Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 312 (2015) 215–230

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Formalisation in Constructive Type Theory of Stoughton's Substitution for the Lambda Calculus

Álvaro Tasistro [1] Ernesto Copello [2] Nora Szasz [3]

Universidad ORT Uruguay, Montevideo, Uruguay

Abstract

In [25], Alley Stoughton proposed a notion of (simultaneous) substitution for the Lambda calculus as formulated in its original syntax –i.e. with only one sort of symbols (names) for variables– and 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 the connection to α-conversion. In this paper we present a formalisation of Stoughton's work in Constructive Type Theory using the language Agda, which reaches up to the Substitution Lemma for α-conversion. The development has been quite inexpensive e.g. in labour cost, and we are able to formulate some improvements over the original presentation. For instance, our definition of α-conversion is just syntax directed and we prove it to be an equivalence relation in an easy way, whereas in [25] the latter was included as part of the definition and then proven to be equivalent to an only nearly structural definition as corollary of a lengthier development. 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.

Keywords: Formal Metatheory, Lambda Calculus, Constructive Type Theory

1 Introduction

The Lambda calculus was introduced by Church [5] without a definition of substitution. The complexity of this operation was actually a prime motivation for Curry and Feys to provide the first definition in [7], somewhat as follows:

[1] Email: tasistro@ort.edu.uy
[2] Email: copello@ort.edu.uy
[3] Email: szasz@ort.edu.uy

http://dx.doi.org/10.1016/j.entcs.2015.04.013

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

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

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

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 lesser 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. This is extremely difficult to formalise in any of the several proof assistants available. Besides, there is the inconvenience that proofs of properties of the substitution operation have to be conducted by induction on the size of terms and have generally three subcases, with two invocations to the induction hypothesis in the subcase considered above. These observations prompt the search for a simpler definition.

As is well known, several of the proposed solutions take the path of modifying the syntax of the language as used above. Such a decision is indeed well motivated, especially if the alternative is to employ for the local or bound names a type of symbol different from the one of the variables: that was, to begin with, Frege's choice in the first fully fledged formal language [11], which featured universal quantification as a binder, and was later made again by at least Gentzen [12], Prawitz [22] and Coquand [6]. Within the field of machine-checked meta-theory, McKinna and Pollack[18] 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. Another alternative is of course de Bruijn's nameless syntax [8] or its more up-to-date version *locally nameless* syntax [2,4], which uses names for the free or global variables and the indices counting up to the binding abstractor for the occurrences of local parameters. That is to say that locally nameless syntax is a variation of Frege style syntax in which the local parameters are nameless. The overhead in this case is the following: a well-formedness predicate ensures that valid terms do not contain too large indices and, besides substitution for the free names, an "opening" operation is needed for consuming a λ-abstractor with a term. Opening with a fresh name is used to ensure that only well-formed terms in which indices never refer to non-local abstractors are ever used and therefore the need of shifting indices during reduction is avoided. A dual operation of variable closing is also necessary, which abstracts a name from a term, replacing it with a bound variable, i.e. an index. There certainly is in this case a relief in not having to consider α-conversion; but, at the same time, the nameless syntax seriously affects readability of terms by humans, so that one can say that it stays too distant from a natural syntax. The same has to be said of the map representation introduced in [24].

Meanwhile, it remains interesting to investigate how well it is possible to do with the original, ordinary syntax of the Lambda calculus. In the informal setting (e.g. [3,14]) the standard way to proceed is to identify α-equivalent terms, choosing each time a convenient representative, according to the so-called *Barendregt's variable convention*: just choose the bound variables mutually distinct and also distinct from any free variable in the current context. Now, this convention needs careful formulation in order for it to be compatible with structural induction and recursion, for the latter demand induction steps to work for *arbitrary* and not just conveniently chosen variables. The corresponding formalisation has been provided in form of α-induction and recursion principles in e.g. [21,20], and has been implemented in the form of a package on the proof assistant Isabelle-HOL [26] as well as in Coq [1]. If, however, one insists in preserving Curry and Feys' basic approach and work directly on concrete terms, then there appears a thesis worth testing, namely that it was Stoughton [25] 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 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: 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 [9], whereas the one of uniform renaming is originally presented in [23]. The resulting theory of substitutions and α-conversion in the context of the Lambda calculus has many good 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. Stoughton's work starts with a definition of α-conversion as the least congruence induced by appropriate renaming of bound variable and ends up with its characterisation in the form of a (nearly) syntax-directed inductive definition. In the course of this main development, the so-called Substitution Lemma for α-conversion is proven, which expresses the compatibility between substitution and α-equivalence.

