M1 d'Informatique Fondamentale
École Normale Supérieure de Lyon

Internship report
Chantal Keller

Substitutions for simply typed λ-calculus

Parallel and hereditary substitutions

Abstract

Substitutions play an important role in simply typed λ-calculus, as a key part of the reduction rules, which make this calculus be strongly normalising.

Parallel substitutions are a nice presentation of substitutions as first-class objects of the calculus, and make it easy to adapt proofs to proof assistants. We will also deal with hereditary substitutions, a new approach to normalisation that is structurally recursive.

During this internship, we implemented these two different kinds of substitutions using the proof assistant Agda, and proved that parallel substitutions form a category with finite products, and that the $\beta\eta$-equivalence was decidable using hereditary substitutions.

Keywords

functional programming — $\beta\eta$-equivalence — substitutions
simply typed λ-calculus — decidability

Supervisors: Thorsten Altenkirch (Reader at the School of Computer Science and Information Technology of the University of Nottingham)
Nicolas Oury (Marie Curie research fellow at the School of Computer Science of the University of Nottingham)

June, 5th 2008 - August, 13th 2008

Contents

Thanks

I would like to thank my supervisors, for their kindness and availability. I also thank the whole Functional Programming team of the University of Nottingham for their welcome !

Introduction

Simply typed λ-calculus and substitutions

The λ-calculus was introduced in the 1930s. It is a powerful while concise model of computation, but it is logically unsound. The simply typed λ-calculus is a subset of λ-calculus that is sound and strongly normalising.

Substitutions play an important role in λ-calculus and in its normalisation. Classically, substitution is central in the β-rule:

$$(\lambda x.f)\ t \to f[x := t]$$

Here f and t represent two terms, and $f[x := t]$ stands for f in which all the free occurrences of x are replaced by t.

There are different approaches of substitutions. Historically, point-wise substitutions were first introduced, but other presentations of the same feature are useful to give different points of view and to put different lights over substitutions.

We will focus on two different kinds of substitutions. Parallel substitutions form an elegant approach, giving an abstraction of substitutions as first-class objects of the calculus. On the other side, hereditary substitutions, implemented using point-wise substitutions, give an algorithm for normalisation that has the nice property to be structurally recursive.

Background of the internship

It was a research internship about type theory and functional programming in the Functional Programming laboratory of the School of Computer Science in Nottingham, United Kingdom. It lasted ten weeks.

The objective of the internship was to study different approaches of substitutions for simply typed λ-calculus, and to prove important properties about those substitutions using the proof assistant Agda. Hence we first had to implement substitutions and the properties we wanted to prove in this language, and then to perform the proofs.

This work is part of a global work that consists in the implementation of functional features and proofs using proof assistants. As a result, it justifies the development of such tools: on the one hand, it shows the interest of this kind of software both for development and utilisation, and on the other hand, it gives a guide for its development. We had to think about the implementation of substitutions in a certain proof assistant, whereas substitutions previously remained mostly quite hand-made.

It is also a global view of what can be nowadays done about substitutions, presenting both parallel substitutions, which are now well established and known to have good properties, and hereditary substitutions, a recent approach that gains to be known as it offers a solution to some problems about normalisation.

The work was done using the last realease of Agda 2 [1], a proof assistant and a programming language based on dependant types. It is a very useful tool to perform proofs concerning functional programming, thanks to its approach based on terms (contrary to other proof assistants such as Coq, based on logical progression) and its syntax closed to functional programming languages such as Haskell and OCaml.

We chose to present our work using a formal presentation (for instance, inference rules) rather than an Agda presentation: indeed, our work is very general and independent from Agda's syntax and possibilities. However, the Agda implementation was a main part of the work we did during this internship; as a result, we will present Agda's syntax and main pieces of our code in appendix D.

1 The simply typed λ-calculus

The λ-calculus is a formal system designed to investigate functions definition and application (including recursion) [2]. It was introduced by Alonzo Church and Stephen Kleene in the beginning of the twentieth century as a means to describe the foundations of mathematics, but soon became a paradigm of programming called functional programming. Although this system is inconsistent, it is a powerful model to describe recursive functions.

Soon after, types over λ-calculus were introduced by Haskell Curry then by Alonzo Church [3]. They are syntactic objects that can be assigned to some of the λ-terms. A type can be seen as a specification of a program or, in a logical point of view, as a proposition whose proofs are the λ-terms which have this type.

Here we will only focus on simply typed terms, that is to say the types will only be base types (elements of a finite set) and function types (a relation between two types usually written using an arrow). The simply typed λ-calculus is related to minimal logic through the Curry-Howard isomorphism: the types inhabited by close terms are exactly the tautologies of minimal logic.

We will use a directly typed syntax, rather than first defining pure λ-calculus and then adding types, as it is commonly done. This prevents us from defining terms we are not interested in, and gives a more natural and easy-to-manipulate presentation of typed terms. To define those terms, we will use De Bruijn notation (see section 1.2 for details).

In section 1.2, we will define our notations for terms of the simply typed λ-calculus. In section 1.3, we will introduce two reduction rules, the β-reduction and the η-reduction, and see how we can define normal forms according to this set of rules. In section 1.4, we will see how to transform these rules into an equivalence relation between terms: the $\beta\eta$-equivalence. But first, we will define propositional equality between two elements of a same set.

1.1 Propositional equality

The propositional equality between two elements of a set A is an inductive data-type that has only one constructor, that is the reflexivity (one element is equal to itself):

$$\frac{a, b : A}{a ==_A b : \text{Set}} \qquad \frac{}{==_A -refl : a ==_A a}\ \text{REFL}$$

This equality is an equivalence: one can prove easily that it is symmetric and transitive by pattern matching on the proof (as the only constructor is $==_A -refl$).

We can prove the same way that this equality is compatible with all the constructors we are going to define in the following parts.

1.2 Notations for the λ-calculus

As we consider only simply typed λ-calculus, types are defined using an inductive data-type with two constructors (we have one single base type):

$$\frac{}{Ty : \mathrm{Set}} \qquad \frac{}{\circ : Ty}\ \text{BASE} \qquad \frac{\sigma, \tau : Ty}{\sigma \to \tau : Ty}\ \text{FUN}$$

To represent terms, we will use the de Bruijn notation [4]. In this latter, variables are represented by integers which stand for the distance to the binding constructor λ. It prevents troubles caused by the constant need to rename bound variables, for instance during substitutions. In this system, α-conversion is not required.

As a result, contexts under which terms are going to be typed are lists of the types of free variables in the terms, in the order corresponding to the numbering of those variables.

$$\frac{}{Con : \mathrm{Set}} \qquad \frac{}{\varepsilon : Con}\ \varepsilon \qquad \frac{\Gamma : Con \qquad \sigma : Ty}{\Gamma, \sigma : Con}\ \text{EXT}$$

Hence, we can define variables and terms as follows.

$$\frac{\Gamma : Con \qquad \sigma : Ty}{Var\ \Gamma\ \sigma : \mathrm{Set}} \qquad \frac{}{vz : Var\ (\Gamma, \sigma)\ \sigma}\ \emptyset \qquad \frac{x : Var\ \Gamma\ \tau}{vs\ x : Var\ (\Gamma, \sigma)\ \tau}\ \text{WEAK}$$

$$\frac{\Gamma : Con \qquad \sigma : Ty}{Tm\ \Gamma\ \sigma : \mathrm{Set}} \qquad \frac{v : Var\ \Gamma\ \sigma}{var\ v : Tm\ \Gamma\ \sigma}\ \text{VAR} \qquad \frac{t : Tm\ (\Gamma, \sigma)\ \tau}{\lambda t : Tm\ \Gamma\ (\sigma \to \tau)}\ \lambda$$

$$\frac{t_1 : Tm\ \Gamma\ (\sigma \to \tau) \qquad t_2 : Tm\ \Gamma\ \sigma}{app\ t_1\ t_2 : Tm\ \Gamma\ \tau}\ \text{APP}$$

1.3 β-reduction, η-expansion and normal forms

The β-reduction and the η-expansion are two rules over λ-calculus terms. They can be seen as computational rules, and make the simply typed λ-calculus be strongly normalising according to these rules (and verify the subject reduction). The β-reduction rule needs to first define substitutions over terms.

1.3.1 Substitutions

A substitution in λ-calculus is a way to substitute a free variable x of a term t for another term u (that does not contains x as a free variable). In simply typed λ-calculus, we impose that x and u has the same type, so that the resulting term would also be a typed term.

