ON STEPWISE EXPLICIT SUBSTITUTION

International Journal of Foundations of Computer Science 4(3), **197-240, 1993.**

FAIROUZ KAMAREDDINE[*]

Department of Computing Science, University of Glasgow

17 Lilybank Gardens, Glasgow G12 8QQ, Scotland

and

ROB NEDERPELT

Department of Mathematics and Computing Science

Eindhoven University of Technology

P.O.Box 513, 5600 MB Eindhoven, the Netherlands

ABSTRACT

This paper starts by setting the ground for a lambda calculus notation that strongly mirrors the two fundamental operations of term construction, namely abstraction and application. In particular, we single out those parts of a term, called items in the paper, that are added during abstraction and application. This item notation proves to be a powerful device for the representation of basic substitution steps, giving rise to different versions of β-reduction including local and global β-reduction. In other words substitution, thanks to the new notation, can be easily formalised as an object language notion rather than remaining a meta language one. Such formalisation will have advantages with respect to various areas including functional application and the partial unfolding of definitions. Moreover our substitution is, we believe, the most general to date. This is shown by the fact that our framework can accommodate most of the known reduction strategies, which range from local to global. Finally, we show how the calculus of substitution of Abadi et al., can be embedded into our calculus. We show moreover that many of the rules of Abadi et al. are easily derivable in our calculus.

Keywords: lambda calculus, item notation, term trees, substitution, reduction.

[*]Kamareddine is grateful to the Department of Mathematics and Computing Science, Eindhoven University of Technology, for their financial support and hospitality from October 1991 to September 1992 and during December/January and the summer of 1993.

1 Introduction

A system of lambda calculus consists of a set of terms (*lambda terms*) and a set of relations between these terms (*reductions*). Terms are constructed on the basis of two general principles: *abstraction*, by means of which free variables are bound, thus generating some sort of functions; and *application*, being in a sense the opposite operation, formalizing the application of a function to an argument. By observing these two operations, we provide a new notation for lambda terms which will be very influential for many notions of interest in the lambda calculus, such as type theory and logic. In order to avoid the well-known problems caused by variables, we make use of de Bruijn-indices rather than variables (see [5]). Section 2 introduces both de Bruijn's indices and the new notation. Results concerning this new notation and illustrative examples are given throughout.

The use of our framework as the most general vehicle for describing the well known type theories (such as those of the Barendregt's λ-cube in [3]) is discussed in [24]. In fact, [24] shows that our approach enables a unified framework for representing all the systems of the λ-cube in that any of these systems is just a copy of ours but where some parameters are changed.

It seems natural to study the framework further in order to see what it can offer to a very important notion of the λ-calculus: *substitution.* In fact, substitution is the most basic operation of the λ-calculus. Manipulation of λ-terms depends on substitution. The α- and β-axioms are given in terms of substitution. What substitution are we talking about? Substitution in the λ-calculus is usually defined (up to some variation) as follows (see [2]):

Definition 1.1 *(Substitution in the λ-calculus)*
If t, t' are lambda terms and x is a variable, then we define the result of substituting t' for all the free occurrences of x in t as follows:

$$t[x := t'] = \begin{cases} t' & \text{if } t \equiv x \\ y & \text{if } t \equiv y \not\equiv x \\ t_1[x := t']t_2[x := t'] & \text{if } t \equiv t_1 t_2 \\ \lambda_{x:t_2[x:=t']}.t_1 & \text{if } t \equiv \lambda_{x:t_2}.t_1 \\ \lambda_{y:t_2[x:=t']}.(t_1[x := t']) & \text{if } t \equiv \lambda_{y:t_2[x:=t']}.t_1, x \not\equiv y, \\ & (x \notin FV(t_1) \text{ or } y \notin FV(t')) \\ \lambda_{z:t_2[x:=t']}.(t_1[y := z][x := t']) & \text{if } t \equiv \lambda_{y:t_2}.t_1, x \not\equiv y, \\ & (x \in FV(t_1) \text{ and } y \in FV(t')), \\ & \text{and } z \notin FV(t_1 t') \end{cases}$$

Here $FV(t)$ is the set of free variables of t.

So what is happening in $t[x := t']$? We are replacing all free occurrences of x in t by t', but without any disastrous side effects such as binding occurrences of variables which were originally free. Take for example $xx[x := y]$. This will result in yy. $(\lambda_{y:u}.xy)[x := y]$ will result in $\lambda_{z:u}.yz$. So this process of substitution works fine.

It is a metalevel process however. That is, this substitution takes t, x, t' and returns a final result $t[x := t']$. The various stages of moving from the t, x, t' to $t[x := t']$ are lost and nothing matters but the result. This works fine for many applications but fails in areas which are now becoming vital in Computer Science. In functional programming for example, there is an interest in partial evaluation. That is, given $xx[x := y]$, we may not be interested in having yy as the result of $xx[x := y]$ but rather only $yx[x := y]$. In other words, we only substitute one occurrence of x by y and continue the substitution later. This issue of being able to follow substitution and decide how much to do and how much to postpone, has become a major one in functional language implementation (see [25]). However, in order to have this spreading control over substitution and to be able to manipulate those partially substituted terms, we must render the latter from being a metalevel notion to an object level notion. It turns out that our new notation will enable such rendering efficiently and will enable the representation of various forms of substitution: *local, global, implicit* and *explicit*.

Based on this discussion, this paper will introduce substitution which is object level but which can evaluate λ-terms fully obtaining the result of the metalevel substitution. More precisely, in section 3 we introduce the process of *stepwise substitution*, which is meant to refine reduction procedures. Since substitution is the fundamental operation in β-reduction, being in its turn the most important relation in lambda calculus, we are in the heart of the matter. The stepwise substitution is embedded in the calculus, thus giving rise to what is nowadays called *explicit substitution*. It is meant as the final refinement of β-reduction, which has – to our knowledge – not been studied before to this extent.

This substitution relation, being the formalization of a process of stepwise substitution, leads to a natural distinction between a global and a local approach. With **global substitution** we mean the intended replacement of a whole class of bound variables (all bound by the same abstraction-λ) by a given term; for **local substitution** we have only one of these occurrences in view. Both kinds of substitution play a role in mathematical applications, global substitution in the case of function application and local substitution for the "unfolding" of a particular instance of a defined name. We discuss several versions of stepwise substitution and the corresponding reductions. We also extend the usual notion of β-reduction, an extension which is an evident consequence of local substitution. The framework for the description of terms, as explained before, is very adequate for this matter.

Finally in section 4, we interpret the approach of [1] in our framework concluding that ours is more general. In fact, we believe that our account of substitution is the most refined and general one to date.

2 The Calculus

In this section, we start by introducing the reader to the lambda calculus augmented with de Bruijn's indices. We will explain the use of these indices in both the typed and untyped λ-calculus. As the type free λ-calculus can be considered as a special case of the typed λ-calculus, we concentrate on the latter in this paper but will be able to account for the type free λ-calculus very easily as will be mentioned (via ε). We move on to provide a translation of the typed λ-calculus in a novel representation. The novel representation will be generalised to a new notation that will prove to be a powerful vehicle for the representation of substitution, implicitly, explicitly, locally and globally, together with the ability of tracing all stages of substitution, *stepwise substitution*.

2.1 The lambda calculus with de Bruijn's indices

Terms of the untyped lambda calculus are constructed as follows: $t ::= x \mid (\lambda_x.t) \mid (tt)$. Parentheses are omitted if no confusion can arise. Terms of the typed lambda calculus are similar except that the type information is contained in the abstraction. That is, instead of $\lambda_x.t$ we restrict x to have some type say t_1 by writing $\lambda_{x:t_1}.t$. Of course special attention has to be paid in order to construct well-typed terms. Moreover, in the typed calculus, we can abstract over types as well as over terms. For example, $\Lambda_A.\lambda_{x:A}.x$ is the polymorphic identity function for every type A.

The basic axiom of the (typed or untyped) lambda calculus, is β-conversion which is as follows:[a] $(\lambda_x.t_1)t_2 = t_1[x := t_2]$,[b] where substitution has been defined in a way which deals with the problem of variable clashes (see Definition 1.1). For example, $(\lambda_x.\lambda_y.xy)y = (\lambda_y.xy)[x := y] = \lambda_z.xy[y := z][x := y] = \lambda_z.xz[x := y] = \lambda_z.yz$. This process of renaming variables such as changing $\lambda_y.xy$ to $\lambda_z.xz$ can be avoided by the use of de Bruijn's indices. In fact, de Bruijn noted that due to the fact that terms as $\lambda_x.x$ and $\lambda_y.y$ are the "same", we can find a λ-notation modulo α-conversion, where the axiom (α) is: $\lambda_x.t = \lambda_y.t[x := y]$ for y not free in t. That is, following de Bruijn, we can abandon variables and use indices instead. Examples 2.1, 2.2 below show how lambda terms can be denoted using de Bruijn's indices and example 2.3 illustrates how β-conversion works using such indices.

Example 2.1 Consider (in "classical" notation) the lambda term $(\lambda_x.x)$. In this term, the x following λ_x is a variable bound by this λ. In de Bruijn's notation, $\lambda_x.x$ and all its α-equivalent expressions can be written as $\lambda.1$. The bond between the bound variable x and the operator λ is expressed by the number 1; the position of this number in the term is that of the bound variable x, and the value of the number ("*one*") tells us how many lambda's we have to count, going leftwards in

[a]For the sake of clarity, we ignore in this section abstraction over types.

[b]In the case of the typed calculus, the principle is: $(\lambda_{x:t}.t_1)t_2 = t_1[x := t_2]$ where t and t_2 are related.

the term, starting from the mentioned position, to find the binding place (in this case: the *first* λ to the left is the binding place).

Moreover, de Bruijn's notation can be used for the typed λ-calculus. We illustrate here how the two terms $(\lambda_{x:y}.x)u$ and $\Lambda_A.\lambda_{x:A}.x$ can be represented using de Bruijn's indices.

Example 2.2 The term $(\lambda_{x:y}.x)u$ is written as $(\lambda 2.1)1$ under the assumption that y comes before u in the free variable list (see below). As in Example 2.1, the final x in $\lambda_{x:y}.x$ is represented by the final index 1 in $\lambda 2.1$ since the binding λ is the first to the left. The free variables u and y in the typed lambda term are translated into the number 1 (occurring after the term in parentheses), and the number 2: they refer to "invisible" lambda's that are not present in the term, but may be thought of to *preceed* the term, binding the free variables in some arbitrary, but fixed order (these invisible lambda's form a **free variable list**).

Some type theories insist on distinguishing types and terms and so use λ to abstract over terms and Λ over types. Hence the typed term $\Lambda_A.\lambda_{x:A}.x$ can be written as $\Lambda.\lambda 1.1$ where the 1 adjacent to λ, says that Λ is the binding operator and the final 1 replaces the variable bound by λ.

The described way of omitting binding variables, and rendering bound and free variables by means of so-called **reference numbers**, is precisely how de Bruijn's notation works. Next we see how β-reduction works in this notation.

Example 2.3 In ordinary lambda calculus, the term $(\lambda_{x:z}.(xy))u$ β-reduces to uy, i.e. the result of substituting "argument" u for x in xy. In de Bruijn's notation this becomes, — under the assumption that the free variable list is $\lambda_y, \lambda_z, \lambda_u$: $(\lambda 2.\ 14)1$ reduces to 13. Here the contents of the subterm 14 changes: 4 becomes 3. This is due to the fact that a λ-item, viz. $(\lambda 2)$, disappeared (together with the argument 1). The first variable 1 did not change; note, however, that the λ binding this variable has changed "after" the reduction; it is the last λ in the free variable list ("λ_u") and no longer the λ inside the original term ("λ_x"). The reference changed, but the number stayed (by chance) the same.[c]

We have in examples 2.1, 2.2 and 2.3 introduced de Bruijn's indices and how they work for β-reduction. In what follows we shall introduce a new notation which uses de Bruijn's indices but assumes a layered representation of terms and which groups term constituents (so-called "items") together in a novel way. This new notation will prove powerful for many applications, of which we study substitutions in detail in this paper.

[c]In more complicated examples, there are more cases in which variables must be "updated". This updating of variables is an unpleasant consequence of the use of de Bruijn-indices. It is the price we have to pay for the banishing of actual variable names (taking reference numbers instead). We will however provide an update function which does the work for us.

As the new notation might prove unreadable at first, and as it is very general so as to accommodate not only the λ-calculus described above but all of the λ-systems of the λ-cube of Barendregt (see [24]), we would rather introduce this notation in steps. We start by introducing a less general representation LT in which we translate the above λ-calculus and then we give the generalised new notation which is based on LT but also includes de Bruijn's indices. We call this calculus BLT, and we give its abstract definition, and the various notions related to its terms.

2.2 Translating λ-terms into layered structures

2.2.1 The classical calculus T_λ

Usually, the typed λ-calculus is considered in this form:

Definition 2.4 *(T_λ)*
We consider T_λ to be the set of the following terms:
$t ::= x \mid (\lambda_{x:t}.t) \mid (tt)$
We drop parentheses when no confusion occurs.

These λ-terms are then drawn using binary trees which are defined below, $TREE$ being the collection of these trees and $tree$ being the association from a λ-term to a binary tree.

Definition 2.5 *($TREE$)*
We define the domain $TREE$ to be the domain of binary trees which have for leaves $x, y, \ldots$ and for nodes $\delta, \lambda_x, \lambda_y, \ldots$. We let $\omega, \omega_1, \ldots$ range over these nodes, which we also call operators.

Let us associate with each term of T_λ, its binary tree in $TREE$ as follows:

Definition 2.6 *($tree$)*
If t is a term of the typed lambda calculus T_λ, then $tree(t)$ is defined recursively as follows:

$$\begin{array}{lcl} tree(x) & = & x \\ tree(t_1 t_2) & = & \delta(tree(t_1), tree(t_2)) \\ tree(\lambda_{x:t_1}.t_2) & = & \lambda_x(tree(t_2), tree(t_1)) \end{array}$$

Example 2.7 *$tree((\lambda_{x:z}.xy)u)$ is $\delta(\lambda_x(\delta(x,y),z),u)$ and its graphical representation is to be found in Figure 1.*

Of course these trees are all binary trees:

Lemma 2.8 *For every term t in T_λ, tree(t) is a binary tree.*
Proof: *Left to the reader.* □

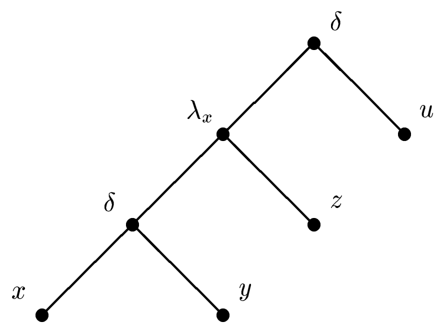


Figure 1: binary tree of $(\lambda_{x:z}.xy)u$

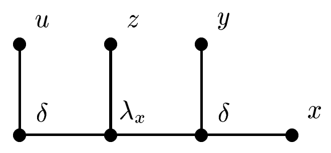


Figure 2: term tree of $(\lambda_{x:z}.xy)u$

Now, instead of drawing trees as above, we will rotate them anticlockwise by 135 degree hence obtaining for Example 2.7, the picture given in Figure 2.