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. Such formalisation has been undertaken by Lee in [16] 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 [25]) and defines the propositional identity of substitutions as their extensional equivalence by formulating a postulate in Coq. We are rather unsatisfied with this strategy, 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 by actually working on restrictions of substitutions to (the free variables of) terms. This notion of restriction turns out actually to constitute the relevant one concerning substitutions. 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: As already said, Stoughton's version is as the least congruence generated by a simple renaming of bound variable, which is eventually proven equivalent to a definition directed by the structure of terms, only that including two different cases for abstractions. On the other hand, Lee's formalisation gives a structural definition and then takes considerable effort to prove it an equivalence relation. Our definition of α-equivalence directly follows the structure of terms and the proof that it is an equivalence relation is quite simple, although it requires induction on the length of the terms at one point. As different from both Stoughton and Lee, we do not need the already mentioned substitution lemma for proving that the syntax directed definition of α-conversion gives rise to a congruence. We nevertheless conduct our development up to that result, 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 leading to the Church-Rosser theorem.

In the next section the formal development is presented with as much detail as we consider appropriate, next to which we include a final section with concluding comments.

2 Formalisation

We use the language Agda [19]. The present is actually a literate Agda document, where we hide some code for reasons of conciseness. The entire code is available at:

`http://fi.ort.edu.uy/innovaportal/file/17663/1/lsfa.lagda`.

Agda implements Constructive Type Theory [17] (*type theory* for short). It is actually a functional programming language in which:

(i) 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.

(ii) 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.

(iii) Functions on inductive types are defined by *pattern-maching* equations.

(iv) 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.

(v) 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.

2.1 Syntax

The set Λ of terms is as usual. It is built up from a denumerable set of variables V.

```
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. We import the terminology and notation from the works on nominal abstract syntax, see e.g. [27]. Parameters to a function written between curly brackets can be omitted when invoking the function.

```
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, which we write _*_, is as usual. Freshness and freedom are of course the negation of each other. We work by defining two positive notions instead of using negation. Or we can say: in programming terms, we proceed just naturally by introducing the two types. The programming way is to our mind the natural practice to carry out in a constructive mathematics setting.

```
data _*_ : V → Λ → Set where
  var  : {x y : V} → y ≡ x →
```

```
          x * var y
   appl : {x : V} {M N : Λ} → x * M →
          x * (app M N)
   appr : {x : V} {M N : Λ} → x * N →
          x * (app M N)
   abs  : {x y : V} {M : Λ} → x * M → y ≢ x →
          x * (abs y M)
```

Sameness of free variables is an important relation between terms, which is defined below. The symbol $\times$ stands for the type of ordered pairs which, under the interpretation of propositions as types is the logical conjunction.

```
_∼*_ : (M M' : Λ) → Set
_∼*_ M M' = (∀ x → x * M → x * M') × (∀ x → x * M' → x * M)
```

This definition should be compared with the standard practice (as in Stoughton's work) of defining the set of free variables and then using equality of (finite) sets.

Of course sameness of fresh variables is defined equally. We omit the details.

2.2 Substitutions

Substitutions are functions from variables to terms.

```
Σ = V → Λ
```

We actually need finite, identity almost everywhere functions. So we shall consider functions 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
```

The latter defines the term corresponding to each variable in an updated substitution.

Now, most of the relevant properties of substitutions concern their *restrictions* to (the free variables of) given terms. We write R the type of restrictions and $\sigma \downharpoonright M$ the restriction of substitution σ to a term M.