As we will see in sections 2 and 3, there are different approaches to define substitutions. According to the implementations and the proofs one wants to perform, one way or another is preferable.

Here, we will present explicit substitutions with a syntactic approach, so it can be instantiated using parallel substitutions (see section 2) or point-wise substitutions (see section 3).

Here is the defining rule of a substitution (we suppose that $x \notin FV(u)$):

$$\frac{t : Tm\ \Gamma\ \sigma \qquad x : Var\ \Gamma\ \tau \qquad u : Tm\ \Gamma'\ \tau}{t[x := u] : Tm\ \Gamma'\ \sigma}\ \text{SUBST}$$

The next two sections give two different approaches of the relationship between Γ and Γ'. Substitution must respect the following rules:

$$\begin{cases} (var\ x)[x := u] = u & (var\ y)[x := u] = var\ y \text{ if } y \neq x \\ (\lambda t)[x := u] = \lambda t[vs\ x := weak\ u] & (app\ t_1\ t_2)[x := u] = t_1[x := u]\ t_2[x := u] \end{cases}$$

We let away how to define $weak\ u$, as it depends on the kind of substitution we use. It matches the following definition:

$$\frac{t : Tm\ \Gamma\ \sigma}{weak\ t : Tm\ (\Gamma, \tau)\ \sigma}\ \text{WEAK}$$

1.3.2 β-reduction

β-reduction expresses the idea of function application. The rule is defined as follows:

$$app\ (\lambda t)\ u \to t[vz := u]$$

Inside a term, a sub-term which matches the form $app\ (\lambda t)\ u$ is called a β-redex. A normal form according to this rule must not contain β-redexes.

1.3.3 η-expansion

η-expansion expresses the idea of extensionality, that is to say two functions are the same if and only if they give the same result applied to the same argument. The rule is defined as follows:

$$t \to \lambda(app\ (weak\ t)\ (var\ vz))$$

t must have a function type $(\sigma \to \tau)$, and the term inside the λ-abstraction has type τ. A normal form according to this rule must be as much η-expanded as possible so that the term inside all the λ-abstractions would have base type $(\circ)$.

1.3.4 Normal forms

We quickly saw what normal forms according to these rules looked like. One can prove that normal forms are of two kinds:

- the λ-abstraction of a normal form;
- the application of a variable to normal forms so that the result would have base type (neutral terms).

As a result, normals forms form a subset of the simply-typed λ-calculus we can formally define:

$$\frac{\Gamma : Con \qquad \sigma : Ty}{Nf\ \Gamma\ \sigma : \text{Set}} \qquad \frac{t : Nf\ (\Gamma, \sigma)\ \tau}{\lambda n\ t : Nf\ \Gamma\ (\sigma \to \tau)}\ \lambda\text{N} \qquad \frac{x : Var\ \Gamma\ \sigma \qquad \vec{ts} : Sp\ \Gamma\ \sigma\ \circ}{x\ \vec{ts} : Nf\ \Gamma\ \circ}\ \text{NE}$$

We introduce the set of spines, which are successions of normal forms:

$$\frac{\Gamma : Con \qquad \sigma, \tau : Ty}{Sp\ \Gamma\ \sigma\ \tau : \text{Set}} \qquad \frac{}{\vec{\varepsilon} : Sp\ \Gamma\ \sigma\ \sigma}\ \vec{\varepsilon} \qquad \frac{t : Nf\ \Gamma\ \sigma \qquad \vec{ts} : Sp\ \Gamma\ \tau\ \rho}{t, \vec{ts} : Sp\ \Gamma\ (\sigma \to \tau)\ \rho}\ \text{EXT}$$

Spines are represented using two types: if $\vec{ts} : Sp\ \Gamma\ \sigma\ \tau$, σ represents the type of the variable you must feed the spine with, obtaining that way an element of type τ. This latter will be a normal form only if $\tau == \circ$.

1.4 $\beta\eta$-equivalence between terms

Considering the β-reduction and the η-expansion both senses, we can define a relation between terms. We want this relation to be an equivalence, so we need the three axioms of an equivalence (reflexivity, symmetry, transitivity); we also need it to be compatible with the term constructors λ and *app*. Hence we get the $\beta\eta$-equivalence:

$$\frac{t, u : Tm\ \Gamma\ \sigma}{t \equiv u : \text{Set}} \qquad \frac{}{refl : t \equiv t}\ \text{REFL} \qquad \frac{p : t \equiv u}{sym\ p : u \equiv t}\ \text{SYM} \qquad \frac{p_1 : t \equiv u \qquad p_2 : u \equiv v}{trans\ p_1\ p_2 : t \equiv v}\ \text{TRANS}$$

$$\frac{p : t \equiv u}{cong\lambda\ p : \lambda t \equiv \lambda u}\ \text{CONG}\lambda \qquad \frac{p_1 : t_1 \equiv u_1 \qquad p_2 : t_2 \equiv u_2}{congApp\ p_1\ p_2 : app\ t_1\ t_2 \equiv app\ u_1\ u_2}\ \text{CONGAPP}$$

$$\frac{}{beta : app\ (\lambda t)\ u \equiv t[vz := u]}\ \beta \qquad \frac{}{eta : \lambda(app\ (weak\ t)\ (var\ vz)) \equiv t}\ \eta$$

We will see in section 3 one proof of the decidability of this equivalence.

2 The category of parallel substitutions in simply typed λ-calculus

In this section, we will focus on one possible definition of substitutions: parallel substitutions. A substitution of this kind transforms a term typed in a context Δ into a term typed in a context Γ, by replacing each free variable of Δ with a term in Γ.

We will define the identity substitution *id* and the composition of two substitutions, written using the symbol $\circ$. We intend to prove that those substitutions form a category, which means we have to prove the following three theorems:

$$\begin{cases} id \circ s & == & s & \text{(neutrality on the left)} \\ s \circ id & == & s & \text{(neutrality on the right)} \\ (u \circ v) \circ w & == & u \circ (v \circ w) & \text{(associativity)} \end{cases}$$

for any substitutions s, u, v and w. We also want to prove that this category has finite products.

In section 2.1, we will define this kind of substitutions, both for variables and terms, with some of the needed features (weakening a substitution and composition). Section 2.2 will be devoted to definitions specific to variable substitutions (the complete proof of the fact that variable substitutions form a category with finite product is available in appendix A). In section 2.3, we are going to explain the way we adapted proofs about variable substitutions to proofs about term substitutions.

All the proofs have been checked using the proof assistant Agda [1]. The source code and a report explaining it are available online [5]. We exploit as much as possible the code factorization ideas from [6].

The fact that substitutions in simply typed λ-calculus form a category which has finite products is not new. The interest of our study is nonetheless multiple:

- we use a directly typed syntax for λ-calculus;
- we present parallel substitutions, an elegant view of substitutions;
- all the proofs are checked in Agda, contributing to the interest of developing such tools as proof assistants.

2.1 Parallel substitutions

A parallel substitution in λ-calculus is a way to transform a term which has free variables in a context Δ into a term which has free variables in another context Γ. To do so, it substitutes each variable in Δ by a term in Γ. In simply typed λ-calculus, we impose that the term in Γ has the same type as the variable it substitutes, so that the resulting term would also be a typed term.

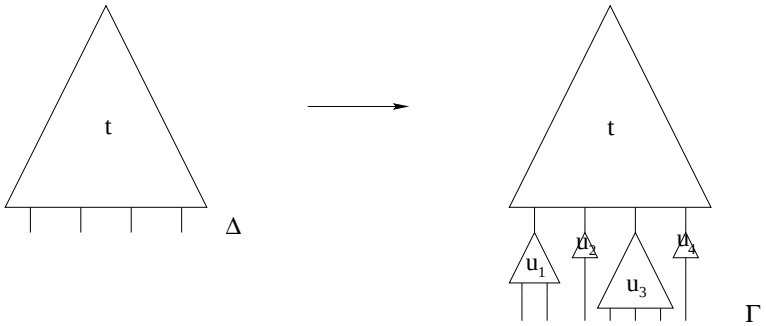

Figure 1: An example of the application of a parallel substitution (terms are represented with trees whose nodes are constructors and leafs are free variables).

