On stepwise explicit substitution

Eindhoven University of Technology

Department of Mathematics and Computing Science

On stepwise explicit substitution

by

R.P.Nederpelt and F.Kamareddine

92/08

Computing Science Note 92/08
Eindhoven, April 1992

COMPUTING SCIENCE NOTES

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

On stepwise explicit substitution

Rob Nederpelt
and
Fairouz Kamareddine
Department of Mathematics and Computing Science
Eindhoven University of Technology
Eindhoven, the Netherlands

April 23, 1992

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, lazy evaluation and the partial unfolding of definitions. Moreover our substitution is, we believe, the most general up to date. This is shown by the fact that our framework can accommodate most of the known reduction strategies.

Keywords: *lambda calculus, item notation, substitution, reduction.*

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 turn out to 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 [de Bruijn 72]). Section 2 introduces both de Bruijn's indices and the new notation. Illustrative examples are given throughout.

The use of our notation for many important notions of the lambda calculus will be left to another paper. In this paper however, we will be concentrating on the advantages of such a notation for a lambda theory where substitution is an object level concept rather than a meta level one. 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 [Abadi et al. 90] 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 up to date.

2 The calculus

2.1 The lambda calculus with de Bruijn's indices

Terms of the untyped lambda calculus are constructed as follows: t ::= $x \mid (\lambda_x.t) \mid (t_1 t_2)$. Parenthesis 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 lambda calculus, whether typed or untyped, is β-conversion which is as follows:[1] $(\lambda_x.t_1)t_2 = t_1[x := t_2]$,[2] where substitution has been defined in a way which deals with the problem of variable clashes. 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. Example 2.1 below shows how lambda terms can be denoted using de Bruijn's indices and example 2.2 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 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.

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). Here the number 2 tells us how many λs we have to count from (and excluding) the λ to the left of 2. The free variables u and y in the typed lambda term are translated into the number 1 (occurring after the term in parenthesis), 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 2.2 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.[3]

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

[2]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.

[3]In more complicated examples, there are more cases in which variables must be "updated". This updating

We have in examples 2.1 and 2.2 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 linear 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.

2.2 The new notation

We shall construct terms in typed lambda calculus as a free structure. That is to say, we consider the two main constructive principles for such terms, viz. abstraction and application, as *operations* on terms. Moreover, we allow different kinds of abstractions and applications, denoted by operators $\lambda, \lambda_1, \lambda_2, \ldots$ for abstraction, $\delta, \delta_1, \delta_2, \ldots$ for application, and $\omega, \omega_1, \omega_2, \ldots$, for both kinds of operators. We refer to the set of **λ-operators** by Ω_λ and to the set of **δ-operators** by Ω_δ. We assume that Ω_λ and Ω_δ are disjoint and finite and write Ω (or $\Omega_{\lambda\delta}$) for their union.

Operators will be written in *infix*-notation; that is we shall consider abstraction as a *binary* operation, linking types to terms (*in this order*; see Example 2.8). Application is a binary operation as well, linking "argument" to "function" again *in this order*, that is to say in the style that writes af instead of fa (or $f(a)$) for function f applied to argument a. This is not only a matter of taste; it will turn out to have essential advantages in developing a term, theoretically as well as in practical applications of typed lambda calculi.[4] Moreover, we take Ξ to be the set of **variables:** $\Xi = \{\varepsilon, 1, 2, \ldots\}$ and use $x, x_1, y, \ldots$ to denote variables.[5]

Definition 2.3 *(terms) $F_\Omega(\Xi)$ is the free Ω-structure generated by Ξ, i.e. the set of symbol strings obtained in the usual manner on the basis of Ξ, the operators in Ω and parentheses. Elements of $F_\Omega(\Xi)$ are called* **terms** *or* **$\Omega_{\lambda\delta}$-terms** *that we shall denote by $t, t_1, \ldots$.*

Examples of terms are: ε, 3, $(2\delta(\varepsilon\lambda 1))$ (where we assume that $\lambda \in \Omega_\lambda$ and $\delta \in \Omega_\delta$).[6]