```
R = Σ × Λ
```

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

$$\begin{array}{l}\_\equiv\_ \ : \ \mathsf{R} \to \mathsf{R} \to \mathsf{Set} \\ (\sigma, \mathsf{M}) \equiv (\sigma', \mathsf{M}') \ = \ (\mathsf{M} \sim^{*} \mathsf{M}') \times ((\mathsf{x} \ : \ \mathsf{V}) \to \mathsf{x} * \mathsf{M} \to \sigma\, \mathsf{x} \equiv \sigma'\, \mathsf{x})\end{array}$$

In a similar way, freshness and freedom of variables are extended to restrictions, as follows:

$$\begin{array}{l}\_\#\downharpoonleft\_ \ : \ \mathsf{V} \to \mathsf{R} \to \mathsf{Set} \\ \mathsf{x} \#\downharpoonleft (\sigma, \mathsf{M}) \ = \ (\mathsf{y} \ : \ \mathsf{V}) \to \mathsf{y} * \mathsf{M} \to \mathsf{x} \# (\sigma\, \mathsf{y})\end{array}$$

$$\begin{array}{l}\_*\downharpoonleft\_ \ : \ \mathsf{V} \to \mathsf{R} \to \mathsf{Set} \\ \mathsf{x} *\downharpoonleft (\sigma, \mathsf{M}) \ = \ \exists\, (\lambda\, \mathsf{y} \to \mathsf{y} * \mathsf{M} \times \mathsf{x} * (\sigma\, \mathsf{y}))\end{array}$$

Sameness of free variables between two restrictions is equally straightforward:

$$\begin{array}{lcl}\_\sim^{*}\downharpoonleft\_ \ : \ \mathsf{R} \to \mathsf{R} \to \mathsf{Set} & & \\ (\sigma, \mathsf{M}) \sim^{*}\downharpoonleft (\sigma', \mathsf{M}') & = & ((\mathsf{x} \ : \ \mathsf{V}) \to \mathsf{x} *\downharpoonleft (\sigma, \mathsf{M}) \to \mathsf{x} *\downharpoonleft (\sigma', \mathsf{M}')) \times \\ & & ((\mathsf{x} \ : \ \mathsf{V}) \to \mathsf{x} *\downharpoonleft (\sigma', \mathsf{M}') \to \mathsf{x} *\downharpoonleft (\sigma, \mathsf{M}))\end{array}$$

And the same can be done for sameness of fresh variables. We omit the definition.

2.3 The choice function.

A mechanism is necessary for selecting a variable appropriate for renaming in order to avoid capture when performing substitutions. Stoughton postulates a function *choice* which acts on non-empty sets of variables. Then, when a substitution σ is to act on an abstraction $\lambda x.M$, a variable y is selected that does not occur free in the restriction of σ to $\lambda x.M$ and x is renamed to y in a way that we shall show in the next subsection. Stoughton formulates y as the result of applying the postulated *choice* function to the complement of the set of free variables of the mentioned restriction of the substitution σ. We prefer to simplify the formulation and implement a *choice* function χ that acts on a restriction returning the first variable fresh in it. Therefore, the result depends actually only on finite sets of variables, i.e. χ behaves the same for restrictions possessing the same free variables. Formally, our development uses the following facts about the χ function:

(i) $\chi : \mathsf{R} \to \mathsf{V}$.

(ii) $\chi(\sigma \downharpoonleft M)\#(\sigma \downharpoonleft M)$.

(iii) $(\sigma \downharpoonleft M) \sim_{*} (\sigma' \downharpoonleft M') \Rightarrow \chi(\sigma \downharpoonleft M) = \chi(\sigma' \downharpoonleft M')$.

These results follow from the implementation of χ, which we explain in detail in the rest of this subsection.

Since the variables form an enumeration, we can always choose the first one not belonging to the finite list of the free variables of $\sigma \downharpoonleft M$. This list is calculated by concatenating the result of mapping the function fv ∘ σ onto the list of free variables of M.

```
χ : R → V
χ (σ, M) = χ' (concat (map (fv ∘ σ) (fv M)))
```

Here the function χ' chooses the first variable not belonging to the finite list of variables given to it as argument. Let us now assume for convenience of presentation that the variables are just natural numbers. Then χ' is implemented using the auxiliary χaux function, which linearly searches up from a given number n the first variable (number) not in the given list xs. To define this function by primitive recursion, an extra argument m is given, which bounds the number of attempts to be effected until the desired variable is found. Initially, this argument is instantiated with the length of the given list, which is indeed such a bound whenever χ' is called with n= 0. The argument m decreases at each step, which grants termination of the function. If one looks at the type of the result of this function, one will notice that, apart from the desired fresh variable, two extra values are returned: One is a proof that either the returned variable does not belong to the list or that the maximum desired number of attempts has been reached, while the other tells that variables under the one selected are all in the input list. These two results are guaranteed by the invariants expressed by the two last parameters of the function. The code of χ' is shown here below after the type of χaux.