2.1.1 Definition

Substitutions can be defined both for variables and for terms. Given a prototype $T : Con \to Ty \to Set$ (which will be instantiated as Var or Tm), substitutions over T are hence defined

as follows:

$$\frac{\Gamma, \Delta : Con}{Subst\ T\ \Gamma\ \Delta : \text{Set}} \qquad \frac{}{\varepsilon : Subst\ T\ \Gamma\ \varepsilon}\ \varepsilon \qquad \frac{s : Subst\ T\ \Gamma\ \Delta \qquad t : T\ \Gamma\ \sigma}{s, t : Subst\ T\ \Gamma\ (\Delta, \sigma)}\ \text{EXT}$$

$$\frac{s : Subst\ T\ \Gamma\ \Delta \qquad t : T\ \Delta\ \sigma}{t[s] : T\ \Gamma\ \sigma}\ \text{SUBST}$$

ε can go from any context to the empty context, as a term typed in the empty context has no free variable. When you extend a substitution s with a term t, it means that the obtained substitution will replace the last variable ($\emptyset$) with t.

2.1.2 Weakening

We want to be able to extend the codomain of a substitution, it is to say to weaken the substitution:

$$\frac{s : Subst\ T\ \Gamma\ \Delta \qquad \sigma : Ty}{s^{+\sigma} : Subst\ T\ (\Gamma, \sigma)\ \Delta}\ \text{WEAK}$$

which is defined by pattern matching on s:

$$\begin{cases} \varepsilon^{+\sigma} & = & \varepsilon \\ (s, t)^{+\sigma} & = & s^{+\sigma}, (weak\ t) \end{cases}$$

2.1.3 Composition

We can also compose substitutions:

$$\frac{u : Subst\ T\ \Gamma\ \Delta \qquad v : Subst\ T\ \Delta\ \Theta}{u \circ v : Subst\ T\ \Gamma\ \Theta}\ \text{COMP}$$

which is defined by pattern matching on v:

$$\begin{cases} u \circ \varepsilon & = & \varepsilon \\ u \circ (v, t) & = & u \circ v, t[u] \end{cases}$$

2.1.4 Proofs

We want to prove that variable and term substitutions both form categories with finite products. We previously exposed what we need to establish in order to prove they form categories, but not in order to prove these categories have finite products. We consider the function that extends a substitution as a pairing function:

$$\begin{array}{rcccccc} ext & : & Subst\ T\ \Gamma\ \Delta & \rightarrow & T\ \Gamma\ \sigma & \rightarrow & Subst\ T\ \Gamma\ (\Delta, \sigma) \\ & & s & \mapsto & t & \mapsto & s, t \end{array}$$

We will have to find two projectors:

$$\begin{cases} \pi_1 & : & Subst\ T\ \Gamma\ (\Delta, \sigma) \rightarrow Subst\ T\ \Gamma\ \Delta \\ \pi_2 & : & Subst\ T\ \Gamma\ (\Delta, \sigma) \rightarrow T\ \Gamma\ \sigma \end{cases}$$

which match the axioms for surjective pairing:

$$\begin{cases} \pi_1\ (ext\ s\ t) & == & s & \text{(first projector)} \\ \pi_2\ (ext\ s\ t) & == & t & \text{(second projector)} \\ ext\ (\pi_1\ u)\ (\pi_2\ u) & == & u & \text{(surjective pairing)} \end{cases}$$

for any substitutions s and u and any element t of T.

2.2 The category of variable substitutions

In this section, we will give the last remaining definitions we need in order to prove that variable substitutions form a category with finite products. This proof is completely detailed in appendix A.

In this proof, we are only interested in applying a variable substitution to a variable, as defined by the rule SUBST (see section 2.1.1). However, this definition can be extended to any substitution: given a prototype $T : Con \to Ty \to Set$, we can apply a substitution over T to a variable and then obtain an element of T:

$$\frac{s : Subst\ T\ \Gamma\ \Delta \qquad v : Var\ \Delta\ \sigma}{v[s] : T\ \Gamma\ \sigma}\ \text{SUBSTVAR}$$

As v is defined in Δ, s cannot be an empty substitution. Applying a substitution to a variable is defined by pattern matching on this variable:

$$\begin{cases} vz[s,t] & = & t \\ (vs\ v)[s,t] & = & v[s] \end{cases}$$

It is easy to define the identity substitution, with the following prototype:

$$\frac{\Gamma : Con}{idv_\Gamma : Subst\ Var\ \Gamma\ \Gamma}\ \text{IDV}$$

by pattern matching on Γ:

$$\begin{cases} idv_\varepsilon & = & \varepsilon \\ idv_{\Gamma,\sigma} & = & idv_\Gamma^{+\sigma}, vz \end{cases}$$

The two projectors we will consider for the category of variable substitutions are the following ones:

$$\begin{cases} \pi_1 & : & s & \mapsto & s \circ idv^{+\sigma} \\ \pi_2 & : & s & \mapsto & vz[s] \end{cases}$$

2.3 The category of term substitutions

We now want to prove that term substitutions form a category with finite products. The main idea of all the proofs is to bring the proofs for term substitutions back to proofs for variable substitutions, as those latter are very easy to perform. We will here present this technique. But first, we will define the features that will be necessary so as to implement term substitutions.

2.3.1 Definitions

As explained in the rule SUBST, we can apply a term substitution to a term. We do it by pattern matching on the term:

$$\begin{cases} (var\ v)[s] & = & v[s] \\ (\lambda t)[s] & = & \lambda t[s^{+\sigma}, var\ vz] \\ (app\ t_1\ t_2)[s] & = & app\ t_1[s]\ t_2[s] \end{cases}$$

for any substitution s, variable v and terms t, t_1 and t_2. Here we see the interest of defining SUBSTVAR for any kind of substitution.

We have to define how to weaken a term (see section 1.3.1). Here is the first example of the way to bring features for terms back to features for variables: we will weaken a term by applying the weakened identity for variables. To do so, we have to define how to apply a variable substitution to a term:

$$\frac{s : Subst\ Var\ \Gamma\ \Delta \qquad t : Tm\ \Delta\ \sigma}{t[s] : Tm\ \Gamma\ \sigma}\ \text{SUBSTTM}$$

with a definition similar to the previous one:

$$\begin{cases} (var\ v)[s] & = & var\ v[s] \\ (\lambda t)[s] & = & \lambda t[s^{+\sigma}, vz] \\ (app\ t_1\ t_2)[s] & = & app\ t_1[s]\ t_2[s] \end{cases}$$

We can now weaken terms:

$$weak\ t = t[idv_\Gamma^{+\sigma}]$$

where $t : Tm\ \Gamma\ \tau$ and σ is a type.

We could define the identity substitution for terms as we defined the identity substitution for variables (with, in the second case: $idt_{\Gamma,\sigma} = idt_\Gamma^{+\sigma}, var\ vz$), but once more, we will try to go back to variables. We will obtain the identity substitution by embedding the identity for variables into a term substitution, using this function:

$$\frac{s : Subst\ Var\ \Gamma\ \Delta}{\lceil s \rceil : Subst\ Tm\ \Gamma\ \Delta}\ \text{EMB}$$

defined as follows:

$$\begin{cases} \lceil \varepsilon \rceil & = & \varepsilon \\ \lceil s, v \rceil & = & \lceil s \rceil, var\ v \end{cases}$$

The identity substitution is then:

$$idt_\Gamma = \lceil idv_\Gamma \rceil$$

We will finally need to compose variable and term substitutions, in both senses:

$$\frac{s : Subst\ Var\ \Gamma\ \Delta \qquad s' : Subst\ Tm\ \Delta\ \Theta}{s \circ s' : Subst\ Tm\ \Gamma\ \Theta}\ \text{C}_1 \qquad \frac{s : Subst\ Tm\ \Gamma\ \Delta \qquad s' : Subst\ Var\ \Delta\ \Theta}{s \circ s' : Subst\ Tm\ \Gamma\ \Theta}\ \text{C}_2$$

with the same definition as for the rule COMP.

Projectors for surjective pairing are the same as for variables:

$$\begin{cases} \pi_1 & : & s & \mapsto & s \circ idt^{+\sigma} \\ \pi_2 & : & s & \mapsto & (var\ vz)[s] \end{cases}$$

2.3.2 Proofs

We need two kinds of lemmas.

1. Obviously, we will need the same lemmas as for proofs for variables (see appendix A): they will also constitute the skeleton of the proofs for terms. We are not going to present those lemmas, as we explain in the appendix their roles for variables. Moreover, one can check the report on-line [5] to have the complete proof: section 4.3 of this report explains with great details how these lemmas are constructed, related and proved.