Notation 2.4 *(Item-notation)* We shall place parentheses in an unorthodox manner: we write $(t_1\omega)t_2$ instead of $(t_1\omega t_2)$. This will be called the **itemized** or **item**-notation. 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, according to the infix notation of terms in $F_\Omega(\Xi)$, $t'\lambda t$ is obtained by prefixing $t'\lambda$ to t.
- the application tt' (in "classical" notation) is obtained by postfixing the argument-item t' to the term t. Now in $F_\Omega(\Xi)$, $t'\delta t$ is obtained by prefixing $t'\delta$ to t.

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).

[4]This observation is due to de Bruijn, see [de Bruijn 70] or [de Bruijn 80].

[5]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)

[6]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 $\Lambda = \lambda_1$.

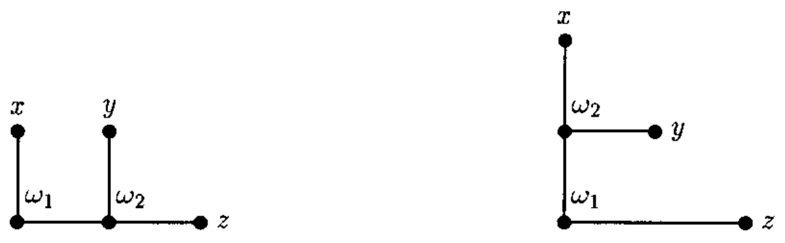
In item-notation we write in these cases $(t'\lambda)t$ and $(t'\delta)t$, respectively. Here both items $(t'\lambda)$ and $(t'\delta)$ are *prefixed* to the term t. We use $s, s_1, s_i, \ldots$ as meta-variables for items.

Notation 2.5 *(parentheses)* Note the intended parsing convention:

In the term $(s_1 \ldots s_n x\omega)s'_1 \ldots s'_m y$, the operator ω combines the *full* term $s_1 \ldots s_n x$ with the *full* term $s'_1 \ldots s'_m y$.

So the $\Omega_{\lambda\delta}$-term $(x\omega_1(y\omega_2 z))$ becomes in item-notation: $(x\omega_1)(y\omega_2)z$. Analogously, the $\Omega_{\lambda\delta}$-term $((x\omega_2 y)\omega_1 z)$ becomes $((x\omega_2)y\omega_1)z$. More concretely: $(2\delta(\varepsilon\lambda 1))$ becomes $(2\delta)(\varepsilon\lambda)1$.[7]

Notation 2.6 *(tree notation)* One can also consider terms as trees, in the usual manner (in this case we shall speak of **term trees**). In term trees, parentheses are superfluous (see figure 1). In this figure, 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 linear term. This has also advantages in the sections to come. The item-notation suggests a partitioning of the term tree in vertical layers. 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.



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

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

Figure 1: Term trees, with normal linear notation and item-notation

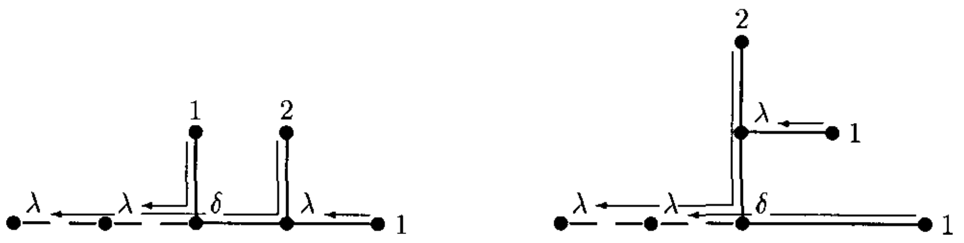
Notation 2.7 *(name carrying terms)* For ease of reading, we occasionally use customary variable names like x, y, z and u instead of reference numbers, thus creating name-carrying terms in item-notation, such as $(u\delta)(y\lambda_x)x$ in Example 2.8. The symbols used as subscripts for λ in this notation are only necessary for establishing the place of reference; they do not "occur" as variables in the term.