```
χaux : (n m k : V) → (xs : List V) →
       n + m ≡ k → ((y : V) → y < n → y ∈ xs) →
       ∃ (λ v → (v ∉ xs ∨ v ≡ k) × ((y : V) → y < v → y ∈ xs))
  --
χ' : List V → V
χ' xs = proj₁ (χaux 0 (length xs) (length xs) refl xs (y<0⇒y∈xs xs))
```

Besides the present code, we have written a full proof of the correctness of the choice function χ.

2.4 The substitution action

The action of substitutions on terms is defined by *structural* recursion. Uniform capture-avoiding renaming is performed by using the already introduced choice function χ. Notice that the renaming extends the originally given substitution. It is clear that the result of substituting in $\lambda x.M$ is independent of $\sigma\, x$ and that, consequently, the selected new name y must not capture variables in the restriction of σ to the abstraction $\lambda x.M$.

```
_•_ : 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, M) ≡ (σ', M) → (M • σ) ≡ (M • σ')
```

i.e. equal substitutions acting on the term on which they coincide yield the same result. The following lemmas can be read as expressing the avoidance of capture in substitution:

```
lemmafreeσ→ : {x : V} {M : Λ} {σ : Σ} → x * (M • σ) → x *⌊ (σ, M)
```

```
lemmafreeσ← : {x : V} {M : Λ} {σ : Σ} → x *⌊ (σ, M) → x * (M • σ)
```

They follow by easy inductions on the _*_ relation.

2.5 *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 symmetry of this 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, say from left to right, under appropriate circumstances. The α-equivalence of substitutions is also properly defined on restrictions:

$$(\sigma \downharpoonleft M) \sim_\alpha (\sigma' \downharpoonleft M') \Rightarrow M \sim_* M' \wedge (\forall x)(x * M \Rightarrow \sigma\ x \sim_\alpha \sigma'\ x)$$

2.6 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 explicitly.

Lemma 1 *$M \sim_\alpha M' \Rightarrow M \sim_* M'$.*

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

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

The proof is by induction on the relation _∼α_, and requires three freshness lemmas besides the third χ lemma.

Lemma 2 *$M \sim_\alpha M' \Rightarrow M\sigma = M'\sigma$,*

i.e. any substitution *equalizes* α convertible terms, which is due to the uniform capture-avoiding renaming. Formally:

```
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_ and benefits itself from the symmetry of the rule of abstractions. It requires the following lemma:

```
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 3 *$M\iota = M'\iota \Rightarrow M \sim_\alpha M'$.*

```
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, which is necessary because of the λ rule of the definition of ∼α. Indeed, notice that the premise of that rule does not mention the bodies of the abstractions involved, but the results of renamings applied to them. 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. We could alternatively have used Agda's sized types.

The following result is not necessary in our formalisation but it is a nice one, and 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.

Corollary 2.1 $M \sim_\alpha M' \Leftrightarrow M\iota = M'\iota$.

Lemma 4 $\sim_\alpha$ *is a congruence.*

It is enough to show that $\sim_\alpha$ is an equivalence relation. We include the full code, which is very short. The proof uses just the corresponding properties of equality (of terms), whose proofs are here named refl, sym and trans.

```
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))
```

We end up with the lemma showing that substitution is compatible with α-conversion both of the term wherein the substitution is performed and of the substituted terms. For convenience we use the notation $\sigma \sim_\alpha \sigma' \downharpoonleft M$ to stand for $\sigma \downharpoonleft M \sim_\alpha \sigma' \downharpoonleft M$.

Lemma 5 (Substitution Lemma for $\sim_\alpha$) $\left.\begin{array}{r} M \sim_\alpha M' \\ \sigma \sim_\alpha \sigma' \downharpoonleft M \end{array}\right\} \Rightarrow M\sigma \sim_\alpha M'\sigma'.$

The proof given below is in a convenient calculational style. It uses a lemma: $\sigma \sim_\alpha \sigma' \downharpoonleft M \Rightarrow M\sigma \sim_\alpha M\sigma'$, whose proof is shown further down. 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

Our formalisation in Constructive Type Theory of Stoughton's theory of substitutions and α-conversion presents the following features:

(i) It bases itself upon the notion of *restriction* of a substitution to (the free variables of) a term. For instance, the substitution lemma which is our final result is about α-equivalent restrictions to those α-equivalent terms wherein they are performed. As a consequence we have used the corresponding finite notions of equality and α-equivalence, whereas Stoughton and the formalisation by Lee [16] 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 behaviour of substitutions manifests itself in interaction with terms.