The proposed way of drawing trees will turn out to have essential advantages in developing a term, theoretically as well as in practical applications of typed lambda calculi.[d] We call such trees **term trees**. Note that these trees will help to visualize the structure of the terms. They are not however formal components of the calculus.

2.2.2 The layered terms LT

Now the concepts *tree* and *term tree* of a term t in T_λ are quite obvious. However, we like to consider the set of layered terms by itself.

Definition 2.9 *(LT)*

Let us define LT to be the set of layered terms as follows:

- *Variables,* $x, y, \ldots$ *are in LT.*
- *If* t_1, t_2 *are in LT and* ω *is an operator which is either* δ *or any of* $\lambda_x, \lambda_y, \ldots$, *then* $(t_1 \omega t_2)$ *is in LT.*

[d]This observation is due to de Bruijn, see [4] or [6].

Note that we use the same variables for both T_λ and LT and that we use $t, t_1, \ldots$ to denote both terms in T_λ and layered terms in LT. Note that in discussing T_λ and LT, we will not make a distinction between object and meta-variables. We will do so however in BLT.

Example 2.10 The term rendered in Figure 2 has the following representation in LT: $(u\delta(z\lambda_x(y\delta x)))$.

What we are looking for further is a way of writing the term t which will be more advantageous and efficient. In fact, note that in $t \equiv (\lambda_{x:z}.xy)u$, the term $\lambda_{x:z}.xy$ is applied to u. This application provokes β-conversion and hence will result in reducing the term. In fact, applying an abstraction (as in $\lambda_{x:z}.xy$) to an element (such as u) is important in the λ-calculus. This however, is not obvious in the way we have written terms as in $(\lambda_{x:z}.xy)u$. If we mimic $tree(t)$ in a different manner and write t as $(u\delta)(z\lambda_x)(y\delta)x$, then we can give a special name to the pair $(u\delta)(z\lambda_x)$. We will call them $\delta\lambda$-items and they will be the pairs which enable us to carry out β-reduction.

Notation 2.11 *(Item Notation)*
We shall place parentheses in LT in an unorthodox manner: we write $(t_1\omega)t_2$ instead of $(t_1\omega t_2)$. The reason for using this format is, that both abstraction and application can be seen as the process of fixing a certain part (an "**item**") to a term:

- the abstraction $\lambda_{x:t'}.t$ is obtained by prefixing the abstraction-item $\lambda_{x:t'}$ to the term t. Hence, $(t'\lambda_x t)$ is obtained by prefixing $t'\lambda_x$ to t.
- the application tt' is obtained by postfixing the argument-item t' to the term t. Now $(t'\delta t)$ is obtained by prefixing $t'\delta$ to t.

In item-notation we write in these cases $(t'\lambda_x)t$ and $(t'\delta)t$, respectively. Here both $(t'\lambda_x)$ and $(t'\delta)$ are *prefixed* to the term t.[e]

Definition 2.12 *(Items)*
If t is a layered term in item-notation and ω is an operator, then $(t\omega)$ is an item. We use $s, s_1, s_i, \ldots$ as meta-variables for items.

Notation 2.13 *(parentheses)* Note the intended parsing convention:
In $(s_1 \ldots s_n x\omega)s'_1 \ldots s'_m y$, the operator ω combines the *full* term $s_1 \ldots s_n x$ with the *full* term $s'_1 \ldots s'_m y$.

Example 2.14 The term $(x\omega_1(y\omega_2 z))$ becomes in item-notation: $(x\omega_1)(y\omega_2)z$. Analogously, the term $((x\omega_2 y)\omega_1 z)$ becomes $((x\omega_2)y\omega_1)z$.

[e]In the *Automath*-tradition (see [6]), an abstraction-item $\lambda_{x:t'}$ (or $(t'\lambda_x)$ in our new notation) is called and *abstractor* and denoted as $[x : t']$. An argument-item t' (or $(t'\delta)$ in our notation) is called an *applicator* and denoted either as $\{t'\}$ or as $< t' >$.

Lemma 2.15 *Every layered term has the form $(t_1\omega_1)\dots(t_n\omega_n)x$ for $t_1,\dots,t_n$ layered terms, $\omega_1,\dots,\omega_n$ operators, $n \geq 0$ and x a variable. In other words, every layered term is either a variable or has the form $s_1, s_2, \dots, s_n x$, for $s_1, s_2, \dots, s_n$ items and x a variable.*

Proof: *Easy.* □

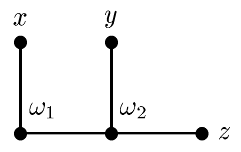
Definition 2.16 *For any tree t in $TREE$, we define $lin(t)$ in LT as follows:*

$$lin(t) = \begin{cases} t & \text{if } t \text{ is a leaf} \\ (lin(t_1)\omega_1)lin(t_2) & \text{if } t \equiv \omega_1(t_2, t_1) \end{cases}$$

Example 2.17 *If t is the tree of Example 2.7, then*

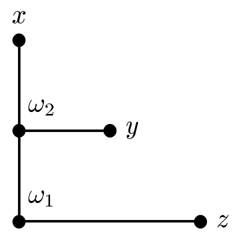
$$\begin{aligned} lin(t) &= (u\delta)lin(\lambda_x(\delta(x,y),z)) \\ &= (u\delta)(z\lambda_x)lin(\delta(x,y)) \\ &= (u\delta)(z\lambda_x)(y\delta)x \end{aligned}$$

Now let us see the relation between $tree(t)$ and $lin(t)$, for every term t. Recall that in $TREE$, we deviate from the normal way to depict a tree; for example: we position the root of the tree in the lower left hand corner. We have chosen this manner of depicting a tree in order to maintain a close resemblance with the layered terms. This has also advantages in the sections to come. The item-notation suggests a partitioning of the term tree in vertical layers (see Figure 3). For $(x\omega_1)(y\omega_2)z$, these layers are: the parts of the tree corresponding with $(x\omega_1)$, $(y\omega_2)$ and z (connected in the tree with two edges). For $((x\omega_2)y\omega_1)z$ these layers are: the part of the tree corresponding with $((x\omega_2)y\omega_1)$ and the one corresponding with z.



$(x\omega_1(y\omega_2 z))$

$(x\omega_1)(y\omega_2)z$



$((x\omega_2 y)\omega_1 z)$

$((x\omega_2)y\omega_1)z$

Figure 3: Term trees, with normal layered notation and item-notation

Lemma 2.18 *$lin : TREE \longrightarrow LT$ is well defined.*

Proof: *Obvious.* □

Note that in the rest of this paper we will write terms of LT in item-notation.

2.2.3 Translating T_λ in LT

Now we are ready to translate terms of T_λ into layered terms of LT (in item-notation) as follows:

Definition 2.19 *If t is in T_λ then $l(t) = lin(tree(t))$.*

Lemma 2.20 *For any t in T_λ, $l(t)$ is well defined and $l : T_\lambda \longrightarrow LT$ is bijective.*

Proof: *We shall only prove the surjectivity of l.*
For this, we prove by induction that for any t in LT, $(\exists t'$ in $T_\lambda)[l(t') = t]$.

- *If t is x then $t' = x$.*
- *If $t = (t_1\delta)t_2$ where $(\exists t'_1, t'_2 \in T_\lambda)[l(t'_1) = t_1 \wedge l(t'_2) = t_2]$ then let $t' = (t'_2 t'_1)$. Now,*
$$\begin{array}{rcl} l(t') & = & lin(tree(t'_2 t'_1)) \\ & = & lin(\delta(tree(t'_2), tree(t'_1))) \\ & = & (lin(tree(t'_1))\delta) lin(tree(t'_2)) \\ & = & (l(t'_1)\delta) l(t'_2) = (t_1\delta)t_2 = t \end{array}$$
- *If $t = (t_1\lambda_x)t_2$ where $(\exists t'_1, t'_2 \in T_\lambda)[l(t'_1) = t_1 \wedge l(t'_2) = t_2]$ then let $t' = \lambda_{x:t'_1}.t'_2$. Now,*
$$\begin{array}{rcl} l(t') & = & lin(tree(\lambda_{x:t'_1}.t'_2)) \\ & = & lin(\lambda_x(tree(t'_2), tree(t'_1))) \\ & = & (lin(tree(t'_1))\lambda_x) lin(tree(t'_2)) \\ & = & (l(t'_1)\lambda_x) l(t'_2) = (t_1\lambda_x)t_2 = t \end{array}$$

□

So far, we have translated all terms from the usual λ-calculus into layered terms. This translation is moreover bijective, so we can take any layered term into a term of the usual λ-calculus. The following are examples:

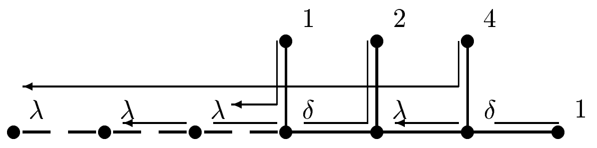
Example 2.21

$$\begin{array}{lcl} l((\lambda_{x:y}.x)u) & = & (u\delta)(y\lambda_x)x \\ l(u(\lambda_{x:y}.x)) & = & ((y\lambda_x)x\delta)u \\ l((\lambda_{y:z}.\lambda_{x:z}.y)u) & = & (u\delta)(z\lambda_y)(z\lambda_x)y \end{array}$$

2.3 A notation based on layered structures and de Bruijn's indices

Now that we have a bijective translation from T_λ to LT, let us see how we can get rid of the variables and replace them by de Bruijn's indices. This would mean of course that we no longer would need each λ to carry the index x, y or so on with it, but rather, the number would point to which λ binds which occurrence. The best way here is to start with an example. We take the layered term t with its graph in term tree as in Figure 2. We need to remove x, y, z, u and to replace them by numbers. For this, as we see that u, z, y are free variables, we need a free variable list. We take the convention (arbitrarily)[f] that y comes before z which in turn comes before u in the free variable list. This list is represented by three extra λ's: λ_y, λ_z and λ_u (in this order), intended to "bind" the free variables y, z and u. We append three extra nodes and dashed lines to our term tree to show this.

Now for each variable, we draw thin lines ending in arrows, pointing at the λ binding the variable. These lines follow the path which leads from the variable to the root following the *left side* of the branches of the tree. Only λ's count, the δ's do not. For example, we draw the thin line going from x following the path which leads from x to the root, until we reach λ_x, the λ binding x. We end the arrow there and as we have only passed one λ, the x should be replaced by 1. This is the only x we have in the tree, so we replace λ_x by λ. For y, in drawing the thin line going from y following the path which leads from y to the root, keeping to the left side of the branches until we reach λ_y, we see that we pass four λs. Hence, the y should be replaced by 4. Now replacing u and z will be left as exercises. Figure 4 is now self explanatory.



$(\lambda_{x:z}.xy)u$

$(u\delta)(z\lambda_x)(y\delta)x$

$(1\delta)(2\lambda)(4\delta)1$

Figure 4: Term tree with de Bruijn's indices

2.3.1 Layered terms with de Bruijn's indices, BLT

Our new notation will be exactly that of layered terms, but increased with de Bruijn's indices. Let us start by giving the definition of our layered terms with de Bruijn's indices.

Definition 2.22 *(BLT)*
Let us define BLT to be the set of layered terms using de Bruijn's indices as follows:

[f] Note that we lose bijectivity here, since a different order in the variable list gives a different representation for the same term.

- *Every element in Ξ is in BLT. Here, we take Ξ to be the set of* **variables***: $\Xi = \{\varepsilon, 1, 2, \ldots\}$ and use $x, x_1, y, \ldots$ to denote variables.*

- *If t, t' are in BLT and ω is an operator which is either δ or λ, then $(t\omega t')$ is in BLT.*

Remark 2.23 ε is a special variable that denotes the "empty term". It can be used for rendering ordinary (untyped) lambda calculus; take all types to be ε. Another use is as a "final type", like $\Box$ in Barendregt's cube or in Pure Type Systems (PTS's). Note moreover that, as ε can be used to render ordinary (untyped) lambda calculus, we were fine in concentrating on the typed calculus in this paper.

Now we take the same notational conventions as those for LT given in Notations 2.11 and 2.13, and we define items similarly. Simple examples of terms are: ε, 3, $(2\delta)(\varepsilon\lambda)1$.[g] Moreover, in $(t\omega)$, we may drop t in case $t \equiv \varepsilon$. Hence the last mentioned example can also be written as $(2\delta)(\lambda)1$. Here is another example:

Example 2.24 Consider the typed lambda term $(\lambda_{x:z}.x)u$. In item-notation with name-carrying variables this term becomes $(u\delta)(z\lambda_x)x$. In item-notation with de Bruijn-indices, it is denoted as $(1\delta)(2\lambda)1$.

The typed lambda term $u(\lambda_{x:z}.x)$ is denoted as $((z\lambda_x)x\delta)u$ in our name-carrying item-notation and as $((2\lambda)1\delta)1$ in item-notation with de Bruijn-indices. The free variable list, in the name-carrying version, is λ_z, λ_u, in both examples.

The term trees of these lambda terms are given in figure 5. In each of the two pictures, the references of the three variables in the term have been indicated: thin lines, ending in arrows, point at the λ's binding the variables in question.

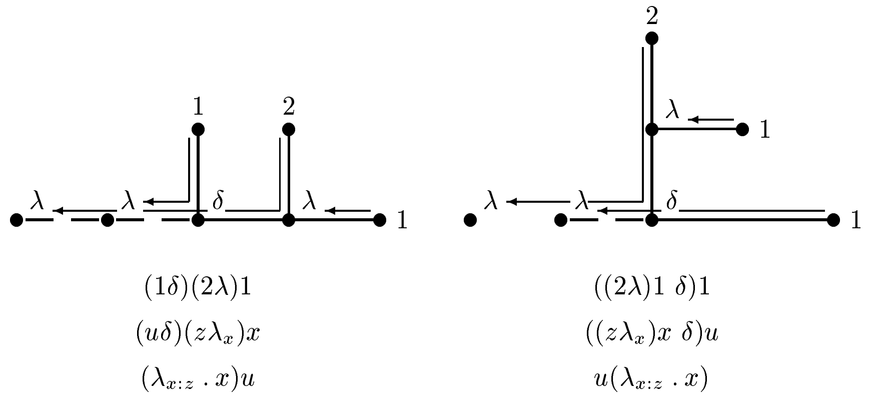


Figure 5: Term trees with explicit free variable lists and reference numbers

[g] There can be different (finitely many) λ's and/or δ's in terms. In the present paper we shall consider only one of each, denoted λ and δ, respectively. Different λ's can be used, for example, in second-order theories: write $\lambda = \lambda_2$ and $\Lambda = \lambda_1$.