Example 2.8 Consider the typed lambda term $(\lambda_{x:y}.x)u$. In item-notation with name-carrying variables this term becomes $(u\delta)(y\lambda_x)x$. In item-notation with de Bruijn-indices, it is denoted as $(1\delta)(2\lambda)1$.

The typed lambda term $u(\lambda_{x:y}.x)$ is denoted as $((y\lambda_x)x\delta)u$ 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_y, λ_u, in both examples.

[7]In the *Automath*-tradition (cf. [de Bruijn 80]), an abstraction-item $\lambda_{x:t'}$ (or $(t'\lambda)$ in our new notation) is called an *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 $< t' >$.

The term trees of these lambda terms are given in figure 2. 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. Note that these lines follow the path which leads from the variable to the root following *the left side* of the branches of the tree. Only the λ's met do count, the δ's do not.



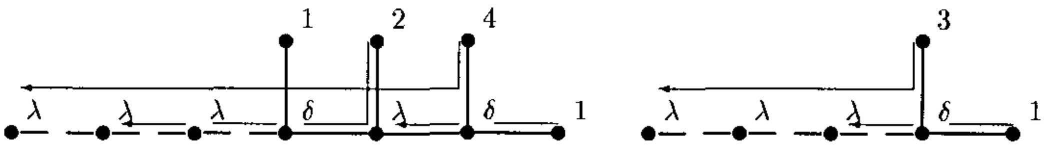
$(1\delta)(2\lambda)1$ $\qquad$ $((2\lambda)1\ \delta)1$

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

$(\lambda_{x:y}\ .\ x)u$ $\qquad$ $u(\lambda_{x:y}\ .\ x)$

Figure 2: Term trees with explicit free variable lists and reference numbers

Example 2.9 Now for β-reduction, the term $(\lambda_{x:z}.(xy))u$ β-reduces to uy. In our sugared item-notation this becomes: $(u\delta)(z\lambda_x)(y\delta)x$ reduces to $(y\delta)u$ (see figure 3). 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: $(u\delta)(y\lambda_x)$) is the signal for a possible β-reduction. The "unsugared" version reads: the term $(1\delta)(2\lambda)(4\delta)1$ reduces to $(3\delta)1$.



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

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

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

Figure 3: 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.

3 Reduction

Nothing so far has been said about the axioms and rules that we assume over our set of terms. In fact, the two indispensable axioms of the λ-calculus, that is β- and α-conversion, have only lightly been touched in the preceding section. The axiom (α) got a better deal than (β) because de Bruijn's indices that were introduced in 2.1 and used in 2.2 make α-conversion unnecessary. What about (β)? Such an axiom (β): $(\lambda_x.t_1)t_2 = t_1[x := t_2]$ where substitution $t_1[x := t]$ is defined by certain metarules has been used in most of the theories of the λ-calculus. 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 hence 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. For example, the notion "continuity of a function" 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.[8] 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. [Peyton Jones 87].

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

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

[8]See also [de Bruijn 87]

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 3.1 *($\Omega_{\lambda\delta\sigma}$-terms) We extend the set $\Omega_{\lambda\delta}$ with the finite set Ω_σ (the set of* **σ-operators***). The arity of these operators is two. The new Ω is called $\Omega_{\lambda\delta\sigma}$. We keep to the same meta level notation of 2.2, but let $\omega, \omega_1, \omega_2, \ldots$ range over λ-, δ- and σ-operators.*

In this paper, we take Ω_σ to have only one element: σ. We use σ as an indexed *operator, numbered with upper indices: $\sigma^{(1)}, \sigma^{(2)}, \ldots$. Hence a σ-item has the form: $(t'\sigma^{(i)})$. Our terms now are $\Omega_{\lambda\delta\sigma}$-terms.*

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. More on this will follow. Before we do so however, we need to have a new look at the structure of terms taking into account the new substitution items. While doing so we complement the informal definition of terms given in 2.2 and give a number of definitions regarding certain substrings of terms.