(ii) 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 [16]. 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, which we prove for its own sake as a relevant piece of meta-theory. The induction on the size of terms could be straightforwardly encoded in Agda using library functions.

(iii) The *choice* function needed for selecting an appropriate variable for renaming during substitution in order to avoid capture is given both a simpler type and an explicit implementation, with a full correctness proof.

Our work was primarily directed towards investigating the brevity of the development up to the substitution lemma for α-conversion when sticking to the conventional syntax of the Lambda calculus, and without quotienting the set of terms by the relation of α-equivalence. It could be interesting to point out that the present formalisation took one man-week to complete, which is one aspect in which the convenience of the approach manifests itself.

Within the general approach to syntax chosen, the main work to compare – besides the one by Lee already commented upon– is Vestergaard and Brotherston [28] which is indeed quite successful in using modified rules of α-conversion and β reduction to formally prove the Church-Rosser theorem in Isabelle-HOL. We are yet to complete a similar development but, in first appreciation, we prefer Stoughton's less contrived formulations. We are in fact ready to claim that substitution ought to be considered in Stoughton's multiple form instead of the conventional unary one that brings about so many inconveniences.

Peter Homeier [13] has tried to recreate Barendregt Variable Convention using Stoughton's multiple substitution in a proof of the Church-Rooser theorem in Isabelle-HOL. His work defines $\sim_\alpha$ without using the substitution operation, by means of only three syntax directed rules. Unfortunately, the definition seems to relate more terms than just α-equivalent ones. For example, the term $\lambda y.\lambda x.\lambda x.x\ y$ is $\sim_\alpha$ in his definition with $\lambda z.\lambda u.\lambda v.u\ z$, while they are clearly not equivalent. In this work, Homeier uses a HOL package to implement the quotient type of terms under $\sim_\alpha$. This package provides a mapping function between raw terms and quotient classes, so that if two raw terms are related by $\sim_\alpha$, then their corresponding quotient objects are equal. The translation –which might be implemented via some kind of de Bruijn representation– is not quite transparent: for every function defined at the concrete term level an often complicated proof of adequacy must be given to obtain a similar function at the type of the quotient classes.

Ford and Mason [10] evaluated the PVS proof assistant proving Church-Rooser in a call by value lambda-calculus in the spirit of Landin's ISWIM [15]. The substitution operation uses the renaming operation on bound variables to prevent capture in the abstraction case, and thus proceeds by recursion on the term's size. In this work a relevant technique in the inductive definition of the $\sim_\alpha$ relation is introduced, namely *cofinite quantification*. The premise in the abstraction case (fig. 1) says 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 abstractions' bodies, it should hold that both bodies are in the $\sim_\alpha$ relation after renaming the bound variables to x.

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

Fig. 1. Ford and Mason abstraction case of the α-conversion definition.

It is important to point out that the use of renaming in the implementation of the substitution operation leads to the necessity of generally employing induction on the size when induction on terms is to be carried out. The same is the case if swapping of bound names, i.e. bijective renaming, is used, but of course not with Stoughton's definition. The issue can also be simplified by use of α-induction and recursion principles as formulated by [21,20] in the context of the framework of Nominal Abstract Syntax. In this connection we are working on formulations of α-induction and recursion principles in constructive type theory using Agda, trying to elaborate on implementations like [26] and [1].

A natural continuation of the present work would, to begin with, consist in completing some of the fundamental theorems of the meta-theory of the Lambda-calculus, namely Church-Rosser as well as preservation of typing under α-equivalence and β-reduction for simple type systems.

It is also interesting to investigate a formulation of the present theory that makes use of dependently typed features in a more intensive and specific way than the present paper. This had originally the intention to stay close to Stoughton's presentation which is carried out in standard, and thus first-order, mathematics but it can be elaborated further by considering for instance types of terms indexed by contexts of free variables as well as substitutions "typed" by such contexts.

Finally, we are also interested in extending Stoughton's substitutions to a general framework of languages with binders, set up in very much the same way as the nominal framework mentioned above.

References

[1] Aydemir, B., A. Bohannon and S. Weirich, *Nominal reasoning techniques in Coq*, Electron. Notes Theor. Comput. Sci. **174** (2007), pp. 69–77.
URL http://dx.doi.org/10.1016/j.entcs.2007.01.028

[2] Aydemir, B., A. Charguéraud, B. C. Pierce, R. Pollack and S. Weirich, *Engineering formal metatheory*, in: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08 (2008), pp. 3–15.
URL http://doi.acm.org/10.1145/1328438.1328443

[3] Barendregt, H., "The Lambda Calculus: Its Syntax and Semantics," Studies in Logic and the Foundations of Mathematics, Elsevier Science, 1985.