Now it is obvious that Lemma 2.15 holds also for BLT, where the terms are now terms of BLT. More precisely:

Lemma 2.25 *Every layered term has the form $(t_1\omega_1)\dots(t_n\omega_n)x$ for $t_1,\dots,t_n$ layered terms, $\omega_1,\dots,\omega_n$ operators, $n \geq 0$ and x a variable.*

Proof: *Easy.* □

2.4 α- and β-reduction in T_λ, LT and BLT

The fundamental axioms of the λ-calculus are (α) and (β). Other axioms such as (η) (which is needed together with another axiom to derive extensionality) are optional. Therefore, we shall only concentrate on (α) and (β).

2.4.1 Reduction in T_λ

In T_λ, the axioms (α) and (β) are as follows:

$$\begin{array}{ll} (\alpha) & \lambda_{x:t}.t' \to_\alpha \lambda_{y:t}.t'[x := y] \text{ where } y \notin FV(t') \\ (\beta) & (\lambda_{x:t}.t')t'' \to_\beta t'[x := t''] \end{array}$$

We say that $t \to_\alpha t'$ (respectively $t \to_\beta t'$) just in case (α) (respectively (β)) takes t to t'. We call the reflexive transitive closure of $\to_\alpha$ (respectively $\to_\beta$), $\twoheadrightarrow_\alpha$ (respectively $\twoheadrightarrow_\beta$).

2.4.2 Reduction in LT

In LT, these axioms are the same but written in item notation as follows:

$$\begin{array}{ll} (\alpha') & (t\lambda_x)t' \to_{\alpha'} (t\lambda_y)t'[x := y]' \text{ where } y \notin FV(t') \\ (\beta') & (t''\delta)(t\lambda_x)t' \to_{\beta'} t'[x := t'']' \end{array}$$

Of course $t[x := t']'$ is the substitution in LT of t' for all free occurrences of x in t. Free and bound variables/occurrences in LT are easy to define and we will of course obtain the following lemma:

Lemma 2.26 *For any $t \in T_\lambda$, $FV(t) = FV(l(t))$.*

Proof: *Obvious.* □

The notion of substitution in LT is also easy to define. This is done as follows:

Definition 2.27 *(Substitution in LT)*

If t, t' are layered terms and x is a variable we define the result of substituting t' for all the free occurrences of x in t as follows:

$$t[x := t']' = \begin{cases} t' & \text{if } t \equiv x \\ y & \text{if } t \equiv y \not\equiv x \\ (t_1[x := t']'\delta)t_2[x := t']' & \text{if } t \equiv (t_1\delta)t_2 \\ (t_2[x := t']'\lambda_x)t_1 & \text{if } t \equiv (t_2\lambda_x)t_1 \\ (t_2[x := t']'\lambda_y)(t_1[x := t']') & \text{if } t \equiv (t_2\lambda_y)t_1, x \not\equiv y, \\ & (x \not\in FV(t_1) \text{ or } y \notin FV(t')) \\ (t_2[x := t']'\lambda_z)(t_1[y := z]'[x := t']') & \text{if } t \equiv (t_2\lambda_y)t_1, x \not\equiv y, \\ & (x \in FV(t_1) \text{ and } y \in FV(t')), \\ & \text{and } z \notin FV(t_1t') \end{cases}$$

Now the following lemma holds:

Lemma 2.28

- *If t, t' are in T_λ then $l(t[x := t']) = l(t)[x := l(t')]'$*
- *If t, t' are in LT then $l^{-1}(t[x := t']') = l^{-1}(t)[x := l^{-1}(t')]$*

Proof: *Left to the reader.* □

In LT, we define $\rightarrow_{\alpha'}$ (respectively $\rightarrow_{\beta'}, \twoheadrightarrow_{\alpha'}, \twoheadrightarrow_{\beta'}$) similarly to that of T_λ, but using (α') and (β') instead. Now the following lemma holds:

Lemma 2.29 *For $t, t' \in T_\lambda$, we have: $t \rightarrow_\alpha t'$ (respectively $\rightarrow_\beta, \twoheadrightarrow_\alpha, \twoheadrightarrow_\beta$) iff $l(t) \rightarrow_{\alpha'} l(t')$ (respectively $\rightarrow_{\beta'}, \twoheadrightarrow_{\alpha'}, \twoheadrightarrow_{\beta'}$) .*

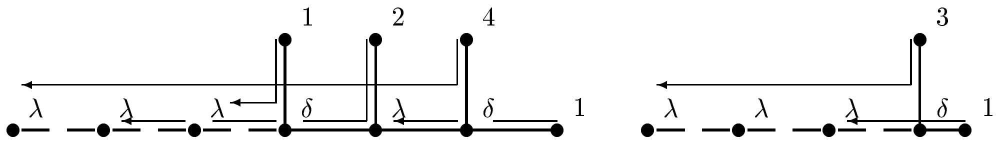
Proof: *Left to the reader.* □

Of course now, all the theorems of T_λ, such as the Church-Rosser theorem, the fixed point theorems, the undefinability results, and so on, hold for LT. Let us see now what would happen to BLT.

2.4.3 Reduction in BLT

α-reduction is not needed for BLT, precisely because we no longer have variables (de Bruijn's indices got rid of them). So now, we no longer have different ways of writing the same term as we have taken the equivalent classes so that $\lambda_{x:t}.x, \lambda_{y:t}.y, \ldots$ all are represented by $(t\lambda)1$. For β-reduction, this is a bit more complicated. Let us start by an example:

Example 2.30 Now for β-reduction, the term $(\lambda_{x:z}.(xy))u$ of T_λ, β-reduces to uy. In LT, this becomes: $(u\delta)(z\lambda_x)(y\delta)x$ reduces to $(y\delta)u$ (see figure 6). Note that the presence of a so-called δ-λ-segment (i.e. a δ-item immediately followed by a λ-item), in this example: $(u\delta)(z\lambda_x)$ is the signal for a possible β-reduction. Using de Bruijn's indices, this becomes: $(1\delta)(2\lambda)(4\delta)1$ reduces to $(3\delta)1$.



$(\lambda_{x:z}.xy)u$ $\qquad$ uy

$(u\delta)(z\lambda_x)(y\delta)x$ $\qquad$ $(y\delta)u$

$(1\delta)(2\lambda)(4\delta)1$ $\qquad$ $(3\delta)1$

Figure 6: β-reduction in our notation

We can see from the above example that the convention of writing the argument *before* the function has a practical advantage: the δ-item and the λ-item involved in a β-reduction occur *adjacently* in the term; they are not separated by the "body" of the term, that can be extremely long! It is well-known that such a δ-λ-segment can code a definition occurring in some mathematical text; in such a case it is very desirable for legibility that the coded definiendum and definiens occur very close to each other in the term.

Before we define β-conversion in BLT, we need to define substitution and free occurrences of variables. For this, and for the next section on explicit substitution, we need to give a number of definitions regarding certain substrings of terms. This is done next.

2.5 The formal machinery of *BLT*

2.5.1 Items and Segments

Recall that we defined items in Definition 2.12. Items together with the following notion are basic for our machinery.

Definition 2.31 *(segments)*
A concatenation of zero or more items is a **segment**.[h]

Remark 2.32 Note that if $\overline{s}$ is a segment of BLT, then either $\overline{s} = \emptyset$ or $\overline{s} = s_1 s_2 \ldots s_n$ where $1 \leq n$ and s_i is an item for $1 \leq i \leq n$.

Recall that we use $s, s_1, \ldots$ as meta-variables for items. Moreover, we use $\overline{s}, \overline{s}_1, \overline{s}_i, \ldots$ as meta-variables for segments. Now we are ready to give an abstract formulation (definition 2.33) of all the notions that have been defined so far. We let $\mathcal{V}$ stand for the set of variables, $\mathcal{O}$ for operators, $\mathcal{T}$ for terms, $\mathcal{I}$ for items and $\mathcal{S}$ for segments.

Definition 2.33 *(variables, operators, terms, items, segments)*

[h]In [8] an item is called a *wagon* and a segment is called a *train*.

$\mathcal{V} = \{\varepsilon, 1, 2, \ldots\}$
$\mathcal{O} = \{\delta, \lambda, \ldots\}$
$\mathcal{T} = \mathcal{V} \mid \mathcal{I}\ \mathcal{T}$
$\mathcal{I} = (\mathcal{T}\ \mathcal{O})$
$\mathcal{S} = \emptyset \mid \mathcal{I}\ \mathcal{S}$

For the next section, when we introduce substitution, we will assume the same sets $\mathcal{V}$, $\mathcal{T}$, $\mathcal{I}$, and $\mathcal{S}$. $\mathcal{O}$ however will be increased by two more operators σ and φ which will deal with substitution. In other applications, we use more than one δ and one λ. [24] is an example where different λ's and δ's are introduced and needed. It was mentioned earlier moreover that we may take two λ's, λ_1 and λ_2 where in second order theories, the first represents Λ and the second represents λ.

We define a number of concepts connected with terms, items and segments. These will be used in the rest of the paper.

Definition 2.34 *(main items, main segments, empty segments, ω-items, $\omega_1 \ldots \omega_n$-segments)*

- *Each term t is the concatenation of zero or more items and a variable: $t \equiv s_1 \ldots s_n x$ (see Lemma 2.25). These items $s_1 \ldots s_n$ are called the* **main items** *of t.*
- *Analogously, a segment $\overline{s}$ is a concatenation of zero or more items: $\overline{s} \equiv s_1 \ldots s_n$ (see Remark 2.32); again, these items $s_1 \ldots s_n$ (if any) are called the* **main items***, this time of $\overline{s}$.*
- *A concatenation of adjacent main items (in t or $\overline{s}$), $s_m \ldots s_{m+k}$, is called a* **main segment** *(in t or $\overline{s}$).*
- *An item $(t\ \omega)$ is called an* **ω-item***. Hence, we may speak about* **λ-items***,* **δ-items** *(and later on about* **σ-items** *and* **φ-items***).*
- *A segment $\overline{s}$ such that $\overline{s} \equiv \emptyset$ is called an* **empty segment***; other segments are* **non-empty***. A* **context** *is a segment consisting of only λ-items.*
- *If a segment consists of a concatenation of an ω_1-item up to an ω_n-item, this segment may be referred to as being an* **ω_1-...-ω_n-segment***. An important case is that of a* **δ-λ-segment***, being a δ-item immediately followed by a λ-item.*

All these definitions are easy and obvious. The reader can now think more of the structure of terms and see some enligthening but trivial results such as: every term is of the form $\overline{s}x$ where $\overline{s}$ is a segment and x is a variable. The following is an example of some of these notions.

Example 2.35 Let the term t be defined as $(\varepsilon\lambda)((1\delta)(\varepsilon\lambda)1\delta)(2\lambda)1$ and let the segment $\overline{s}$ be $(\varepsilon\lambda)((1\delta)(\varepsilon\lambda)1\delta)(2\lambda)$. Then the main items of both t and $\overline{s}$ are $(\varepsilon\lambda)$, $((1\delta)(\varepsilon\lambda)1\delta)$ and (2λ), being a λ-item, a δ-item, and another λ-item. Moreover, $((1\delta)(\varepsilon\lambda)1\delta)(2\lambda)$ is an example of a main segment of both t and $\overline{s}$, which is not a context (i.e. not a purely λ-segment), but a δ-λ-segment. Also, $\overline{s}$ is a λ-δ-λ-segment, which is a main segment of t.

Definition 2.36 *(body, end variable, end operator)*

- *Let $t \equiv \overline{s}x$ be a term. Then we call $\overline{s}$ the* **body** *of t, denoted* $\mathtt{body}(t)$*, and x the* **end variable** *of t, or* $\mathtt{endvar}(t)$*. It follows that $t \equiv \mathtt{body}(t)\ \mathtt{endvar}(t)$.*
- *Let $s \equiv (t\omega)$ be an item. Then we call t the* **body** *of s, denoted* $\mathtt{body}(s)$*, and ω the* **end operator** *of s, or* $\mathtt{endop}(s)$*. Hence, $s \equiv (\mathtt{body}(s)\ \mathtt{endop}(s))$.*

Note that we use the word ‘body’ in two meanings: the body of a term is a segment, and the body of an item is a term.

Items and segments play an important role in many applications. As explained before, a λ-item is the part joined to a term in an abstraction, and a δ-item is the part joined in an application. In using typed lambda calculi for e.g. mathematical reasoning, λ-items may be used for assumptions or variable introductions and a δ-λ-segment may express a definition or a theorem (See [22], [23] and [24]).

2.5.2 Bound and free variables

After defining our items and segments and the various notions related to them, we need to discuss the notion of free and bound variables. In LT, these notions are similarly definable to that of T_λ. In BLT, variables are indices and λ's do not have any reference to the variables they bind. Rather it is the number which is indicative of the binding λ. Calculating bound and free variables in BLT will turn out to be an easy, mechanisable and efficient affair.

We start by defining $sieveseg_\omega$ which will gather all the main ω-items in a term, in the order in which they occur in the term.

Definition 2.37 *(sieveseg)*
Let $\overline{s}$ be a segment, or let t be a term with body $\overline{s}$.
Then $\mathtt{sieveseg}_\omega(\overline{s}) = \mathtt{sieveseg}_\omega(t) =$ *the segment consisting of all main ω-items of $\overline{s}$, concatenated in the same order in which they appear in $\overline{s}$.*

Example 2.38 In the term $t = (\varepsilon\lambda)((1\delta)(\varepsilon\lambda)1\delta)(2\lambda)1$,
$\mathtt{sieveseg}_\lambda(t) \equiv (\varepsilon\lambda)(2\lambda)$ and $\mathtt{sieveseg}_\delta(t) \equiv ((1\delta)(\varepsilon\lambda)1\delta)$.

