Substitution in the λ-Calculus and the role of the Curry School

Fairouz Kamareddine

Abstract

Substitution plays a prominent role in the foundation and implementation of mathematics and computation. In the λ-calculus, we cannot define α-congruence without a form of substitution but for substitution and reduction to work, we need to assume a form of α-congruence (e.g., when we take λ-terms modulo bound variables). Students on a λ-calculus course usually find this confusing. The elegant writings and research of the Curry school [7, 14, 11] have settled this problem very well. This article is an ode to the contributions of the Curry school (especially the excellent book of Hindley and Seldin [12, 13]) on the subject of α-congruence and substitution.

1 Introduction

Jonathan Seldin completed his PhD thesis (*Studies in Illative Combinatory Logic* [20]) in 1968 under the supervision of Haskell Curry at the University of Amsterdam, and to this day, he has safeguarded all the manuscripts of Curry as well as his legacy. This article is dedicated to Jonathan for the great role he has played throughout the evolution of combinatory logic and the λ-calculus.

The first half of the twentieth century had seen an explosion of new novel ideas that would shape the foundations of mathematics and would

This paper appears in *A Century since Principia's Substitution Bedazzled Haskell Curry*, Fairouz Kamareddine (ed). College Publications (11 July 2023). ISSN-10 1848904363, ISBN 13 978-1848904361. Here is the list of contents: `http://collegepublications.co.uk/contents/tbt00051.pdf`

lead to the birth of computability. During those impressive times, new formal languages, logics, and tools were invented that up to this day still represent the undisputed standards for formalisation and computation. These standards include the Turing machine, the λ-calculus, category theory, combinatory logic and Gödel's incompleteness. Although invented by different people, they are all interconnected and each sheds light on the *Entscheidungsproblem* in its own way. Alonzo Church gave us the λ-calculus which is the language of the computable (f is computable iff f is λ-definable), Haskell Curry gave us Combinatory Logic which is another computation model that is equivalent to the λ-claculus. Curry also gave us a number of concepts that we continue to heavily use today (e.g., the Curry-Howard isomorphism which gives a correspondence between proofs in proof systems and terms in models of computation, and the Currying notion where a function of many arguments can be evaluated as a sequence of unary-functions, as well as the *functionality* concept, which became the basis of what nowadays is called type assignment). Turing gave us another equivalent model of computation: the Turing machine which is the machine of the computable ($f : \mathbb{N} \mapsto \mathbb{N}$ is computable iff there is a Turing machine M_f that takes an input n and halts with output $f(n)$).[1] In this paper, we only focus on the λ-calculus and on some of the lessons from the Curry school on subtitution.

Substitution plays a prominent role in the foundation and implementation of mathematics and computation. Bertrand Russell and others made a number of attempts at defining substitution but most, if not all, of the attempts to correctly define substitution with bound variables before the publication in 1934 of Hilbert and Bernays foundations of mathematics [10] were erroneous. Although Russell's type free substitutional theory of 1905 [19] enabled him to prove the axiom of infinity, it also allowed contradictions and he was frustrated by the paradoxes which as he explained in a letter to his friend the mathematician Ralph Hawtrey "pilled the substitution theory" (see [21]). The notion of substitution remained unsatisfactorily explained in his Principia Mathematica.

[1]Samuel Eilenberg and Saunders Mac Lane gave us category theory which is an elegant foundation of mathematics. And there was the new developments of set theory and type theory which appropriately claim their crucial role in the foundations of mathematics and computation.

This problem of substitution was the motivation of Curry[2] to develop combinatory logic. In 1922, as a graduate student, Curry noticed in the first chapter of Principia [22] that the rule of substitution of well-formed formulas for propositional variables (which does not even involve the well known complication of bound variables) was considerably more tricky than the rule of detachment (which is equivalent to modus ponens). The complication Curry noticed in the rule of substitution in Chapter 1 of Principia is now considered to be the complication of its implementation by a computer program although there were no electronic computers when Curry noticed this.[3]

This article discusses substitution in the λ-calculus and the role of the Curry school. One of the first hurdles after introducing students to the extremely simple syntax of the type-free pure λ-calculus (variables, abstraction and application) is how/when to introduce the substitution rule and α-congruence. In particular, should one use α-congruence (yes they should because α-congruence is needed for the Church-Rosser property of β-reduction[4]), and then, in which order should one introduce α-conversion and substitution? We cannot define α-congruence without a form of substitution but for substitution and reduction to work, we need to assume a form of α-congruence (e.g., when we take λ-terms modulo bound variables). Students on a λ-calculus course usually find this confusing. Moreover, to them, this does not fit with a purely syntactical account where much of the work should be carried out using rules that can be automated. This was behind most research on expliciting the notion of substitution in the λ-calculus in order to bridge theory and implementation.

The more the students appreciate the enormous intellectual debates triggered by the paradoxes and the labour that led to the birth of com-

[2]For Curry's PhD thesis (which was originally written in German) translated into English, see [15].

[3]It is precisely when Curry started looking for simpler forms of substitution [3, 4], that he introduced some of the combinators and defined in [5], what is nowadays called the *bracket abstract* $[x_1, x_2, \cdots, x_n]\mathfrak{X}$. Curry was able to prove that for any term $\mathfrak{X}$ where free variables $x_1, x_2, \cdots, x_n$ may appear, there is a term X in which none of these variables appear with the property that $Xx_1x_2\cdots x_n = \mathfrak{X}$. This term X is unique by extensionality and is denoted by $[x_1, x_2, \cdots, x_n]\mathfrak{X}$.

[4]See Examples A1.1 and A1.2 of [13].

puters and the foundations and limits of computation,[5] the more they question representations that might confuse them. Students prefer rules that they can apply or even implement to being told to work things in their head. They get confusd if they are told at the start that λ-terms will be taken modulo the name of their bound variables and then shown how to build α-reduction using substitution and to give substitution modulo α-equivalence classes. They ask which is first (α or substitution). For them this is bothersome because they feel that the dependence between α-reduction and substitution is self-referential.

The elegant writings and research of the Curry school [7, 14, 11] have settled this problem very well. This article is an ode to the contributions of the Curry school (especially the excellent book of Hindley and Seldin [12, 13]) on the subject of α-reduction and substitution.

In Section 2 we introduce the basic syntax (with the purely syntactic identity $=_{\mathcal{M}}$), notational conventions and denotational semantics of the λ-calculus. In Section 3 we move to computation and calculation with λ-terms and explain the problem of the grafting $A\{v := B\}$.[6] Then we introduce bound and free occurrences of variables as well as the replacement $A\langle\langle v := B\rangle\rangle$ based on an ordered variable list.[7] We note that although calculating (which we called $\overline{\beta}$-reducing) λ-terms using the substitution based on an ordered variable list avoids the problem of variable capture, it does not work on its own because the order in which a term is computed will affect the answer. In Section 4 we discuss two alternatives to identify terms modulo bound variables. One approach ($=_\alpha$) builds α-congruence directly from the replacement based on ordered variables while the other is based on safe applications of grafting to build a notion $=_{\alpha'}$ of term equivalence. Interestingly, both approaches give the same notion of syntactic equivalence denoted by $\equiv$ where terms that only differ in the name of their bound variables are

[5]Ranging from Frege's abstraction principle, his general concept of a function and his formalisations [8, 9], to Russell's paradox in Frege's work and his monumental Principia Mathematica [22], to Hilbert's Entscheidungsproblem and the languages/models of Curry, Turing, Church and others.

[6]Which is the purely syntactic replacement/substitution of a variable v inside a term A by a term B.

[7]Which replaces/substitutes a variable v inside a term A by a term B using an ordered variable list to avoid the capture of free variables.

equivalent. In Section 5 we introduce the β' and β'' reduction relations. The basic (β') rule is the same as $(\overline{\beta})$ but the reflexive transitive closure $\twoheadrightarrow_{\beta'}$ adds α-reduction in the sense that if $A \twoheadrightarrow_{\beta'} B$ then there is C such that $A \twoheadrightarrow_{\overline{\beta}} C \twoheadrightarrow_{\alpha} B$. The basic (β'') adds α'-reduction to transform the redex into a so-called clean term before any beta reduction takes place and then safely uses grafting since this does not cause any problems with clean terms. Not only the basic (β'') rule is based on α'-reduction, but also the reflexive transitive closure $\twoheadrightarrow_{\beta''}$. So, for grafting to work, we use α'-reduction at the basic reduction stage (to clean the term) and at the reflexive transitive stage. Unlike $\overline{\beta}$-reduction, both β' and β'' reductions now satisfy CR. However, there is an oddity about β'' compared to β'. (β') is a function in the sense that, if $(\lambda v.A)B \rightarrow_{\beta'} C$ then C is unique because the replacement relation for β' is based on an ordered variable list.[8] On the other hand, (β'') is not a function since (β'') uses α'-reduction to clean terms. For example, we can clean $(\lambda x.x(\lambda x.x))x$ into either $(\lambda z.z(\lambda y.y))x$ or $(\lambda y.y(\lambda z.z))x$ and so we get $(\lambda x.x(\lambda x.x))x \rightarrow_{\beta''} x(\lambda y.y)$ and $(\lambda x.x(\lambda x.x))x \rightarrow_{\beta''} x(\lambda z.z)$ yet $x(\lambda y.y) \neq_{\mathcal{M}} x(\lambda z.z)$. But since both β' and β'' reductions have used α/α' reduction and since we have built and understood the priorities and order of substitution and equivalence classes modulo the names of bound variables (α/α'), we can now move to the stage at which courses on the λ-calculus usually start and take terms modulo the names of bound variables. So in the rest of Section 5, we introduce the usual notion of substitution and β-reduction and give a summary and comparison of all the β-reductions introduced so far. For the sake of completeness, in Section 6 we present the λ-calculus with de Bruijn indices where the de Bruijn indices incorporate the equivalence classes of terms. We look at the substitution and reduction rules with de Bruijn indices which will lead us naturally to a calculus with explicit substitutions where we take the classical λ-calculus with de Bruijn indices exactly as it is, but simply turn its meta-updating and meta-substitution to the object level to obtain a calculus of explicit substitutions. Extending λ-calculi with explicit substitutions is essential for the implementations of these calculi.

[8] Note that although (β') is a function, $\rightarrow_{\beta'}$ is not unless we take it as a function of 2 arguments, the term and the particular redex-occurrence. For example, $(\lambda x.\underline{(\lambda y.y)x})u \rightarrow_{\beta'} (\lambda x.x)u$ and $\underline{(\lambda x.(\lambda y.y)x)u} \rightarrow_{\beta'} (\lambda y.y)u$ but $(\lambda x.x)u \neq_{\mathcal{M}} (\lambda y.y)u$.

We conclude in Section 7.

