Gentle Formalisation of Stoughton's Lambda Calculus Substitution

Álvaro Tasistro
Universidad ORT Uruguay
tasistro@ort.edu.uy

Ernesto Copello
Universidad ORT Uruguay
copello@ort.edu.uy

Nora Szasz
Universidad ORT Uruguay
szasz@ort.edu.uy

ABSTRACT

In [5], Allen Stoughton proposed a notion of substitution for the Lambda calculus formulated in its original syntax with only one sort of symbols (names) for variables –without identifying α-convertible terms. According to such formulation, the action of substitution on terms is defined by simple structural recursion and an interesting theory arises concerning e.g. α conversion. In this paper we present a formalisation of Stoughton's development in Constructive Type Theory using the language Agda, up to the Substitution Lemma for α conversion. The code obtained is quite concise and we are able to formulate some improvements over the original presentation. For instance, our definition of α conversion is just syntax directed but we are nevertheless able to prove in an easy way that it gives rise to an equivalence relation, whereas in [5] the latter was included as part of the definition. As a result of this work we are inclined to assert that Stoughton's is the right way to formulate the Lambda calculus in its original, conventional syntax and that it is a formulation amenable to fully formal treatment.

1. INTRODUCTION

The Lambda calculus was introduced in [1] without a definition of substitution. The complexity of this operation was actually a prime motivation for [2], which provided the first formal definition, somewhat as follows:

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

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

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

The complexity lies in the last case: The recursion is on the *length* of the term wherein the substitution is performed. But, to ascertain that $M[x := z]$ is of a length lesser than that of $\lambda x.M$, a proof has to be composed, which must be therefore simultaneous to the justification of the well-foundedness of the definition. Also consequently, proofs of properties of the substitution operation have generally to be conducted on the length of terms. This prompts the search for a simpler definition, especially if one is interested in actually carrying out the formal meta-theory of the lambda calculus to a substantial extent, e.g. employing some of the proof assistants at hand.

As is well known, several of the proposed solutions take the path of modifying the actual syntax of the language. Such path is indeed well motivated, especially if the choice is to employ a type of symbols different from that of the variables for the local or bound names: That was actually Frege's choice in the first fully fledged formal language with a binder [3]. But it is also interesting to investigate how well it is possible to do with the original, ordinary syntax of the Lambda calculus. In this respect, it can be argued that it was Stoughton [5] who provided the right formulation of substitution.

The prime observation is simple: In the difficult case where renaming of the bound variable is necessary, structural recursion is recovered if one lets substitutions to grow *multiple* (*simultaneous*) instead of 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. The resulting theory has many pleasant properties, besides radically simplifying the method of reasoning. Possibly the most interesting result is that the identity substitution normalizes terms with respect to α-

conversion, which is due precisely to the method of uniform renaming.

The purpose of the present paper is to explore the formalisation in Constructive Type Theory of Stoughton's formulation and subsequent theory of substitution in the Lambda calculus. We shall carry it out up to the proof of the so-called *Substitution Lemma* for α conversion, i.e. the result establishing that the substitution operation is compatible with α conversion. Such formalisation has been undertaken in [4] but we believe we are making some substantial reformulation thereof. In particular, that work represents the multiple substitutions as (total) functions from variables to terms (the same as in [5]) and defines the propositional identity of substitutions as their extensional equivalence. We are rather unsatisfied with this stand, for we prefer the propositional identity to reflect the definitional, and thus decidable, equality. We shall also represent the multiple substitutions as functions, but will avoid using such fully extensional equivalence. We also introduce simplifications with respect to both Stoughton's original formulation and the just mentioned formalisation. Foremost among these is the definition of α conversion: Stoughton's version is as the least congruence generated by a simple renaming of bound variable. On the other hand, the formalisation [4] gives a structural definition and then takes considerable effort to prove it an equivalence relation. Our definition is a quite simple inductive one, directly following the structure of terms, and the proof that it is an equivalence relation is very short, although it requires induction on the length of the terms at one point.

From a general point of view, one can assume a relaxed, didactic style of presentation which explores the various concepts and choices involved in a bottom-up fashion or, alternatively, one can direct oneself as efficiently as possible towards an established objective. In this paper we take the second approach, mainly due to our interest in finding out how inexpensive a formal proof of the Substitution Lemma for α conversion can be if we stick to the classical syntax of Lambda calculus. The interest in the mentioned Substitution Lemma is due to the fact that it is an important piece in the subsequent development of the meta-theory of the Lambda calculus, particularly in the various lemmas conducting to the Church-Rosser theorem.

The rest of the paper consists just in the next section, where the formal development is presented with as much detail as we consider appropriate, and a final section on concluding comments.