Definition 3.2 *(items, segments)*
If ω is an operator and t a term, then $(t\omega)$ is an **item***.*
A concatenation of zero or more items is a **segment***.*[9]

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 3.3) 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 3.3 *(variables, operators, terms, items, segments)*
$\mathcal{V} = \varepsilon \mid 1 \mid 2 \mid \ldots$
$\mathcal{O} = \delta \mid \lambda \mid \sigma \mid \ldots$
$\mathcal{T} = \mathcal{V} \mid \mathcal{I}\ \mathcal{T}$
$\mathcal{I} = (\mathcal{T}\ \mathcal{O})$
$\mathcal{S} = \emptyset \mid \mathcal{I}\ \mathcal{S}$

We define a number of concepts connected with terms, items and segments.

Definition 3.4 *(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: $t \equiv s_1 \ldots s_n x$. 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: $\overline{s} \equiv s_1 \ldots s_n$; 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* **σ-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.*

[9]In [de Bruijn 9x] an item is called a *wagon* and a segment is called a *train*.

*If a segment consists of a concatenation of an ω_1-item up to an ω_n-item, $\omega_i \in \Omega$, 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.*

Example 3.5 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, but a δ-λ-segment. Also, $\overline{s}$ is a λ-δ-λ-segment, which is a main segment of t.

Definition 3.6 *(body, end variable, end operator) Let $t \equiv \overline{s}x$ be a term. Then we call $\overline{s}$ the* **body** *of t, denoted* body(t)*, and x the* **end variable** *of t, or* endvar(t)*. It follows that $t \equiv$* body(t) endvar(t).

Let $s \equiv (t\omega)$ be an item. Then we call t the **body** *of s, denoted* body(s)*, and ω the* **end operator** *of s, or* endop(s)*. Hence, it holds that $s \equiv$* (body(s) 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.

Definition 3.7 is an abstract formulation of the above definition.

Definition 3.7 *(body, end variable, body, end operator)*

$$\texttt{body}(t) = \begin{cases} \emptyset & \textit{if } t \in \mathcal{V} \\ i\ \texttt{body}(t') & \textit{if } t = it' \textit{ for some } i \in \mathcal{I},\ t' \in \mathcal{T} \end{cases}$$

$$\texttt{endvar}(t) = \begin{cases} t & \textit{if } t \in \mathcal{V} \\ \texttt{endvar}(t') & \textit{if } t = it' \textit{ for some } i \in \mathcal{I},\ t' \in \mathcal{T} \end{cases}$$

$$\texttt{body}(i) = t \textit{ if } i = (t\omega) \textit{ for some } t \in \mathcal{T},\ \omega \in \mathcal{O}$$
$$\texttt{endop}(i) = \omega \textit{ if } i = (t\omega) \textit{ for some } t \in \mathcal{T},\ \omega \in \mathcal{O}$$

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.[10]

3.3 Step-wise substitution

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.

[10]See also [Nederpelt 90].

Definition 3.8 *(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:)
$(t_1\sigma^{(i)})(t_2\lambda) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\lambda)$ *($\sigma_{0\lambda}$-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)*
(σ-destruction rules:)
$(t_1\sigma^{(i)})i \rightarrow_\sigma \mathrm{ud}^{(i)}(t_1)$
$(t_1\sigma^{(i)})x \rightarrow_\sigma x$ *if* $x \neq i$.

The following details about these rules are to be noted. Firstly, in the σ-generation rule the so-called **δ-λ-segment** or **reduceable 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 reduceable 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 reduceable 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: the σ-item can move *inside* the item met (upwards in the tree; the cases σ_0), it can jump *over* the item (to the right in the tree; σ_1), or do both things at the same time (σ_{01}). 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).