2. We will also need commutation lemmas that exploit the definitions of weakening and identity. Those lemmas will be used to link the lemmas of type 1 together. We are now going to detail them.

Embedding and weakening a substitution commute

Lemma 2.1 *For any variable substitution s and type σ:*

$$\lceil s \rceil^{+\sigma} == \lceil s^{+\sigma} \rceil$$

Proof *By induction on s.* □

Embedding and applying a substitution commute

Lemma 2.2 *For any variable substitution s and term t:*

$$t[\lceil s \rceil] = t[s]$$

Proof *By induction on t and using lemma 2.1.* □

Weakening and applying a substitution commute To prove this property, we first need two lemmas:

Lemma 2.3 *For any substitution s and type σ:*

$$\begin{cases} idv^{+\sigma} \circ s == s^{+\sigma} \\ s^{+\sigma} == (s^{+\sigma}, var\ vz) \circ idv^{+\sigma} \end{cases}$$

Proof *By induction on s.* □

We can now establish the following property:

Lemma 2.4 *For any substitution s, term t and type σ:*

$$weak\ t[s] == (weak\ t)[s^{+\sigma}, var\ vz]$$

Proof *We have the following equalities:*

$$\begin{array}{lcll} weak\ t[s] & == & (t[s])[idv^{+\sigma}] & \textit{(definition)} \\ & == & t[idv^{+\sigma} \circ s] & \textit{(variant of the lemma A.3 for terms)} \\ & == & t[(s^{+\sigma}, var\ vz) \circ idv^{+\sigma}] & \textit{(lemma 2.3)} \\ & == & (t[id^{+\sigma}])[s^{+\sigma}, var\ vz] & \textit{(variant of the lemma A.3 for terms)} \\ & == & (weak\ t)[s^{+\sigma}, var\ vz] & \textit{(definition)} \end{array}$$

□

With all these lemmas, we can prove variants of the lemmas presented in section A: we can commute *weak* $\bullet$, $\bullet^{+\sigma}$, $\bullet[\bullet]$ and $\lceil\bullet\rceil$ when needed. And then we get our main theorem:

Theorem 2.1 (Category of term substitutions) *Term substitutions form a category with finite products.*

3 Decidability of the $\beta\eta$-equivalence using hereditary substitutions

Hereditary substitutions constitute a kind of substitutions over typed λ-calculus that are structurally recursive. It provides an algorithm for normalisation whose termination can be proved by a simple lexicographic induction. The main idea of this algorithm is to simultaneously substitute and re-normalise.

It was introduced in the beginning of the 21$^{\text{st}}$ century by K. Watkins as a normaliser for the Concurrent Logical Framework [7]. It was then used for different purposes, for instance to implement an interpreter for λ-calculus whose termination is easy to prove [8].

Our purpose is to provide an elegant implementation of hereditary substitutions for simply-typed λ-calculus, and use this kind of normalisation to prove decidability of $\beta\eta$-equivalence. We will introduce how to transform a term into a normal form (normalisation) and how to transform a normal form into a term (embedding):

$$\frac{t : Tm\ \Gamma\ \sigma}{nf\ t : Nf\ \Gamma\ \sigma}\ \text{NF} \qquad \frac{n : Nf\ \Gamma\ \sigma}{\lceil n\rceil : Tm\ \Gamma\ \sigma}\ \text{EMB}$$

and then prove the following two properties[1]:

$$\frac{}{\lceil nf\ t\rceil \equiv t}\ \text{COMPLETENESS} \qquad \frac{t \equiv u}{nf\ t == nf\ u}\ \text{SOUNDNESS}$$

Those properties will lead to the decidability of the $\beta\eta$-equivalence. We will also establish stability (a rule that does not ensue from completeness and soundness):

$$\frac{}{nf\ \lceil t\rceil == t}\ \text{STABILITY}$$

[1] The names of these two properties are sometimes inverted.

The proof of stability is detailed in appendix B.

Contrary to parallel substitutions, which substitute all the free variables simultaneously, we use point-wise substitutions, that substitute only one free variable.

In section 3.1, we will introduce hereditary substitutions and define normalisation (the *nf* function). We will prove its termination. In section 3.2, we will present the main ideas of the proof of decidability of $\beta\eta$-equivalence.

Those proofs have been checked using the proof assistant Agda [1]. The source code is available online [5].

The interest of our study is to adapt a well-know property (decidability of the $\beta\eta$-equivalence) to a particular kind of substitutions, hereditary substitutions, which present the particularity of being structurally recursive. Once more, it uses Agda and contributes to the development of this tool.

3.1 Hereditary substitutions

3.1.1 Point-wise substitutions

We have to adapt point-wise substitutions presented in section 1.3.1, that is to say to define Γ' and the fact that $x \notin FV(u)$. Commonly, lots of implementations use a notation that adds a variable into a context. Here, we will present another notation, that removes a variable from a context. This presentation is quite elegant for simply-typed λ-calculus, but does not lend itself to other typed λ-calculus.

We have to define how to remove a variable from a context:

$$\frac{\Gamma : Con \qquad x : Var\ \Gamma\ \sigma}{\Gamma - x : Con}\ \text{MIN}$$

As x is in Γ, Γ cannot be empty. The function is really natural to define with the de Bruijn notation:

$$\begin{cases} (\Gamma, \sigma) - vz & = & \Gamma \\ (\Gamma, \sigma) - (vs\ x) & = & \Gamma - x, \sigma \end{cases}$$

In this framework, weakening a variable, a term, a normal form or a spine has a different meaning:

$$\frac{x : Var\ \Gamma\ \tau \qquad t : T\ (\Gamma - x)\ \sigma}{t^{+x} : T\ \Gamma\ \sigma}\ \text{WEAK} \qquad \frac{x : Var\ \Gamma\ \rho \qquad \vec{ts} : Sp\ (\Gamma - x)\ \sigma\ \tau}{\vec{ts}^{+x} : Sp\ \Gamma\ \sigma\ \tau}\ \text{WEAKSP}$$

where $T : Con \to Ty \to Set$. It is defined by pattern matching on t:

$$T \leftarrow Var : \begin{cases} y^{+vz} & = & vs\ y \\ vz^{+vs\ x} & = & vz \\ (vs\ y)^{+vs\ x} & = & vs\ y^{+x} \end{cases} \qquad T \leftarrow Tm : \begin{cases} (var\ v)^{+x} & = & var\ v^{+x} \\ (\lambda t)^{+x} & = & \lambda t^{+vs\ x} \\ (app\ t_1\ t_2)^{+x} & = & app\ t_1^{+x}\ t_2^{+x} \end{cases}$$

$$T \leftarrow Nf : \begin{cases} (\lambda n\ t)^{+x} & = & \lambda n\ t^{+vs\ x} \\ (y\ \vec{ts})^{+x} & = & y^{+x}\ \vec{ts}^{+x} \end{cases} \qquad Sp : \begin{cases} \vec{\varepsilon}^{+x} & = & \vec{\varepsilon} \\ (t, \vec{ts})^{+x} & = & t^{+x}, \vec{ts}^{+x} \end{cases}$$

and the SUBST rule is now:

$$\frac{t : T\ \Gamma\ \sigma \qquad x : Var\ \Gamma\ \tau \qquad u : T\ (\Gamma - x)\ \tau}{t[x := u] : T\ (\Gamma - x)\ \sigma}\ \text{SUBST}$$

$$\frac{\vec{ts} : Sp\ \Gamma\ \sigma\ \rho \qquad x : Var\ \Gamma\ \tau \qquad u : Nf\ (\Gamma - x)\ \tau}{\vec{ts}[x := u] : Sp\ (\Gamma - x)\ \sigma\ \rho}\ \text{SUBSTSP}$$

where T will be instantiated as Tm or Nf (we do not especially need a variable substitution).

SUBST where T is instantiated with Tm is the algorithm given section 1.3.1. We need to be able to compare two variables, and that is where the "-" for the contexts reveals its elegance:

- either the two variables are the same;
- either one exists in a context where the other has been removed (indeed, if y is in $(\Gamma - x)$, $y \neq x$ by definition).