2. FORMALISATION

We use the language Agda[?]. The present is actually a literate Agda document, where we hide some code for reasons of conciseness[1].

2.1 Syntax

[1]The entire code is available at:
http://docentes.ort.edu.uy/perfil.jsp?docenteId=2264

V is an inductive type with infinitely many objects.

```
data Λ : Set where
  var : V → Λ
  app : Λ → Λ → Λ
  abs : V → Λ → Λ
```

The following is called the *freshness* relation. It holds when a variable does not occur free in a term.

```
data _#_ : V → Λ → Set where
  var  : {x y : V} → y ≢ x →
         x # var y
  app  : {x : V} {M N : Λ} → x # M → x # N →
         x # (app M N)
  abse : {x y : V} {M : Λ} → x ≡ y →
         x # (abs y M)
  abs  : {x y : V} {M : Λ} → x # M →
         x # (abs y M)
```

The notion of free variable is as usual. We could do with only one of the two notions of freedom and freshness. Due to our already commented interest in proceeding as straightforwardly as possible, we avoid the corresponding discussion.

```
data _free_ : V → Λ → Set where
  var  : {x y : V} → y ≡ x →
         x free var y
  appl : {x : V} {M N : Λ} → x free M →
          x free (app M N)
  appr : {x : V} {M N : Λ} → x free N →
          x free (app M N)
  abs  : {x y : V} {M : Λ} → x free M → y ≢ x →
         x free (abs y M)
```

2.2 Substitutions

Substitutions are functions from variables to terms.

```
Σ = V → Λ
```

We actually work with finite, identity almost everywhere functions. So they will be generated by an update operation <+ up from the identity function ι.

```
ι : Σ
ι = id ∘ var

_<+_ : Σ → V × Λ → Σ
(σ <+ (x, M)) y with x ≟ y
... | yes _ = M
... | no _ = σ y
```

Moreover, most of the relevant properties of substitutions concern their *restrictions* to (the free variables of) given terms. We write P the type of restrictions and $\sigma \downharpoonright M$ the restriction of substitution σ to a term M. We refer sometimes as the *codomain* of a restriction $\sigma \downharpoonright M$ to the (list of) terms N such that $\sigma x = N$ for x *free* M. Restrictions, however, are not functions, but just finite fragments thereof that it is convenient to consider for most of the definitions to be given. For instance, freshness in a restriction is defined as follows:

```
_#_⫰_ : V → Σ → Λ → Set
x # σ ⫰ M = (y : V) → y free M → x # (σ y)
```

The right notion of identity of substitutions has to be formulated for restrictions:

```
_≡_⫰_ : Σ → Σ → Λ → Set
σ ≡ σ' ⫰ M = (x : V) → x free M → σ x ≡ σ' x
```

The *choice* function: We postulate a *choice* function χ that returns a variable fresh in the codomain of a given restriction. It is actually a function of the codomain in question, i.e. it depends in fact only on a finite set of variables, returning a variable not in such set. Formally, it must satisfy the following:

1. $\chi : \mathsf{P} \to \mathsf{V}$.
2. $\chi(\sigma \downharpoonright M) \overline{\#}\ \sigma \downharpoonright M$.
3. $\chi(\sigma \downharpoonright M) = \chi(\sigma' \downharpoonright M') \Leftrightarrow$
   $[x \textit{ free } M \ \Leftrightarrow\ x \textit{ free } M'$
   $\wedge$
   $(\forall x \textit{ free } M)(y \textit{ free } (\sigma x) \Leftrightarrow\ y \textit{ free } (\sigma' x))]$.

The Agda code corresponding to these axioms is straightforward but we choose not to show it here for reasons of layout.

χ is indeed implementable as the variables form an enumeration.

The substitution action: The action of substitutions on terms is defined by *structural* recursion. The χ function is used to perform uniform capture-avoiding renaming.

```
_•_ : Λ → Σ → Λ
(var x) • σ = σ x
(app M N) • σ = app (M • σ) (N • σ)
(abs x M) • σ = abs y (M • (σ <+ (x, var y)))
  where y = χ (σ, abs x M)
```

It follows that:

```
lemma-subst-σ≡ : {M : Λ} {σ σ' : Σ} →
  σ ≡ σ' ⫰ M → (M • σ) ≡ (M • σ')
```

i.e. equal substitutions acting on a term yield the same result.