Lemma 2.39 *Let $\overline{s}$ be a segment, then all variables in $sieveseg_\lambda(\overline{s})$ will point to the same λ's that they pointed at in $\overline{s}$.*
Proof: *Easy.* □

Let us now define the restriction of a term to a particular variable occurrence. We will not give the formal definition here, but it can be found in [23]. We shall only explain what it is and how we get it without the formal machinery.

Definition 2.40 *(term restriction)*
Let t be a term in BLT and x° be a particular occurrence of a variable in t. We define $t \upharpoonright x^\circ$ the term restriction of t to x° to be the original term, from which we remove all the things to the right of x° and then we remove all extra parentheses.

Example 2.41 Let t be the term $(\varepsilon\lambda_x)((x\lambda_u)((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y\lambda_v)u$. $t \upharpoonright x^\circ$, the restriction of t to x° is $(\varepsilon\lambda_x)(x\lambda_u)(u\delta)(x\lambda_t)x^\circ$.

Definition 2.42 *(bound and free variables, type, open and closed terms)*

- *Let x° be a variable occurrence in t such that $x \not\equiv \varepsilon$ and $\texttt{sieveseg}_\lambda(t \upharpoonright x^\circ) \equiv s_m \ldots s_1$ (for convenience numbered downwards). Then x° is* **bound** *in t if $x \leq m$; the* **binding item** *of x° in t is s_x and the λ that* **binds** *x° in t is* $\texttt{endop}$*(s_x). The* **type** *of x° in t is* $\texttt{body}$*(s_x). Furthermore, x° is* **free** *in t if $x > m$.*
- *The variable ε is neither bound nor free in a term.*
- *Term t is* **closed** *when all occurrences of variables in t different from ε are bound in t. Otherwise t is* **open** *or* **has free variables**.

Example 2.43 $t \equiv (\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2^\circ\delta)(1\lambda)1 \Rightarrow t \upharpoonright 2^\circ \equiv (\lambda)(1\lambda)(2\delta)(\lambda)(3\lambda)2^\circ$. So $\texttt{sieveseg}_\lambda(t \upharpoonright 2^\circ) \equiv s_4s_3s_2s_1 \equiv (\lambda)(1\lambda)(\lambda)(3\lambda)$. Hence, 2° is bound in t since $2 \leq 4$. Moreover, the type of 2° in t is $\texttt{body}(s_2) \equiv \varepsilon$. There are no free variables in t, hence t is closed.

We see from this example that one can easily account for free and bound variables, just by calculation. Note that restriction does not affect whether a variable occurrence is free or bound.

Now, we have all the machinery to define substitution in our system, not only as the known substitution described in Definition 1.1 but in all its forms, local, global, explicit and implicit. From substitution, we can define β-reduction, again local and global.

3 Reduction

Recall that in Def 1.1, substitution $t[x := t']$ was defined by certain metarules. Such metalevel substitution however, is unsatisfactory for many reasons, some of which we mention in 3.1. In the rest of the section, we make substitution a part of the formal language for our terms, providing thereby a means by which we avoid the disadvantages mentioned in 3.1.

3.1 Global vs. local β-reduction

The traditional β-reduction causes a substitution for *all* variables bound by a certain λ. This is not always what is desired. In the case when a definition is coded, it is clear that this kind of β-reduction is too radical: one sometimes wishes to "unfold" a definition at a certain place, but such an unfolding should not concern *all* places where the same definition is used. The following example illustrates the point:

Example 3.1 The notion "continuity" of a function may be defined as a λ-term in constructive mathematics but needs a rather complicated definition. Now sometimes, e.g. in a proof, one "goes back to the definition" by substituting the text body of this definition, in which the definiens is expressed. In such a case one certainly does not want as a side effect that the word "continuity" will be replaced by its definiens at all places in the text where it appears.

This is the reason for admitting another kind of β-reduction, called *local* β-reduction, where only one bound variable can be replaced (See also [7]). To emphasize the difference between this local β-reduction and the usual one, we shall call the latter *global* β-reduction.

Another wish is to execute substitutions only when necessary. For this purpose one may decide to postpone substitutions as long as possible ("lazy evaluations"). This can yield profits, since substitution is an inefficient, maybe even exploding, process by the many repetitions it causes. This is the ground for the so-called graph reduction, see e.g. [25].

We shall describe substitution as a step-by-step procedure, giving the user the possibilities to use it as he wishes. Our step-wise treatment of substitution and reduction is connected with the wish to unravel these processes in atomary steps. This is no restriction, since we can also combine these steps into the ordinary β-relations.

3.2 Adding substitution items

Recall that we had two kinds of items, the λ-items, of the form $(t\lambda)$ and the δ-items of the form $(t\delta)$. In order to be able to push substitutions ahead, step by step, we shall introduce a new kind of items, called **substitution items** (or σ-items). These σ-items can move through the branches of the term, step-wise, from one node to an adjacent one, until they reach a leaf of the tree. At the leaf, if appropriate, a σ-item can cause the desired substitution effect. In this manner these substitution items can bring about different kinds of β-reductions.

Definition 3.2 *(σBLT)*
We extend the set of operators with σ, whose arity is two. Terms of σBLT are exactly those of BLT except that new terms can now be formed using not only λ and δ but also σ. We keep to the same meta level notation of Section 2.3, but let $\omega, \omega_1, \omega_2, \ldots$ range over λ, δ and σ.

Now, if one goes back to Definition 2.33, the only set which changes is $\mathcal{O}$ which gets σ as an extra element. To be more precise, it is not only one σ that is added, rather an infinite number of them, $\sigma^{(i)}, i \in \mathcal{N}$. Based on this observation, Lemma 2.25 holds for σBLT. Moreover, all the definitions and results of Section 2.5 (and in particular Section 2.5.2) hold here.

We use σ as an *indexed* operator, numbered with upper indices: $\sigma^{(1)}, \sigma^{(2)}, \ldots$. Hence a σ-item has the form: $(t'\sigma^{(i)})$.

The intended meaning of a σ-item $(t'\sigma^{(i)})$ is: term t' is a candidate to be substituted for one or more occurrences of a certain variable; the index i selects the appropriate occurrences. In fact, the index i preserves the variable that has to be substituted for. More on this will follow.

3.3 Step-wise substitution

3.3.1 One-step σ-reduction

Now we can give the rules for *one-step σ-reduction*. This relation is denoted by the symbol $\rightarrow_\sigma$. The relation *σ-reduction* is the reflexive and transitive closure of one-step substitution. It is denoted by $\twoheadrightarrow_\sigma$. We introduce $\rightarrow_\sigma$ as a relation between segments, although it is meant to be a relation between terms. The rules must be read as follows: rule $\overline{s} \rightarrow_\sigma \overline{s'}$ states that $t \rightarrow_\sigma t'$ when a segment of the form $\overline{s}$ occurs in t, where t' is the result of the replacement of this $\overline{s}$ by $\overline{s'}$ in t. Otherwise said, we omit so-called compatibility rules (see [2]).

Definition 3.3 *(σ-reduction)*
(σ-generation rule:)
$(t_1\delta)(t_2\lambda) \rightarrow_\sigma (t_1\delta)(t_2\lambda)(t_1\sigma^{(1)})$
(σ-transition rules:)

$$\begin{array}{ll}
(t_1\sigma^{(i)})(t_2\lambda) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\lambda) & (\sigma_{0\lambda} - transition) \\
(t_1\sigma^{(i)})(t_2\lambda) \rightarrow_\sigma (t_2\lambda)(t_1\sigma^{(i+1)}) & (\sigma_{1\lambda} - transition) \\
(t_1\sigma^{(i)})(t_2\lambda) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\lambda)(t_1\sigma^{(i+1)}) & (\sigma_{01\lambda} - transition) \\
(t_1\sigma^{(i)})(t_2\delta) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\delta) & (\sigma_{0\delta} - transition) \\
(t_1\sigma^{(i)})(t_2\delta) \rightarrow_\sigma (t_2\delta)(t_1\sigma^{(i)}) & (\sigma_{1\delta} - transition) \\
(t_1\sigma^{(i)})(t_2\delta) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\delta)(t_1\sigma^{(i)}) & (\sigma_{01\delta} - transition)
\end{array}$$

(σ-destruction rules:)
$(t_1\sigma^{(i)})i \rightarrow_\sigma \mathtt{ud}^{(i)}(t_1)$
$(t_1\sigma^{(i)})x \rightarrow_\sigma x$ *if* $x \neq i$.

Compare the σ-generation rule with (β) . Our rule, does not get rid of $(t_1\delta)(t_2\lambda)$ but keeps it because we are not necessarily going to perform a global β-reduction, so some variables may still be bound by the λ in $(t_2\lambda)$ (see Example 3.18). The addition of $(t_1\sigma^{(1)})$ moreover, is to fire substitutions which will, according to the transition rules be to the right of the tree of the term, upwards in the tree or both. The destruction rule is for the case where we have reached a leaf and σ cannot propagate any longer, then substitution takes place. The following details about these rules elaborate more these points.

- Firstly, the σ-generation rule adds a σ-item to the term, as the start of a possible reduction. Note that in this rule, the so-called **δ-λ-segment** or **reducible segment** $(t_1\delta)(t_2\lambda)$ stays where it is; this is different from ordinary β-reduction, where both argument and corresponding λ disappear. The reason for not removing this reducible segment is, of course, that we want to keep a binding λ and the corresponding argument (i.e. δ-item) in a term, as long as there still are variables in the term that are bound by that λ. When the substitution process is on its way, existing bonds are maintained. Moreover, when we choose to perform *local* β-reduction, then one bound variable disappears in the substitution process, but other bound occurrences of the same variable, which are also possible clients for the same substitution, may stay. We shall see in 3.5 how we can dispose of a reducible segment when there are no more customers for the λ involved, i.e. when there is no variable bound by this λ in the term.

- Secondly, the σ-transition rules occur in two triples, one triple for the case that a σ-item meets a λ-item, and one for the case where a σ-item meets a δ-item. In each triple the following three possibilities are covered:

  1. The σ-item can move *inside* the item met (upwards in the tree; the cases σ_0), this is when we are interested only in inside reductions.
  2. The σ-item can jump *over* the item (to the right in the tree; σ_1), this is when we are interested only in reductions to the right of the tree.
  3. The σ-item can do both things at the same time (σ_{01}), this is when we are interested in both reductions.

  For the time being, all possibilities may be effectuated. Only in the case that the σ-item jumps over a λ-item (i.e. in the cases $\sigma_{1\lambda}$ and $\sigma_{01\lambda}$), the index of the σ increases by one. This is because that index counts the number of λ's actually passed, in order to find the right (occurrence of the) variable involved. The index is also of use in the process of updating the substituted term t_1 (see below).

- Thirdly, the σ-destruction rules apply when the σ-item has reached a leaf of the tree. When the index i of the σ is in accordance with the value of the variable, then we have met an intended occurrence of the variable; the substitution of t_1 for i takes place, accompanied with an updating (`ud`) of the variables in t_1. This updating is necessary, in order to restore the right correspondences between variables in t_1 and λ's. When the index of σ and the variable in question do not match, then nothing happens to the variable, and the σ-item vanishes without effect.

It is not hard to see that the **update function** $\mathtt{ud}^{(i)}$ should have the following effect on term t_1: all *free* variables in t_1 must increase by an amount of i. (The σ-generation rule initialized i with value 1, for obvious reasons.) This updating is a simple process.

The following lemma shows that σ-reduction reaches eventually all occurrences to be substituted. I.e., there is a path for global β-reduction, but we may not take it.

Lemma 3.4 *In $(t_1\delta)(t_2\lambda)t_3$, σ-reduction can substitute t_1 for all occurrences of the variable bound by the λ of $(t_2\lambda)$ in t_3.*

Proof: *The proof is by an easy induction on t_3 in $(t_1\delta)(t_2\lambda)(t_1\sigma^{(1)})t_3$.* □

The examples below demonstrate how σ-reduction works.

3.3.2 Examples of one-step σ-reduction

Example 3.5 Let us take example 2.30 and see how σ-reduction works here too. This example is not very interesting from the point of view of different possibilities of substitution, due to the presence of just one occurrence of the x to be substituted. It will however, demonstrate the working of the rules, in the case where a σ-item meets a δ-item.

There are 3 cases to consider, depending on the choice concerning the σ-transition rules.

case 1 (using $\sigma_{0\omega}$-transition rules only)
$(1\delta)(2\lambda)(4\delta)1 \to_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(4\delta)1 \to_\sigma$
$(1\delta)(2\lambda)((1\ \sigma^{(1)})4\delta)1 \to_\sigma$
$(1\delta)(2\lambda)(4\delta)1$
case 2 (using $\sigma_{1\omega}$-transition rules only)
$(1\delta)(2\lambda)(4\delta)1 \to_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(4\delta)1 \to_\sigma$
$(1\delta)(2\lambda)(4\delta)(1\ \sigma^{(1)})1 \to_\sigma$
$(1\delta)(2\lambda)(4\delta)\mathtt{ud}^{(1)}(1) \to_\sigma$
$(1\delta)(2\lambda)(4\delta)2.$
case 3 (using $\sigma_{01\omega}$-transition rules only)
$(1\delta)(2\lambda)(4\delta)1 \to_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(4\delta)1 \to_\sigma$
$(1\delta)(2\lambda)((1\ \sigma^{(1)})4\delta)(1\ \sigma^{(1)})1 \to_\sigma$
$(1\delta)(2\lambda)(4\delta)\mathtt{ud}^{(1)}(1) \to_\sigma$
$(1\delta)(2\lambda)(4\delta)2.$

The first case which only carries out reductions upwards in the tree, has missed the occurrence of x to the right of the tree, and so no reductions have been carried out. The second case does the reduction to the right of the tree, so it does substitute the x. The third case carries out reductions both upwards and to the right. But upwards results in nothing new so we obtain the same result as in case 2.

In the second and third cases, $(1\delta)(2\lambda)$ is useless and once we remove it, we should decrease the free variables in $(4\delta)2$ obtaining hence $(3\delta)1$ (see Figure 7 which you should also note its similarity to Figure 6).

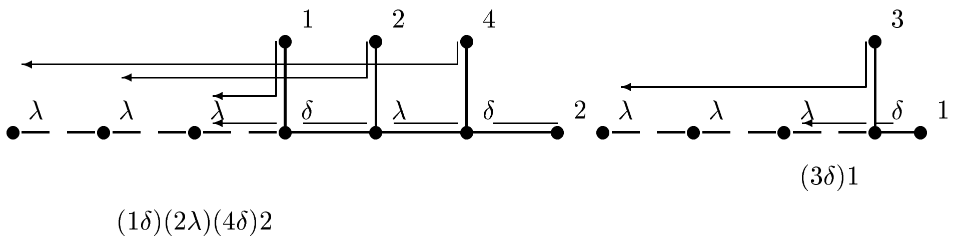


Figure 7: σ-reduction when a σ-item meets a δ-item

There are, of course, more possibilities than these three cases, if we use a mixture of the $\sigma_{0\omega}$-, $\sigma_{1\omega}$- and $\sigma_{01\omega}$-transition rules.