2 The Syntax and Denotational Semantics of the λ-calculus

Convention 1. *If Symb ranges over a set of entities $\mathcal{A}$ then also Symb$'$, Symb$''$, Symb$_1$, Symb$_2$, etc., range over $\mathcal{A}$.*

Definition 2 (λ-terms in $\mathcal{M}$)**.**

- *Let $\mathcal{V} = \{x, y, z, x', y', z', x_1, y_1, z_1, \dots\}$ be an infinite set of* term variables and let *meta variables u, v range over $\mathcal{V}$. By Convention 1, also u', v_1, etc., range over $\mathcal{V}$.*
  Note that $\mathcal{V}$ is a set and hence all its elements are pairwise distinct.
- *The set of classical λ-terms (or λ-expressions) $\mathcal{M}$ is given by:*

  $$\mathcal{M} ::= \mathcal{V} \mid (\lambda\mathcal{V}.\mathcal{M}) \mid (\mathcal{M}\mathcal{M}).$$

  We let capital letters $A, B, C, D, \cdots$ range over $\mathcal{M}$.
- *The set of contexts with one hole $\mathcal{C}$ is given by:*

  $$\mathcal{C} ::= [\,] \mid (\lambda\mathcal{V}.\mathcal{C}) \mid (\mathcal{C}\mathcal{M}) \mid (\mathcal{M}\mathcal{C}).$$

  We call $[\,]$ a hole and let $C[\,], C'[\,], C_1[\,], \cdots$ range over $\mathcal{C}$.
  We write $C[A]$ to denote the filling of the hole in $C[\,]$ with term A. For example, if $C[\,]$ is $(\lambda x.([\,]x))$ then $C[x]$ is $(\lambda x.(xx))$, $C[y]$ is $(\lambda x.(yx))$, and $C[(\lambda y.y)]$ is $(\lambda x.((\lambda y.y)x))$.
- *The* length *of a term A (written $\#A$) is defined inductively as follows:*
  $\#v = 1 \qquad \#(AB) = \#A + \#B \qquad \#(\lambda v.A) = 1 + \#A.$

Definition 3 (Compatibility)**.** *We say that a relation*[9] *R on λ-terms is* compatible *if the following hold for any λ-terms A, B and C and for any variable v:*

$$\frac{ARB}{(AC)R(BC)} \qquad \frac{ARB}{(CA)R(CB)} \qquad \frac{ARB}{(\lambda v.A)R(\lambda v.B)}$$

[9]Note that if R is a compatible relation and ARB, then $C[A]RC[B]$ for any context $C[\,]$.

Here, $\frac{\text{above}}{\text{below}}$ means that if "above" holds then "below" holds too.

Definition 4 (Strict Equality $=_{\mathcal{M}}$)**.** *Strict equaliy $=_{\mathcal{M}}$ on the set $\mathcal{M}$ of λ-terms is defined as the reflexive, transitive, symmetric and compatible closure under*

$$\frac{v \in \mathcal{V}}{v = v} \quad (\textit{Base } =_{\mathcal{M}})$$

Note that Strict Equality $=_{\mathcal{M}}$ is $\{(A, A) | A \in \mathcal{M}\}$.

To explain the meaning of terms, let us imagine a model $\mathcal{D}$ where every λ-term denotes an element of that model. We let $\mathbf{d}$ range over $\mathcal{D}$ (and hence by Convention 1, $\mathbf{d}'$, $\mathbf{d}''$, $\mathbf{d}_1$, $\mathbf{d}_2$ etc., also range over $\mathcal{D}$).

The meaning of terms depends very much on the values we assign to variables. This is why we need the so-called *environment*.

Definition 5 (`ENV`, Changing Environments)**.** *We define the set of* environments *`ENV` as the set of total functions from $\mathcal{V}$ to $\mathcal{D}$. We let σ range over `ENV`.*

$$\mathit{ENV} = \mathcal{V} \xrightarrow{tot} \mathcal{D}.$$

We define the new environment $\sigma[\boldsymbol{d}/v] \in \mathcal{V} \xrightarrow{tot} \mathcal{D}$ as follows:

$$\sigma[\boldsymbol{d}/v](v') = \begin{cases} \boldsymbol{d} & \textit{if } v' = v \\ \sigma(v') & \textit{otherwise} \end{cases}$$

Note that $\sigma[\mathbf{d}/v][\mathbf{d}'/v](v) = \mathbf{d}'$, $\sigma[\mathbf{d}/v][\mathbf{d}'/v] = \sigma[\mathbf{d}'/v]$ and if $v \neq v'$ then $\sigma[\mathbf{d}/v][\mathbf{d}'/v'] = \sigma[\mathbf{d}'/v'][\mathbf{d}/v]$.

Definition 6 (Denotational meaning of terms)**.** *We define the following:*[10]

- *The function $[\![\ ]\!] \in \mathcal{M} \xrightarrow{tot} (\mathit{ENV} \xrightarrow{tot} \mathcal{D})$ as follows:*

[10]Note that we defined our environments to be total functions on $\mathcal{V}$ (`ENV` $= \mathcal{V} \xrightarrow{\text{tot}} \mathcal{D}$) and we defined our meaning function $[\![\ ]\!] \in \mathcal{M} \xrightarrow{\text{tot}}$ (`ENV` $\xrightarrow{\text{tot}} \mathcal{D}$) to be a total function on $\mathcal{M}$ and hence, for any $\sigma \in$ `ENV` and $A \in \mathcal{M}$, $[\![A]\!]_\sigma$ is defined and is an element of $\mathcal{D}$. We also see that when A is an abstraction $(\lambda v.B)$, then $[\![(\lambda v.B)]\!]_\sigma$ is a partial function in $\mathcal{D} \xrightarrow{\circ} \mathcal{D}$.

- $[\![v]\!]_\sigma = \sigma(v)$.
- $[\![(AB)]\!]_\sigma = [\![A]\!]_\sigma([\![B]\!]_\sigma)$.
- $[\![(\lambda v.A)]\!]_\sigma = \boldsymbol{f} \in \mathcal{D} \overset{\circ}{\longrightarrow} \mathcal{D}$ *such that* $\forall \boldsymbol{d} \in \mathcal{D}$, $\boldsymbol{f}(\boldsymbol{d}) = [\![A]\!]_{\sigma[\boldsymbol{d}/v]}$.

- A *and* B have the same meaning in environment σ *iff* $[\![A]\!]_\sigma = [\![B]\!]_\sigma$.
- $[\![A]\!] = [\![B]\!]$ *iff* $[\![A]\!]_\sigma = [\![B]\!]_\sigma$ *for every environment* σ.

We will assume the usual parenthesis convention where application associates to the left, outer parenthesis can be dropped, the body of a λ includes everything that comes after it and a sequence of λ's is compressed to one. So, $ABCD$ denotes $(((AB)C)D)$ and $\lambda vv'.AB$ denotes $\lambda v.(\lambda v'.(AB))$.

3 Computing λ-terms

The abstraction term $(\lambda v.A)$ alone is a bachelor waiting for its partner. As soon as a B stands to the right of the abstraction $(\lambda v.A)$ we get a partnered couple $(\lambda v.A)B$. When the time comes and we are ready for computation, $(\lambda v.A)B$ can produce a child $A_{v:=B}$ which is the body A of the abstraction in which all the occurrences of v are replaced by B as follows:

$$(\lambda v.A)B \text{ computes to } A_{v:=B} \tag{1}$$

This replacement $A_{v:=B}$ still needs to be defined.

3.1 Defining $A_{v:=B}$ as Grafting $A\{v := B\}$ does not work

One might immediately think of defining $A_{v:=B}$ by a strict syntactic replacement (a.k.a. grafting) as given in Definition 7 below, however, this is the wrong definition as we will see in Table 1.

Definition 7 (Grafting $A\{v := B\}$)**.** *For any A, B, v, we define the* grafting *relation $A\{v := B\}$ to be the result of syntactically replacing B*

for every occurrence of v in A, as follows:[11]

$$\begin{array}{llll}
1.\ v\{v := B\} & =_{def} & B & \\
2.\ v'\{v := B\} & =_{def} & v' & \textit{if } v \neq_{\mathcal{M}} v' \\
3.\ (AC)\{v := B\} & =_{def} & A\{v := B\}C\{v := B\} & \\
4.\ (\lambda v.A)\{v := B\} & =_{def} & \lambda v.A & \\
5.\ (\lambda v'.A)\{v := B\} & =_{def} & \lambda v'.A\{v := B\} & \textit{if } v \neq_{\mathcal{M}} v'
\end{array}$$

Let σ be an environment.
Let $\mathbf{f}, \mathbf{g}, \mathbf{g}'$ and $\mathbf{h}$ be such that $\forall \mathbf{d}, \mathbf{d}' \in \mathcal{D}$: $\mathbf{f}(\mathbf{d})(\mathbf{d}') = \mathbf{d}(\mathbf{d}')$, $\mathbf{g}(\mathbf{d}) = [\![y]\!]_\sigma(\mathbf{d})$, $\mathbf{g}'(\mathbf{d}) = [\![x]\!]_\sigma(\mathbf{d})$ and $\mathbf{h}(\mathbf{d}) = \mathbf{d}(\mathbf{d})$.
Clearly $\mathbf{g}$, $\mathbf{g}'$ and $\mathbf{h}$ are different and $[\![\lambda z.xz]\!]_\sigma = [\![\lambda y.xy]\!]_\sigma = \mathbf{g}'$.
If we allow Definition 6 to cover the $=_{def}$ of Grafting Definition 7 (i.e., if $A\{v := B\} =_{def} C$ then $[\![A\{v := B\}]\!]_\sigma = [\![C]\!]_\sigma$), then:

- $[\![(\lambda z.xz)\{x := y\}]\!]_\sigma = [\![\lambda z.yz]\!]_\sigma = \mathbf{g}$
- $[\![(\lambda y.xy)\{x := y\}]\!]_\sigma = [\![\lambda y.yy]\!]_\sigma = \mathbf{h}$.
- So, $[\![\lambda z.xz]\!]_\sigma = [\![\lambda y.xy]\!]_\sigma$ but $[\![(\lambda z.xz)\{x := y\}]\!]_\sigma \neq [\![(\lambda y.xy)\{x := y\}]\!]_\sigma$.
  This is not good.
- If $(\lambda v.A)_{v:=B}$ is the grafting $A\{v := B\}$ of Definition 7, then by (1):
  1. $(\lambda xz.xz)y$ computes to $(\lambda z.xz)\{x := y\} =_{def} \lambda z.yz$.
     And, $[\![(\lambda xz.xz)y]\!]_\sigma = \mathbf{g} = [\![\lambda z.yz]\!]_\sigma$.
  2. $(\lambda xy.xy)y$ computes to $(\lambda y.xy)\{x := y\} =_{def} \lambda y.yy$.
     But, $[\![(\lambda xy.xy)y]\!]_\sigma = \mathbf{g} \neq [\![\lambda y.yy]\!]_\sigma = \mathbf{h}$.

This is bad and so we cannot use the computation rule (β^w):

$$(\lambda v.A)B \to_{\beta^w} A\{v := B\}$$

because $(\lambda xy.xy)y$ and $(\lambda xz.xz)y$ have the same meaning $\mathbf{g}$, $(\lambda xy.xy)y \to_{\beta^w} \lambda y.yy$ and $(\lambda xz.xz)y \to_{\beta^w} \lambda z.yz$ but $\lambda z.yz$ and $\lambda y.yy$ have different meanings ($\mathbf{g}$ resp. $\mathbf{h}$).

Table 1: Grafting does not work

[11] We use $=_{def}$ here instead of $=_{\mathcal{M}}$ to draw attention that this definition will not work as we will see in the rest of this paper.

Table 1 gives examples where equation (1) is used with the grafting of Definition 7 resulting in comptation rule (β^w). As we see, computing $(\lambda xz.xz)\textcircled{y}$ to $(\lambda z.xz)\{x := \textcircled{y}\} =_{def} \lambda z.\textcircled{y}z$ is correct whereas computing $(\lambda xy.xy)\textcircled{y}$ to $(\lambda y.xy)\{x := \textcircled{y}\} =_{def} \lambda y.\textcircled{y}y$ is wrong. Since λ is a binder (like $\forall$), the $\textcircled{y}$ which is free in the original term $(\lambda xy.xy)\textcircled{y}$ is now bound in its computation $\lambda y.\textcircled{y}y$. So $(\lambda y.xy)\{x := \textcircled{y}\}$ gives the wrong answer and we need to find a different definition of replacement which respects the free status of $\textcircled{y}$.

