Theoretical Computer Science 685 (2017) 65–82

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Formal metatheory of the Lambda calculus using Stoughton's substitution

Ernesto Copello, Nora Szasz, Álvaro Tasistro *

Universidad ORT Uruguay, 11100 Montevideo, Uruguay

ARTICLE INFO

Article history:
Received 17 July 2015
Received in revised form 1 August 2016
Accepted 16 August 2016
Available online 19 September 2016

Keywords:
Formal metatheory
Lambda calculus
Type Theory

ABSTRACT

We develop metatheory of the Lambda calculus in Constructive Type Theory, using a first-order presentation with one sort of names for both free and bound variables and without identifying terms up to α-conversion. Concerning β-reduction, we prove the Church–Rosser theorem and the Subject Reduction theorem for the system of assignment of simple types. It is thereby shown that this concrete approach allows for gentle full formalisation, thanks to the use of an appropriate notion of substitution due to A. Stoughton. The whole development has been machine-checked using the system Agda.

1. Introduction

The Lambda calculus was originally formulated with *one* sort of names to serve both as real (free) and apparent (bound) variables [5]. Such design brought about two different issues:

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

Historically too, substitution was granted the more fundamental place within the couple above, since it was used in the definition of α-conversion. Substitution itself was just left undefined by Church [5] in the original formulation of the calculus but later its complexity became a prime motivation for Curry and Feys [8], which provided the first definition, somewhat as follows:

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

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

* Corresponding author.
E-mail addresses: copello@ort.edu.uy (E. Copello), szasz@ort.edu.uy (N. Szasz), tasistro@ort.edu.uy (Á. Tasistro).

http://dx.doi.org/10.1016/j.tcs.2016.08.025
0304-3975/

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

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 smaller than that of $\lambda x.M$, a proof has to be given and, since the renaming is effected by the very same operation of substitution that is being defined, such a proof must be simultaneous to the justification of the well-foundedness of the whole definition. The procedure is indeed intricate and, incidentally, hardly ever mentioned as such in the several texts introducing the Lambda calculus along this line.

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

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

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

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

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

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

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

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

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

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

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

One alternative along this view is to apply the principles of what has become known as *nominal abstract syntax* [21,13,12,22,23]: α-conversion is defined in terms of a basic operation of *name permutation* which acts uniformly on free and bound names. Then principles of induction and recursion are formulated that allow, under natural conditions, to work on abstract α-classes of terms, thus formalising informal practice as carried out in most textbooks. With this method it is possible, for instance, to formally define substitution in the simple way given place to by the application of the so-called *Barendregt variable convention*. The approach has been implemented on machine in [30,1,31,6].

The former is *nominal* syntax as opposed to the "nameless" terms of de Bruijn [9] or its more up-to-date version, *locally nameless syntax* [2,4]. These works take the radical approach to eliminate bound names and therefore α-conversion altogether. The result is a more machine-like presentation of the calculus for which a certain overhead necessary to ensure soundness (particularly in handling substitution) cannot be entirely avoided.

Finally, probably the first proposal of formalisation of reasoning modulo α-conversion is [15]. This work gives an axiomatisation of the syntax of the Lambda calculus in which equality embodies α-conversion and for whose resulting (abstract) terms a method of definition by recursion is provided. It ultimately rests upon the use of higher-order abstract syntax within the system HOL.

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

A first way out within this perspective consists in employing for the local or bound names a type of symbol different from the one of the real variables: that was, to begin with, Frege's choice in the first fully fledged formal language [11], which featured universal quantification as a binder, and was later made again by at least Gentzen [14], Prawitz [25] and Coquand [7]. Within the field of machine-checked meta-theory, McKinna and Pollack [19] used the approach to develop substantial work in the proof assistant Lego, concerning both the pure Lambda calculus and Pure Type Systems. Now, the method is not without some overhead: there must be one substitution operation for each kind of name and a well-formedness predicate to ensure that bound names do not occur unbound — so that induction on terms becomes in fact induction on this predicate.

If, still, one persists in sticking to the original syntax, there first appears the idea, employed in [26], of introducing an operation of *renaming* consisting in the replacement of a bound name by another. Renaming is simpler than substitution, as it proceeds just naïvely, and can be used to implement α-conversion provided the new name is chosen so that it does not at all occur in the term in question. It is also sufficient, under the same proviso, to implement the complex case of substitution. This latter use of renaming provides a way out of the complexity of the justification of the definition of substitution as exposed above. But, on the other hand, it cannot avoid the need of induction on the size of terms, both for justifying the recursion employed in substitution and in the subsequent proofs involving this operation.