Example 3.6 Now let us see how σ-reduction works when we have that a σ-item meets a λ-term. Take for example: $(\lambda_{y:z}.\lambda_{x:z}.y)u$. In BLT notation this is $(1\delta)(2\lambda)(3\lambda)2$ and in LT notation, it is: $(u\delta)(z\lambda_y)(z\lambda_x)y$ (see Figure 8). This term reduces to $\lambda_{x:z}.u$ or in BLT notation $(2\lambda)2$ and in LT notation $(z\lambda_x)u$. σ-reduction on this term results in the following 3 cases.

case 1
$(1\delta)(2\lambda)(3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)((1\ \sigma^{(1)})3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)(3\lambda)2$
case 2
$(1\delta)(2\lambda)(3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)(3\lambda)(1\ \sigma^{(2)})2 \to_\sigma$
$(1\delta)(2\lambda)(3\lambda)\mathtt{ud}^{(2)}(1) \to_\sigma$
$(1\delta)(2\lambda)(3\lambda)3.$
case 3
$(1\delta)(2\lambda)(3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)((1\ \sigma^{(1)})3\lambda)(1\ \sigma^{(2)})2 \to_\sigma$
$(1\delta)(2\lambda)(3\lambda)\mathtt{ud}^{(2)}(1) \to_\sigma$
$(1\delta)(2\lambda)(3\lambda)3.$

Again the first case didn't carry out any substitutions as there was none in the upward part of the tree. The second and third cases are similar to those of

Example 3.5. Moreover, $(1\delta)(2\lambda)$ is useless and once we remove it, we should decrease the free variables in $(3\lambda)3$ obtaining $(2\lambda)2$ (see Figure 8). Note that what actually happens in Figure 8 is that the part of the tree with nodes $1, 2, \delta, \lambda$ is removed and the part of the tree with nodes $\lambda, 3, 3$ replaces it but with the variables updated to point at the correct λ's.

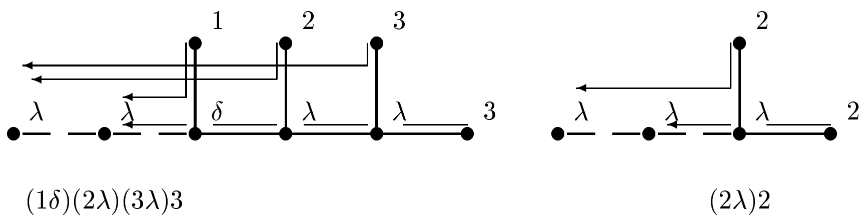


Figure 8: σ-reduction when a σ-item meets a λ-item

The definition of σ-reduction could be simplified further as follows:

Definition 3.7 *($\overline{\sigma}$-reduction)*
($\overline{\sigma}$-generation rule:)
$(t_1\delta)(t_2\lambda) \to_{\overline{\sigma}} (t_1\delta)(t_2\lambda)(t_1\overline{\sigma}^{(1)})$
($\overline{\sigma}$-transition rules:)

$$\begin{array}{ll} (t_1\overline{\sigma}^{(i)})(t_2\lambda) \to_{\overline{\sigma}} ((t_1\overline{\sigma}^{(i)})t_2\lambda)(t_1\overline{\sigma}^{(i+1)}) & (\overline{\sigma}_{01\lambda} - transition) \\ (t_1\overline{\sigma}^{(i)})(t_2\delta) \to_{\overline{\sigma}} ((t_1\overline{\sigma}^{(i)})t_2\delta)(t_1\overline{\sigma}^{(i)}) & (\overline{\sigma}_{01\delta} - transition) \end{array}$$

($\overline{\sigma}$-destruction rules:)

$$\begin{array}{ll} (t_1\overline{\sigma}^{(i)})i \to_{\overline{\sigma}} \mathtt{ud}^{(i)}(t_1) & (\overline{\sigma}_0 - destruction) \\ (t_1\overline{\sigma}^{(i)})x \to_{\overline{\sigma}} x \text{ if } x \neq i & (\overline{\sigma}_1 - destruction) \\ (t_1\overline{\sigma}^{(i)}) \to_{\overline{\sigma}} \emptyset & (\overline{\sigma}_2 - destruction) \end{array}$$

That is, in Definition 3.3, we get rid of $\sigma_{0\omega}$ and $\sigma_{1\omega}$ for $\omega = \lambda$ or δ, and add $\overline{\sigma}_2$-destruction.

Now it is obvious to see that $\sigma_{0\omega}$ and $\sigma_{1\omega}$ are special cases of $\overline{\sigma}_{01\omega}$, in the presence of $\overline{\sigma}_2$-destruction. In fact the following lemma holds:

Lemma 3.8 *For any t_1, t_2, if $t_1 \to_\sigma t_2$ then $t_1 \twoheadrightarrow_{\overline{\sigma}} t_2$.*
Proof: *Left to the reader.* □

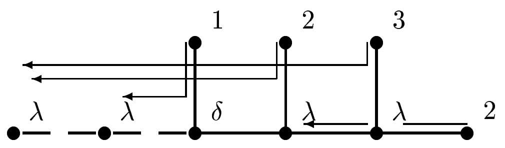
3.3.3 Drawbacks of the one-step σ-reduction

We want to ensure that references are always maintained correctly, even before substitution takes place. This is not the case as we shall see below.

We note that our updating is less complicated, but also less general than in the original treatment of de Bruijn-indices (see [5]), where the usual β-reduction is applied (the global relation) and substitution is not presented as a step-wise process. In explicit substitution procedures as in [1], the more general, but complicated update functions are used.

Our loss of generality has the following cause. A σ-item $(t\sigma^{(i)})$ is supposed to be "cut off" from the rest of the term. Variables in t may have lost their reference value; in case a variable x in t is bound by a λ outside t, then this binding λ can only be found by taking also the index i into consideration. That is: variables inside a σ-segment are shut off from the "outer world", meaning that their value need not reflect the exact binding place. Only after application of the σ-destruction rule, the updating restores the proper value of such variables. The following example illustrates the point:

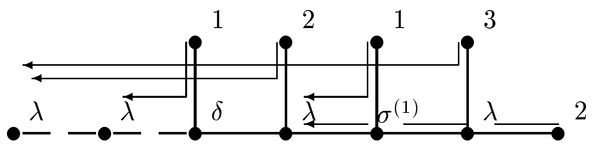
Example 3.9 Let us look back at Example 3.6. The term discussed there was $(1\delta)(2\lambda)(3\lambda)2$ and its term tree is pictured in Figure 9.



$(1\delta)(2\lambda)(3\lambda)2$

Figure 9: The term tree of $(1\delta)(2\lambda)(3\lambda)2$

When adding the σ-item $(1\sigma^{(1)})$, we messed up the references. In fact, the 1 in $(1\sigma^{(1)})$ tells us that it is bound by the λ in (2λ) but this is not as it should be: its reference must be the same as the original one in (1δ), namely the second λ in the free variable list. The term tree of $(1\delta)(2\lambda)(1\sigma^{(1)})(3\lambda)2$ is to be found in Figure 10.



$(1\delta)(2\lambda)(1\sigma^{(1)})(3\lambda)2$

Figure 10: A term tree where references are not as intended

In the following subsection we propose a solution for step-wise substitution that does not suffer from the mentioned drawbacks.

3.4 A general step-wise substitution

3.4.1 Step-by-step update or φ-reduction

In order to avoid the disadvantages mentioned in subsection 3.3, we shall describe the effect of the update function by means of a step-by-step approach. For this purpose we use a (unary prefix) function symbol $\varphi^{(k,i)}$ with two parameters k and i. The intention of the indices is the following.

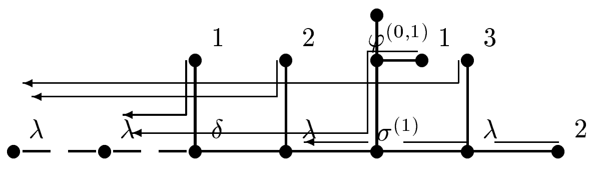
- Index i preserves the value of the update desired (i = '**increment**').
- Index k counts the λ's that are internally passed by (k = '**threshold**').

The effect of the updating must be that all free variables in t_1 increase with an amount of i; the k is meant to identify the free variables in t_1.

Now, instead of $\mathtt{ud}^{(i)}(t_1)$, we write $(\varphi^{(0,i)})t_1$. We extend our set of operators in Definition 2.33 with φ. As explained above, we use the φ's with a double index: $\varphi^{(k,i)}; k, i \in \mathcal{N}$. We call all $(\varphi^{(k,i)})$'s **φ-items**. Note that the body of a φ-item is always the empty term.

Before we set up the φ-rules, let us go back to Example 3.9 and let us show how the use of φ will fix the references.

Example 3.10 In Figure 10, let us replace the 1 above $\sigma^{(1)}$ by a tree which has 2 branches, the upwards being empty, the right branch having 1 as a leaf, and the root being $\varphi^{(0,1)}$. Now when tracing from the 1 (above $\sigma^{(1)}$), the λ which binds it, we pass through $\varphi^{(0,1)}$. This is indicative that the λ can be found by the combination of the index 1 and the item $\varphi^{(0,1)}$. The 0 in $(0,1)$ tests if index 1 is free or bound. If index 1 was bound (i.e. if $1 \leq 0$) then we forget about the 1 in $(0,1)$ and look for the first λ. 1 however is not ≤ 0 and so it is free and its binding λ is the one refered to by $1+1$, being the sum of the variable 1 itself and the second projection of $(0,1)$. Figure 11 shows the right references.



$(1\delta)(2\lambda)((\varphi^{(0,1)})1\sigma^{(1)})(3\lambda)2$

Figure 11: A term tree where references are as intended

Now, the use of the φ-items is established in the following rules.

Definition 3.11 *(φ-reduction)*
(σ-destruction/φ-generation rule:)
$(t_1\sigma^{(i)})i \to_\varphi (\varphi^{(0,i)})t_1$
(φ-transition rules:)
$(\varphi^{(k,i)})(t'\lambda) \to_\varphi ((\varphi^{(k,i)})t'\lambda)(\varphi^{(k+1,i)})$
$(\varphi^{(k,i)})(t'\delta) \to_\varphi ((\varphi^{(k,i)})t'\delta)(\varphi^{(k,i)})$
(φ-destruction rules:)
$(\varphi^{(k,i)})x \to_\varphi x + i$ *if* $x > k$
$(\varphi^{(k,i)})x \to_\varphi x$ *if* $x \leq k$ *or* $x \equiv \varepsilon$.

The following details about these rules are to be noted.

- In the φ-generation rule, t_1 is to substitute i, the variable bound by the *ith* λ to the left of i. t_1 has passed these i λ's and so all its free variables must be increased by i. Therefore, we use $\varphi^{(0,1)}$.
- A term of the form $(\varphi^{(k,i)})t$ will be either such that t is a variable or a λ-item or a δ-item. In the case of a δ-item or a λ-item, we have to update all the variables so that we keep the right references.
- The case where $(\varphi^{(k,i)})$ is to the left of a variable, we use one of two φ-destruction rules, the first for the case that x is free in t_1 (then a real update occurs), the second for the case that x is bound in t_1 or $x \equiv \varepsilon$ (then nothing happens with x).

Now, in order to keep the references inside a σ-item correct during the process of σ-transition, a φ-item $(\varphi^{(k,i)})$ is added inside the σ-item, as follows: $((\varphi^{(k,i)})t\sigma^{(j)})$. We shall give the rules of this general σ-reduction below.

For convenience sake, we may drop the first index or both indices of the φ, according to the following definition:

Definition 3.12 *(φ-abbreviation)*
For all $i \in \mathcal{N}$, $\varphi^{(i)}$ denotes $\varphi^{(0,i)}$. Moreover, φ denotes $\varphi^{(1)}$ (hence $= \varphi^{(0,1)}$).

3.4.2 General σ-reduction

Now the rules for σ-items can be adapted as follows (cf. Definition 3.3):

Definition 3.13 *(general σ-reduction)*
(general σ-generation rule:)
$(t_1\delta)(t_2\lambda) \to_\sigma (t_1\delta)(t_2\lambda)((\varphi)t_1\sigma^{(1)})$
(general σ-transition rules:)

$$
\begin{array}{ll}
(t_1\sigma^{(i)})(t_2\lambda) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\lambda) & (\sigma_{0\lambda} - \mathit{transition}) \\
(t_1\sigma^{(i)})(t_2\lambda) \rightarrow_\sigma (t_2\lambda)((\varphi)t_1\sigma^{(i+1)}) & (\sigma_{1\lambda} - \mathit{transition}) \\
(t_1\sigma^{(i)})(t_2\lambda) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\lambda)((\varphi)t_1\sigma^{(i+1)}) & (\sigma_{01\lambda} - \mathit{transition}) \\
(t_1\sigma^{(i)})(t_2\delta) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\delta) & (\sigma_{0\delta} - \mathit{transition}) \\
(t_1\sigma^{(i)})(t_2\delta) \rightarrow_\sigma (t_2\delta)(t_1\sigma^{(i)}) & (\sigma_{1\delta} - \mathit{transition}) \\
(t_1\sigma^{(i)})(t_2\delta) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\delta)(t_1\sigma^{(i)}) & (\sigma_{01\delta} - \mathit{transition})
\end{array}
$$

(general σ-destruction rules:)
$(t_1\sigma^{(i)})i \rightarrow_\sigma t_1$
$(t_1\sigma^{(i)})x \rightarrow_\sigma x$ *if* $x \neq i$.

Note that a term $t_1 \equiv t'$ changes into $(\varphi)t'$ when passing a λ; see e.g. the $\sigma_{1\lambda}$-rule. The reason is that the *free* variables in t' must be increased by an amount of 1 (remember that $\varphi = \varphi^{(0,1)}$, hence the increment is 1). The obtained $(\varphi)t'$ is again a term, so one may take $t_1 \equiv (\varphi)t'$ in the next step.

Now the following lemma shows that the right bond between variables and their binding λ's are maintained.