First, we define the notions of *free* and *bound* occurrences of variables.

3.2 Free and bound occurrences of variables

Definition 8 (Term Occurrence)**.** *We define an* occurrence relation *between λ-terms as follows:*

- *A occurs in A.*
- *If A occurs in either B or C then A occurs in BC.*
- *If A occurs in B or $A =_{\mathcal{M}} v$ then A occurs in $\lambda v.B$.*

We can number the occurrences as in $\overline{xy}^{\circ_1}(\lambda x.y(\lambda y.z)(\overline{xy}^{\circ_2}))$. When the term A is a variable, we leave out the overline as in $x^{\circ_1}(\lambda x^{\circ_2}.y(\lambda y.z)(x^{\circ_3}z))$.

Definition 9 (Scope, Free/Bound Occurrences, Combinator)**.**

1. *For a particular occurrence of a $\lambda v.A$ in a term C, we call the occurrence of A* the scope *of the occurrence of the λv.*
2. *We call an occurrence of a variable v in a term C,*
   (a) bound *if it is in the scope of a λv in C.*[12]
   (b) bound and binding*, if it is the v of a λv in C.*
   (c) free *if it is not bound.*
3. *We say that v is bound (resp. free) in C if v has at least one binding (resp. free) occurrence in C. We write $BV(C)$ (resp. $FV(C)$) for the set of bound (resp. free) variables of C.*

[12]That is, if the occurrence of v is inside the term A of a $\lambda v.A$ which occurs in C.

4. *A* closed expression *is an expression in which all occurrences of variables are bound. A closed expression is also called a* combinator.

3.3 Defining $A_{v:=B}$ as Replacement $A\langle\langle v := B\rangle\rangle$ Using Ordered Variables

In order to avoid the problem of grafting as we saw in Table 1, replacement needs to be handled with care due to the distinct roles played by bound and free occurrences of variables. Since $(\lambda y.xy)_{x:=\textcircled{y}}$ must not return $\lambda y.\textcircled{y}y$ and we should not change the free $\textcircled{y}$, then we should change the name of the bound variable y in $\lambda y.xy$ to $v \notin \{x, y\}$ obtaining $\lambda v.xv$. Then, replacing the x of $\lambda v.xv$ by $\textcircled{y}$ gives $\lambda v.\textcircled{y}v$ which is fine because $v \notin \{x, y\}$. Clearly we need to change the following rule 5. of Definition 7:

$$5.\ (\lambda v'.A)\{v := B\} =_{def} \lambda v'.A\{v := B\} \qquad \text{if } v \neq_{\mathcal{M}} v'.$$

We could split 5. into 2 rules as follows:

$$\begin{array}{lll}
5.\ (\lambda v'.A)\{v := B\} & =_{def} & \lambda v'.A\{v := B\} \qquad \text{if } v \neq_{\mathcal{M}} v' \\
 & & \text{and } (v' \notin FV(B) \text{ or } v \notin FV(A)) \\
6.\ (\lambda v'.A)\{v := B\} & =??? & \lambda v''.A\{v' := v''\}\{v := B\} \qquad \text{if } v \neq_{\mathcal{M}} v' \\
 & & \text{and } (v' \in FV(B) \text{ and } v \in FV(A)) \\
 & & \text{and } v'' \notin FV(AB)
\end{array}$$

We still need to precise the v'' of rule 6. Clearly if we stick to the strict equality of Definition 4, then v'' must be unique since if $\lambda v''_1.yv''_1 =_{\mathcal{M}} \lambda v''_2.yv''_2$ then $v''_1 =_{\mathcal{M}} v''_2$ follows from the following consequence of Definition 4:

$$(\lambda v.A) =_{\mathcal{M}} (\lambda v'.A') \text{ iff } (v =_{\mathcal{M}} v' \text{ and } A =_{\mathcal{M}} A') \qquad (2)$$

One way to get a unique result in rule 6. would be to order the list of variables $\mathcal{V}$ and then to take v'' to be the first variable in the ordered list $\mathcal{V}$ which is different from v and v' and which occurs after all the free variables of AB. To do this, we add to clause 6 the condition that v'' is the first variable in the ordered list $\mathcal{V}$ which satisfies the conditions of clause 6. That is, we replace Definition 7 by:

Definition 10 (Replacement $A\langle\langle v := B\rangle\rangle$ using ordered variables)**.** *We define $A\langle\langle v := B\rangle\rangle$ to be the result of replacing B for every free occurrence of v in A:*

1. $v\langle\langle v := B\rangle\rangle =_{\mathcal{M}} B$
2. $v'\langle\langle v := B\rangle\rangle =_{\mathcal{M}} v'$ *if* $v \neq_{\mathcal{M}} v'$
3. $(AC)\langle\langle v := B\rangle\rangle =_{\mathcal{M}} A\langle\langle v := B\rangle\rangle C\langle\langle v := B\rangle\rangle$
4. $(\lambda v.A)\langle\langle v := B\rangle\rangle =_{\mathcal{M}} \lambda v.A$
5. $(\lambda v'.A)\langle\langle v := B\rangle\rangle =_{\mathcal{M}} \lambda v'.A\langle\langle v := B\rangle\rangle$ *if* $v \neq_{\mathcal{M}} v'$ *and* $(v' \notin FV(B)$ *or* $v \notin FV(A))$
6. $(\lambda v'.A)\langle\langle v := B\rangle\rangle =_{\mathcal{M}} \lambda v''.A\langle\langle v' := v''\rangle\rangle\langle\langle v := B\rangle\rangle$ *if* $v \neq_{\mathcal{M}} v'$ *and* $(v' \in FV(B)$ *and* $v \in FV(A))$ *and* v'' *is the first variable in the ordered variable list* $\mathcal{V}$ *such that* $v'' \notin FV(AB)$.

For example, if the ascending order in $\mathcal{V}$ is

$$x, y, z, x', y', z', x'', y'', z'', \ldots$$

then z the first variable in this list which is different from x and y and which is not free in either y or xy. So, $(\lambda y.xy)\langle\langle x := y\rangle\rangle$ can only be $(\lambda z.yz)$.

The next lemma plays some initial steps of comparing grafting with replacement using ordered variables. Note especially item 3 which shows that under some strict conditions, grafting is replacement.

Lemma 1.

1. *If $v \notin FV(A)$ then for any B, $A\langle\langle v := B\rangle\rangle =_{\mathcal{M}} A$.*[13]
2. *If $v \in FV(A)$ then $FV(A\langle\langle v := B\rangle\rangle) = (FV(A) \setminus \{v\}) \cup FV(B)$.*[14]
3. *If $v' \notin FV(vA)$, $v, v' \notin BV(A)$ then $A\{v := v'\} =_{\mathcal{M}} A\langle\langle v := v'\rangle\rangle$.*

[13]Also, if $v \notin FV(A)$ then $A\{v := B\} =_{\mathcal{M}} A$.

[14]This does not hold for $A\{v := B\}$. E.g., $(FV(\lambda x.y) \setminus \{y\}) \cup \{x\} = \{x\} \neq FV((\lambda x.y)\{y := x\}) = FV(\lambda x.x) = \emptyset$. However, $FV(A\{v := B\}) \subseteq (FV(A) \setminus \{v\}) \cup FV(B)$.

4. *If $v \neq v'$, $v \notin FV(C)$ and whenever $\lambda v''.D$ occurs in A then $v'' \notin FV(BC)$ (i.e., no bound variable of A occurs free in BC), then $A\langle\langle v := B\rangle\rangle\langle\langle v' := C\rangle\rangle =_{\mathcal{M}} A\langle\langle v' := C\rangle\rangle\langle\langle v := B\langle\langle v' := C\rangle\rangle\rangle\rangle$.*

Proof. 1. and 2. are by induction on the derivation $A\langle\langle v := B\rangle\rangle$. 3. is by induction on the derivation $A\{v := v'\} =_{def} C$. 4. is by induction on A. □

With this lemma, we are starting to see some of the complications of having to take the first relevant variable in the ordered variable list in the definition of $A\langle\langle v := B\rangle\rangle$. In fact, without the condition "whenever $\lambda v''.D$ occurs in A then $v'' \notin FV(BC)$", we would not be able to prove Lemma 1.4. This defeats the purpose of having an ordered list of variables.

However, there is a more substantial reason as to why an ordered list of variables on its own will not work (see Example 1 below). As you recall, we are trying to define the replacement $A_{v:=B}$ in order to define the computation of equation (1). If we use the replacement Definition 10 to define computation, we get this definition:

Definition 11. *We define $\rightarrow_{\overline{\beta}}$ as the least compatible relation closed under:*

$$(\overline{\beta}) \qquad (\lambda v.A)B \rightarrow_{\overline{\beta}} A\langle\langle v := B\rangle\rangle$$

We call this reduction relation $\overline{\beta}$-reduction. We define $\twoheadrightarrow_{\overline{\beta}}$ as the reflexive transitive closure of $\rightarrow_{\overline{\beta}}$.

With this definition we would lose the so-called Church-Rosser (CR) Property which is defined for a relation R as follows:

Definition 12. *We say that a relation R on $\mathcal{M}$ enjoys the CR property if whenever ARB and ARC then there is D such that BRD and CRD.*

$\twoheadrightarrow_{\overline{\beta}}$ does not enjoy the CR property. This can be seen as follows (example is taken from [13]):

Example 1. *$(\lambda xy.yx)((\lambda z.x')y) \twoheadrightarrow_{\overline{\beta}} \lambda y.yx'$ and $(\lambda xy.yx)((\lambda z.x')y) \twoheadrightarrow_{\overline{\beta}} \lambda y'.y'x'$ (assuming the ordered variable list given on page 12). It is clear that $\lambda y.yx' \neq_{\mathcal{M}} \lambda y'.y'x'$ and there is no D such that $\lambda y.yx' \twoheadrightarrow_{\overline{\beta}} D$ and $\lambda y'.y'x' \twoheadrightarrow_{\overline{\beta}} D$. Note that $[\![\lambda y.yx']\!] = [\![\lambda y'.y'x']\!]$.*

4 Syntactic identity revised, Searching for $\equiv$

In the previous section we discussed computation using either grafting (β^w which does not work) or replacement based on ordered variables ($\overline{\beta}$ which has its complications, and moreover, will still not work as we saw in Example 1).

In this section we will discuss two alternatives both of which identify terms modulo bound variables. One approach builds the so-called α-congruence $=_\alpha$ directly from the replacement based on ordered variables given in Definition 10 (see Definition 13) while the other is based on safe applications of the grafting of Definition 7 (see Definition 14) to build a notion $=_{\alpha'}$ of term equivalence. Interestingly, both approaches give the same notion of syntactic equivalence denoted by $\equiv$ where terms that only differ in the name of their bound variables are equivalent. For example: $\lambda y.xy \equiv \lambda z.xz$, $\lambda x.x \equiv \lambda y.y$, $\lambda x'.yx' \equiv \lambda y'.yy'$ and $\lambda x'.yx' \equiv \lambda z.yz$, etc. Note that $\lambda x.xy \neq_{\mathcal{M}} \lambda z.xz$, $\lambda x.x \neq_{\mathcal{M}} \lambda y.y$, etc.