2.3 Alpha conversion

The inductive definition is simple and follows the structure of terms:

```
data _∼α_ : Λ → Λ → Set where
  var : {x : V} →
    (var x) ∼α (var x)
  app : {M M' N N' : Λ} → M ∼α M' → N ∼α N' →
    (app M N) ∼α (app M' N')
  abs : {M M' : Λ} {x x' y : V} → y # (abs x M) →
    y # (abs x' M') →
    (M • (ι <+ (x, var y))) ∼α (M' • (ι <+ (x', var y))) →
    (abs x M) ∼α (abs x' M')
```

We show the last rule in a more friendly notation. Notice its symmetry:

$$\frac{M(\iota{<}{+}(x,z)) \sim_\alpha M'(\iota{<}{+}(x',z))}{\lambda x M \sim_\alpha \lambda x' M'}\ z\#\lambda x M, \lambda x' M'$$

The α equivalence of substitutions is also defined on restrictions:

```
_∼α_⫰_ : Σ → Σ → Λ → Set
σ ∼α σ' ⫰ M = (x : V) → x free M → σ x ∼α σ' x
```

2.4 Lemmas

The following are the main results of the development. We give all the statements in mathematical notation and in Agda. We give comments about the proofs and show some of them explicitely.

Lemma 1. $M \sim_\alpha M' \Rightarrow \chi(\sigma \downharpoonright M) = \chi(\sigma \downharpoonright M')$.

```
lemma-χ : {M M' : Λ} {σ : Σ} → M ∼α M' →
  χ (σ, M) ≡ χ (σ, M')
```

Lemma 2. $M \sim_\alpha M' \Rightarrow (x \textit{ free } M \Leftrightarrow x \textit{ free } M')$.

```
lemmaM∼M'→free→ : {M M' : Λ} → M ∼α M' →
  (z : V) → z free M → z free M'
lemmaM∼M'→free← : {M M' : Λ} → M ∼α M' →
  (z : V) → z free M' → z free M
```

The proof is by induction on the relation _∼α_, and requires three freshness lemmas besides the preceding one.

Lemma 3. $M \sim_\alpha M' \Rightarrow M\sigma = M'\sigma$.

```
lemmaM∼M'→Mσ≡M'σ : {M M' : Λ} {σ : Σ}
  → M ∼α M' → M • σ ≡ M' • σ
```

The proof is again by induction on the relation _∼α_. It requires the χ lemma as well as the following:

```
lemma<+ : {x y z : V} {M : Λ} {σ : Σ} →
  z # (abs x M) →
  M • (σ <+ (x, var y)) ≡
  (M • (ι <+ (x, var z))) • (σ <+ (z, var y))
```

LEMMA 4. $M\iota = M'\iota \Rightarrow M \sim_\alpha M'$.

```
lemmaMι≡M'ι→M∼M' : {M M' : Λ} →
  M • ι ≡ M' • ι → M ∼α M'
```

This is the only proof done by induction on length of the term. We use the Agda 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.

COROLLARY 1. $M \sim_\alpha M' \Leftrightarrow M\iota = M'\iota$.

The previous result is not necessary in our formalisation but it is a nice result, which is immediate from the former lemmas. In particular, the direction from left to right amounts to a normalizing property of ι with respect to ∼α.

LEMMA 5. $\sim_\alpha$ *is a congruence.*

It is enough to show that $\sim_\alpha$ is an equivalence relation. We include the code, which is very short and simple enough to be read directly:

```
ρ : Reflexive _∼α_
ρ {M} = lemmaMι≡M'ι→M∼M' refl

σ : Symmetric _∼α_
σ {M} {N} M∼N
  = lemmaMι≡M'ι→M∼M'
    (sym (lemmaM∼M'→Mσ≡M'σ M∼N))

τ : Transitive _∼α_
τ {M} {N} {P} M∼N N∼P
  = lemmaMι≡M'ι→M∼M'
    (trans (lemmaM∼M'→Mσ≡M'σ M∼N)
      (lemmaM∼M'→Mσ≡M'σ N∼P))
```

The proof uses just the corresponding properties of equality.

LEMMA 6 (SUBSTITUTION LEMMA FOR $\sim_\alpha$). $M \sim_\alpha M'$, $\sigma \sim_\alpha \sigma' \downharpoonright M \Rightarrow M\sigma \sim_\alpha M'\sigma'$.