Lemma 3.14 *In $\overline{s}(t_1\delta)(t_2\lambda)((\varphi)t_1\sigma^{(1)})t_3$, all variable occurrences are bound by the same λ's which bound them in $\overline{s}(t_1\delta)(t_2\lambda)t_3$.*

Proof: *We will only show how some cases can be carried out. The rest will be an easy exercise left to the reader. Let x be a variable in $(t_1\delta)(t_2\lambda)((\varphi)t_1\sigma^{(1)})$. There are only two cases to consider.*

- *case x occurs in $(t_1\delta)(t_2\lambda)$, then nothing to prove, as nothing has changed for that occurrence.*
- *case x occurs in $((\varphi^{(0,1)})t_1\sigma^{(1)})$, in particular in t_1, then a bound variable in t_1 clearly remains bound by the same λ in t_1. A free variable x in t_1 becomes updated by 1 by the $\varphi^{(0,1)}$. This is exactly what is intended, since there is one extra λ that one has to go through on the way from x to its λ. That is, the λ of $(t_2\lambda)$.*

□

Example 3.15 Let us go through example 3.6 but using φ-reduction.
case 1
$(1\delta)(2\lambda)(3\lambda)2 \rightarrow_\sigma$
$(1\delta)(2\lambda)((\varphi)1\ \sigma^{(1)})(3\lambda)2 \rightarrow_\sigma$
$(1\delta)(2\lambda)(((\varphi)1\ \sigma^{(1)})3\lambda))2 \rightarrow_\sigma$
$(1\delta)(2\lambda)(3\lambda)2$
case 2
$(1\delta)(2\lambda)(3\lambda)2 \rightarrow_\sigma$
$(1\delta)(2\lambda)((\varphi)1\ \sigma^{(1)})(3\lambda)2 \rightarrow_\sigma$

$(1\delta)(2\lambda)(3\lambda)((\varphi)(\varphi)1\ \sigma^{(2)})2 \to_\sigma$
$(1\delta)(2\lambda)(3\lambda)(\varphi)(\varphi)1 \twoheadrightarrow_\varphi$
$(1\delta)(2\lambda)(3\lambda)3$
case 3
$(1\delta)(2\lambda)(3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)((\varphi)1\ \sigma^{(1)})(3\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)(((\varphi)1\ \sigma^{(1)})3\lambda)((\varphi)(\varphi)1\ \sigma^{(2)})2 \twoheadrightarrow_{\sigma,\varphi}$
$(1\delta)(2\lambda)(((\varphi)1\ \sigma^{(1)})3\lambda)3 \to_\sigma$
$(1\delta)(2\lambda)(3\lambda)3$

It is not hard to see that this definition gives the same results as Definition 3.3 in the case that we apply the φ-transition rules *after* all possible σ-transition rules have been applied. However, we have now the possibility to "update" the σ-item at any instance, thus re-establishing the correct bond between bound variable and binding λ. It is also more easy now to find the binding λ of a certain variable in t_1 *before* updating: following the path from the variable to the root, we just add j for every $(\varphi^{(j)})$ encountered.

Again here, we may use the simplified version of general σ-reduction, which consists of the same σ-generation rule, only of $\sigma_{01\lambda}$ and $\sigma_{01\delta}$ as σ-transition rules, and the same destruction rules together with $(t_1\sigma^{(i)}) \to_\sigma \emptyset$.

Finally, we note that our transition rules as given here do not allow for σ-items to "pass" other σ-items. The reason for this is, that we wish to prevent undesired effects, like an infinite exchange of two adjacent σ-items.

3.4.3 Remarks on φ

The mentioned $(\varphi^{(j)})$ may originate as combinations of "simple" (φ)-items. Let us assume for a moment that only one-step σ-reductions are applied to a given term, and no φ-reductions. Then a σ-item, "travelling" through this term, "collects" as many φ-items (φ) as it has passed λ-items. These φ-items may be combined, since $(\varphi)\ldots(\varphi)$ (i times) $= (\varphi)^i = (\varphi^{(i)})$.

We can make a few more remarks in this respect.

1. First, it is not necessary to update t_1 completely. One can easily convince oneself that φ-items with equal first index are *additive*, in the sense that $(\varphi^{(k,m)})(\varphi^{(k,n)})$ has the same effect as $(\varphi^{(k,m+n)})$, for all $k, m, n \in \mathcal{N}$. In particular, $(\varphi^{(m)})(\varphi^{(n)})$ "is" $(\varphi^{(m+n)})$. Hence, one may split up $(\varphi^{(j)})$ into $(\varphi^{(j')})$ and $(\varphi^{(j'')})$ in case $j > 1$ and $j' + j'' = j$, and update with $(\varphi^{(j'')})$. This process can be repeated at many places. Moreover, a φ-transition can be executed for one or more steps, or left alone, whichever one likes.

   Things become more complicated if we desire to combine two adjacent φ-items like $(\varphi^{(k,i)})$ and $(\varphi^{(l,m)})$ in one new update function. We do not consider these matters, in order to maintain a simple system.

2. Second, we note, that it is quite natural to add a third φ-transition rule for the case that we desire to update a term starting with a σ-item:

   Definition 3.16 *(φ-transition rule for σ-items:)*
   $(\varphi^{(k,i)})(t'\sigma^{(l)}) \rightarrow_\varphi ((\varphi^{(k,i)})t'\sigma^{(l)})(\varphi^{(k,i)})$ *if* $l \leq k$ *and*
   $(\varphi^{(k,i)})(t'\sigma^{(l)}) \rightarrow_\varphi ((\varphi^{(k,i)})t'\sigma^{(l+i)})(\varphi^{(k,i)})$ *if* $l > k$.

   So far, we showed that σ-items and φ-items have obtained the same status as the original λ- and δ-items. The σ- and φ-items have become, so to say, "first class citizens". There is, however, still a slight scent of discrimination, in the sense that some items can blockade the transition of other items. For example, σ-items cannot pass φ-items. These matters have to be investigated, especially as regards the consequences for normalization. At this moment, these questions are not yet solved.

3. A third remark is, that there is with this general σ-reduction a feasible possibility for the addition of a $\sigma_{01\sigma}$-transition. This can be done, since the bodies of σ-items now contain the correct references, by the extra φ-items added. Hence, we can allow that σ-items intrude other σ-items:

   Definition 3.17 *($\sigma_{01\sigma}$-transition)*
   $(t_1\sigma^{(i)})(t_2\sigma^{(k)}) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\sigma^{(k)})(t_1\sigma^{(i)})$ *if* $i \neq k$

3.5 Substitution and β-reduction

So far, we have explained using our reduction $\twoheadrightarrow_\sigma$, how a term containing a δ-λ-segment can be transformed to another term. We have not yet explained how we can get local and global β-reduction out of such reduction. Moreover, so far in our approach, the reducible segment is not removed. We still have to supply the tools for eliminating useless reducible segments. In this section we explain how reducible segments are removed and how local and global β-reduction are obtained.

3.5.1 Local and global substitution

We recall here that with **global substitution** we mean the intended replacement of a whole class of bound variables (all bound by the same abstraction-λ) by a given term; for **local substitution** we have only one of these occurrences in view. By restricting the choice we have in the σ-transition rules we get local and global reduction. Let us give an example.

Example 3.18 Take the term $(\lambda_{x:z}.xx)u$. There are three possibilities here, either we can have global β-reduction and then obtain uu, or we can have local β-reduction where the first x of the body xx is replaced by u, or we can have local β-reduction where the second x is replaced by u. Those three cases are easily obtainable from our σ-reduction. Here is how:

The term in our notation is $(1\delta)(2\lambda)(1\delta)1$. Applying σ-reduction we get the following cases:

case 1
$(1\delta)(2\lambda)(1\delta)1 \to_\sigma$
$(1\delta)(2\lambda)((\varphi)1\ \sigma^{(1)})(1\delta)1 \to_\sigma$
$(1\delta)(2\lambda)(((\varphi)1\ \sigma^{(1)})1\delta)1 \to_\sigma$
$(1\delta)(2\lambda)((\varphi)1\delta)1 \to_\varphi$
$(1\delta)(2\lambda)(2\delta)1$
case 2
$(1\delta)(2\lambda)(1\delta)1 \to_\sigma$
$(1\delta)(2\lambda)((\varphi)1\ \sigma^{(1)})(1\delta)1 \to_\sigma$
$(1\delta)(2\lambda)(1\delta)((\varphi)1\ \sigma^{(1)})1 \to_\sigma$
$(1\delta)(2\lambda)(1\delta)(\varphi)1 \to_\varphi$
$(1\delta)(2\lambda)(1\delta)2.$
case 3
$(1\delta)(2\lambda)(1\delta)1 \to_\sigma$
$(1\delta)(2\lambda)((\varphi)1\ \sigma^{(1)})(1\delta)1 \to_\sigma$
$(1\delta)(2\lambda)(((\varphi)1\ \sigma^{(1)})1\delta)((\varphi)1\ \sigma^{(1)})1 \twoheadrightarrow_{\sigma,\varphi}$
$(1\delta)(2\lambda)(2\delta)2$

- Case one comes from using $\sigma_{0\omega}$-transition and is the local substitution for the second x in xx resulting in $(\lambda_{x:z}xu)u$.
- Case 2 comes from using $\sigma_{1\omega}$-transition and is the local substitution of the first x resulting in $(\lambda_{x:z}ux)u$.
- The third case comes from using $\sigma_{01\omega}$-transition and is the global substitution resulting in $(\lambda_{x:z}.uu)u$ which should of course be rewritten as uu (we still have not removed useless segments). That is: the reducible segment $(1\delta)(2\lambda)$ in the result of case 3 should be removed and $(2\delta)2$ should be changed to $(1\delta)1$. Below we will see how to do this.

Note however that in cases 1 and 2, we cannot remove $(1\delta)(2\lambda)$ because we only carried out local substitution on one occurrence of the bound variable and there are occurrences that are still bound by the same λ.

3.5.2 Efficiency considerations

For local β-reduction, as is seen from the example above, we have to make a choice between either σ_0 or σ_1, both when meeting a λ- or a δ- item, in order to follow the right path to the intended (occurrence of the) variable. Such a path may be coded by a string of 0's and 1's in an obvious manner. For global β-reduction we also have a choice. Syntactically the simplest thing is to choose always the

σ_{01}-rules, dispersing the σ-item over all branches to come. However, in the case that we know beforehand which branches lead to an occurrence of the substitutable variable in question, and which do not, we can, at each λ- or δ-item met, make the appropriate choice between σ_0, σ_1 or σ_{01}. The last possibility is efficient as regards the σ-transitions; it depends, however, on the implementation whether the mentioned information about branches and variables is present. Alas however, the generation and maintenance of this information has its price as well.

Of course, there exists a scale of possibilities between local and global: e.g., one may formalize substitution for a *number* of designated occurrences of a certain variable.

A *one-step* local β-reduction of a term consists of one σ-generation and a local reduction as described above, executed until the σ in question (and the corresponding φ's) have disappeared. Cases 1 and 2 of example 3.18 are instances of a one-step local β-reduction. A one-step global β-reduction is defined analogously. Case 3 of example 3.18 is an instance of a one-step global reduction. Note that, in both cases, the reducible segment is not (yet) removed.

An option is to distinguish from the beginning between (possible) local and global β-reductions, by using different λ's and/or δ's (see [24], for the use of various λ's and δ's).

Example 3.19 We could use $\lambda_{\texttt{loc}}$ for a future destination in local reductions and $\lambda_{\texttt{glo}}$ for global reductions. A "definition" then could be rendered as a δ-λ-segment $(t_1\delta_{\texttt{loc}})(t_2\lambda_{\texttt{loc}})$, ready for local reduction. A "function" could start with a λ-item $(t_2\lambda_{\texttt{glo}})$, whereas an "argument" for this function could have the form of a δ-item $(t_1\delta_{\texttt{glo}})$.[i]

Now, for example, the general σ-generation rule of Definition 3.13 obtains two versions:

Definition 3.20 *(local vs. global σ-generation)*
$(t_1\delta_i)(t_2\lambda_i) \to_\sigma (t_1\delta_i)(t_2\lambda_i)((\varphi)t_1\sigma_i^{(1)})$, *for* $i = \texttt{loc}, \texttt{glo}$.

As regards the σ-transition rules, either the σ_0-transition or the σ_1-transition is chosen for $\sigma_{\texttt{loc}}$'s, according to the path in the tree that has been prescribed. And σ_{01}-transition is reserved for $\sigma_{\texttt{glo}}$'s. The σ-destruction rules are adapted with an index to the σ, in an obvious manner.

The possibility of labelling λ's and δ's as above is an evidence of the flexibility of our account.

3.5.3 Removing the useless reducible segments

Let us keep in mind the two reductions strategies (local and global) and remember that they can overlap. For example, when we have one unique occurrence of the

[i]See [22] for an explanation of these notions "definition", "function" and " argument" with respect to typed lambda calculus.

variable to be substituted, as in $(\lambda_x.x)u$ where we have one unique occurrence of the x in the body, then both local and global substitutions are the same. Let us hence take some standpoints as to how we are going to treat such an overlap and when we should remove the useless segments.

It will be clear that, in applying local β-reduction, we have a certain reducible segment and an occurrence of one goal-variable in view, connected by means of a path in the tree. Hence we know that the reducible segment has actual reductional potencies, i.e. the main λ of the segment binds at least one occurrence of a variable.

As regards global β-reduction, the situation is different. Here the reducible segment may be "without customers". Then σ-generation is undesirable since this leads to useless efforts. Hence it seems a wise policy to restrict the use of the σ-generation rule to those cases where the main λ of the reducible segment does actually bind at least one variable. When this is *not* the case, we shall speak of a **void δ-λ-segment**. Such a segment may be removed. One may compare this case to the application of a constant function to some argument; the result is always the (unchanged) body of the function in question. For this purpose we define the **void β-reduction**:

Definition 3.21 *(void β-reduction)*

Assume that a δ-λ-segment $\overline{s}$ occurs in a layered term t, where the final operator λ° of $\overline{s}$ does not bind any variable in t. Let t_1 be the scope of $\overline{s}$, i.e. rightarg(λ°). Then t reduces to the term t', obtained from t by removing $\overline{s}$ and replacing t_1 by $(\varphi^{(-1)})t_1$.

