomas F. Melham. Five Axioms of Alpha-Conversion. In *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, pages 173–190, 1996.

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

[30] Peter V. Homeier. A Proof of the Church-Rosser Theorem for the λ-calculus in Higher Order Logic. In *TPHOLs'01: Supplemental Proceedings*, pages 207–222, 2001.

[31] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *In ESOP'98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag.

[32] Jean-Louis Krivine. *Lambda-Calculus, Types and Models.* Ellis Horwood series in computers and their applications. Masson, 1993.

[33] Gyesik Lee. Proof Pearl: Substitution Revisited, Again. Hankyong National University, Korea. http://formal.hknu.ac.kr/Publi/Stoughton.pdf.

[34] Gyesik Lee, Bruno C. D. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. *GMeta: A Generic Formal Metatheory Framework for First-Order Representations*, pages 436–455. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[35] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[36] Daniel R. Licata and Robert Harper. A universe of binding and computation. In *International Conference on Functional Programming (ICFP)*, pages 123–134, September 2009.

[37] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes.* Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[38] Helmut Schwichtenberg Masahiko Sato, Randy Pollack and Takafumi Sakurai. Viewing Lambda-terms Through Maps, 2013. Available from http://homepages.inf.ed.ac.uk/rpollack/export/Maps_SatoPollackSchwichtenbergSakurai.pdf.

[39] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2012. Version 8.0.

[40] J. McKinna and R. Pollack. Some λ-calculus and Type Theory Formalized. *Journal of Automated Reasoning*, 23(3–4), November 1999.

[41] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. Technical report, Durham, NC, USA, 1987.

[42] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.

[43] Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs*, TYPES'04, pages 252–267, Berlin, Heidelberg, 2006. Springer-Verlag.

[44] Tobias Nipkow. Programming and Proving in Isabelle/HOL, 2016. Available from https://isabelle.in.tum.de/doc/prog-prove.pdf.

[45] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

[46] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[47] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[48] Michael Norrish. Recursive Function Definition for Types with Binders. In *In Seventeenth International Conference on Theorem Proving in Higher Order Logics*, pages 241–256, 2004.

[49] H. Pfeifer and H. Ruess. Polytypic abstraction in type theory. In Roland Backhouse and Tim Sheard, editors, *Workshop on Generic Programming (WGP'98).* Dept. of Computing Science, Chalmers Univ. of Technology, and Göteborg Univ., June 1998.

[50] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

[51] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[52] A. M. Pitts. Nominal Techniques. *ACM SIGLOG News*, 3(1):57–72, January 2016.

[53] Andrew M. Pitts. Nominal Logic, a First 0rder Theory of Names and Binding. *Information and Computation*, 186(2):165–193, 2003.

[54] Andrew M. Pitts. Alpha-Structural Recursion and Induction. *Journal of the ACM*, 53(3):459–506, May 2006.

[55] Randy Pollack. Closure under alpha-conversion. In *Proceedings of the International Workshop on Types for Proofs and Programs*, TYPES '93, pages 313–332, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[56] Dag Prawitz. *Natural Deduction: a Proof-Theoretical Study.* Number 3 in Stockholm Studies in Philosophy. Almquist and Wiskell, 1965.

[57] György E. Révész. *Lambda-Calculus, Combinators and Functional Programming*, volume 4 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1988.

[58] Carsten Schurmann, Joelle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.*, 266(1-2):1–57, September 2001.

[59] Mark R. Shinwell. Fresh o'caml: Nominal abstract syntax for the masses. *Electronic Notes in Theoretical Computer Science*, 148(2):53 – 77, 2006.

[60] Mark R. Shinwell, Andrew M. Pitts, and Murdoch James Gabbay. Freshml: programming with binders made simple. *SIGPLAN Notices*, 38(9):263–274, 2003.

[61] A. Stoughton. Substitution revisited. *Theor. Comput. Sci.*, 59:317–325, 1988.

[62] M. Takahashi. Parallel Reductions in λ-Calculus. *Information and Computation*, 118(1):120 – 127, 1995.

[63] Alvaro Tasistro, Ernesto Copello, and Nora Szasz. Principal type scheme for session types. *International Journal of Logic and Computation*, 3(1):34–43, 2012.

[64] Alvaro Tasistro, Ernesto Copello, and Nora Szasz. Formalisation in Constructive Type Theory of Stoughton's Substitution for the λ-calculus. *Electronic Notes in Theoretical Computer Science*, 312:215–230, 2015.

[65] Christian Urban and Michael Norrish. A formal treatment of the Barendregt variable convention in rule inductions. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning About Languages with Variable Binding*, MERLIN '05, pages 25–32, New York, NY, USA, 2005. ACM.

[66] Christian Urban and Christine Tasson. Nominal Techniques in Isabelle/HOL. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer Berlin Heidelberg, 2005.

[67] Paul van der Walt and Wouter Swierstra. *Engineering Proof by Reflection in Agda*, pages 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[68] René Vestergaard and James Brotherston. A Formalised First-Order Confluence Proof for the λ-Calculus using One-Sorted Variable Names. *Information and Computation*, 183(2):212–244, 2003.

[69] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73 – 93, 2007.