We maintain that the genuinely historical approach to the metatheory of the lambda calculus is the one initiated by Curry and Feys [8] and later pursued at least partly in [16]. And that, when trying to fully formalize this theory and give it an implementation on machine, there appears a thesis worth testing, namely that it was Stoughton [27] who provided the right formulation of substitution. The prime insight is simple: In the difficult case where renaming of a bound variable is necessary, structural recursion is recovered if one lets substitutions become *multiple* (*simultaneous*) instead of keeping them just unary. Moreover, further simplification is achieved if one does not bother in distinguishing so many cases when considering the substitution in an abstraction and just performs uniformly the renaming of the bound variable: indeed, given that equivalence under renaming of bound variables is natural and necessary, it makes no point to try to preserve as much as possible the identity of the concrete terms, as in the Curry–Feys definition. As pointed out by Stoughton, the idea of using multiple substitution comes from [10], whereas the one of uniform renaming is originally presented in [26].

In this paper we present proofs of fundamental results of the metatheory of the Lambda calculus employing Stoughton's definition of substitution. Specifically, concerning β-reduction we prove the Church–Rosser theorem and the Subject Reduction theorem for the system of assignment of simple types. The definitions and proofs have been fully formalised in Constructive Type Theory [18] and machine-checked employing the system Agda [20]. The corresponding code is available at https://github.com/ernius/formalmetatheory-stoughton. In the subsequent text we give the proofs in English with a considerable level of detail so that they serve for clarifying their formalisation. We hope thereby to show that what we have called the historical approach to the metatheory of the calculus is indeed formally and machine-tractable, thanks to the adequate reformulation of the substitution operation. Indeed, the proofs of both principal theorems follow standard strategies, and are formalised in a gentle manner.

The structure of the paper is as follows: in section 2 we begin by introducing the syntax of the Lambda calculus and the definition of substitution, together with some basic propositions. We also present a little theory concerning the composition of substitutions which is not indispensable for establishing our main results but allows for a more elegant presentation thereof and bears some interest in itself. Section 3 is about α-conversion, which is given an inductive definition directed by the structure of terms. We then establish two results, namely that the so defined α-conversion is a congruence and that it is compatible with substitution (the Substitution Lemma). Thereby the two first sections constitute themselves into a reformulation of Stoughton's original paper [27] in which we give alternative definitions and proofs obtaining what we believe is a simpler structure of the whole. Section 4 introduces the notion of β-reduction and proves the Church–Rosser theorem by using the standard method due to Tait and Martin-Löf which involves the formulation and study of the parallel β-reduction. In section 5 we formulate the system of assignment of simple types to terms, and then also prove that it is compatible with substitution. From this result, the closure of typing under α-conversion and the subject reduction property of β-reduction are also proven. The overall conclusions are exposed in section 6.

This paper is an extended and revised version of [29], which only included the treatment of α-conversion up to the Substitution Lemma. Additional material includes Section 4 on β-reduction and the Church–Rosser theorem, Section 5 on the simple typed system and the Subject Reduction theorem, and the part of Section 2 on composition of substitutions. Also, as already mentioned, we use natural mathematical syntax instead of Agda's in order to improve readability.

2. Substitution

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proposition 1.

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

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

We next extend freedom and freshness of variables to restrictions:

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

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

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

Proposition 2.

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

The proof is similar to the one of preceding proposition. □

We likewise extend sameness of free variables to restrictions:

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

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

Proposition 3. $\sigma \restriction M \cong \sigma' \restriction M' \Rightarrow (\sigma \restriction M) \sim_* (\sigma' \restriction M')$. □

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

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

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

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

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

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

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

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

Lemma 1. $\sigma \cong \sigma' \restriction M \Leftrightarrow M\sigma = M\sigma'$.

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

For the case of a variable x, we have:

$\sigma \cong \sigma' \lfloor x$
$\Leftrightarrow$ (Proposition 1)
$\sigma\, x = \sigma' x$
$\Leftrightarrow$ (Action of substitutions)
$x\sigma = x\sigma'$.

For applications MN we observe:

$\sigma \cong \sigma' \lfloor MN$
$\Leftrightarrow$ (Proposition 1)
$\sigma \cong \sigma' \lfloor M \wedge \sigma \cong \sigma' \lfloor N$
$\Leftrightarrow$ (Induction)
$M\sigma = M\sigma' \wedge N\sigma = N\sigma'$
$\Leftrightarrow$ (Application formation)
$M\sigma\ N\sigma = M\sigma'\ N\sigma'$
$\Leftrightarrow$ (Action of substitutions)
$(MN)\sigma = (MN)\sigma'$.

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

$\sigma \cong \sigma' \lfloor \lambda x.M$
$\Rightarrow$ (Proposition 3)
$\sigma \sim_* \sigma' \lfloor \lambda x.M$
$\Rightarrow$ (Choice axiom 2)
$\chi\,(\sigma \lfloor \lambda x.M) = \chi\,(\sigma' \lfloor \lambda x.M)$.