Notation: $t \rightarrow_{\emptyset} t'$.[j]

We can also describe void β-reduction in the previously given format:

Definition 3.22 *($\delta\lambda$-destruction rule)*

$(t'\delta)(t''\lambda) \rightarrow_{\emptyset} (\varphi^{(-1)})$ *if* $(t'\delta)(t''\lambda)$ *is void.*

Note the fact that updating here occurs with a *negative* amount of -1. The reason is that the disappearance of the λ has to be compensated. We note that this negative updating is not without complications. For example:

- The second φ-transition rule of Definition 3.16 is no longer valid.
- Additivity of φ-items (see Section 3.4) does not hold for negative indices. E.g. $(\varphi^{(1,1)})(\varphi^{(1,-1)})$ is not equal to $(\varphi^{(1,0)})$ (the identity), since

$$(\varphi^{(1,1)})(\varphi^{(1,-1)})(2\delta)1 \twoheadrightarrow_{\varphi} (1\delta)1$$

- The same example shows that negative indices can have the effect that *different* variables become identified:

$$(\varphi^{(1,-1)})(2\delta)1 \twoheadrightarrow_{\varphi} (1\delta)1$$

  Hence, updating is no longer an injection, which can be highly undesirable.

We note, however, that the mentioned unpleasant effects do not occur in the setting presented above: a φ-item with a negative exponent only occurs after the clean-up of a void δ-λ-segment, hence with a λ that does not bind any variable. Therefore, the injective property of updating is not threatened.

We shall give an example which demonstrates how void segments can disappear.

Example 3.23 Take example 3.5. After σ-reduction we obtained $(1\delta)(2\lambda)(4\delta)2$ in the cases 2 and 3 (see Figure 7). In this latter term, call it t, the δ-λ-segment $(1\delta)(2\lambda)$ occurs and its λ does not bind any variable in t. Moreover, $(4\delta)2$ is the scope of $(1\delta)(2\lambda)$ and if in t we remove $(1\delta)(2\lambda)$ and replace $(4\delta)2$ by $(\varphi^{(-1)})(4\delta)2$ we get $(3\delta)1$. Hence t reduces to $(3\delta)1$.

Lemma 3.24 *If $t \rightarrow_{\emptyset} t'$ then all occurrences of variables in t' are bound by the same λ's that bind them in t.*

Proof: *Left to the reader.* □

[j]This reduction was introduced in [21], where it was called β_2-reduction. De Bruijn defines a *mini-reduction* as being either a one-step local β-reduction or a void reduction; see [7].

Now we can describe the usual one-step β-reduction as a combination of σ-steps and φ-steps:

Definition 3.25 *(one-step β-reduction)*
One-step β-reduction of a layered term is the combination of one σ-generation from a δ-λ-segment $\overline{s}$, the transition of the generated σ-item through the appropriate subterm in a global manner, followed by a number of destructions, and updated by φ-items until again a layered term is obtained.
Finally, there follows one void β-reduction for the disposal of $\overline{s}$.

Notation 3.26 We denote one-step β-reduction by $t \to_{\sigma\beta} t'$, and (ordinary) β-reduction — its reflexive and transitive closure — by $t \twoheadrightarrow_{\sigma\beta} t'$. We write $=_\beta$ for the equivalence relation generated by $\twoheadrightarrow_{\sigma\beta}$.

Remark 3.27 About the normalisation properties of our system (concerning the termination of β-reduction sequences) we note the following.
We first recall some well-known concepts:
A *redex* in a term is a subterm which starts with a δ-λ-segment.
A *normal form* is a term without a redex (hence without a δ-λ-segment).
A term t is *strongly normalizing* if *all* β-reduction sequences, starting from t, terminate (in a normal form).
A term t is *weakly normalizing* if *some* β-reduction sequence, starting from t, does terminate (in a normal form).
In general: the property *strong normalization* refers to the *necessary* termination, for each term, of all β-reduction sequences starting from that term, and the property *weak normalization* refers to the *possible* termination, for each term, of a β-reduction sequence starting from that term.

Now we discuss normalization with respect to our system of rules.

The σ-generation rule, as given in Definition 3.3, can be applied indefinitely many times. A similar remark holds for the $\sigma_{01\sigma}$-transition rule, which permits an eternal reshuffling between adjacent σ-items. Hence, strong normalization is not guaranteed without extra provisions.

This may be an awkward matter, especially in (typed) systems that "normally" *do* strongly normalize. Hence, it may be advisable to *restrict* the use of these rules in order to prevent the mentioned effects. For the latter rule (the $\sigma_{01\sigma}$-transition rule) this is easy: just forbid its use, maybe with the exception that it can be used in one-step local β-reductions. For the former rule one might formulate the condition that a σ-item may only be generated by a δ-λ-segment if this segment is not void, and if it cannot become void by substitutions which are "on the way", i.e. by the application of φ- and σ- reductions which are due to φ- and σ-items which are already present in the term under consideration.

It will be clear that our rules do not hamper *weak* normalization . Indeed, if a terminating sequence exists, starting from a term t, we can always choose an appropriate strategy for step-wise substitution in order to "follow the path" of this normalizing β-reduction sequence.

4 Comparison with the explicit substitution of Abadi, Cardelli, Curien and Lévy

4.1 The calculus of Abadi, Cardelli, Curien and Lévy

In [1], the $\lambda\sigma$-calculus is introduced, where explicit substitutions are dealt with in an algebraic manner. We give a short survey of the operators that the authors introduce and we discuss some features of the equational theory that is proposed in the paper.

The authors use de Bruijn-indices and define substitutions as index manipulations. A substitution is an infinite list of substitution instructions, one for each natural number greater than 0. For example, $s = \{a_1/1, a_2/2, a_3/3, \ldots\}$ is a notation for the substitution of the terms a_i for the indices i. When s is considered as a function, then $s(i)$, the "substituand" for i, is a_i. Another notation for $s(i)$ is $i[s]$.

Such an infinite substitution must be thought of as being a *simultaneous* substitution of all a_i for i.

It will be clear that infinite substitutions are meant as *meta*-notations for actual simultaneous substitutions, the latter ones being finite and therefore executable. In fact, for any term with de Bruijn-indices there is a maximal number N that can occur as an index; as one can easily see, this number N is equal to the number of λ's occurring in the term plus the maximal reference place in the free variable list, of the different free variables that occur in the term. Hence, an infinite substitution for a given term can always be pruned to a finite explicit substitution.

Apart from id — the identity substitution $\{i/i\}$ or $\{1/1, 2/2, \ldots\}$ — [1] introduces three other index manipulations:[k]

- $\uparrow$ (*shift*), the substitution $\{(i+1)/i\}$.
- $\cdot$, as in $a \cdot s$, the *cons* of a onto s; here a is a term and s a substitution. The substitution $a \cdot s$ is the substitution $\{a/1, s(i)/(i+1)\}$, that is to say: a is alloted to index 1, and all substituands $s(i)$ are alloted to an index which is one more than the original one (i). For example:
  $1 \cdot \uparrow = \{1/1, \uparrow(1)/2, \uparrow(2)/3, \ldots\} = id$.
- $\circ$, as in $s \circ t$, the *composition* of s and t; here both s and t are substitutions, and $s \circ t = \{t(s(i))/i\}$. For example:
  $\uparrow \circ (a \cdot s) = \{(a \cdot s)(\uparrow(i))/i\} = \{(a \cdot s)(i+1)/i\} = \{s(i)/i\} = s$.

[k]The examples are taken from [1]. Note how the operations can be used for algebraic manipulations.

4.2 A soundness proof

With the help of our system, we can give a soundness proof for the equality axioms in [1]. Therefore we translate the above operations into the notation introduced in the present paper. We have no direct means to render infinite substitutions, but we introduce *parallel σ-items* for this purpose. Such a parallel σ-item is an infinity of $\sigma^{(i)}$-items, one for each number $i > 0$. The notation that we use is $(t_i\sigma^{(\overline{\imath})})$. The "vector" upper bar ($\overline{\imath}$) abbreviates a universal quantification. By $(t_i\sigma^{(\overline{\imath})})$ we mean the same as Abadi et al. mean with the substitution $\{t_1/1, t_2/2, \ldots\}$, i.e. the simultaneous substitution of t_i for i for all i. Similarly, $(t_i\sigma^{(\overline{\imath}>1)})$ denotes the same as $\{t_2/2, t_3/3, \ldots\}$, and so on.

Hence, the definition of the parallel σ-item $(t_i\sigma^{(\overline{\imath})})$ is that for any variable k, $(t_i\sigma^{(\overline{\imath})})k = t_k$.

We may split such a parallel σ-item in a finite head and an infinite tail, connected with the symbol $\oplus$. For example:
$(t_i\sigma^{(\overline{\imath})}) = (t_1\sigma^{(1)}) \oplus (t_i\sigma^{(\overline{\imath}>1)})$.

We define a function $[\![\ldots]\!]$, mapping terms from [1] to terms in our calculus. We define moreover index manipulation functions to parallel σ-items.

For terms, the definition is:

$$\begin{array}{lcl} [\![i]\!] & = & i \\ [\![\lambda a]\!] & = & (\lambda)[\![a]\!] \\ [\![ab]\!] & = & ([\![b]\!]\delta)[\![a]\!] \end{array}$$

For the index manipulation function we have the following:
Let a be a term, $[\![s]\!] = (t_i\sigma^{(\overline{\imath})})$ and $[\![s']\!] = (t'_i\sigma^{(\overline{\imath})})$. Then:

$$\begin{array}{lcl} [\![id]\!] & = & (i\sigma^{(\overline{\imath})}) \\ [\![\uparrow]\!] & = & ((i+1)\sigma^{(\overline{\imath})}) \\ [\![a \cdot s]\!] & = & (a\sigma^{(1)}) \oplus (t_{i-1}\sigma^{(\overline{\imath}>1)}) \\ [\![s \circ s']\!] & = & (t'_j\sigma^{(\overline{\jmath})})(t_i\sigma^{(\overline{\imath})}) \end{array}$$

Finally, $[\![a[s]]\!] = [\![s(a)]\!] = [\![s]\!][\![a]\!]$. It is not hard to see that $(t'_j\sigma^{(\overline{\jmath})})(t_i\sigma^{(\overline{\imath})}) = ((t'_j\sigma^{(\overline{\jmath})})t_i\sigma^{(\overline{\imath})})$, so that we have an alternative translation for $s \circ s'$.

Moreover, it will be clear that (φ) and $\uparrow$ (or $(\ i+1\ \sigma^{(\overline{\imath})})$) have the same effect. The same holds, in general, for $(\varphi^{(k,l)})$ and $(\ i+l\ \sigma^{(\overline{\imath}>k)})$.

We show that we can justify the algebraic manipulations of Abadi et al. in this setting. Moreover, the equations that the authors give as an axiomatic basis for their equational theory, can all be *derived* in our approach. In our opinion, this is an important result in favor of the treatment that we propose in this paper.

Moreover, we claim that the introduction of parallel σ-items is only apparently an extension of the system that we discussed in the present paper:
— the infinity of σ-items can be reduced to a finite number for every given term (we explained this above);
— the "parallel" (simultaneous) character of the substitutions is embodied in our

φ-items; this is the only "global" substitution operator for de Bruijn-indices that we need, the σ-items being the vehicles for the substitution.

The latter property follows from the fact that we discriminate between *updating* of de Bruijn-indices and *actual substitutions*. This distinction, absent in [1], simplifies matters considerably.

A comparison between the two systems gives the following results:

- The system of Abadi et al. is based on a set of algebraic equality rules, which are treated with the usual term rewriting techniques. It only works for the usual (global) β-reduction.
- Our system has a wider range of application, since it is also suited for local reduction. Moreover, it seems that the separation of real substitution and simple updates makes things less complex; we also have the feeling that our system is, in a sense, more "natural".

We give four rules from [1] and show their justification in our setting. Those rules are:

- *VarCons:* $1[a \cdot s] = a$.
- *Abs:* $(\lambda a)[s] = \lambda(a[1 \cdot (s\circ \uparrow)])$.
- *SCons:* $1[s] \cdot (\uparrow \circ s) = s$.
- *Beta:* $(\lambda a)b = a[b \cdot id]$

To show Beta, we need the equation
$(\varphi^{(-1)})((\varphi)t_1\sigma^{(1)}) = (t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)})$.
That this equality holds is shown by the following Lemma:

Lemma 4.1 *In $\varphi\sigma BLT$, the following holds:*
$(\varphi^{(-1)})((\varphi)t_1\sigma^{(1)}) = (t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)})$.

Proof:
$(\varphi^{(-1)})((\varphi)t_1\sigma^{(1)}) =$
$(\ i-1\ \sigma^{(\overline{\imath})})((\ j+1\ \sigma^{(\overline{\jmath})})t_1\sigma^{(1)}) =$
(differentiate between the effect of this substitution on index 1 and on indices > 1, respectively)
$(\ i-1\ \sigma^{(\overline{\imath})})((\ j+1\ \sigma^{(\overline{\jmath})})t_1\sigma^{(1)}) \oplus (\ i-1\ \sigma^{(\overline{\imath}>1)}) =$
(since, as noted above: $(t_i\sigma^{(\overline{\imath})})(t'_j\sigma^{(\overline{\jmath})}) = ((t_i\sigma^{(\overline{\imath})})t'_j\sigma^{(\overline{\jmath})})$)
$((\ i-1\ \sigma^{(\overline{\imath})})(\ j+1\ \sigma^{(\overline{\jmath})})t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)}) =$
(by additivity, which holds in this case)

$((j\sigma^{(\overline{\jmath})})t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)}) =$
$(t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)}).$ □

Now, here is how the above four rules can be derived in our system:

Lemma 4.2 *In $\varphi\sigma BLT$, the rules* VarCons, Abs, SCons *and* Beta *are derivable.*
Proof:

- VarCons:
  $[\![1[a \cdot s]]\!] = [\![(a \cdot s)1]\!] = (([\![a]\!]\sigma^{(1)}) \oplus (t_{i-1}\sigma^{(\overline{\imath}>1)}))1 \to_\sigma [\![a]\!].$
- Abs:
  $[\![(\lambda a)[s]]\!] = (t_i\sigma^{(\overline{\imath})})(\lambda)[\![a]\!] \to_\sigma (\lambda)((\varphi)t_{i-1}\sigma^{(\overline{\imath}>1)})[\![a]\!],$
  since $(t_i\sigma^{(i)})(\lambda)[\![a]\!] \to_\sigma (\lambda)((\varphi)t_i\sigma^{(i+1)})[\![a]\!]$ *for each* i*;*
  $[\![\lambda(a[1 \cdot (s\circ \uparrow)]]\!] = (\lambda)((1\sigma^{(1)}) \oplus ((\varphi)t_{i-1}\sigma^{(\overline{\imath}>1)}))[\![a]\!] =$
  $= (\lambda)((\varphi)t_{i-1}\sigma^{(\overline{\imath}>1)})[\![a]\!],$
  since $[\![s\circ \uparrow]\!] = ((\ j+1\ \sigma^{(\overline{\jmath})})t_i\sigma^{(\overline{\imath})}) = ((\varphi)t_i\sigma^{(\overline{\imath})}).$
- SCons:
  $[\![1[s] \cdot (\uparrow \circ s)]\!] = ((t_i\sigma^{(\overline{\imath})})1\sigma^{(1)}) \oplus ((t_j\sigma^{(\overline{\jmath})})i\sigma^{(\overline{\imath}>1)}) =$
  $= (t_1\sigma^{(1)}) \oplus (t_i\sigma^{(\overline{\imath}>1)}) = (t_i\sigma^{(\overline{\imath})}) = [\![s]\!].$
- Beta:
  The traditional rule of β-reduction has the following form in our system
  $(t_1\delta)(t_2\lambda) \to_\sigma (\varphi^{(-1)})((\varphi)t_1\sigma^{(1)}).$
  This enables us directly to derive the translation of the Beta*-rule:*
  $[\![(\lambda a)b]\!] = ([\![b]\!]\delta)(\lambda)[\![a]\!] \to_\sigma (\varphi^{(-1)})((\varphi)[\![b]\!]\sigma^{(1)})[\![a]\!];$
  $[\![a[b \cdot id]]\!] = (([\![b]\!]\sigma^{(1)}) \oplus (\ i-1\ \sigma^{(\overline{\imath}>1)}))[\![a]\!] =$
  $(([\![b]\!]\sigma^{(1)}) \oplus (\varphi^{(1,-1)}))[\![a]\!].$
  Hence, $[\![(\lambda a)b]\!] = [\![a[b \cdot id]]\!]$ *from Lemma 4.1.*

□

This section hence, showed that the whole of [1]'s system can be translated into ours and that some of what they take as rules are easily derivable in our system. This shows that our system is more general than theirs.