4.1 Defining $=_\alpha$ using the replacement based on ordered variables

Definition 13. *We define $\to_\alpha$ as the least compatible relation closed under:*

$$(\alpha) \qquad \lambda v.A \to_\alpha \lambda v'.A\langle\langle v := v'\rangle\rangle \qquad \text{where } v' \notin FV(A)$$

We call this reduction relation α-reduction, $\lambda v.A$ an α-redex and $\lambda v'.A\langle\langle v := v'\rangle\rangle$ its α-contractum. If R is an α-redex, we write $\Gamma_\alpha[R]$ for its α-contractum. We define $\twoheadrightarrow_\alpha$ (resp. $=_\alpha$) as the reflexive transitive (resp. equivalence) closure of $\to_\alpha$.

Note that $\to_\alpha$ is not symmetric. E.g., using the variable list of page 12:

- $\lambda xy.xy \to_\alpha \lambda y.(\lambda y.xy)\langle\langle x := y\rangle\rangle =_{\mathcal{M}}$
  $\lambda y.\lambda z.(xy)\langle\langle y := z\rangle\rangle\langle\langle x := y\rangle\rangle =_{\mathcal{M}} \lambda yz.yz$.
- $\lambda yz.yz \to_\alpha \lambda x.(\lambda z.yz)\langle\langle y := x\rangle\rangle =_{\mathcal{M}} \lambda xz.xz \neq_{\mathcal{M}} \lambda xy.xy$.
- So, $\lambda xy.xy \to_\alpha \lambda yz.yz$ but $\lambda yz.yz \not\to_\alpha \lambda xy.xy$.
  However, $\lambda yz.yz \twoheadrightarrow_\alpha \lambda xy.xy$.

In fact, $\twoheadrightarrow_\alpha$ is symmetric and hence $=_\alpha$ is the same relation as $\twoheadrightarrow_\alpha$. See Lemma 4.8.

Note also that $=_\alpha$ is closed under replacement (Definition 10), and denotational meaning (Definition 6) and we can now remove the bound variable conditions in Lemma 1.4 as long as we use $=_\alpha$ instead of $=_\mathcal{M}$.

Lemma 2. *1. If $A =_\alpha B$ then $FV(A) = FV(B)$ and $[\![A]\!] = [\![B]\!]$.*

2. If $v \neq v'$, $v' \notin FV(A)$ then $A\langle\langle v := v'\rangle\rangle\langle\langle v' := B\rangle\rangle =_\alpha A\langle\langle v := B\rangle\rangle$.

3. If $A =_\alpha B$ then $C\langle\langle v := A\rangle\rangle =_\alpha C\langle\langle v := B\rangle\rangle$ and $A\langle\langle v := C\rangle\rangle =_\alpha B\langle\langle v := C\rangle\rangle$.

4. If $v \neq v'$ and $v \notin FV(C)$, then $A\langle\langle v := B\rangle\rangle\langle\langle v' := C\rangle\rangle =_\alpha A\langle\langle v' := C\rangle\rangle\langle\langle v := B\langle\langle v' := C\rangle\rangle\rangle\rangle$.

Proof. 1. is by induction on the derivation $A =_\alpha B$ using Lemma 1.(1 and 2) for the case $A =_\mathcal{M} \lambda v.A' \rightarrow_\alpha \lambda v'.A'\langle\langle v := v'\rangle\rangle =_\mathcal{M} B$ where $v' \notin FV(A')$.
2. is by induction on A using Lemma 1.(1 and 2). This also involves a number of sublemmas, all are proved by straightforward induction (see [6]).
3. The proof of $C\langle\langle v := A\rangle\rangle =_\alpha C\langle\langle v := B\rangle\rangle$ is by induction on $\#A$. The proof of $A\langle\langle v := C\rangle\rangle =_\alpha B\langle\langle v := C\rangle\rangle$ is by induction on the derivation $A =_\alpha B$. Both proofs use 1. above.
4. By induction on $\#A$ using Lemma 1.(1 and 2). This also involves a number of sublemmas, all are proved by straightforward induction. $\square$

4.2 Defining $=_{\alpha'}$

In Section 3 we stated that the use of a replacement/substitution via an ordered list of variables suffers from complications and in Lemma 2 we saw that we can remove the bound variables conditions in Lemma 1.4 as long as we use $=_\alpha$ instead of $=_\mathcal{M}$. But $=_\alpha$ (Definition 13) is still built on the replacement/substitution via an ordered list of variables (Definition 10).

Can we forget completely about the ordered variable list and the replacement notion using the ordered variable list (step 1 in the above approach), and simply use grafting (Definition 7) to define an alternative to α-conversion that we can use to define β-reduction?

The next definition attempts to introduce syntactic equivalence $=_{\alpha'}$ which uses grafting but will coincide with the α-congruence (which is based on the replacement which uses an ordered variable list). The use of grafting here will not cause problems since we are applying grafting in a well controlled situation which is guaranteed by the preconditions of the (α') rule (recall Lemma 1.3).

Definition 14. *We define $\to_{\alpha'}$ as the compatible closure of the following rule:*

$$(\alpha') \quad \lambda v.A \to_{\alpha'} \lambda v'.A\{v := v'\} \qquad \textit{if } v' \not\in FV(vA) \textit{ and } v, v' \not\in BV(A)$$

We define $\twoheadrightarrow_{\alpha'}$ (resp. $=_{\alpha'}$) as the reflexive transitive (resp. equivalence) closure of $\to_{\alpha'}$.

Lemma 3. *If $v' \not\in FV(vA)$ and $v, v' \not\in BV(A)$ then $A\{v := v'\}\{v' := v\} = A$ and hence $\to_{\alpha'}$ (resp. $\twoheadrightarrow_{\alpha'}$) is symmetric and so, $\twoheadrightarrow_{\alpha'}$ is the same relation as $=_{\alpha'}$ on λ-terms.*

Proof. The first part is by induction on A. Symmetry of $\to_{\alpha'}$ is by induction on $\to_{\alpha'}$ using what we just proved. Symmetry of $\twoheadrightarrow_{\alpha'}$ then follows. □

Here is a lemma that establishes that using grafting inside the α' rule is safe and that $=_\alpha$ is the same relation as $=_{\alpha'}$.

Lemma 4.
1. *If $A \to_{\alpha'} B$ then $\#A = \#B$, $FV(A) = FV(B)$ and if $A = \lambda v.A'$ then $B = \lambda v'.B'$.*
2. *If $v' \not\in FV(vA)$ and $v, v' \not\in BV(A)$ then $A\{v := v'\} =_{\mathcal{M}} A\langle\langle v := v'\rangle\rangle$.*
3. *If $A \to_{\alpha'} B$ then $A \to_\alpha B$. Hence if $A \twoheadrightarrow_{\alpha'} B$ (resp. $A =_{\alpha'} B$) then $A \twoheadrightarrow_\alpha B$ (resp. $A =_\alpha B$).*
4. *For any A, $v_1, v_2, \cdots v_n$, we can find A' such that $A \twoheadrightarrow_{\alpha'} A'$ and $BV(A') \cap \{v_1, v_2, \cdots v_n\} = \emptyset$.*

5. *For any A, v and $v' \notin FV(vA)$, there is A' such that $v' \notin FV(vA')$, $v, v' \notin BV(A')$, $A \twoheadrightarrow_{\alpha'} A'$ and $A\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} A'\langle\langle v := v'\rangle\rangle$.*

6. *If $A \to_\alpha B$ then $A \twoheadrightarrow_{\alpha'} B$. Hence if $A \twoheadrightarrow_\alpha B$ (resp. $A =_\alpha B$) then $A \twoheadrightarrow_{\alpha'} B$ (resp. $A =_{\alpha'} B$).*

7. *$=_\alpha$ is the same relation as $=_{\alpha'}$*

8. *$\twoheadrightarrow_\alpha$ is symmetric.*

Proof. 1. Easy induction.

2. By induction on A.

3. By induction on the derivation $A \to_{\alpha'} B$ resp. $A \twoheadrightarrow_{\alpha'} B$ resp. $A =_{\alpha'} B$.

4. By induction on A. We only do the case $A =_{\mathcal{M}} \lambda v.B$. By induction hypothesis, there is B' such that $B \twoheadrightarrow_{\alpha'} B'$ and $BV(B') \cap (\{v_1, v_2, \cdots v_n\} \cup \{v\}) = \emptyset$. Let $v' \notin (FV(vB') \cup \{v_1, v_2, \cdots v_n\} \cup BV(B'))$. Then, $A =_{\mathcal{M}} \lambda v.B \twoheadrightarrow_{\alpha'} \lambda v.B' \to_{\alpha'} \lambda v'.B'\{v := v'\}$. Let $A' =_{\mathcal{M}} \lambda v'.B'\{v := v'\}$. Now, $BV(A') \cap \{v_1, v_2, \cdots, v_n\} = (\{v'\} \cup BV(B')) \cap \{v_1, v_2, \cdots, v_n\} = (\{v'\} \cap \{v_1, v_2, \cdots, v_n\}) \cup (BV(B') \cap \{v_1, v_2, \cdots, v_n\}) = \emptyset$.

5. By induction on $\#A$. We only do the case $A = \lambda v''.B$.

   - Case $v'' \notin \{v, v'\}$ then by the induction hypothesis (IH), there is B' such that $v' \notin FV(vB')$, $v, v' \notin BV(B')$, $B \twoheadrightarrow_{\alpha'} B'$ and $B\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} B'\langle\langle v := v'\rangle\rangle$. Hence $v' \notin FV(v(\lambda v''.B'))$, $v, v' \notin BV(\lambda v''.B')$, $\lambda v''.B \twoheadrightarrow_{\alpha'} \lambda v''.B'$ and $\lambda v''.B\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v''.B'\langle\langle v := v'\rangle\rangle$. But $(\lambda v''.B)\langle\langle v := v'\rangle\rangle =_{\mathcal{M}} \lambda v''.B\langle\langle v := v'\rangle\rangle$ and $(\lambda v''.B')\langle\langle v := v'\rangle\rangle =_{\mathcal{M}} \lambda v''.B'\langle\langle v := v'\rangle\rangle$. We are done.
   - Case $v'' = v$ (i.e., $A =_{\mathcal{M}} \lambda v.B$) then by 4. and 1. above, there is $\lambda v_1.B'$ such that $\lambda v.B \twoheadrightarrow_{\alpha'} \lambda v_1.B'$, $\#B = \#B'$, $FV(\lambda v.B) = FV(\lambda v_1.B')$ and $BV(\lambda v_1.B') \cap \{v, v'\} = \emptyset$. By IH, let C be such that $B' \twoheadrightarrow_{\alpha'} C$, $B'\langle\langle v := v'\rangle\rangle \to\to_{\alpha'} C\langle\langle v := v'\rangle\rangle$, $v' \notin FV(C)$ and $v, v' \notin BV(C)$. Hence $\lambda v.B \twoheadrightarrow_{\alpha'} \lambda v_1.B' \twoheadrightarrow_{\alpha'} \lambda v_1.C$, $v' \notin FV(v(\lambda v_1.C))$, $v, v' \notin$

$BV(\lambda v_1.C)$, and since $v \notin FV(\lambda v_1.C)$, then $(\lambda v.B)\langle\langle v := v'\rangle\rangle =_{\mathcal{M}} \lambda v.B \twoheadrightarrow_{\alpha'} \lambda v_1.C =_{\mathcal{M}} (\lambda v_1.C)\langle\langle v := v'\rangle\rangle$.