To compare two variables, we hence define a data structure with two constructors corresponding to the two cases:

$$\frac{x : Var\ \Gamma\ \sigma \qquad y : Var\ \Gamma\ \tau}{Eq\ x\ y : Set} \qquad \frac{}{same : Eq\ x\ x} = \qquad \frac{x : Var\ \Gamma\ \sigma \qquad y : Var\ (\Gamma - x)\ \tau}{diff\ x\ y : Eq\ x\ (y^{+x})} \neq$$

and compare variables:

$$\frac{x : Var\ \Gamma\ \sigma \qquad y : Var\ \Gamma\ \tau}{eq\ x\ y : Eq\ x\ y}\ \text{EQ}$$

by pattern matching:

$$\begin{cases} eq\ vz\ vz & = & same & \\ eq\ vz\ (vs\ y) & = & diff\ vz\ y & \\ eq\ (vs\ x)\ vz & = & diff\ (vs\ x)\ vz & \\ eq\ (vs\ x)\ (vs\ y) & = & same & \text{if } eq\ x\ y = same \\ eq\ (vs\ x)\ (vs\ y) & = & diff\ (vs\ a)\ (vs\ b) & \text{if } eq\ x\ y = diff\ a\ b \end{cases}$$

Substitutions for normal forms remain to be defined. Normal forms are not closed under a substitution defined as for variables:

$$(y\ (z, \vec{\varepsilon}))[y := \lambda n\ vz] = (\lambda n\ vz)(z, \vec{\varepsilon})$$

which contains a β-redex and is not a normal form. It means we have to normalise when applying a substitution, and especially in this case : after the substitution properly speaking, $(\lambda n\ vz)$ must be successively **applied** to the elements of the spine until we get a normal form. The operator for this application will be written as follows:

$$\frac{u : Nf\ \Gamma\ \sigma \qquad \vec{ts} : Sp\ \Gamma\ \sigma\ \tau}{u \vec{@} \vec{ts} : Nf\ \Gamma\ \tau}\ \vec{@}$$

This function is mutually recursive with SUBST for normal forms and SUBSTSP:

$$\textsc{subst} : \begin{cases} (\lambda n\ t)[x := u] & = & \lambda n\ t[x^{+vz} := u^{+vz}] & \\ (y\ \vec{ts})[x := u] & = & u\vec{@}\vec{ts}[x := u] & \text{if } eq\ x\ y = same \\ (y\ \vec{ts})[x := u] & = & b\ \vec{ts}[x := u] & \text{if } eq\ x\ y = diff\ a\ b \end{cases}$$

$$\textsc{substSp} : \begin{cases} \vec{\varepsilon}[x := u] & = & \vec{\varepsilon} \\ (t, \vec{ts})[x := u] & = & t[x := u], \vec{ts}[x := u] \end{cases}$$

$$\vec{@} : \begin{cases} u\vec{@}\vec{\varepsilon} & = & u \\ (\lambda n\ u)\vec{@}(t, \vec{ts}) & = & (u[vz := t])\vec{@}\vec{ts} \end{cases}$$

In the last case, the normal form must have a function type, hence it must be a λn-abstraction.

It is now time to prove what we affirmed in the introduction:

Theorem 3.1 (Hereditary substitution) *Hereditary substitutions are structurally recursive and terminate.*

Proof *We only have to consider the last three definitions. For* SUBST*, we will focus on the pair (type of x,t); for* SUBSTSP*, on (type of x,$\vec{ts}$); and for $\vec{@}$, on (type of u). It is obvious to see that when each function calls itself and when* SUBST *and* SUBSTSP *call one another, all the measures decrease for the lexical order.*

The remaining point is to prove that when SUBST *and $\vec{@}$ call each other, the measures decrease for the lexicographical order. Those two functions can call one another only in this case:*

$$\begin{cases} (x\ (t, \vec{ts}))[x := \lambda n\ u] & = & (\lambda n\ u)\vec{@}(t', \vec{ts}') & \text{where } (t', \vec{ts}') = (t, ts)[x := u] \\ (\lambda n\ u)\vec{@}(t', \vec{ts}') & = & (u[vz := t'])\vec{@}\vec{ts}' & \end{cases}$$

When $(x\ (t, \vec{ts}))[x := \lambda n\ u]$ calls $\vec{@}$, x and $(\lambda n\ u)$ have type $\sigma \to \tau$. When $(\lambda n\ u)\vec{@}(t', \vec{ts}')$ calls SUBST*, vz has type σ, which is strictly inferior to $\sigma \to \tau$; and when $(\lambda n\ u)\vec{@}(t', \vec{ts}')$ calls $\vec{@}$ $(u[vz := t'])$ has type τ, which is strictly inferior to $\sigma \to \tau$. So the measures decrease for the lexicographical order.* □

FIG. 2 is a summary of this proof. When a function calls itself using any path in the graph, its measure strictly decreases.

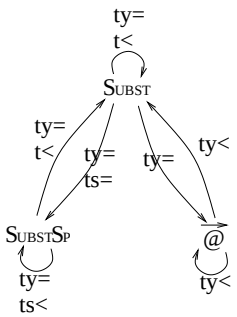

Figure 2: Call-graph of hereditary substitutions.

3.1.2 Normalisation

We are now going to define the *nf* function. But first, we need to normalise neutral terms (a variable applied to many normal forms) when it does not have the base type:

$$\frac{x : Var\ \Gamma\ \sigma \qquad \vec{ts} : Sp\ \Gamma\ \sigma\ \tau}{nf'\ x\ \vec{ts} : Nf\ \Gamma\ \circ}\ nf'$$

We do as many η-expansions as needed to bring neutral terms back to the base type:

$$\begin{cases} nf'\ x\ \vec{ts} &=& x\ \vec{ts} & \text{if } \vec{ts} : Sp\ \Gamma\ \sigma\ \circ \\ nf'\ x\ \vec{ts} &=& \lambda n\ (nf'\ x\ (append\ \vec{ts}^{+vz}\ (nf'\ vz\ \vec{\varepsilon}))) & \text{if } \vec{ts} : Sp\ \Gamma\ \sigma\ (\tau \to \rho) \end{cases}$$

where *append* adds a normal form at the end of a spine:

$$\frac{\vec{ts} : Sp\ \Gamma\ \rho\ (\sigma \to \tau) \qquad u : Nf\ \Gamma\ \sigma}{append\ \vec{ts}\ u : Sp\ \Gamma\ \rho\ \tau}\ \textsc{append}$$

with the following definition:

$$\begin{cases} append\ \vec{\varepsilon}\ u &=& u, \vec{\varepsilon} \\ append\ (t, \vec{ts})\ u &=& t, append\ \vec{ts}\ u \end{cases}$$

This will be used to normalise variables in the definition of *nf*:

$$\begin{cases} nf\ (var\ v) &=& nf'\ v\ \vec{\varepsilon} \\ nf\ (\lambda t) &=& \lambda n\ (nf\ t) \\ nf\ (app\ t_1\ t_2) &=& u[vz := nf\ t_2] & \text{where } nf\ t_1 = \lambda n\ u \end{cases}$$

3.1.3 Embedding

We have to define the reverse operation: embedding a normal form into a term. The function $\lceil \bullet \rceil$ will be mutually recursive with a function doing the same thing for a spine:

$$\frac{\vec{ts} : Sp\ \Gamma\ \sigma\ \tau \qquad t : Tm\ \Gamma\ \sigma}{embSp\ \vec{ts}\ t : Tm\ \Gamma\ \tau}\ \textsc{embSp}$$

with the following definitions:

$$\lceil \bullet \rceil : \begin{cases} \lceil \lambda n\ u \rceil &=& \lambda \lceil u \rceil \\ \lceil x\ \vec{ts} \rceil &=& embSp\ \vec{ts}\ (var\ x) \end{cases}$$

$$embSp : \begin{cases} embSp\ \vec{\varepsilon}\ t &=& t \\ embSp\ (u, \vec{us})\ t &=& embSp\ \vec{us}\ (app\ t\ \lceil u \rceil) \end{cases}$$

3.2 Proofs

We established that our normaliser is actually terminating. Each normal form can be written a unique way. As a result, we can use it to prove decidability of $\beta\eta$-equivalence: to check if two (typed) terms are $\beta\eta$-equivalent, we just have to normalise them and check if their normal forms are propositionally equal... provided this calculus is complete and sound! This is why we propose to show completeness and soundness.

The completeness and soundness proofs will all rely on common syntactic rewriting laws. These rules are a bit tricky to write and to prove because of the "-" notation for contexts; as a result, we are only going to informally write it, and more details can be found in appendix C.