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

$\sigma \cong \sigma' \lfloor \lambda x.M$
$\Leftrightarrow$ (Proposition 1)
$(\sigma, x{:=}y) \cong (\sigma', x{:=}y) \lfloor M$
$\Leftrightarrow$ (Induction)
$M(\sigma, x{:=}y) = M(\sigma', x{:=}y)$
$\Leftrightarrow$ (Abstraction formation)
$\lambda y.M(\sigma, x{:=}y) = \lambda y.M(\sigma', x{:=}y)$
$\Leftrightarrow$ (Action of substitutions)
$(\lambda x.M)\sigma = (\lambda x.M)\sigma'$. □

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

Lemma 2 *(No capture).* $y * M\sigma \;\Leftrightarrow\; y * (\sigma \lfloor M)$.

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

$y * (\lambda x.M)\sigma$
$\Leftrightarrow$ (Action of substitutions)
$y * \lambda z.M(\sigma, x{:=}z)$
$\Leftrightarrow$ (Definition of $*$)
$y * M(\sigma, x{:=}z) \wedge z \neq y$
$\Leftrightarrow$ (Induction)
$y * (\sigma, x{:=}z \lfloor M) \wedge z \neq y$
$\Leftrightarrow$ (Proposition 2)
$y * (\sigma \lfloor \lambda x.M)$. □

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

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

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

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

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

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

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

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

The proof is by some calculation from the definitions. □

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

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

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

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

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

Proposition 7 *(Distributivity of composition over update).*

$$\sigma' \circ (\sigma, x{:=}M) \cong (\sigma' \circ \sigma), x{:=}M\sigma'. \qquad \square$$

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

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

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

$$\begin{aligned}
&(\lambda x.M)(\sigma' \circ \sigma)\\
=\ &\text{(Action of substitutions)}\\
&\lambda y.M((\sigma' \circ \sigma), x{:=}y),
\end{aligned}$$

and, on the other hand,

$$\begin{aligned}
&((\lambda x.M)\sigma)\sigma'\\
=\ &\text{(Action of substitutions, with } x' \#\sigma \restriction \lambda x.M)\\
&\lambda y.(M(\sigma, x{:=}x'))(\sigma', x'{:=}y)\\
=\ &\text{(Induction)}\\
&\lambda y.M(\sigma', x'{:=}y \circ \sigma, x{:=}x')\\
=\ &\text{(Propositions 7 and 6 and Lemma 1)}\\
&\lambda y.M((\sigma', x'{:=}y \circ \sigma), x{:=}y).
\end{aligned}$$

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

$$\begin{aligned}
&((\sigma', x'{:=}y \circ \sigma), x{:=}y)\, z\\
=\ &(z \neq x)\\
&(\sigma', x'{:=}y \circ \sigma)\, z\\
=\ &\text{(Composition of substitutions)}\\
&(\sigma\, z)(\sigma', x'{:=}y)\\
=\ &\text{(Proposition 4, using } x' \#\sigma\, z)\\
&(\sigma\, z)\sigma'\\
=\ &\text{(Composition of substitutions)}\\
&(\sigma' \circ \sigma)\, z\\
=\ &(z \neq x)\\
&((\sigma' \circ \sigma), x{:=}y)\, z. \qquad \square
\end{aligned}$$

The following is obtained by direct calculation:

Proposition 9.

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

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

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

The proof involves simple calculations and distinction of cases. □

3. Alpha-conversion

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

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

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

This definition is inspired in one given in [22] and it is also similar to the one in [24]. The symmetry of the abstraction rule favours certain proofs as we shall comment shortly. It is not present in Stoughton's definition, which renames the bound variable of one of the abstractions into the one of the other under appropriate circumstances. Stoughton's definition includes rules for reflexivity, symmetry and transitivity of the relation right from the beginning and then one of the final results of the work is an almost syntax directed characterisation — namely with two different rules for abstractions. We instead shall show that the present syntax-directed definition gives an equivalence relation, which is the first important result to be reached below. The other one is that α-conversion is compatible with substitution. As we comment in detail in the final section, our development is considerably simpler, although at the price of employing size induction at one point, which we shall indicate. Stoughton's development, meanwhile, is entirely free from size induction.

Our first result is:

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

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

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

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

which follows by just logical calculations. □

We then have:

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

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

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

and

$$
\begin{aligned}
& (\lambda x'.M')\sigma \\
= \ & \text{(Action of substitutions)} \\
& \lambda y.M'(\sigma, x':=y).
\end{aligned}
$$

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

$$
\begin{aligned}
& M(\sigma, x:=y) \\
= \ & \text{(Proposition 10, using } z \# \lambda x.M) \\
& M((\sigma, z:=y) \circ (\iota, x:=z)) \\
= \ & \text{(Action of the composition of substitutions)} \\
& (M(\iota, x:=z))(\sigma, z:=y) \\
= \ & \text{(Induction)} \\
& (M'(\iota, x':=z))(\sigma, z:=y) \\
= \ & \text{(Action of the composition of substitutions)} \\
& M'((\sigma, z:=y) \circ (\iota, x':=z)) \\
= \ & \text{(Proposition 10, using } z \# \lambda x'.M') \\
& M'(\sigma, x':=y). \quad \square
\end{aligned}
$$

We can now get to the following important result:

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

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

$$
\begin{aligned}
& M(\iota, x:=y) = M'(\iota, x':=y) \\
\Rightarrow \ & \text{(Congruence of substitution action)} \\
& (M(\iota, x:=y))\iota = (M'(\iota, x':=y))\iota \\
\Rightarrow \ & \text{(Induction)} \\
& M(\iota, x:=y) \sim_\alpha M'(\iota, x':=y) \\
\Rightarrow \ & (\sim_\alpha, \text{ using } y \# \lambda x.M, \lambda x'.M') \\
& \lambda x.M \sim_\alpha \lambda x'.M'. \quad \square
\end{aligned}
$$

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

The following is now immediate:

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

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

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

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

We hereby achieve a syntax directed characterisation of $\sim_\alpha$, plus the result that it is a congruence, in a much more direct way than the one in Stoughton's paper or its formalisation in [17]. The conciseness achieved justifies, to our mind, the use we have made of size induction; we do, however, remark that this is not essential, in the sense that we could have formalised the whole development as originally by Stouhgton in [27], only that in a much lengthier way.

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

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

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

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

Now we state:

Lemma 7. $\sigma \sim_\alpha \sigma' \upharpoonright M \Rightarrow M\sigma \sim_\alpha M\sigma'$.

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

$(M\sigma)\iota$
= (Action of composition of substitutions)
$M(\iota \circ \sigma)$
= (Since $\sigma \sim_\alpha \sigma' \upharpoonright M$, using Corollary 2 and Lemma 1)
$M(\iota \circ \sigma')$
= (Action of composition of substitutions)
$(M\sigma')\iota$. □

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

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

The proof is now a very short calculation:

$M\sigma$
$\sim_\alpha$ (Lemma 7)
$M\sigma'$
$\sim_\alpha$ (Lemma 4 and reflexivity of $\sim_\alpha$)
$M'\sigma'$. □

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

Corollary 3.

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

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

Corollary 4.

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

As with equality before, the second line above could as well be just $(\iota \circ \sigma) \sim_\alpha \sigma \upharpoonright x$.
Finally, we give three results to be employed in the next section:

Corollary 5.

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

4. Beta-reduction and the Church–Rosser theorem

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

The proof rests upon the property of confluence of the so-called *parallel* reduction, which we present here below as an inductive definition. The rules are essentially the same as in [3] or [28] but we have to add a rule allowing explicit α-conversion, since we are working with concrete terms, i.e. not identified under such relation. The definition below can also be regarded as an inductive formalisation of the version in [16]:

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

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

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

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

We have, to begin with, the following:

Lemma 9.

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

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

It easily follows that

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

Now we prove that parallel reduction is compatible with substitution:

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

The proof is by induction on $\rightrightarrows$. We spell it in full.
For a variable x, we have:

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

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

$(MN)\sigma$
$=$ (Action of substitutions)
$M\sigma\ \ N\sigma$
$\rightrightarrows$ (Induction and $\rightrightarrows$ of applications)
$M'\sigma\ \ N'\sigma$
$=$ (Action of substitutions)
$(M'N')\sigma$.

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

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

$\lambda y.M(\sigma, x{:=}y)$
$\rightrightarrows$ (Induction, with $\sigma, x{:=}y \rightrightarrows \sigma', x{:=}y \restriction M$, and $\rightrightarrows$ of abstractions)
$\lambda y.M'(\sigma', x{:=}y)$
$\sim_\alpha$ (Corollary 5.2, using $y \# \sigma' \restriction \lambda x.M'$ which follows from $y \# \sigma \restriction \lambda x.M$ by Corollary 6)
$(\lambda x.M')\sigma'$.

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

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

$((\lambda x.M)N)\sigma$
$=$ (Action of substitutions, with $y \# \sigma \restriction \lambda x.M$)
$(\lambda y.M(\sigma, x{:=}y))N\sigma$
$\rightrightarrows$ (Induction, with $\sigma, x{:=}y \rightrightarrows \sigma', x{:=}y \restriction M$, and outermost application of the β-rule of $\rightrightarrows$)
$(M'(\sigma', x{:=}y))(\iota, y{:=}N'\sigma')$
$\sim_\alpha$ (Corollary 5.1, using $y \# \sigma' \restriction \lambda x.M'$ which follows from $y \# \sigma \restriction \lambda x.M$ by Corollary 6)
$M'(\sigma', x{:=}N'\sigma')$
$=$ (Composition of substitutions and distributivity over update)
$(M'(\iota, x{:=}N'))\sigma'$.