- Case $v'' = v'$ (i.e., $A =_{\mathcal{M}} \lambda v'.B$) and $v \notin FV(B)$ then by 4. and 1. above, there is $\lambda v_1.B'$ such that $\lambda v'.B \twoheadrightarrow_{\alpha'} \lambda v_1.B'$, $\#B = \#B'$, $FV(\lambda v'.B) = FV(\lambda v_1.B')$ and $BV(\lambda v_1.B') \cap \{v, v'\} = \emptyset$. Since $v' \notin FV(B') \subseteq (FV(B) \setminus \{v'\}) \cup \{v_1\}$, then by IH, there is C such that $B' \twoheadrightarrow_{\alpha'} C$, $B'\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} C\langle\langle v := v'\rangle\rangle$, $v' \notin FV(C)$ and $v, v' \notin BV(C)$ and by 1. above, $v \notin FV(C)$. Hence $\lambda v'.B \twoheadrightarrow_{\alpha'} \lambda v_1.B' \twoheadrightarrow_{\alpha'} \lambda v_1.C$, $v' \notin FV(v(\lambda v_1.C))$, $v, v' \notin BV(\lambda v_1.C)$, and since $v \notin FV(v'v_1BC)$, $(\lambda v'.B)\langle\langle v := v'\rangle\rangle =_{\mathcal{M}} \lambda v'.B \twoheadrightarrow_{\alpha'} \lambda v_1.C =_{\mathcal{M}} (\lambda v_1.C)\langle\langle v := v'\rangle\rangle$.

- Case $v'' = v'$ (i.e., $A =_{\mathcal{M}} \lambda v'.B$) and $v \in FV(B)$ then $(\lambda v'.B)\langle\langle v := v'\rangle\rangle =_{\mathcal{M}} \lambda v_1''.B\langle\langle v' := v_1''\rangle\rangle\langle\langle v := v'\rangle\rangle$ where v_1'' is the first variable such that $v_1'' \notin FV(v'B)$ (and so, $v_1'' \neq v$). By Lemma 1.(1 and 2), $v' \notin FV(B\langle\langle v' := v_1''\rangle\rangle)$. By IH, there is C such that $v' \notin FV(C)$, $v, v' \notin BV(C)$, $B\langle\langle v' := v_1''\rangle\rangle \twoheadrightarrow_{\alpha'} C$,
  $\lambda v_1''.B\langle\langle v' := v_1''\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v_1''.C$ and
  $B\langle\langle v' := v_1''\rangle\rangle\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} C\langle\langle v := v'\rangle\rangle$.

  Hence $\lambda v_1''.B\langle\langle v' := v_1''\rangle\rangle\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v_1''.C\langle\langle v := v'\rangle\rangle$ and so, $(\lambda v'.B)\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} (\lambda v_1''.C)\langle\langle v := v'\rangle\rangle$.

  All that is left now is to show that $\lambda v'.B \twoheadrightarrow_{\alpha'} \lambda v_1''.C$.

  Since $v_1'' \notin FV(v'B)$ then by IH, there is D such that $v', v_1'' \notin BV(D)$, $v_1'' \notin FV(v'D)$, $B \twoheadrightarrow_{\alpha'} D$, and $B\langle\langle v' := v_1''\rangle\rangle \twoheadrightarrow_{\alpha'} D\langle\langle v' := v_1''\rangle\rangle$. Hence, $\lambda v'.B \twoheadrightarrow_{\alpha'} \lambda v'.D$, and
  $\lambda v_1''.B\langle\langle v' := v_1''\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v_1''.D\langle\langle v' := v_1''\rangle\rangle$.
  Since $v', v_1'' \notin BV(D)$, $v_1'' \notin FV(v'D)$, then by 2. above,
  $\lambda v'.D \rightarrow_{\alpha'} \lambda v_1''.D\{v' := v_1''\} =_{\mathcal{M}} \lambda v_1''.D\langle\langle v' := v_1''\rangle\rangle$.
  Since $\twoheadrightarrow_{\alpha'}$ is symmetric (Lemma 3), then
  $\lambda v_1''.D\langle\langle v' := v_1''\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v_1''.B\langle\langle v' := v_1''\rangle\rangle$. Hence:
  $\lambda v'B. \twoheadrightarrow_{\alpha'} \lambda v'.D \rightarrow_{\alpha'} \lambda v_1''.D\langle\langle v' := v_1''\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v_1''.B\langle\langle v' := v_1''\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v_1''.C$ and we are done.

6. The proof of $\rightarrow_\alpha \subseteq \twoheadrightarrow_{\alpha'}$ is by induction on the derivation $A \rightarrow_\alpha B$. We only do one case.

Assume $\lambda v.C \to_\alpha \lambda v'.C\langle\langle v := v'\rangle\rangle$ where $v' \notin FV(vC)$ (the case $v = v'$ is trivial). Then by 5. above, there is C' such that $v' \notin FV(vC')$, $v, v' \notin BV(C')$, $C \twoheadrightarrow_{\alpha'} C'$ and $C\langle\langle v := v'\rangle\rangle \to\to_{\alpha'} C'\langle\langle v := v'\rangle\rangle$. Hence $\lambda v.C \twoheadrightarrow_{\alpha'} \lambda v.C'$ and $\lambda v'.C\langle\langle v := v'\rangle\rangle \to\to_{\alpha'} \lambda v'.C'\langle\langle v := v'\rangle\rangle$. By symmetricity Lemma 3, $\lambda v'.C'\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v'.C\langle\langle v := v'\rangle\rangle$. By 2. above, $\lambda v'.C'\{v := v'\} =_{\mathcal{M}} \lambda v'.C'\langle\langle v := v'\rangle\rangle$.
Hence $\lambda v.C \twoheadrightarrow_{\alpha'} \lambda v.C' \to_{\alpha'} \lambda v'.C'\langle\langle v := v'\rangle\rangle \twoheadrightarrow_{\alpha'} \lambda v'.C\langle\langle v := v'\rangle\rangle$.

7. Use 3 and 6 above.

8. If $A \twoheadrightarrow_\alpha B$ then by 6. above, $A \twoheadrightarrow_{\alpha'} B$ and by symmetric Lemma 3, $B \twoheadrightarrow_{\alpha'} A$ which mean by 3. above, $B \twoheadrightarrow_\alpha A$.

$\square$

Now that $=_\alpha$ is the same as $=_{\alpha'}$, we use $\equiv$ to denote these relations.

Definition 15 ($\equiv$)**.** *We write $A \equiv B$ iff $A =_{\alpha'} B$ iff $A =_\alpha B$.*
When $A \equiv B$, we say that A and B are syntactically equivalent.

5 Beta Reduction

So far, β^w-reduction of Table 1 does not work and $\overline{\beta}$-reduction of Definition 11 does not satisfy CR. But, we are ready to define the computation that will work and will guarantee CR.

5.1 Computation β'-reduction based on $=_\alpha$

In this section we define the β-reduction relation (called here β') given in [13]. Note that although $\to_{\beta'}$ is the same as $\to_{\overline{\beta}}$, a β'-redex (resp. β'-contractum) is also a $\overline{\beta}$-redex (resp. $\overline{\beta}$-contractum) and vice-versa. However, the reflexive transitive closure $\twoheadrightarrow_{\beta'}$ incorporates also α-reduction (unlike $\twoheadrightarrow_{\overline{\beta}}$).

Definition 16. *We define $\to_{\beta'}$ as $\to_{\overline{\beta}}$ and similarly define a β'-redex, a β'-contractum and $\Gamma_{\beta'}[R]$ when R is a β'-redex.*

We define $\twoheadrightarrow_{\beta'}$ as the reflexive transitive closure of $\to_{\beta'} \cup \to_\alpha$.

Unlike $\twoheadrightarrow_{\overline{\beta}}$ (see Example 1), this $\twoheadrightarrow_{\beta'}$ relation satisfies the CR property (see [12]).

The next help lemma which is based on lemmas 1.12, 1.13 and 1.14 of [13] establishes the closure of α-congruence under β'-reduction.

Lemma 5. *1. If R is a β'-redex and v, A are such that $FV(vA) \cap BV(R) = \emptyset$ then $R\langle\langle v := A\rangle\rangle$ is a β'-redex and $\Gamma_{\beta'}[R\langle\langle v := A\rangle\rangle] =_\alpha (\Gamma_{\beta'}[R])\langle\langle v := A\rangle\rangle$.*

2. If R is a β'-redex and $R =_\alpha R'$ then R' is also a β'-redex and $\Gamma_{\beta'}[R] =_\alpha \Gamma_{\beta'}[R']$.

3. If $A =_\alpha B$ and β'-redex R occurs in A, then a β'-redex R' occurs in B such that if $A \to_{\beta'} A'$ using β'-redex R and $B \to_{\beta'} B'$ using β'-redex R', then $A' =_\alpha B'$.

Proof. 1. Assume $R =_{\mathcal{M}} (\lambda v'.B)C$. Then, $v' \neq v$ and $v' \notin FV(A)$. Hence, $R\langle\langle v := A\rangle\rangle =_{\mathcal{M}} (\lambda v'.B)\langle\langle v := A\rangle\rangle C\langle\langle v := A\rangle\rangle =_{\mathcal{M}} (\lambda v'.B\langle\langle v := A\rangle\rangle)C\langle\langle v := A\rangle\rangle$ is a β'-redex. Moreover, $\Gamma_{\beta'}[R\langle\langle v := A\rangle\rangle] =_\alpha B\langle\langle v := A\rangle\rangle\langle\langle v' := C\langle\langle v := A\rangle\rangle\rangle\rangle =_\alpha^{Lemma\ 1.4} B\langle\langle v' := C\rangle\rangle\langle\langle v := A\rangle\rangle =_\alpha (\Gamma_{\beta'}[R])\langle\langle v := A\rangle\rangle$.

2. By induction on the derivation $R =_\alpha R'$ using Lemmas 2.(2, 3).

3. See [13], Lemma A1.14 (b).

□

In the next lemma we connect both the β' and $\overline{\beta}$ relations.

Lemma 6. *1. $A \to_{\beta'} B$ iff $A \to_{\overline{\beta}} B$.*

2. If $A \twoheadrightarrow_{\overline{\beta}} B$ then $A \twoheadrightarrow_{\beta'} B$.

3. If $A \twoheadrightarrow_{\beta'} B$ then there is C such that $A \twoheadrightarrow_{\overline{\beta}} C \twoheadrightarrow_\alpha B$.

Proof. 1. is obvious since (β') and $(\overline{\beta})$ rules are the same.
2. is a corollary of 1.
3. By definition of $\twoheadrightarrow_{\beta'}$, $A \twoheadrightarrow_{\beta'} B$ is a sequence (possibly empty) of $\to_{\beta'}$ and $\to_\alpha$ steps. We will show that any $A_1 \to_\alpha A_2 \to_{\beta'} A_3$ can be written as $A_1 \to_{\beta'} A_4 \twoheadrightarrow_\alpha A_3$ for some A_4.

Since $A_1 \to_\alpha A_2 \to_\beta A_3$ then $A_1 =_\alpha A_2$ and $A_2 \to_{\beta'} A_3$ (say by β'-redex R). By Lemma 5.3, there is a β'-redex R' in A_1 such that $A_1 \to_{\beta'} A_4$ using β'-redex R' and $A_4 =_\alpha A_3$. But $=_\alpha$ is the same as relation as $\twoheadrightarrow_\alpha$ and therefore, $A_1 \to_{\beta'} A_4 \twoheadrightarrow_\alpha A_3$.