It is also possible to give a translation the other way round. To achieve that purpose, we have to express σ-items $(t\sigma^{(i)})$ and φ-items $(\varphi^{(k,l)})$ by means of the operators id, $\uparrow$, $\cdot$ and $\circ$. Here below we give these translations, where we adopt the convention that the $\cdot$- operation is *associating to the right*, so $a \cdot b \cdot s$ means $a \cdot (b \cdot s)$.

Then the following correspondences hold (here we identify the notations t and $[\![t]\!]$):

- $(t\sigma^{(i)}) = 1 \cdot 2 \cdot \ldots \cdot (i-1) \cdot t \cdot (\uparrow)^i$ and

- $(\varphi^{(k,l)}) = 1 \cdot 2 \cdot \ldots \cdot k \cdot (\uparrow)^{k+l}$.

In particular, $(t\sigma^{(1)}) = t \cdot \uparrow$.
Also, $(\varphi^{(n)}) = (\varphi^{(0,n)}) = (\uparrow)^n$, $(\varphi) = (\varphi^{(1)}) = \uparrow$ and $(\varphi^{(0,-1)}) = (\uparrow)^{-1} = 1 \cdot id$.
When we define $k!$ to be $1 \cdot 2 \cdot \ldots \cdot k$, then the above rules can be simplified to

- $(t\sigma^{(i)}) = (i-1)! \cdot t \cdot (\uparrow)^i$ and
- $(\varphi^{(k,l)}) = k! \cdot (\uparrow)^{k+l}$,

provided that we add the rule $0! \cdot s = s$.

Finally, we give the correspondence between our system and the $\Uparrow$-operator of [10]:

- If $s = \Sigma_i(t_i\sigma^{(i)})$, then $\Uparrow(s) = \Sigma_i((\varphi)t_i\sigma^{(i+1)})$.

The translation from our system to Abadi et al. was only carried out for the sake of completeness and because it is the norm that has to be carried out when comparing two systems. Our claims still hold, that is, we can translate all of their system in our and we can show that some of their rules are derivable in ours. It is not likely that we have redundant rules which can be derived in their system. In fact, this is also the difference between the two systems. Ours just has the rules of the λ-calculus. Theirs, has many many rules which as we have seen here, can be got rid of in our system.

5 Advantages, Conclusions and Further Work

We believe that the notation in this paper deserves attention. We showed how it can facilitate the introduction of substitution as an object level notion in the lambda calculus resulting in a system which can accommodate most substitution strategies. We showed for example, how local and global substitution can be obtained in a unique formulation which can provide the most general substitution in the λ-calculus and all desirable forms of substitution. This is an important step on its own. We have shown that the substitution calculus of [1] can all be translated into ours, together with the result that many of their axioms are easily derivable in our calculus. This again is a nice result. Also, our calculus accommodates explicit substitution in a calculus very close to the classical formulation of the λ-calculus, whereas [1] uses a notation that is not easily grasped at first for those who are unfamiliar with it. Below, we discuss the advantages of our notation and explicit substitution together with an insight into further work. The advantages of the new notation do not stop at substitution, but extend to all branches of the λ-calculus. The layered representation of terms can be a natural basis for the allocation of the free and bound occurrences and for the λ's binding particular variables. It can also be used to restrict the attention to those subterms of a term relevant for a particular application.

5.1 Advantages of the notation

We started in Section 2 with a novel description of term formation, regarding abstraction and application as binary operations. The item-notation of terms enabled us to create a term progressively, or module-like, so to say, in analogy with the manner in which mathematical and logical ideas are developed. Variables and variable bindings obtained a natural place in this setting, both in the name-carrying and in the name-free version, the latter by means of de Bruijn-indices.

Two notational features are of great advantage in this respect: the first is to give the argument prior to (i.e. in front of) the function; the second, of minor importance, is that a type precedes the variable which it regards.

The advantages of our new notation are summarized below. The reader however, will appreciate the new notation more through [23] and [24].

- The convention of writing the argument *before* the function has a practical advantage: the δ-item and the λ-item involved in a β-reduction occur *adjacently* in the term; they are not separated by the "body" of the term, that can be extremely long! It is well-known that such a δ-λ-segment can code a definition occurring in some mathematical text; in such a case it is very desirable for legibility that the coded definiendum and definiens occur very close to each other in the term.

- The notation provides a general vehicle for describing many type theories and calculi. This point has been elaborated in [24] where systems from Barendregt's cube are special instances of our own. Further, we showed there how theorem proving in the calculus of constructions (see [9]) could be more easily done in our framework.

- Bound and free variables are easily accounted for as can be seen from Example 2.43.

- Items and segments play an important role in many applications. As explained before, a λ-item is the part joined to a term in an abstraction, and a δ-item is the part joined in an application. In using typed lambda calculi for e.g. mathematical reasoning, λ-items may be used for assumptions or variable introductions and a δ-λ-segment may express a definition or a theorem (See [22], [23] and [24]).

- There are further advantages, but for the purpose of this paper, we decided to concentrate on explicit substitution. We will below summarize what we did relating to this subject.

5.2 Advantages of explicit substitution and of our formulation of it

In Section 3 we focussed on the relation of reduction. We differentiated between several versions of β-reduction, for example between global β-reduction (the ordinary one) and local β-reduction, necessary for unfolding a defined name in only one place.

In describing these versions of β-reduction, we defined the notion of step-wise substitution, being the utmost refinement of the reduction-concept. For this step-wise reduction we introduced σ-items as a part of the term syntax, thus making substitution an explicit procedure.

When using de Bruijn-indices, we have to make sure that the references in a term are updated during or after a substitution. For this purpose we introduced φ-items, which again do their job in a step-wise fashion.

We also gave a general step-wise substitution, with the purpose of keeping the references (by de Bruijn-indices) unimpaired, also inside the σ-items. As to the reducible segments, we keep them present until they are no longer necessary, then we get rid of them using the notion of void β-reduction.

In Section 4, we introduced the calculus of [1], and showed that it is only a special case of our calculus by providing a translation of the first in the second. This translation can also be viewed as a soundness proof. We showed moreover that many of the axioms that are postulated in [1] are very easily derivable in our system, which shows that our calculus is more attractive. In fact, it is our conviction that the step-wise substitution as introduced in this paper is easier and more manegeable than the proposal for explicit substitution in [1]. Our approach is very close to intuition, yet the formulation remains simple. Here is a summary of the usefulness of explicit substitution and of the advantages of our formulation of it.

- β-reduction is too radical in the case when a definition is coded (see Example 3.1). Therefore local forms of reduction are needed.

- Substitution is inefficient and may be exploding. Therefore we might wish to postpone substitutions as long as possible. The ability hence to control what substitutions to carry out and when is very important.

- The step-wise character of our reduction relation and of our many described procedures enables a flexible approach, in the sense that the user may choose how to combine basic steps into combined ones, depending on the circumstances. For instance, global β-reduction amounts to the generation of one σ-item, and subsequently chasing this item along all possible paths in the direction of the leaves of the term tree, until no descendants of the original σ-item are left. For local β-reduction the σ-item has to follow precisely one path, in the direction of the variable that is chosen as a candidate for substitution.

- The possibility of labelling λ's, δ's and σ's so as to control which local substitutions to carry out is an evidence of the flexibility of our account.

- The step-wise substitution introduced in this paper is more manageable than that of [1].

- Our substitution allows most strategies (local, global, in between) and all is controlled by the user.

5.3 Further work

Now, as for further work, it is known that substitution plays an important role in logic (in quantifier introduction or elimination, to give an example). The notions of free/bound variables are also very important there. Moreover, combining λ-calculus with logic leads to inconsistencies if no restrictions are made. Based on these facts, [11], ..., [20] provide various theories which attempt at combining λ-calculus with logic and at avoiding the paradoxes through types, or through λ-abstraction. These attempts are also applied to various notions of logic, programming and natural languages, such as polymorphism, fixed point theorems, quantifiers, determiners, undefinability results and unification. Explicit substitution has not taken place yet in these areas and this will be followed in the future.

To give one example from Computer Science which would benefit from explicit substitution, we take *pattern matching* and *unification* as used in functional and logic programming. We have not yet tried to study the implications of our system on pattern matching and unification, but we plan to do so in the near future.

As for more foundational issues, we know that the Church Rosser theorem holds for our calculus but we would like to work out the details. We have no doubt that this is a straightforward process similar to the usual proof of Church Rosser. As for the semantics of explicit substitution, and the models of our calculus, this too is an area that we will investigate in the very near future.

6 Acknowledgments

We would like to thank Bert van Benthem Jutting, Inge Bethke, Tijn Borghuis, Kees Hemerik, John Launchbury, Erik Poll and the anonymous referees for their helpful comments and recommendations. We are also grateful for the help received from various members of Eindhoven University, concerning Latex. In particular, we thank Jos Coenen and Jan Joris Vereyken, for their help.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, "Explicit Substitutions", *Functional Programming 1 (4)* (1991) 375-416.

[2] H. Barendregt, *Lambda Calculus: its Syntax and Semantics* (North-Holland, Amsterdam, 1984).

[3] H. Barendregt, "Lambda calculi with types", in *Handbook of Logic in Computer Science II*, eds. S. Abramsky, D.M. Gabbay and T.S.E. Maibaum (Oxford University Press, 1992).

[4] N.G. de Bruijn, "The mathematical language AUTOMATH, its usage and some of its extensions", in *Symposium on Automatic Demonstration, IRIA, Versailles, 1968*, Lecture Notes in Mathematics, 125, (Springer, Berlin, 1970) 29-61.

[5] N.G. de Bruijn, "Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem", *Indagationes Math. 34, (5)* (1972) 381-392.

[6] N.G. de Bruijn, "A survey of the project AUTOMATH", in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, eds. J.R. Hindley and J.P. Seldin (Academic Press, New York/London, 1980) 29-61.

[7] N.G. de Bruijn, "Generalizing Automath by means of a lambda-typed lambda calculus", in: *Mathematical Logic and Theoretical Computer Science*, eds. D.W. Kueker, E.G.K. Lopez-Escobar and C.H. Smith, Lecture Notes in Pure and Applied Mathematics, 106, (Marcel Dekker, New York, 1978) 71-92.

[8] N.G. de Bruijn, "Algorithmic definition of lambda-typed lambda calculus". In preparation.

[9] T. Coquand and G. Huet, "The calculus of constructions", *Informations and Computation 76* (1988), 95-120.

[10] Th. Hardin, and J.-J. Lévy, "A confluent calculus of substitutions", Lecture notes of the INRIA-ICOT symposium, Izu, Japan, November (1989).

[11] F. Kamareddine, "Semantics in a Frege structure, PhD thesis, University of Edinburgh, 1989.

[12] F. Kamareddine, "A system at the cross roads of logic and functional programming", *Science of Computer Programming 19*, (1992), 239-279.

[13] F. Kamareddine, " λ-terms, logic, determiners and quantifiers", *Logic, Language and Information 1 (1)* (1992) 79-103.

[14] F. Kamareddine, "Set Theory and Nominalisation, Part I", *Logic and Computation 2 (5)* (1992) 579-604.

[15] F. Kamareddine, "Set Theory and Nominalisation, Part II", *Logic and Computation 2 (6)* (1992) 687-707.

[16] F. Kamareddine, "Are types needed for Natural Language?", proceedings for the applied Logic conference, Amsterdam, December (1992). The proceedings will appear (in a revised form) as a book.

[17] F. Kamareddine, and E. Klein, "Polymorphism, Type containment and Nominalisation", *Logic, Language and Information 2* (1993) 171-215.

[18] F. Kamareddine, "Non well foundedness and type freeness can unify the interpretation of functional application", to appear in *Logic, Language and Information*, 1993.

[19] F. Kamareddine, "Themes in the λ-calculus", paper presented at the 1992 logic Colloquium, Vesprem, Hungary. Abstract to appear in the *Journal of Symbolic Logic.*

[20] F. Kamareddine and E. Klein, "Polymorphism and Logic in Programming and Natural languages", to appear in the proceedings of the 1988 conference on unification, Titisee, Germany.

[21] R.P. Nederpelt, "Strong normalisation in a typed lambda calculus with lambda structured types", Ph.D. thesis, Eindhoven University of Technology, Eindhoven, 1973.

[22] R.P. Nederpelt, "Type systems — basic ideas and applications", in: *CSN '90, Computing Science in the Netherlands 1990*, ed. A.J. van de Goor (Stichting Mathematisch Centrum, Amsterdam, 1990).

[23] R.P. Nederpelt, and F. Kamareddine, "A useful lambda notation", paper presented at the 1992 logic Colloquium, Vesprem, Hungary. Abstract to appear in *the Journal of Symbolic Logic.*

[24] R.P. Nederpelt and F. Kamareddine, "A unified approach to type theory through a refined λ-calculus", paper presented at the 1992 conference on *Mathematical Foundations of Programming Semantics*, submitted for publication in the proceedings.

[25] S.L. Peyton Jones, *The Implementation of Functional Programming Languages*, (Prentice-Hall, Englewood Cliffs, 1987).