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

$M\sigma$
$\rightrightarrows$ (Induction)
$M'\sigma'$
$=$ (Lemma 4)
$M''\sigma'$. $\square$

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

Now we stand quite close to establishing the confluence of $\rightrightarrows$, i.e. that $M \rightrightarrows M'$ and $M \rightrightarrows M''$ imply the existence of P such that $M' \rightrightarrows P$ and $M'' \rightrightarrows P$. Actually, a proof by induction on $M \rightrightarrows M'$ with subordinate case analysis of $M \rightrightarrows M''$ can be found in [3] and could easily be adapted — except for the fact that it does not consider steps of $\sim_\alpha$ conversion, since it identifies terms up to such relation. However, the mentioned proof may actually be considered as proceeding by induction on our $\rightrightarrows_0$ relation of reduction and used as a basis for formalisation within our framework. We therefore adopt the following strategy, very similar to the one employed in the presentation in [16]:

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

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

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

$M \rightrightarrows M' \wedge M \rightrightarrows M''$
$\Rightarrow$ ($\sim_\alpha$ postponement)
$M \rightrightarrows_0 M'_0 \wedge M \rightrightarrows M''_0$ with $M'_0 \sim_\alpha M' \wedge M''_0 \sim_\alpha M''$
$\Rightarrow$ (confluence, i.e. property 3 above)
$(\exists P)(M'_0 \rightrightarrows P \wedge M''_0 \rightrightarrows P)$ with $M' \sim_\alpha M'_0 \wedge M'' \sim_\alpha M''_0$
$\Rightarrow$ (property 2 above)
$(\exists P)(M' \rightrightarrows P \wedge M'' \rightrightarrows P)$. $\square$

The lemmas of the enumeration above are as follows:

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

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

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

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

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

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

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

$M \sim_\alpha P \wedge P \rightrightarrows N$
$\Rightarrow$ ($\sim_\alpha$ postponement)
$M \sim_\alpha P \wedge P \rightrightarrows_0 P_0 \wedge P_0 \sim_\alpha N$
$\Rightarrow$ (Lemma 13)
$M \rightrightarrows P_0 \wedge P_0 \sim_\alpha N$
$\Rightarrow$ (Rule of $\sim_\alpha$ step)
$M \rightrightarrows N$. □

It only remains to establish the following confluence result:

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

The proof is essentially the standard induction on $M \rightrightarrows_0 M'$ with subordinate case analysis of $M \rightrightarrows_0 M''$, as developed in [3]. Working on $\rightrightarrows_0$, as established by our strategy above, alleviates significantly the number of combinations to be considered. Let us illustrate some of the detail involved by analyzing the case of the rule of parallel reduction of applications. We therefore put ourselves in the position that $MN \rightrightarrows_0 M'N'$, with both $M \rightrightarrows_0 M'$ and $N \rightrightarrows_0 N'$. If now in addition $MN \rightrightarrows_0 Q$, it turns out that there are two cases by way of which this may come about. The first one is that $Q = M''N''$ with both $M \rightrightarrows_0 M''$ and $N \rightrightarrows_0 N''$, i.e. by use of the rule of parallel reduction of applications. But then, just by the induction hypotheses, we get that there exist P_1 and P_2 such that, on the one hand side, $M' \rightrightarrows P_1$ and $M'' \rightrightarrows P_1$ and, on the other $N' \rightrightarrows P_2$ and $N'' \rightrightarrows P_2$. Therefore, $M'N' \rightrightarrows P_1P_2$ and $M''N'' \rightrightarrows P_1P_2$, as desired. The second case is when $MN \rightrightarrows_0 Q$ is arrived at by use of the β-parallel reduction rule. Then it must be $M = \lambda x M_0$ and $Q = M_0''(\iota, x{:=}N'')$ with $M_0 \rightrightarrows_0 M_0''$ and $N \rightrightarrows_0 N''$. Now, in the first place, we get by induction hypothesis that, from $N \rightrightarrows_0 N'$ and $N \rightrightarrows_0 N''$ it follows that there exists P_1 such that both $N' \rightrightarrows P_1$ and $N'' \rightrightarrows P_1$. On the other hand, by analysis of the rules of parallel reduction of abstractions, we must have $M' = \lambda x M_0'$ and, since $M_0 \rightrightarrows_0 M_0''$, also $\lambda x M_0 \rightrightarrows_0 \lambda x M_0''$. Hence, by induction hypothesis, there exists P such that both $\lambda x M_0' \rightrightarrows P$ and $\lambda x M_0'' \rightrightarrows P$. But, by the lemma of postponement of $\sim_\alpha$-conversion steps, we now that there must exist also $P' \sim_\alpha P$ such that both $\lambda x M_0' \rightrightarrows_0 P'$ and $\lambda x M_0'' \rightrightarrows_0 P'$. Further, by analysis of the rules of parallel reduction of abstractions, it must be $P' = \lambda x P_0$ with both $M_0' \rightrightarrows_0 P_0$ and $M_0'' \rightrightarrows_0 P_0$. Therefore what we had at the beginning was that $(\lambda x M_0)N \rightrightarrows_0 (\lambda x M_0')N'$ and $(\lambda x M_0)N \rightrightarrows_0 M_0''(\iota, x{:=}N'')$. But now $(\lambda x M_0')N' \rightrightarrows P_0(\iota, x{:=}P_1)$ by use of the β-parallel reduction rule and, on the other hand, $M_0''(\iota, x{:=}N'') \rightrightarrows P_0(\iota, x{:=}P_1)$ by the Substitution Lemma of $\rightrightarrows$. This gives us the desired confluence. □