This means all the $\to_\alpha$-steps can be postponed till after all the $\to_{\beta'}$-steps. By what we just proved and 1 above, if $A \twoheadrightarrow_{\beta'} B$ then there is C such that $A \twoheadrightarrow_{\overline{\beta}} C \twoheadrightarrow_\alpha B$. □

5.2 β''-reduction based on clean terms and $=_{\alpha'}$

In this section we define the β''-reduction relation. The basic (β'') rule first transforms the redex into a so-called clean term before any beta reduction takes place and uses grafting since clean terms are safe with grafting. Here, the reflexive transitive closure $\twoheadrightarrow_{\beta''}$ also incorporates α'-reduction. So, for grafting to work, we use α'-reduction at the basic reduction stage (to clean the term) and at the reflexive transitive stage.

Definition 17. *A λ-term A is clean iff the following two conditions hold:*

- *$BV(A) \cap FV(A) = \emptyset$.*
- *For any v, λv may occur at most once in A.*

So $\lambda x.\lambda y.(\lambda z.xz(yz))(\lambda y.yz)$ is not clean. However, a clean version is $\lambda x.\lambda y.(\lambda z'.xz'(yz'))(\lambda y'.y'z)$. Of course a term may have different clean versions. E.g., $\lambda x'.\lambda y.(\lambda z'.x'z'(yz'))(\lambda y'.y'z)$ is also a (different) clean version of the term above.

Lemma 7. *For any A, there is a clean B such that $A \twoheadrightarrow_{\alpha'} B$.*

Proof. By Lemma 4.4, we can find A' such that $A \twoheadrightarrow_{\alpha'} A'$ and $BV(A') \cap (FV(A) \cup BV(A)) = \emptyset$. By Lemma 4.1, $FV(A) = FV(A')$. Hence, $BV(A') \cap FV(A') = \emptyset$. If $BV(A') = \emptyset$, we are done. Take B to be A'.

Else, assume that A' has n λs $\lambda v_1, \lambda v_2, \cdots, \lambda v_n$ (note that some of the v_is may be equal) occurring from left to right in that order. Then $A' =_{\mathcal{M}} C_n[\lambda v_n.B_n]$ for some context $C_n[\ ]$ and some term B_n. Let $v'_n \notin FV(A') \cup BV(A')$. Hence, $v'_n \notin FV(v_n B_n) \cup BV(B_n)$ and $A' \to_{\alpha'} C_n[\lambda v'_n.B_n\{v_n := v'_n\}] =_{\mathcal{M}} A'_n$. Note that $FV(A') = FV(A'_n)$, $v'_n \notin$

$FV(A'_n)$ and A'_n has n λs $\lambda v_1, \lambda v_2, \cdots, \lambda v_{n-1}, \lambda v'_n$ occurring from left to right in that order, where $\lambda v'_n$ occurs once in A'_n and $\{v_1, v_2, \cdots, v_{n-1}, v'_n\} \cap FV(A'_n) = \emptyset$.
Repeat the above process for λv_{n-1} where $A'_n =_{\mathcal{M}} C_{n-1}[\lambda v_{n-1}.B_{n-1}]$ for some context C_{n-1} and term B_{n-1}. Let $v'_{n-1} \notin FV(A'_n) \cup BV(A'_n)$. Hence, $v'_{n-1} \notin FV(v_{n-1}B_{n-1}) \cup BV(B_{n-1})$ and $A'_n \to_{\alpha'} C_{n-1} [\lambda v'_{n-1}.B_{n-1}\{v_{n-1} := v'_{n-1}\}] =_{\mathcal{M}} A'_{n-1}$. Note that $FV(A'_n) = FV(A'_{n-1})$, $v'_{n-1} \notin FV(A'_{n-1})$ and A'_{n-1} has n λs λv_1, λv_2, $\cdots$, λv_{n-2}, $\lambda v'_{n-1}, \lambda v'_n$ occurring from left to right in that order, where $v_n \neq v_{n-1}$ and each of $\lambda v'_n$ and $\lambda v'_{n-1}$ occurs once in A'_{n-1} and $\{v_1, v_2, \cdots, v_{n-2}, v'_{n-1}, v'_n\} \cap FV(A'_{n-1}) = \emptyset$.
Like we constructed A'_n, A'_{n-1}, we continue to construct A'_{n-2}, $\cdots$, A'_2, A'_1 such that $A' \to_{\alpha'} A'_n \to_{\alpha'} A'_{n-1} \to_{\alpha'} A'_{n-2} \cdots \to_{\alpha'} A'_1$.
A'_1 is the clean term we are after. □

Lemma 8. *If $(\lambda v.A)B$ is clean then $A\{v := B\} =_{\mathcal{M}} A\langle\langle v' := B\rangle\rangle$.*

Proof. By induction on A. □

Based on Lemmas 7 and 8, we can define beta reduction based on clean terms and α'-reduction as follows:

Definition 18. *We define $\to_{\beta''}$ as the least compatible relation closed under:*

$$(\beta'') \qquad (\lambda v.A)B \to_{\beta''} A'\{v' := B'\}$$
$$\text{where } (\lambda v.A)B \twoheadrightarrow_{\alpha'} (\lambda v'.A')B' \text{ and } (\lambda v'.A')B' \text{ is clean.}$$

We call this reduction relation β''-reduction. We define $\twoheadrightarrow_{\beta''}$ as the reflexive transitive closure of $\to_{\beta''} \cup \to_{\alpha'}$.[15]

Lemma 9. *1. If $A \to_{\beta''} B$ then $A \twoheadrightarrow_{\beta'} B$.*

2. If $A \to_{\beta'} B$ then $A \twoheadrightarrow_{\beta''} B$.

[15]Note that in the above definition of $\twoheadrightarrow_{\beta''}$, the reflexive transitive closure of $\to_{\beta''} \cup \to_{\alpha'}$ is necessary since otherwise, if we take $\twoheadrightarrow_{\beta''}$ as the reflexive transitive closure of $\to_{\beta''}$ then we would lose CR. For example, $(\lambda x.(\lambda x.x)x)x \to_{\beta''} x(\lambda y.y)$ and $(\lambda x.(\lambda x.x)x)x \to_{\beta''} x(\lambda z.z)$ but in the absence of $\to_{\alpha'}$ we can never show that $x(\lambda y.y)$ and $x(\lambda z.z)$ β''-reduce to a common term (say $x(\lambda u.u)$).

Proof. 1. By induction on $A \rightarrow_{\beta''} B$. For the case $(\lambda v.A)B \rightarrow_{\beta''} A'\{v' := B'\}$ where $(\lambda v.A)B \twoheadrightarrow_{\alpha'} (\lambda v'.A')B'$ and $(\lambda v'.A')B'$ is clean, use Lemmas 8 and 4.
2. By induction on $A \rightarrow_{\beta'} B$. We do the case $(\lambda v.A)B \rightarrow_{\beta'} A\langle\langle v := B\rangle\rangle$. By Lemmas 7 and 8, for a clean $(\lambda v'.A')B'$, $(\lambda v.A)B \rightarrow_{\alpha'} (\lambda v'.A')B'$ and $(\lambda v.A)B \rightarrow_{\beta''} A'\langle\langle v' := B'\rangle\rangle$. By Lemma 5.2, since $(\lambda v.A)B =_{\alpha} (\lambda v'.A')B'$ then $A\langle\langle v := B\rangle\rangle =_{\mathcal{M}} \Gamma_{\beta'}[(\lambda v.A)B] =_{\alpha} \Gamma_{\beta'}[(\lambda v'.A')B'] =_{\mathcal{M}} A'\langle\langle v' := B'\rangle\rangle$. Hence by Lemma 4, $A'\langle\langle v' := B'\rangle\rangle \twoheadrightarrow_{\alpha'} A\langle\langle v := B\rangle\rangle$ and so, $(\lambda v.A)B \twoheadrightarrow_{\beta''} A\langle\langle v := B\rangle\rangle$. □

Since β' satisfies CR, by Lemma 9, also β'' satisfies CR.

However, there is an oddity about β'' compared to β'. The latter is a function whereas the former is not. If $A \rightarrow_{\beta'} B$ then B is unique because the replacement relation for β' is based on an ordered variable list. This is not the case for $\rightarrow_{\beta''}$ where for example, $(\lambda x.x(\lambda x.x))x \rightarrow_{\beta''} x(\lambda y.y)$ and $(\lambda x.x(\lambda x.x))x \rightarrow_{\beta''} x(\lambda z.z)$ but $x(\lambda y.y) \neq x(\lambda z.z)$.

5.3 Equating terms modulo $\equiv$

Now that syntactic equivalence $\equiv$ is defined (and is the same relation as α-congruence $=_{\alpha}$ set up in terms of replacement using an ordered variable list and α'-congruence $=_{\alpha'}$ set up in terms of grafting), we could identify terms modulo syntactic equivalence $\equiv$, and then define substitution as a refined form of both replacement using an ordered variable list and grafting. For this, all we need to do is to use $\equiv$ instead $=_{\mathcal{M}}$ in Definition 10 and to remove the statement "and v'' is the first variable in the ordered list $\mathcal{V}$" from clause 6. This gives us the following definition:

Definition 19 (Substitution $A[v := B]$, using $\equiv$)**.** *For any A, B, v, we define $A[v := B]$ to be the result of substituting B for every free occur-*

rence of v in A, as follows:

$$\begin{array}{lcl}
1.\ v[v := B] & \equiv & B \\
2.\ v'[v := B] & \equiv & v' \qquad \textit{if } v \neq v' \\
3.\ (AC)[v := B] & \equiv & A[v := B]C[v := B] \\
4.\ (\lambda v.A)[v := B] & \equiv & \lambda v.A \\
5.\ (\lambda v'.A)[v := B] & \equiv & \lambda v'.A[v := B] \qquad \textit{if } v \neq v' \\
 & & \textit{and } (v' \notin FV(B) \textit{ or } v \notin FV(A)) \\
6.\ (\lambda v'.A)[v := B] & \equiv & \lambda v''.A[v' := v''][v := B] \qquad \textit{if } v \neq v' \\
 & & \textit{and } (v' \in FV(B) \textit{ and } v \in FV(A)) \\
 & & \textit{such that } v'' \notin FV(AB).
\end{array}$$

Recall Lemma 2.3 which implies that if $A \equiv B$ and $C \equiv D$ then $A\langle\langle v := C\rangle\rangle \equiv B\langle\langle v := D\rangle\rangle$. Using $\equiv$ instead of $=_{\mathcal{M}}$ means that we can also move from $A\langle\langle v := B\rangle\rangle$ to $A[v := B]$ as seen by the following lemma which is proven by induction on $\#A$.

Lemma 10. *For any v, A and B, $A[v := B] \equiv A\langle\langle v := B\rangle\rangle$.*

We could even go one step further in our use of $=_\alpha$ and use the so-called variable convention [2] where we assume that no variable name is both free and bound within the same term.[16] So, we will never have terms like $(\lambda v.B)[v := C]$ or $(\lambda v.B)v$. Instead, the λv would be changed to a $\lambda v'$ where $v \neq v'$. This way, clauses 4 and 6 of Definition 19 do not hold and we know that in clause 5, $v \neq v'$ and ($v' \notin FV(B)$ or $v \notin FV(A)$) always hold. Hence, if we always write terms following the variable convention, can define substitution as:

Definition 20. *For any A, B, v, we define $A[v := B]$ to be the result of substituting B for every free occurrence of v in A, as follows:*

$$\begin{array}{lcl}
1.\ v[v := B] & \equiv & B \\
2.\ v'[v := B] & \equiv & v' \qquad \textit{if } v \neq v' \\
3.\ (AC)[v := B] & \equiv & A[v := B]C[v := B] \\
5.\ (\lambda v'.A)[v := B] & \equiv & \lambda v'.A[v := B]
\end{array}$$

[16]Clean terms satisfy the variable convention, but the other way round does not necessarily hold.