Lemma 3.1 *If $T : Con \to Ty \to Set$, for any t of T, for any terms or normal forms u and v and any variables x and y:*

$$\begin{cases} (t^{+x})^{+y} & == & (t^{+y})^{+x} \\ (t[x := u])^{+y} & == & t^{+y}[x^{+y} := u^{+y}] \\ (t[x := u])[y := v] & == & (t[y := v])\big[x := u[y := v]\big] \end{cases}$$

Proof *See appendix C.* □

We will now give some clues to perform the completeness and soundness proofs.

3.2.1 Completeness

In order to prove completeness, we need lemmas that commute embedding and other key functions such as NF, WEAK and SUBST.

Lemma 3.2 *For any variable x, normal form u, spine $\vec{ts}$ and term t:*

$$\begin{cases} \lceil u^{+x} \rceil & \equiv & \lceil u \rceil^{+x} \\ embSp\ \vec{ts}^{+x}\ t^{+x} & \equiv & (embSp\ \vec{ts}\ t)^{+x} \end{cases}$$

Note: The same properties with $==$ instead of $\equiv$ are also true, but we need a weaker result.

Proof *By mutual induction on u and $\vec{ts}$.* □

Lemma 3.3 *For any variable x and spine $\vec{ts}$:*

$$\lceil nf'\ x\ \vec{ts} \rceil \equiv embSp\ \vec{ts}\ (var\ x)$$

Proof *By induction on the second type of $\vec{ts}$ and using lemma 3.2 and the η-law.* □

Lemma 3.4 *For any spine $\vec{ts}$, term t, normal forms u and v, and variable x:*

$$\begin{cases} embSp\ \vec{ts}[x := u]\ t[x := \lceil u \rceil] & \equiv & (embSp\ \vec{ts}\ t)[x := \lceil u \rceil] \\ \lceil u\ \vec{@}\ \vec{ts} \rceil & \equiv & embSp\ \vec{ts}\ \lceil u \rceil \\ \lceil v[x := u] \rceil & \equiv & \lceil v \rceil [x := \lceil u \rceil] \end{cases}$$

Proof *By mutual induction on $\vec{ts}$ and v, and using the beta-law.* □

This leads us to our main theorem:

Theorem 3.2 (Completeness) *For any term t:*

$$\lceil nf\ t \rceil \equiv t$$

Proof *By induction on t and using lemmas 3.3 and 3.4.* □

3.2.2 Soundness

As for completeness, we need lots of commutation lemmas between nf and other functions. We are not going to provide them all, not to be repetitive. The only difference is that it does not deal with $\beta\eta$-equivalence but with propositional equality.

Besides, we need semantic lemmas that prove we actually β-reduced and η-expanded our normal forms as much as possible: it is to say that the β and the η-equalities much stand for normal forms.

It is obvious that the β-equality stands, as each time we obtained a redex, we did the substitution. However, the η-law is really non trivial. We will assume it to be true[2]:

Postulate 3.1 *For any normal form u with a function type:*

$$\lambda n\ (napp\ u^{+vz}\ (nf' vz\ \vec{\varepsilon})) == u$$

where $napp\ (\lambda n\ t)\ u = t[vz := u]$ (definition).

When all this is established, we can complete the proof of our main theorem:

Theorem 3.3 (Soundness) *For any terms t and u:*

$$\text{if } t \equiv u, \text{ then } nf\ t == nf\ u$$

Proof *By induction on the proof that $t \equiv u$.* □

3.2.3 Decidability of $\beta\eta$-equivalence

As mentioned above, completeness and soundness are useful to prove the decidability of the $\beta\eta$-equivalence:

Theorem 3.4 (Decidability of $\beta\eta$-equivalence) *The $\beta\eta$-equivalence is decidable, it is to say that we have an algorithm which, given two terms t and u, says if $t \equiv u$ or $t \not\equiv u$.*

Proof *The algorithm is the following one:*

[2] As we were not able to perform the proof. It is interesting to notice that u must have an arrow type, so it must be a λn-abstraction, and proving this property is similar to proving this one, for any normal form u:

$$u^{+(vs\ vz)}[vz := nf' vz\ \vec{\varepsilon}] == u$$

1. *Normalise t and u to obtain t' and u'.*

2. *$t \equiv u$ if and only if $t' == u'$.*

We proved the first step terminates. We have to prove the second step. As $==$ is decidable, this will automatically lead to the decidability of $\equiv$.

The direct sense is soundness. The reverse sense is trivial: if $t' == u'$, then $\lceil t' \rceil == \lceil u' \rceil$. But $t \equiv \lceil t' \rceil$ and $\lceil u' \rceil \equiv u$ (completeness), so $t \equiv u$ by transitivity. □

Conclusion

Perspectives

This work is part of a global work concerning proof assistants, their development and their use. Indeed, we made obvious that it was really easy to implement functional tools using Agda, a proof assistant. As a result, proofs like those we performed during this internship form a base that can be used to do more complex proofs: our implementation of parallel or hereditary substitutions can form Agda libraries, and the way we conducted the proofs is an example that can be easily generalised to other proofs about parallel or hereditary substitutions.

The concept of proof assistants, and also of proof mechanising, appeared in the beginning of the 20^{th} century, at the same time with λ-calculus and logics, when scientists wondered about the bases of mathematics and its reasoning. Nowadays, proofs assistants exist that are very powerful and can help to make more and more complex proofs, either "mathematical" proofs of proofs of programs.

But proof assistants do not only exist in the research field, but also in the industrial field, as an improving amount automated programs are proved before being on the market field. It might also reach the public field, as languages including dependant types give more expressivity than the existing famous functional languages.

Our work is then very encouraging, as proofs that were previously only formally written are formalised in an assistant proof. It helps to have a new point of view on simply typed λ-calculus, and especially substitutions over it. As mechanisation becomes omnipresent, this point of view is very important to understand the objects we manipulate.

Personal learning

This internship was a source of knowledge for me, above all in the type theory field. I perfected what I previously learned about simply typed λ-calculus, in particular about substitutions. I got a deeper understanding of parallel substitutions, and discovered hereditary substitutions, a kind of substitutions that was completely new for me, and that is likely to be developed in the next years.

I also learned how to implement these objects and how to perform proofs about them in a language including dependant types, Agda, which I did not know at all. I discovered that proofs which where theoretically obvious, had to be adapted a lot to stand using a proof assistant. For instance, I learned and used the de Bruijn notation, which is very useful when implementing, because you do not have α-conversion problems, but sometimes leads to tricky proofs you do not need with the classical notation.

I had a very complete internship, dealing at the same time with programming with a proof assistant, as I really wanted to do, and learning more theoretical concepts about λ-calculus and substitutions. It comforted my will to pursue my studies in the type theory field.

It was also the occasion for me to go to England: on the one hand, I discovered other work methods and habits that you do not have in France, and perfected (a bit) my English, and on the other hand, it was the occasion for me to visit England, where I had never been before.

References

[1] Agda wiki. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php?n=Main.Documentation`.

[2] H. P. Barendregt. The lambda calculus: Its syntax and semantics. *Revised second edition*, 1984.

[3] H. P. Berendregt. Lambda calculi with types. *Handbook of Logic in Computer Science, Volume II, Oxford University Press*, 199-.

[4] N. G. De Bruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the church-rosser theorem. *Indag. Math.*, pages 381–382, 1972.

[5] Proofs in agda. `http://perso.ens-lyon.fr/chantal.keller/Documents-etudes/Stage/Parallel-substitution`.

[6] Conor McBride. Type-preserving renaming and substitution. *Functionnal Pearl*, 2006.

[7] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework i: Judgements and properties. *Tech. rep., School of Computer Science, Carnegie Mellon University, Pittsburgh*, 2003.

[8] A. Abel. Implementing a normalizer using sized heterogeneous types. *Wksh. on Mathematically Structured Functionnal Programming*, 2006.

A Proof that variable substitutions form a category with finite products

In this appendix, we will detail the whole proof that variable substitutions form a category with finite products. Please see section 2 for more details about what we need to prove and variable parallel substitutions.