Now we proceed to the following:

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

The proof is standard, by a double induction. □

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

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

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

from which we finally arrive at:

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

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

1. One-step β-reduction $\rightarrow$ is included in $\rightrightarrows$. The relation $\rightarrow$ is the contextual, i.e. compatible with the syntactic constructors, closure of the simple relation of β-contraction, augmented with $\sim_\alpha$. The proof proceeds by a corresponding simple induction. It follows that β-reduction $\twoheadrightarrow = \rightarrow^*$ is included in $\rightrightarrows^*$.
2. $\rightrightarrows$ is included in $\twoheadrightarrow$. It then follows that $\rightrightarrows^*$ is included in $\twoheadrightarrow^* = \twoheadrightarrow$. This proof is also by a direct induction on $\rightrightarrows$. Let us show the interesting case, corresponding to the β-parallel reduction rule:
We have $M \rightrightarrows M'$ and $N \rightrightarrows N'$ and need to prove $(\lambda xM)N \twoheadrightarrow M'(\iota, x{:=}N')$. Now, by induction hypotheses, we get both $M \twoheadrightarrow M'$ and $N \twoheadrightarrow N'$, and reason as follows:

$M \twoheadrightarrow M'$
$\Rightarrow$ (By compatibility of $\twoheadrightarrow$ with the syntactic constructors)
$\lambda xM \twoheadrightarrow \lambda xM'$
$\Rightarrow$ (Idem, using $N \twoheadrightarrow N'$)
$(\lambda xM)N \twoheadrightarrow (\lambda xM')N'$
$\Rightarrow$ (Transitivity of $\twoheadrightarrow$, using $(\lambda xM')N' \twoheadrightarrow M'(\iota, x{:=}N')$)
$(\lambda xM)N \twoheadrightarrow M'(\iota, x{:=}N')$. $\square$

5. Assignment of simple types

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

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

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

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

$$(\Gamma, x : \alpha)\, x =_{def} \alpha$$
$$(\Gamma, y : \alpha)\, x =_{def} \Gamma\, x \text{ if } y \neq x.$$

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

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

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

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

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

The system satisfies the following *weakening* lemmas:

Proposition 12 *(Weakening).*

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

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

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

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

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

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

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

Proposition 13.

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

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

We finally arrive at the crucial Substitution Lemma:

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

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

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

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

$\Delta \vdash (\lambda x M)\sigma : \alpha \to \beta$
$\Leftarrow$ (action of substitution, with $z = \chi(\sigma \lfloor \lambda x M)$)
$\Delta \vdash \lambda z M(\sigma, x{:=}z) : \alpha \to \beta$
$\Leftarrow$ (typing rule for abstractions)
$\Delta, z : \alpha \vdash M(\sigma, x{:=}z) : \beta$
$\Leftarrow$ (induction hypothesis)
$(\sigma, x{:=}z : \Gamma, x : \alpha \to \Delta, z : \alpha) \lfloor M$
$\Leftarrow$ (Proposition 13, part 5)
$z = \chi(\sigma \lfloor \lambda x M) \# \sigma \lfloor \lambda x M$ and $(\sigma : \Gamma \to \Delta) \lfloor \lambda x M$. □

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

It is now direct to get:

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

$\Gamma \vdash (\lambda x M)N : \beta$
$\Rightarrow$ (typing rules)
$\Gamma, x : \alpha \vdash M : \beta$ and $\Gamma \vdash N : \alpha$
$\Rightarrow$ (Proposition 13, part 2)
$\iota, x{:=}N : \Gamma, x : \alpha \to \Gamma$
$\Rightarrow$ (Substitution Lemma)
$\Gamma \vdash M(\iota, x{:=}N) : \beta$. □

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

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

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

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

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

$\Gamma \vdash (\lambda xM)\iota : \alpha \to \beta$
$\Rightarrow$ (action of substitution, with $z = \chi(\iota \lfloor \lambda xM)$)
$\Gamma \vdash \lambda zM(\iota, x{:=}z) : \alpha \to \beta$
$\Rightarrow$ (typing rules)
$\Gamma, z : \alpha \vdash M(\iota, x{:=}z) : \beta$
$\Rightarrow$ (Proposition 13 part 4 and Lemma 20)
$\Gamma, x : \alpha \vdash M(\iota, x{:=}z)(\iota, z{:=}x) : \beta$
$\Rightarrow$ ($z = \chi(\iota \lfloor \lambda xM)$ $\#\, \iota \lfloor \lambda xM$)
$\Gamma, x : \alpha \vdash M\iota : \beta$
$\Rightarrow$ (induction hypothesis)
$\Gamma, x : \alpha \vdash M : \beta$
$\Rightarrow$ (typing rule for abstractions)
$\Gamma \vdash \lambda xM : \alpha \to \beta$. □