So here, we always use terms modulo names of bound variables and use $\equiv$ (which is the same as $=_\alpha$ and $=_{\alpha'}$). In this case, we could use the replacement given in Definition 19. If we also want our terms to be written according to the variable convention (where the names of bound variables are always different from the free ones) then instead of Definition 19, we can use Definition 20. Whicever style we take here (always modulo bound variable names, with or without variable convention), we define β-reduction as follows:

Definition 21. *We define $\to_\beta$ as the least compatible relation closed under:*

$$(\beta) \qquad (\lambda v.A)B \to_\beta A[v := B]$$

We call this reduction relation β-reduction. We define $\twoheadrightarrow_\beta$ as the reflexive transitive closure of $\to_\beta$.

Let $r \in \{\overline{\beta}, \beta', \beta''\}$. It is easy to show that $\to_r \subseteq \to_\beta$ and that, if $A \to_\beta B$ then there are A', B' such that $A' \equiv A$, $B' \equiv B$, and $A' \to_r B'$.

We end this section by summarising the substitutions and β-reductions introduced in this paper. The basic axioms introduced are:

- (β^w) $(\lambda v.A)B \to_{\beta^w} A\{v := B\}$.
- (α) $\lambda v.A \to_\alpha \lambda v'.A\langle\langle v := v'\rangle\rangle$ where $v' \notin FV(A)$.
- (α') $\lambda v.A \to_{\alpha'} \lambda v'.A\{v := v'\}$ if $v' \notin FV(vA)$ and $v, v' \notin BV(A)$.
- $(\overline{\beta})$ $(\lambda v.A)B \to_{\overline{\beta}} A\langle\langle v := B\rangle\rangle$.
- (β') $(\lambda v.A)B \to_{\beta'} A\langle\langle v := B\rangle\rangle$.
- (β'') $(\lambda v.A)B \to_{\beta''} A'\{v' := B'\}$
  where $(\lambda v.A)B \twoheadrightarrow_{\alpha'} (\lambda v'.A')B'$ and $(\lambda v'.A')B'$ is clean.
- (β) $(\lambda v.A)B \to_\beta A[v := B]$.

The reduction relations based on these axioms are:

- For each $r \in \{\beta^w, \alpha, \alpha', \overline{\beta}, \beta', \beta'', \beta\}$:
  - We define $\to_r$ as the compatible closure of (r) and call this reduction relation r-reduction.

  - We call the term on the left (resp. right) of $\to_r$ in (r), an r-redex (resp. an r-contractum).
  - If R is an r-redex, we write $\Gamma_r[R]$ for its r-contractum.

- For each $r \in \{\beta^w, \alpha, \alpha', \overline{\beta}, \beta\}$, we define $\twoheadrightarrow_r$ (resp. $=_r$) as the reflexive transitive (resp. equivalence) closure of $\to_r$.

- We define $\twoheadrightarrow_{\beta'}$ as the reflexive transitive (resp. equivalence) closure of $\to_{\beta'} \cup \to_\alpha$.

- We define $\twoheadrightarrow_{\beta''}$ as the reflexive transitive closure of $\to_{\beta''} \cup \to_{\alpha'}$.

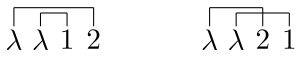
Replacement $A_{v:=B}$	$=$ on $\mathcal{M}$	$(\lambda v.A)B$ $\to_r ?$	Extra	$\twoheadrightarrow_r$	CR
$A\{v := B\}$	$=_\mathcal{M}$	$r = \beta^w$ ✗ ? is $A_{v:=B}$			
$A\langle\langle v := B\rangle\rangle$	$=_\mathcal{M}$	$r = \overline{\beta}$? is $A_{v:=B}$		ref. tran. clos. of $\to_r$	✗
$A\langle\langle v := B\rangle\rangle$	$=_\mathcal{M}$	$r = \beta' = \overline{\beta}$? is $A_{v:=B}$	$=_\alpha$	ref. tran. clos. of $\to_r \cup \to_\alpha$	✓
$A\{v := B\}$	$=_\mathcal{M}$	$r = \beta''$? is $A'_{v':=B'}$, $(\lambda v.A)B =_{\alpha'}$ $(\lambda v'.A')B'$ clean	$=_{\alpha'}$	ref. tran. clos. of $\to_r \cup \to_\alpha$	✓
A[v:=B]	$=_\alpha$	$r = \beta$? is $A_{v:=B}$	var. conv.	ref. tran. clos. of $\to_r$	✓

6 The λ-calculus with de Bruijn indices, towards Explicit Substitutions

We can avoid the problem of variable capture by getting rid of variables and using *De Bruijn indices* which are natural numbers that represent the occurrences of variables in the term. In this section, we introduce the λ-calculus with de Bruijn indices and look at the substitution and reduction rules with de Bruijn indices which will lead us naturally to a calculus with explicit substitutions.

6.1 The classical λ-calculus with de Bruijn indices

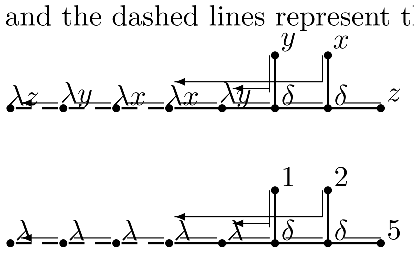
In this approach, an index n in a term A is bound by the nth λ on the left of n. For example, using de Bruijn indices, we write $\lambda xy.yx$ and $\lambda xy.xy$ as follows:



If n is free in A, then we look in a so-called *free variable list.* Say:

$$x, y, z, x', y', z', x'', y'', z'', \ldots$$

For example, the trees of $\lambda x.\lambda y.zxy$ and its translation $\lambda\lambda 521$ are as follows (here δ stands for application, the trees are drawn horizontally to save space, and the dashed lines represent the free variable list):



Definition 22. *We define* Λ*, the* set of terms with de Bruijn indices*, as follows:*

$$\Lambda ::= \mathbb{N} \mid (\Lambda\Lambda) \mid (\lambda\Lambda)$$

As for $\mathcal{M}$*, we use* $A, B, \ldots$ *to range over* Λ*. We also use* $m, n, \ldots$ *to range over* $\mathbb{N}$ *(positive natural numbers).*

We use similar notational conventions and compatibility rules as before. However, here we cannot compress a sequence of λ's to one. While we write $\lambda z.\lambda y.yz$ as $\lambda zy.yz$, we cannot write $\lambda\lambda 12$ as $\lambda 12$ (which is $\lambda y.yx$).

Before we can do substitution, we must learn how to update variables. This will be needed. For example, β-reducing $(\lambda\lambda 521)(\lambda 31)$

should result in $\lambda 5 2 1$ where the variable 2 that was bound by the λ that disappeared is replaced by the argument $\lambda 3 1$. But we cannot just put $\lambda 3 1$ instead of 2 in $\lambda 5 2 1$. First, the 5 should be decreased by 1 because one λ has disappeared from $\lambda\lambda 5 2 1$. Also, the 3 of $\lambda 3 1$ should be increased by 1 when it is inserted in the hole of $\lambda 5[\,]1$ since the $[\,]$ is inside an extra λ and so all the free variables of $\lambda 3 1$ must be increased. The next definition introduces this updating. The intuition behind U^i_k is that k tests for free variables and $i-1$ is the value by which a variable, if free, must be incremented.

Definition 23. *The* meta-updating functions $U^i_k : \Lambda \to \Lambda$ *for* $k \geq 0$ *and* $i \geq 1$ *are defined inductively as follows:*

$$U^i_k(AB) \equiv U^i_k(A)\, U^i_k(B) \qquad\qquad U^i_k(\boldsymbol{n}) \equiv \begin{cases} \boldsymbol{n}+i-1 & \textit{if} \;\; n > k \\ \boldsymbol{n} & \textit{if} \;\; n \leq k\,. \end{cases}$$
$$U^i_k(\lambda A) \equiv \lambda(U^i_{k+1}(A))$$

Using this updating, we define substitution in the obvious way. The first two equalities propagate the substitution through applications and abstractions and the last one carries out the substitution of the intended variable (when $n = i$) by the updated term. If the variable is not the intended one it must be decreased by 1 if it is free (case $n > i$) because one λ has disappeared, whereas if it is bound (case $n < i$) it must remain unaltered.

Definition 24. *The* meta-substitutions at level i, *for* $i \geq 1$, *of a term* $B \in \Lambda$ *in a term* $A \in \Lambda$, *denoted* $A\{\!\{\boldsymbol{i} \leftarrow B\}\!\}$, *is defined inductively on* A *as follows:*

$$\begin{array}{lll}
(A_1A_2)\{\!\{i \leftarrow B\}\!\} & \equiv & (A_1\{\!\{i \leftarrow B\}\!\})\,(A_2\{\!\{i \leftarrow B\}\!\}) \\
(\lambda A)\{\!\{i \leftarrow B\}\!\} & \equiv & \lambda(A\{\!\{i+1 \leftarrow B\}\!\}) \\
n\{\!\{i \leftarrow B\}\!\} & \equiv & \begin{cases} n-1 & \textit{if} \;\; n > i \\ U^i_0(B) & \textit{if} \;\; n = i \\ n & \textit{if} \;\; n < i\,. \end{cases}
\end{array}$$

The following lemma establishes the properties of the meta-substitutions and meta-updating functions. The proof of this lemma is obtained by induction on A and can be found in [16] (the proof of 3 requires 2 with $p = 0$; the proof of 4 uses 1 and 3 both with $k = 0$; finally, 5 with $p = 0$ is needed to prove 6).

Lemma 11.

1. *For* $k < n \leq k + i$ *we have:* $U_k^i(A) \equiv U_k^{i+1}(A)\{\!\{\boldsymbol{n} \leftarrow B\}\!\}$.
2. *For* $p \leq k < j + p$ *we have:* $U_k^i(U_p^j(A)) \equiv U_p^{j+i-1}(A)$.
3. *For* $i \leq n - k$ *we have:* $U_k^i(A)\{\!\{\boldsymbol{n} \leftarrow B\}\!\} \equiv U_k^i(A\{\!\{\boldsymbol{n} - i + 1 \leftarrow B\}\!\})$.
4. *[Meta-substitution lemma] For* $1 \leq i \leq n$ *we have:*
   $A\{\!\{\boldsymbol{i} \leftarrow B\}\!\}\{\!\{\boldsymbol{n} \leftarrow C\}\!\} \equiv A\{\!\{\boldsymbol{n} + 1 \leftarrow C\}\!\}\{\!\{\boldsymbol{i} \leftarrow B\{\!\{\boldsymbol{n} - i + 1 \leftarrow C\}\!\}\}\!\}$.
5. *For* $m \leq k + 1$ *we have:* $U_{k+p}^i(U_p^m(A)) \equiv U_p^m(U_{k+p+1-m}^i(A))$.
6. *[Distribution lemma]*
   For $n \leq k + 1$ *we have:* $U_k^i(A\{\!\{\boldsymbol{n} \leftarrow B\}\!\}) \equiv U_{k+1}^i(A)\{\!\{\boldsymbol{n} \leftarrow U_{k-n+1}^i(B)\}\!\}$.