[4] Charguéraud, A., *The locally nameless representation*, J. Autom. Reasoning **49** (2012), pp. 363–408.
URL http://dx.doi.org/10.1007/s10817-011-9225-2

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

[6] Coquand, T., *An algorithm for testing conversion in type theory*, in: G. Huet and G. Plotkin, editors, *Logical Frameworks*, Cambridge University Press, Cambridge, 1991 pp. 255–279.

[7] Curry, H. B. and R. Feys, "Combinatory Logic, Volume I," North-Holland, 1958, xvi+417 pp., second printing 1968.

[8] de Bruijn, N., *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*, Indagationes Mathematicae (Proceedings) **75** (1972), pp. 381 – 392.
URL http://www.sciencedirect.com/science/article/pii/1385725872900340

[9] Ebbinghaus, H., J. Flum and W. Thomas, "Mathematical logic (2. ed.)," Undergraduate texts in mathematics, Springer, 1994.

[10] Ford, J. M. and I. A. Mason, *Operational techniques in PVS - A preliminary evaluation*, Electr. Notes Theor. Comput. Sci. **42** (2001), pp. 124–142.
URL http://dx.doi.org/10.1016/S1571-0661(04)80882-X

[11] Frege, G., "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.

[12] Gentzen, G., "The Collected Papers of Gerhard Gentzen," Studies in Logic and the Foundations of Mathematics, North-Holland, 1969, edited by M. E. Szabo.

[13] Homeier, P. V., *A proof of the church-rosser theorem for the lambda calculus in higher order logic*, in: *TPHOLs'01: Supplemental Proceedings*, 2001, pp. 207–222.

[14] Krivine, J., "Lambda-calculus, types and models," Ellis Horwood series in computers and their applications, Masson, 1993.

[15] Landin, P. J., *The next 700 programming languages*, Commun. ACM **9** (1966), pp. 157–166.
URL http://doi.acm.org/10.1145/365230.365257

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

[17] Martin-Löf, P. and G. Sambin, "Intuitionistic type theory," Studies in proof theory, Bibliopolis, 1984.
URL http://books.google.com.uy/books?id=_D0ZAQAAIAAJ

[18] McKinna, J. and R. Pollack, *Some lambda calculus and type theory formalized*, Journal of Automated Reasoning **23** (1999).
URL http://homepages.inf.ed.ac.uk/rpollack/export/McKinnaPollack99.ps.gz

[19] Norell, U., "Towards a practical programming language based on dependent type theory," Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (2007).

[20] Pitts, A. M., *Nominal logic, a first order theory of names and binding*, Information and Computation **186** (2003), pp. 165 – 193, theoretical Aspects of Computer Software (TACS 2001).
URL http://www.sciencedirect.com/science/article/pii/S089054010300138X

[21] Pitts, A. M., *Alpha-structural recursion and induction*, J. ACM **53** (2006), pp. 459–506.
URL http://doi.acm.org/10.1145/1147954.1147961

[22] Prawitz, D., “Natural Deduction: a Proof-Theoretical Study,” Number 3 in Stockholm Studies in Philosophy, Almquist and Wiskell, 1965.

[23] Révész, G. E., *Axioms for the theory of lambda-conversion*, SIAM J. Comput. **14** (1985), pp. 373–382. URL http://dx.doi.org/10.1137/0214028

[24] Sato, M., R. Pollack, H. Schwichtenberg and T. Sakurai, *Viewing lambda-terms through maps* (2013), available from http://homepages.inf.ed.ac.uk/rpollack/export/Maps_SatoPollackSchwichtenbergSakurai.pdf.

[25] Stoughton, A., *Substitution revisited*, Theor. Comput. Sci. **59** (1988), pp. 317–325.

[26] Urban, C. and M. Norrish, *A formal treatment of the Barendregt variable convention in rule inductions*, in: R. Pollack, editor, *ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized reasoning about languages with variable binding, MERLIN 2005, Tallinn, Estonia, September 30, 2005* (2005), pp. 25–32.
URL http://doi.acm.org/10.1145/1088454.1088458

[27] Urban, C., A. M. Pitts and M. Gabbay, *Nominal unification*, Theoretical Computer Science **323** (2004), pp. 473–497.

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