Thence we arrive at:

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

using normalisation by ι as follows:

$\Gamma \vdash M : \alpha$
$\Rightarrow$ (Lemma 21)
$\Gamma \vdash M\iota : \alpha$
$\Rightarrow$ (Corollary 1, i.e. $M \sim_\alpha N \Rightarrow M\iota = N\iota$)
$\Gamma \vdash N\iota : \alpha$
$\Rightarrow$ (Lemma 21)
$\Gamma \vdash N : \alpha$. □

We now are able to conclude with

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

which follows by an easy induction using Corollary 8 and Lemma 22. □

6. Conclusions

To our mind, the present work contributes two things:

Firstly, it shows that what we have called the "historical" approach to the meta-theory of the Lambda calculus can be carried out in a completely formal manner so as to scale up to the principal results of the theory. By "historical approach" we mean the one initiated by Curry–Feys [8] and continued at least partly by Hindley and Seldin [16], which consists in treating the calculus in its original syntax — with one sort of names for both free and bound variables —, granting substitution a more basic status than that of α-conversion, and working all the time with concrete terms, i.e. without identifying terms up to α-conversion. The formal treatment is made feasible because of the use of Stoughton's substitution, which has shown therefore to be the appropriate one for this kind of syntax.

Within the general approach to syntax chosen, the main work to compare is the one by Vestergaard and Brotherston [32] which uses modified rules of α-conversion and β-reduction based on unary substitution to formally prove the Church–Rosser theorem in Isabelle–HOL. Substitution does not proceed in cases of capture and they use explicit α-conversion to perform the renaming achieved by our substitution. As a consequence, the Church–Rosser theorem requires an administrative layer of reasoning for showing that α-conversion and β-reduction interact correctly. This consists in a rather complex definition of a new auxiliary relation for α-conversion, which we do not need. On the other hand, they only use structural principles of induction, either on terms or on relations. As already indicated and commented again below, our use of size induction is not essential, but only convenient for simplifying the presentation of the theory of α-conversion with respect to

Stoughton's [27]. Besides, thanks to the use of the multiple form of substitution defined uniformly by structural recursion, we have been able to employ standard strategies for achieving the two principal results, namely confluence and preservation of typing by β-reduction.

Our second contribution consists in presenting Stoughton's theory of substitutions in a new way. First of all, it is based on inductive types and relations, instead of on ordinary set theory. Besides, it presents the following features:

(1) It bases itself upon the notion of *restriction* of a substitution to (the free variables of) a term.

As a consequence we have used the corresponding finite notions of equality and α-equivalence, whereas Stoughton and the formalisation by Lee [17] 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.

(2) Alpha-equivalence is given as a strictly syntax-directed inductive definition, which is easily proven to be an equivalence relation and therefore a congruence. This stands in contrast to Stoughton's work, which starts with a definition of α-conversion as the least congruence generated by a simple renaming of bound variable — 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 [17]. Two lemmas are crucial in the whole development, whatever strategy is taken: The first is the one stating that substitutions *equalize* α-equivalent terms, i.e. $M \sim_\alpha N \Rightarrow M\sigma = N\sigma$. This is very directly proven in our case by induction on $\sim_\alpha$ due to the symmetric character of the rule for abstractions, which is not the case for Stoughton's version of $\sim_\alpha$ and gives rise to the difficulties pointed out above. The second important lemma is the one stating that equality under the identity substitution implies α-equivalence, i.e. $M\iota = N\iota \Rightarrow M \sim_\alpha N$. This one is very easily proven by Stoughton using symmetry and transitivity of $\sim_\alpha$, since these properties are available from the beginning, whereas we need to proceed by induction on the length of M. The latter might be argued to depart from Stoughton's original goals to simplify the methods of reasoning generally employed. Now, as a matter of fact, it has been the only one point in which a principle of induction other than just structural has been used in our proofs and, to our mind, the overall cost of the development pays off such expenditure. Specifically, our proof that $\sim_\alpha$ is a congruence is finally quite concise and down to the point, not needing in particular the Substitution Lemma. The induction on the size of terms could be straightforwardly encoded in Agda using library functions.

As further work, concerning the formalisation of the metatheory of the Lambda calculus, we could complete the presentation with proofs of the Standardisation Theorem and of Strong Normalisation of the system with simple types. We also believe it interesting to investigate the generalisation of the approach to systems of languages with binders as e.g. the one presented in [23].

References

[1] Brian Aydemir, Aaron Bohannon, Stephanie Weirich, Nominal reasoning techniques in Coq, Electron. Notes Theor. Comput. Sci. 174 (5) (June 2007) 69–77.

[2] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, Stephanie Weirich, Engineering formal metatheory, ACM SIGPLAN Notices 43 (1) (January 2008) 3–15.

[3] H.P. Barendregt, The Lambda Calculus, Its Syntax and Semantics, revised edition, Studies in Logic and the Foundations of Mathematics, vol. 103, North Holland, 1984.