Case 4 is the version of Lemma 1.4 using de Bruijn indices.

Definition 25. β_1-reduction *is the least compatible relation on* Λ *generated by:*

$$(\beta_1\text{-rule}) \qquad (\lambda A)\, B \to_{\beta_1} A\{\!\{\mathit{1} \leftarrow B\}\!\}$$

We define $\twoheadrightarrow_{\beta_1}$ *as the reflexive transitive closure of* $\to_{\beta_1}$.

It is easy to check ,that $(\lambda 521)\{\!\{\mathtt{1} \leftarrow (\lambda 31)\}\!\} \equiv \lambda 4(\lambda 41)\mathtt{1}$ and hence $(\lambda\lambda 521)(\lambda 31) \to_{\beta_1} \lambda 4(\lambda 41)\mathtt{1}$.

The λ-calculi with variable names and with de Bruijn indices are isomorphic (there are translation functions between $\mathcal{M}$ and Λ which are inverses of each other and which preserve their respective β-reductions, see [18]).

6.2 The classical λ-calculus with de Bruijn indices and explicit substitutions

Although the λ-calculus with de Bruijn indices is not easy for humans, it is very straightforward for machines to implement its meta-updating, meta-substitution and beta rules. In fact, the λ-calculus with de Bruijn has an important place in the implementations of functional languages. In this section, we take the classical λ-calculus with de Bruijn indices exactly as it is, but simply turn its meta-updating and meta-substitution to

$$
\begin{array}{lrcl}
\sigma\textit{-generation} & (\lambda A)\, B & \longrightarrow & A\, \sigma^1\, B \\
\sigma\textit{-}\lambda\textit{-transition} & (\lambda A)\, \sigma^i B & \longrightarrow & \lambda(A\, \sigma^{i+1}\, B) \\
\sigma\textit{-app-transition} & (A_1\, A_2)\, \sigma^i B & \longrightarrow & (A_1\, \sigma^i B)\, (A_2\, \sigma^i B) \\
\sigma\textit{-destruction} & \mathtt{n}\, \sigma^i B & \longrightarrow & \begin{cases} \mathtt{n}-1 & \text{if } n > i \\ \varphi_0^i\, B & \text{if } n = i \\ \mathtt{n} & \text{if } n < i \end{cases} \\
\varphi\textit{-}\lambda\textit{-transition} & \varphi_k^i(\lambda A) & \longrightarrow & \lambda(\varphi_{k+1}^i\, A) \\
\varphi\textit{-app-transition} & \varphi_k^i(A_1\, A_2) & \longrightarrow & (\varphi_k^i\, A_1)\, (\varphi_k^i\, A_2) \\
\varphi\textit{-destruction} & \varphi_k^i\, \mathtt{n} & \longrightarrow & \begin{cases} \mathtt{n}+i-1 & \text{if } n > k \\ \mathtt{n} & \text{if } n \le k \end{cases}
\end{array}
$$

Figure 1: The λs-rules

the object level to obtain a calculus of explicit substitutions. Extending λ-calculi with explicit substitutions is essential for the implementations of these calculi.

Definition 26 (Syntax of the λs-calculus)**.** *Terms of the λs-calculus are given by:*

$$\Lambda s ::= \mathbb{N} \,|\, (\Lambda s \Lambda s) \,|\, (\lambda \Lambda s) \,|\, (\Lambda s\, \sigma^i \Lambda s) \,|\, (\varphi_k^i \Lambda s) \qquad \textit{where} \quad i \geq 1\,, \quad k \geq 0\,.$$

We use the notational conventions to get rid of unnecessary parenthesis.

Now, we need to include reduction rules that operate on the new terms built with updating and substitutions. Definitions 23 and 24 suggest these rules. The resulting calculus is the explicit substitution calculus λs of [16] whose set of rules is given in Figure 1. Note that these rules are nothing more than β_1 written now as σ-generation, together with the rules of Definitions 23 and 24 oriented as expected.

Definition 27. *The set of rules λs is given in Figure 1. The λs-*calculus *is the reduction system $(\Lambda s, \rightarrow_{\lambda s})$ where $\rightarrow_{\lambda s}$ is the least compatible reduction on Λs generated by the set of rules λs.*

[16] establishes that the s-calculus (i.e., the reduction system whose rules are those of Figure 1 excluding σ-generation) is strongly normalising, that the λs-calculus is confluent, simulates β-reduction and has the property of preservation of strong normalisation PSN (i.e., if a term terminates in the calculus with de Bruijn indices, then it terminates in the λs-calculus).

In Definition 2 we presented contexts with one hole which we used in some of our statements and proofs about λ-terms. In fact, contexts (with one or more holes, also known as open terms) are an important aspect of λ-calculi and their implementations. Since extending λ-calculi with explicit substitutions is crucial for implementations, these extensions must also cover terms with holes. To extend the λs-calculus with open terms, we need to add to the syntax of λs variables $X, Y, \cdots$ that range over terms. However, just adding these variables to the syntax of λs and keeping the rules of Figure 1 as they are does not guarantee confluence. For example $((\lambda X)Y)\sigma^1 1 \to (X\sigma^1 Y)\sigma^1 1$ and $((\lambda X)Y)\sigma^1 1 \to ((\lambda X)\sigma^1 1)(Y\sigma^1 1)$ but $(X\sigma^1 Y)\sigma^1 1$ and $((\lambda X)\sigma^1 1)(Y\sigma^1 1)$ have no common reduct. In addition to extending the syntax with variables that range over terms, we need to extend the rules to guarantee confluence. The extra rules needed are none other than those of Lemma 11 oriented in the obvious way. This results in the calculus λs_e which is confluent on open terms [17]. Like $\lambda\sigma$ of [1], this calculus does not satisfy PSN.

Definition 28 (The λs_e-calculus)**.** *Terms of the λs_e-calculus are given by:*
$\Lambda s_{op} ::= V \mid \mathbb{N} \mid (\Lambda s_{op}\Lambda s_{op}) \mid (\lambda\Lambda s_{op}) \mid (\Lambda s_{op}\,\sigma^j \Lambda s_{op}) \mid (\varphi^i_k \Lambda s_{op})$ *where j, $i \geq 1$, $k \geq 0$ and* **V** *stands for a set of variables, over which X, Y, ... range.*

The set of rules λs_e is obtained by adding the rules given in Figure 2 to the set λs of Figure 1. The λs_e-calculus *is the reduction system $(\Lambda s_{op}, \to_{\lambda s_e})$ where $\to_{\lambda s_e}$ is the least compatible reduction on Λs_{op} generated by the set of rules λs_e.*

7 Conclusion

In this paper we discussed a number of approaches for substitution and reduction in the λ-calculus, Most of which are influenced by the Curry

σ-σ	$(A\,\sigma^i B)\,\sigma^j\,C$	$\longrightarrow$	$(A\,\sigma^{j+1}\,C)\ \sigma^i\ (B\,\sigma^{j-i+1}\,C)$	if	$i \le j$
σ-φ_1	$(\varphi^i_k\ A)\,\sigma^j\ B$	$\longrightarrow$	$\varphi^{i-1}_k\ A$	if	$k < j < k+i$
σ-φ_2	$(\varphi^i_k\ A)\,\sigma^j\ B$	$\longrightarrow$	$\varphi^i_k(A\,\sigma^{j-i+1}\ B)$	if	$k+i \le j$
φ-σ	$\varphi^i_k(A\,\sigma^j\ B)$	$\longrightarrow$	$(\varphi^i_{k+1}\ A)\,\sigma^j\ (\varphi^i_{k+1-j}\ B)$	if	$j \le k+1$
φ-φ_1	$\varphi^i_k\ (\varphi^j_l\ A)$	$\longrightarrow$	$\varphi^j_l\ (\varphi^i_{k+1-j}\ A)$	if	$l+j \le k$
φ-φ_2	$\varphi^i_k\ (\varphi^j_l\ A)$	$\longrightarrow$	$\varphi^{j+i-1}_l\ A$	if	$l \le k < l+j$

Figure 2: The new rules of the λs_e-calculus

school and none of which would make sense without the lessons learned from the Curry tradition. These notes are based on students questions where they wanted to see the build up of terms and computation stepwise from the bottom up and without hidden steps or working modulo classes. By working through grafting, replacement using ordered variable lists, and then different reduction relations based on this grafting and replacement as well as understanding the role of variable renaming, the students are able to appreciate the move to manipulating terms modulo α-classes and then seem to appreciate de Bruijn indices and the λ-calculus à la de Bruijn with or without explicit substitutions. The β' reduction we saw here is the same reduction relation given in Hindley and Seldin's book [13] and is an accurate representation of how we should manage variables, substitution and alpha conversion.

Acknowledgements

I am very appreciative for Jonathan Seldin for all the wonderful discussions and collaboration and friendship we have had for a quarter of a century.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375416, 1991.

[2] Henk Barendregt. *The lambda calculus - its syntax and semantics.* Studies in logic and the foundations of mathematics 103, North-Holland, 1985.

[3] H. B. Curry. An Analysis of Logical Substitution. *American Journal of Mathematics*, 51: 363–384, 1929.

[4] H. B. Curry. On the Definition of Substitution, Replacement, and Allied Notions in an Abstract Formal System. *Revue Philosophique de Louvain*, 50:251—269, 1952.

[5] H. B. Curry. Apparent variables from the standpoint of combinatory logic. *Annals of Mathematics*, 34:381–404, 1933.

[6] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland Publishing Company, Amsterdam, 1958.

[7] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*, volume 2. North-Holland Publishing Company, Amsterdam and London, 1972.

[8] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Nebert, Halle, 1879.

[9] G. Frege. *Grundlagen der Arithmetik, eine logisch-mathematische Untersuchung ¨uber den Begriff der Zahl.* Breslau, 1884.

[10] D. Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume I. Springer, Berlin, 1934.

[11] J. R. Hindley and J. P. Seldin, editors. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, New York, 1980.

[12] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[13] J. R. Hindley and J. P. Seldin. *Lambda-Calculus and Combinators, an Introduction.* Cambridge University Press, 2008.

[14] J. R. Hindley, B. Lercher, and J. P. Seldin. *Introduction to Combinatory Logic.* Cambridge University Press, 1972. London Math. Soc. Lecture Note No. 7.

[15] F. Kamareddine and J. P. Seldin. *Foundations of Combinatory Logic, by H.B.Curry: Translation into English. Haskell Curry, 1930, University of Goettingen, Grundlagen der kombinatorischen Logik (Foundations of Combinatory logic).* Kings College Publications. ISBN 978-1-84890-202-2. 2016.

[16] F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with Explicit Substitutions. *PLILP 1995.* Springer.

[17] F. Kamareddine and A. Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms *Journal of Functional Programming* volume 7, no. (4), pages 395-420, 1997.

[18] M. Mauny. Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML. PhD thesis, Université Paris VII. Paris, 1985.

[19] B. Russell. On Denoting. *Mind*, 14:479-493, 1905.

[20] J. P. Seldin. *Studies in Illative Combinatory Logic.* PhD thesis, University of Amsterdam, 1968.

[21] A. Urquhart. Russell and Gödel. The Bulletin of Symbolic Logic , DECEMBER 2016, Vol. 22, No. 4, pp. 504-520.

[22] A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, Cambridge, England, 1910–1913. Second edition, 1925–1927.