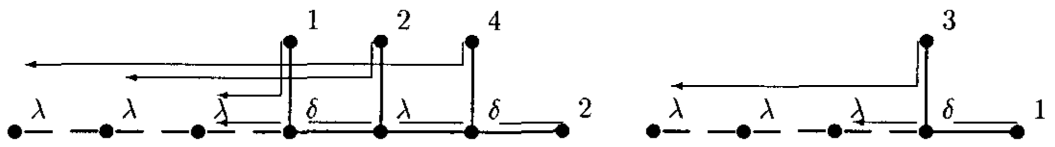
Thirdly, the σ-destruction rules apply when the σ-item has reached a leaf of the tree. When the index i of the σ is in accordance with the value of the variable, then we have met an intended occurrence of the variable; the substitution of t_1 for i takes place, accompanied with an updating (ud) of the 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** $\mathrm{ud}^{(i)}$ 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 examples below demonstrate how σ-reduction works.

Example 3.9 Let us take example 2.9 and see how σ-reduction works here too. There are 3 cases to consider, depending on the choice concerning the σ-transition rules.

case 1 (using σ_{0_ω}-transition rules)
$(1\delta)(2\lambda)(4\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(4\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)((1\ \sigma^{(1)})4\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(4\delta)1$
case 2 (using σ_{1_ω}-transition rules)
$(1\delta)(2\lambda)(4\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(4\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(4\delta)(1\ \sigma^{(1)})1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(4\delta)\mathrm{ud}^{(1)}(1) \rightarrow_\sigma$
$(1\delta)(2\lambda)(4\delta)2.$
case 3 (using σ_{01_ω}-transition rules)
$(1\delta)(2\lambda)(4\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(1\ \sigma^{(1)})(4\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)((1\ \sigma^{(1)})4\delta)(1\ \sigma^{(1)})1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(4\delta)\mathrm{ud}^{(1)}(1) \rightarrow_\sigma$
$(1\delta)(2\lambda)(4\delta)2.$

The first case is looping, 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 4).



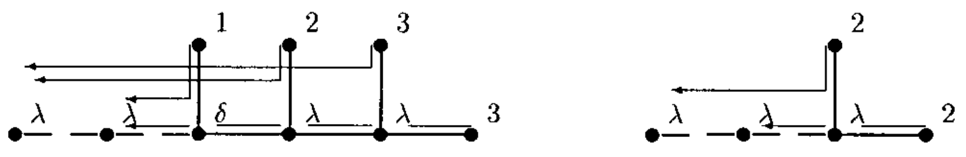
$(1\delta)(2\lambda)(4\delta)2$ $(3\delta)1$

Figure 4: σ-reduction when a σ-item meets a δ-item

Example 3.10 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 item notation this is $(1\delta)(2\lambda)(3\lambda)2$ and in sugared notation, it is: $(u\delta)(z\lambda_y)(z\lambda_x)y$ (see figure 5). This term reduces to $\lambda_{x:z}.u$ or in term notation $(2\lambda)2$ and in sugared notation $(z\lambda)u$. σ-reduction on this term results in the following 3 cases.

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

$(1\delta)(2\lambda)((1\,\sigma^{(1)})3\lambda)2 \rightarrow_\sigma$
$(1\delta)(2\lambda)(3\lambda)2$
case 2
$(1\delta)(2\lambda)(3\lambda)2 \rightarrow_\sigma$
$(1\delta)(2\lambda)(1\,\sigma^{(1)})(3\lambda)2 \rightarrow_\sigma$
$(1\delta)(2\lambda)(3\lambda)(1\,\sigma^{(2)})2 \rightarrow_\sigma$
$(1\delta)(2\lambda)(3\lambda)\mathrm{ud}^{(2)}(1) \rightarrow_\sigma$
$(1\delta)(2\lambda)(3\lambda)3.$
case 3
$(1\delta)(2\lambda)(3\lambda)2 \rightarrow_\sigma$
$(1\delta)(2\lambda)(1\,\sigma^{(1)})(3\lambda)2 \rightarrow_\sigma$
$(1\delta)(2\lambda)((1\,\sigma^{(1)})3\lambda)(1\,\sigma^{(2)})2 \rightarrow_\sigma$
$(1\delta)(2\lambda)(3\lambda)\mathrm{ud}^{(2)}(1) \rightarrow_\sigma$
$(1\delta)(2\lambda)(3\lambda)3.$
Again the first case is looping, the second and third cases are similar to those of example 3.9. That is $(1\delta)(2\lambda)$ is useless and once we remove it, we should decrease the free variables in $(3\lambda)3$ obtaining $(2\lambda)2$ (see figure 5).