[4] Arthur Charguéraud, The locally nameless representation, J. Automat. Reason. 49 (3) (2012) 363–408.

[5] A. Church, A set of postulates for the foundation of logic Part I, Ann. of Math. (2) 33 (2) (1932) 346–366.

[6] Ernesto Copello, Alvaro Tasistro, Nora Szasz, Ana Bove, Maribel Fernández, Alpha-structural induction and recursion for the lambda calculus in constructive type theory, in: 10th Workshop on Logical and Semantic Frameworks with Applications, LSFA 2015, 2015, pp. 51–66.

[7] T. Coquand, An algorithm for testing conversion in type theory, in: G. Huet, G. Plotkin (Eds.), Logical Frameworks, Cambridge University Press, Cambridge, 1991, pp. 255–279.

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

[9] N.G. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with applications to the Church–Rosser theorem, Indag. Math. (Koninglijke Ned. Akad. Wet.) 34 (5) (1972) 381–392.

[10] Heinz-Dieter Ebbinghaus, Jörg Flum, Wolfgang Thomas, Mathematical Logic, 2nd ed., Undergraduate Texts in Mathematics, Springer, 1994.

[11] G. Frege, Begriffsschrift eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens, Halle, 1879;
English translation in: J. van Heijenoort (Ed.), From Frege to Gödel: a Source Book in Mathematical Logic, Harvard University, Cambridge, 1967, pp. 1–82.

[12] Murdoch J. Gabbay, Foundations of nominal techniques: logic and semantics of variables in abstract syntax, Bull. Symbolic Logic 17 (2) (2011) 161–229.

[13] Murdoch J. Gabbay, Andrew M. Pitts, A new approach to abstract syntax with variable binding, Form. Asp. Comput. 13 (3–5) (2002) 341–363.

[14] Gerhard Gentzen, The collected papers of Gerhard Gentzen, in: M.E. Szabo (Ed.), Studies in Logic and the Foundations of Mathematics, North-Holland, 1969.

[15] Andrew D. Gordon, Thomas F. Melham, Five axioms of alpha-conversion, in: Theorem Proving in Higher Order Logics, 9th International Conference, Proceedings, TPHOLs'96, Turku, Finland, August 26–30, 1996, 1996, pp. 173–190.

[16] J. Roger Hindley, Jonathan P. Seldin, Introduction to Combinators and Lambda-Calculus, Cambridge University Press, 1986.

[17] Gyesik Lee, Proof pearl: substitution revisited, again, Hankyong National University, Korea. http://formal.hknu.ac.kr/Publi/Stoughton.pdf.

[18] P. Martin-Löf, G. Sambin, Intuitionistic Type Theory, Studies in Proof Theory, Bibliopolis, 1984.

[19] J. McKinna, R. Pollack, Some lambda calculus and type theory formalized, J. Automat. Reason. 23 (3–4) (November 1999).
[20] Ulf Norell, Towards a practical programming language based on dependent type theory, PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.
[21] A.M. Pitts, Nominal techniques, ACM SIGLOG News 3 (1) (January 2016) 57–72.
[22] Andrew M. Pitts, Nominal logic, a first order theory of names and binding, Inform. and Comput. 186 (2) (2003) 165–193.
[23] Andrew M. Pitts, Alpha-structural recursion and induction, J. ACM 53 (3) (May 2006) 459–506.
[24] Randy Pollack, Closure under alpha-conversion, in: Proceedings of the International Workshop on Types for Proofs and Programs, TYPES '93, Secaucus, NJ, USA, Springer-Verlag New York, Inc., 1994, pp. 313–332.
[25] Dag Prawitz, Natural Deduction: A Proof-Theoretical Study, Stockholm Studies in Philosophy, vol. 3, Almquist and Wiskell, 1965.
[26] György E. Révész, Lambda-Calculus, Combinators and Functional Programming, Cambridge Tracts in Theoretical Computer Science, vol. 4, Cambridge University Press, 1988.
[27] A. Stoughton, Substitution revisited, Theoret. Comput. Sci. 59 (1988) 317–325.
[28] M. Takahashi, Parallel reductions in λ-calculus, Inform. and Comput. 118 (1) (1995) 120–127.
[29] Alvaro Tasistro, Ernesto Copello, Nora Szasz, Formalisation in constructive type theory of Stoughton's substitution for the lambda calculus, Electron. Notes Theor. Comput. Sci. 312 (2015) 215–230.
[30] Christian Urban, Nominal techniques in Isabelle/HOL, J. Automat. Reason. 40 (4) (2008) 327–356, 5.
[31] Christian Urban, Michael Norrish, A formal treatment of the Barendregt Variable Convention in rule inductions, in: ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized Reasoning About Languages with Variable Binding, MERLIN, 2005, pp. 25–32, 2005.
[32] René Vestergaard, James Brotherston, A formalised first-order confluence proof for the λ-calculus using one-sorted variable names, Inform. and Comput. 183 (2) (2003) 212–244.