The full code is:

```
lemma-subst : {M M' : Λ} {σ σ' : Σ} →
  M ∼α M' → σ ∼α σ' ⌊ M → (M • σ) ∼α (M' • σ')
lemma-subst {M} {M'} {σ} {σ'} M∼M' σ∼σ'⌊M
  = begin
      M • σ
      ∼⟨ lemma-subst-σ∼ σ∼σ'⌊M ⟩
      M • σ'
      ≈⟨ lemmaM∼M'→Mσ≡M'σ M∼M' ⟩
      M' • σ'
```

Where:

```
lemma-subst-σ∼ : {M : Λ} {σ σ' : Σ} →
  σ ∼α σ' ⌊ M → (M • σ) ∼α (M • σ')
lemma-subst-σ∼ {M} {σ} {σ'} σ∼ασ'⌊M
  = lemmaMι≡M'ι→M∼M' (begin≡
    (M • σ) • ι
    ≡⟨ lemma· {M} {σ} {ι} ⟩
    M • (ι · σ)
    ≡⟨ lemma-subst-σ≡ {M}
         {ι · σ} {ι · σ'}
         (lemma-σ⌊ σ∼ασ'⌊M) ⟩
    M • (ι · σ')
    ≡⟨ sym (lemma· {M} {σ'} {ι}) ⟩
    (M • σ') • ι
    )
```

3. CONCLUSIONS

The potential contributions of this paper will necessarily have to do with the ease of formalisation of results that are basic and well-known but usually difficult to treat in a fully formal manner. Therefore an assessment must proceed by way of comparison. To begin with, we believe that we have improved on the formalisation [4] already mentioned in the introduction in at least two respects:

1. The use of better founded relations of equivalence between substitutions.
2. The simplicity of the proofs yielding that α conversion is an equivalence relation.

The first item has to do with our preference for finitely given equivalences, determined by restrictions of the substitutions to given terms, instead of the infinite, undecidable extensional identity. The second item, in turn, can be attested by just noting the remarks in [4] concerning the (unexpected) complexity of the proofs of symmetry and, especially, transitivity of the α conversion. The issue points in fact also to an improvement of our work over Stoughton's original presentation, as it is based on a much simpler definition of α conversion: Stoughton's comprises in fact six rules, whereas ours requires three, in correspondence with the structure of terms. This obviously simplifies reasoning by induction on this relation. As already said, Stoughton's remaining rules, which are precisely the ones to the effect that α conversion is an equivalence relation, are obtained from ours via (quite) painless proofs. Central for the simplicity of those proofs is the lemma 3 to the effect that two terms that converge under the effect of the identity substitution are α convertible. This was given in [5] but not used in the way we have done here. We have proven it by well-founded induction on the length of terms because that appeared to be the simplest way available. This method of proof could be straightforwardly encoded in Agda using library functions. It would be interesting to investigate whether Agda's facility of *sized types* could be used to allow a simple structural recursion instead of the general well-founded one.

As we said earlier, our interest in this work lies primarily in investigating the brevity of the development up to the substitution lemma for α conversion when sticking to the conventional syntax of the Lambda calculus. In this respect, the main work to compare is [7] which is indeed quite successful in using modified rules of α conversion and β reduction to formally prove the Church-Rosser theorem. We ought to proceed to a similar development but, in first appreciation, we prefer Stoughton's less contrived formulations. We are in fact ready to sustain that substitution ought to be considered in Stoughton's multiple form instead of the conventional unary one that brings about so many inconveniences.

Another approach worth investigating in the context of the ordinary syntax of the Lambda calculus is an adaptation of the so-called Nominal Abstract Syntax [6]. The important point in this respect is that the renaming of bound variable that is necessary for defining substitution and lies at the origin of α conversion can naturally be conceived as an operation of a nature different from that of full substitution. In particular, the Nominal Abstract Syntax approach would implement it by means of an operation of *swapping*, i.e. of *interchange* of names which is bijective and thus mathematically much better behaved than simple substitution. We are currently working in formalisations along this "name swapping" line.

In fact, the present work takes place within a project aiming at carrying out extensive comparisons of formalisation techniques using the meta-theory of Lambda calculus as testbed. So we aim at eventually obtain conclusions regarding also the use of Frege-like, locally nameless and higher-order syntax. Besides the detailed comparisons that will become possible we aim with this work at improving our skills in formalisation and programming, particularly in dependently typed languages.

4. REFERENCES

[1] A. Church. A set of postulates for the foundation of logic part i. *Annals of Mathematics*, 33(2):346–366, 1932. http://www.jstor.org/stable/1968702Electronic Edition.

[2] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, 1958. Second printing 1968.

[3] 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.

[4] G. Lee. Proof pearl: Substitution revisited, again. Hankyong National University, Korea, http://formal.hknu.ac.kr/Publi/Stoughton.pdf.

[5] A. Stoughton. Substitution revisited. *Theor. Comput. Sci.*, 59:317–325, 1988.

[6] C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.

[7] R. Vestergaard and J. Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names. *Inf. Comput.*, 183(2):212–244, 2003.