Neutrality on the left We first have to establish three lemmas:

Lemma A.1 *For any substitution s, variable v and type σ:*

$$\begin{cases} v[s^{+\sigma}] == vs\ v[s] \\ v[idv^{+\sigma}] == vs\ v \\ v[idv] == v \end{cases}$$

Proof *The first property can be proved by induction on v. The last two properties can be mutually proved by induction on v (using the first property).* □

We can now prove our main theorem:

Theorem A.1 (Neutrality on the left) *For any substitution s:*

$$idv \circ s == s$$

Proof *By induction on s and using lemma A.1.* □

Neutrality on the right We need on single lemma:

Lemma A.2 *For any substitutions s and s', variable v and type σ:*

$$(s, v) \circ s'^{+\sigma} == s \circ s'$$

Proof *By induction on s'.* □

We can now prove the main theorem:

Theorem A.2 (Neutrality on the right) *For any substitution s:*

$$s \circ idv == s$$

Proof *By induction on s and using lemma A.2.* □

Associativity We need a preliminary lemma, which proves that applying the composition of two substitutions is like applying each one successively:

Lemma A.3 *For any variable v and substitutions s and s':*

$$v[s \circ s'] == (v[s'])[s]$$

Proof *By induction on v.* □

Our main theorem is:

Theorem A.3 (Associativity) *For any substitutions u, v and w:*

$$u \circ (v \circ w) == (u \circ v) \circ w$$

Proof *By induction on w and using lemma A.3.* □

First projector We want to prove we chose a good candidate for the first projector:

Theorem A.4 (First projector) *For any substitution s and variable v:*

$$\pi_1(s, v) == s$$

Proof *Lemma A.2 and theorem A.2.* □

Second projector We want to prove we chose a good candidate for the second projector:

Theorem A.5 (Second projector) *For any substitution s and variable v:*

$$\pi_2(s, v) == v$$

Proof *Definition of π_2.* □

Surjective pairing We can now prove the last required theorem:

Theorem A.6 (Surjective pairing) *For all s in Subst Var Γ (Δ, σ):*

$$\pi_1\ s, \pi_2\ s == s$$

Proof *As s is in Subst Var Γ (Δ, σ), s must be an extended substitution. The proof is then completed by theorems A.4 and A.5.* □

B Stability of hereditary substitutions

We will use the same commutation lemmas as for soundness. We propose a quite elegant proof, using concatenation of two spines:

$$\frac{\vec{ts} : Sp\ \Gamma\ \sigma\ \tau \qquad \vec{us} : Sp\ \Gamma\ \tau\ \rho}{\vec{ts}^{\wedge}\vec{us} : Sp\ \Gamma\ \sigma\ \rho}\ \wedge$$

easily defined by pattern matching on $\vec{ts}$:

$$\begin{cases} \vec{\varepsilon}^{\wedge}\vec{us} & = & \vec{us} \\ (t,\vec{ts})^{\wedge}\vec{us} & = & t,\vec{ts}^{\wedge}\vec{us} \end{cases}$$

We need those two lemmas about concatenation:

Lemma B.1 *For any spines $\vec{ts}$ and $\vec{us}$ and any normal form u:*

$$\begin{cases} \vec{ts}^{\wedge}\vec{\varepsilon} & == & \vec{ts} \\ \vec{ts}^{\wedge}(u,\vec{us}) & == & (append\ \vec{ts}\ u)^{\wedge}\vec{us} \end{cases}$$

Proof *By induction on $\vec{ts}$.* □

Lemma B.2 *For any variable x and any spines $\vec{ts}$ and $\vec{us}$:*

$$(nf'\ x\ \vec{ts})\vec{@}\vec{us} == nf'\ x\ (\vec{ts}^{\wedge}\vec{us})$$

Proof *By induction on $\vec{us}$ and using lemma B.1.* □

Our main theorem will be mutually recursive with another property:

Theorem B.1 (stability) *For any spine $\vec{ts}$, term t and normal form u:*

$$\begin{cases} nf\ (embSp\ \vec{ts}\ t) & == & (nf\ t)\vec{@}\vec{ts} \\ nf\ \lceil u \rceil & == & u \end{cases}$$

Proof *By mutual induction on $\vec{ts}$ and u and using lemma B.2.* □

C Rewriting rules for hereditary substitutions

Each proof concerning point-wise substitutions relies on rewriting rules that allow to commute weakening and substitutions in the three possible senses (see section 3.2 for the details of these rules).

Proving these rules is easy to formally perform, but present some problems to adapt to our implementation, and especially to the minus notation. Indeed, even righting the rules is not obvious. For instance the rule:

$$(t^{+x})^{+y} == (t^{+y})^{+x}$$

has no sense with our notations: considering the left hand side of the equality, if y exists in a context Γ, then x must be in $\Gamma - y$ and t in $(\Gamma - y) - x$, and the right hand side is impossible to write.

As a consequence, we have to introduce a new notation to be able to write such a rule. What we miss here is the possibility to know that y exists in a context where x has been removed! So we introduce a new function with the following prototype:

$$\frac{x : Var\ \Gamma\ \sigma \qquad y : Var\ (\Gamma - x)\ \tau}{x^{-y} : Var\ (\Gamma - y^{+x})\ \sigma}\ \text{REM}$$

defined by pattern matching on the two variables:

$$\begin{cases} vz^{-y} & = & vz \\ (vs\ x)^{-vz} & = & x \\ (vs\ x)^{-(vs\ y)} & = & vs\ x^{-y} \end{cases}$$

The idea will be to sometimes consider $(\Gamma - x) - y$ as the same context in which the roles of x and y are inverted; indeed, the following lemma stands:

Lemma C.1 *For any context Γ and any variable x and y:*

$$(\Gamma - x) - y == (\Gamma - y^{+x}) - x^{-y}$$

Proof *By induction on x and y.* □

Note: In the following of the appendix, this proof will be written $\mathcal{P}$.

Using this equality, we can change the context in which a variable, a term or a normal form is typed. To do so, we introduce a last function:

$$\frac{p : \Gamma == \Delta \qquad x : T\ \Gamma\ \sigma}{|\, p > x : T\ \Delta\ \sigma}\ |\ >$$

where $T : Con \to Ty \to Set$. It is defined by pattern matching on the proof p, which can only be the constructor $== -refl$.

The three lemmas informally described in section 3.2 can be written using these new notations:

Lemma C.2 *For any t of T (where $T : Con \to Ty \to Set$), variables x and y and terms or normal forms u, u_1 and u_2:*

$$\begin{cases} \left((|\mathcal{P} > t)^{+x^{-y}}\right)^{+y^{+x}} & == & (t^{+y})^{+x} \\ (|\mathcal{P} > t[y := u])^{+x^{-y}} & == & t^{+x}[y^{+x} := (|\mathcal{P} > u)^{+x^{-y}}] \\ |\mathcal{P} > (t[x := u_1])[y := u_2] & == & (t[y^{+x} := (|\mathcal{P} > u_2)^{+x^{-y}}])\big[x^{-y} := |\mathcal{P} > u_1[y := u_2]\big] \end{cases}$$

Proof *By induction on t.* □

D Agda's implementation

The whole code was written in Agda, a programming language including dependant types, and so useful to check proofs. It also provides an interface that can help to perform the proofs.

In this section, we intend to present the very base of Agda's syntax, and then some of the source code that shows the need to rephrase the usual notations for λ-calculus so as to implement it easily.

A short introduction to Agda

Agda is a proof assistant which most important specificity is to provide a language close to functional languages (such as Haskell and Coq). As a result, it may be more difficult to find the term which is the proof of some property, but using Agda is closer to make programs, and then it makes it more natural to use. As a consequence, it is well adapted to perform proofs about logics and λ-calculus.

Here is a short introduction to Agda basic syntax. A complete tutorial is available on the Agda wiki [1].

Data-type declarations

Inductive data-types can be defined by a formation rule and constructors. For instance, the Agda definition for the propositional equality presented in section 1.1 is:

```
data _==_ {A : Set} : A -> A -> Set where
  ==-refl : {a : A} -> a == a
```

For a given set `A`, the equality on this set is defined as a set of elements that are identical. The use of the braces allows us to define implicit arguments. The syntax `_==_` means that this relation is infix : we can write `a == a`.

Agda checks that everything is well typed and that the induction definitions do terminate.

Function declarations

Functions are defined as follows: the first line defines the name and the type of the function, and the following lines define the core of the function, using generally pattern matching. For instance, we can define the functions that prove that `==` is symmetric and transitive:

```
==-sym : {A : Set} -> {a b : A} -> a == b -> b == a
==-sym ==-refl = ==-refl

==-trans : {A : Set} -> {a b c : A} -> a == b -> b == c -> a == c
==-trans ==-refl ==-refl = ==-refl
```

Agda checks that everything is well typed and that the function definitions do terminate.

Simply typed λ-calculus

We can now implement our definition of the simply typed λ-calculus, using the de Bruijn notation.

Types and contexts:

```
data Ty : Set where
  ∘ : Ty
  _⇒_ : Ty → Ty → Ty

data Con : Set where
  ε : Con
  _,_ : Con → Ty → Con
```

Variables:

```
data Var : Con → Ty → Set where
  vz : forall {Γ σ} → Var (Γ , σ) σ
  vs : forall {τ Γ σ} → Var Γ σ → Var (Γ , τ) σ
```

Terms:

```
data Tm : Con → Ty → Set where
  var : forall {Γ σ} → Var Γ σ → Tm Γ σ
  λ   : forall {Γ σ τ} → Tm (Γ , σ) τ → Tm Γ (σ ⇒ τ)
  app : forall {Γ σ τ} → Tm Γ (σ ⇒ τ) → Tm Γ σ → Tm Γ τ
```

Normal forms (using mutually inductive data-types):

```
mutual
  data Nf : Con → Ty → Set where
    λn  : forall {Γ σ τ} → Nf (Γ , σ) τ → Nf Γ (σ ⇒ τ)
    ne  : forall {Γ} → Ne Γ ∘ → Nf Γ ∘

  data Ne : Con → Ty → Set where
    _,_ : forall {Γ σ τ} → Var Γ σ → Sp Γ σ τ → Ne Γ τ

  data Sp : Con → Ty → Ty → Set where
    ε : forall {σ Γ} → Sp Γ σ σ
    _,_ : forall {Γ σ τ ρ} → Nf Γ τ → Sp Γ σ ρ → Sp Γ (τ ⇒ σ) ρ
```

$\beta\eta$-equivalence:

```
data _βη≡_ {Γ : Con} : {σ : Ty} → Tm Γ σ → Tm Γ σ → Set where
  refl : forall {σ} → {t : Tm Γ σ} → t βη≡ t
  sym : forall {σ} → {t₁ t₂ : Tm Γ σ} → t₁ βη≡ t₂ → t₂ βη≡ t₁
  trans : forall {σ} → {t₁ t₂ t₃ : Tm Γ σ} → t₁ βη≡ t₂ → t₂ βη≡ t₃ →
      t₁ βη≡ t₃
  congλ : forall {σ τ} → {t₁ t₂ : Tm (Γ , σ) τ} → (t₁ βη≡ t₂) →
      λ t₁ βη≡ λ t₂
  congApp : forall {σ τ} → {t₁ t₂ : Tm Γ (σ ⇒ τ)} → {u₁ u₂ : Tm Γ σ} →
      t₁ βη≡ t₂ → u₁ βη≡ u₂ → app t₁ u₁ βη≡ app t₂ u₂
  beta : forall {σ τ} → {t : Tm (Γ , σ) τ} → {u : Tm Γ σ} →
      app (λ t) u βη≡ t[vz := u]
  eta : forall {σ τ} → {t : Tm Γ (σ ⇒ τ)} →
      λ (app t^+vz (var vz)) βη≡ t
```

Parallel substitutions

We will give the code for the definition of parallel substitutions, and one example of a proof.

Substitutions:

```
data Subst (T : Con → Ty → Set) : Con → Con → Set where
  ε : forall {Γ} → Subst T Γ empty
  _,_ : forall {Γ Δ σ} → Subst T Γ Δ → T Γ σ → Subst T Γ (ext Δ σ)
```

Proof that variable substitutions are associative:

```
aCSVar : forall {Γ Δ Θ σ} → (u : Subst Var Γ Δ) → (v : Subst Var Δ Θ) →
      (v' : Var Θ σ) → v'[u ∘ v] == (v'[v])[u]
aCSVar _ ε ()
aCSVar _ (_ , _) vz = ==-refl
aCSVar u (v , _) (vs v') = aCSVar u v v'

assoCompSVar : forall {Γ Δ Θ Ξ} → (u : Subst Var Γ Δ) →
       (v : Subst Var Δ Θ) → (w : Subst Var Θ Ξ) →
       (u ∘ v) ∘ w == u ∘ (v ∘ w)
assoCompSVar _ _ ε = ==-refl
assoCompSVar u v (w , t) = reflSubstExt (assoCompSVar u v w) (aCSVar u v t)
```

The symbol _ means that we do not care the name of the variable it stands for. () means that this case (in the pattern matching) is absurd. reflSubstExt is the following function:

```
reflSubstExt : forall {T Γ Δ σ} → {s₁ s₂ : Subst T Γ Δ} → {t₁ t₂ : T Γ σ}
    → s₁ == s₂ → t₁ == t₂ → s₁ , t₁ == s₂ , t₂
reflSubstExt ==-refl ==-refl = ==-refl
```

Hereditary substitutions

We will present the Agda code for some of the notations we defined in section 3.1.1, and hereditary substitutions.

Minus notation:

```
_-_ : {σ : Ty} -> (Γ : Con) -> Var Γ σ -> Con
ε - ()
(Γ , σ) - vz = Γ
(Γ , τ) - (vs x) = (Γ - x) , τ
```

Weakening:

```
wkv : forall {Γ σ τ} -> (x : Var Γ σ) -> Var (Γ - x) τ -> Var Γ τ
wkv vz     y      = vs y
wkv (vs x) vz     = vz
wkv (vs x) (vs y) = vs (wkv x y)

mutual
  wkNf : forall {σ Γ τ} -> (x : Var Γ σ) -> Nf (Γ - x) τ -> Nf Γ τ
  wkNf x (λn t)       = λn (wkNf (vs x) t)
  wkNf x (ne (y , us)) = ne (wkv x y , wkSp x us)

  wkSp :  forall {σ Γ τ ρ} -> (x : Var Γ σ) -> Sp (Γ - x) τ ρ -> Sp Γ τ ρ
  wkSp x ε       = ε
  wkSp x (u , us) = (wkNf x u) , (wkSp x us)
```

Equality between variables:

```
data EqV {Γ : Con} : {σ τ : Ty} -> Var Γ σ -> Var Γ τ -> Set where
  same : forall {σ} -> {x : Var Γ σ} -> EqV {Γ} {σ} {σ} x x
  diff : forall {σ τ} -> (x : Var Γ σ) -> (y : Var (Γ - x) τ) -> EqV {Γ}
      {σ} {τ} x (wkv x y)

eq : forall {Γ σ τ} -> (x : Var Γ σ) -> (y : Var Γ τ) -> EqV x y
eq vz     vz     = same
eq vz     (vs x) = diff vz x
eq (vs x) vz     = diff (vs x) vz
eq (vs x) (vs y) with eq x y
eq (vs x) (vs .x)          | same       = same
eq (vs .x) (vs .(wkv x y)) | (diff x y) = diff (vs x) (vs y)
```

The with notation allows us to have a view on an object.

Application of a normal form to a spine:

```
appSp : forall {Γ σ τ ρ} -> Sp Γ ρ (σ → τ) -> Nf Γ σ -> Sp Γ ρ τ
appSp ε u = (u , ε)
appSp (t , ts) u = ( t , appSp ts u)
```

Hereditary substitutions:

```
mutual
  substNf : forall {σ Γ τ} -> (Nf Γ τ) -> (x : Var Γ σ) -> Nf (Γ − x) σ ->
      Nf (Γ − x) τ
  substNf (λn t) x u = λn (substNf t (vs x) (wkNf vz u))
  substNf (ne (y , ts)) x u with eq x y
  substNf (ne (x , ts)) .x u | same = appNf u (substSp ts x u)
  substNf (ne (.(wkv x y') , ts)) .x u | diff x y' = ne (y' , substSp ts x
      u)

  substSp : forall {Γ σ τ ρ} -> (Sp Γ τ ρ) -> (x : Var Γ σ) -> Nf (Γ − x) σ
       -> Sp (Γ − x) τ ρ
  substSp ε x u = ε
  substSp (t , ts) x u = (substNf t x u) , (substSp ts x u)

  appNf : forall {τ Γ σ} -> Nf Γ σ -> Sp Γ σ τ -> Nf Γ τ
  appNf t (u , us) = appNf (substNf t vz u) us
  appNf t ε = t
```