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

Figure 5: σ-reduction when a σ-item meets a λ-item

We note that our updating is less complicated, but also less general than in the original treatment of de Bruijn-indices (see [de Bruijn 72]), 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 [Abadi et al. 90], 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 (see case 2 of example 3.10).

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

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 variable that has to be replaced by t_1; one can also say: i is the number of λ's that term t_1 has passed by on his way from the reduceable segment to the leaf in question (i = '**increment**'). Index k counts the λ's that are internally passed by in t_1 (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 $\mathrm{ud}^{(i)}(t_1)$, we write $(\varphi^{(0,i)})t_1$. We extend our set $\Omega_{\lambda\delta\sigma}$ with a set of φ-operators Ω_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.

Our terms are now $\Omega_{\lambda\delta\sigma\varphi}$-terms.

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* $x > k$
$(\varphi^{(k,l)})x \to_\varphi x$ *if* $x \leq k$ *or* $x \equiv \varepsilon$.

There are 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).

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.

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 $= \varphi^{(0,1)}$).

Now the rules for σ-items can be adapted as follows (cf. Definition 3.8):

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:)
$(t_1\sigma^{(i)})(t_2\lambda) \to_\sigma ((t_1\sigma^{(i)})t_2\lambda)$ *($\sigma_{0\lambda}$-transition)*
$(t_1\sigma^{(i)})(t_2\lambda) \to_\sigma (t_2\lambda)((\varphi)t_1\sigma^{(i+1)})$ *($\sigma_{1\lambda}$-transition)*
$(t_1\sigma^{(i)})(t_2\lambda) \to_\sigma ((t_1\sigma^{(i)})t_2\lambda)((\varphi)t_1\sigma^{(i+1)})$ *($\sigma_{01\lambda}$-transition)*
$(t_1\sigma^{(i)})(t_2\delta) \to_\sigma ((t_1\sigma^{(i)})t_2\delta)$ *($\sigma_{0\delta}$-transition)*
$(t_1\sigma^{(i)})(t_2\delta) \to_\sigma (t_2\delta)(t_1\sigma^{(i)})$ *($\sigma_{1\delta}$-transition)*
$(t_1\sigma^{(i)})(t_2\delta) \to_\sigma ((t_1\sigma^{(i)})t_2\delta)(t_1\sigma^{(i)})$ *($\sigma_{01\delta}$-transition)*
(general σ-destruction rules:)
$(t_1\sigma^{(i)})i \to_\sigma t_1$
$(t_1\sigma^{(i)})x \to_\sigma x$ *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.

Example 3.14 Let us go through example 3.10 but using φ-reduction.

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

It is not hard to see that this definition gives the same results as Definition 3.8 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.

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 $(\varphi)\dots(\varphi)$ (i times) $= (\varphi)^i = (\varphi^{(i)})$.

We can make a few more remarks in this respect.

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.

A second 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 3.15 *($\sigma_{01\sigma}$-transition)*
$(t_1\sigma^{(i)})(t_2\sigma^{(k)}) \rightarrow_\sigma ((t_1\sigma^{(i)})t_2\sigma^{(k)})(t_1\sigma^{(i)})$ *if* $i \neq k$

However, as we mentioned already in Subsection 3.3, there is one serious objection against this definition: it can easily give rise to an infinite series of transitions, for example when two σ-items keep playing leap-frog.

In the rest of this paper, we shall consider $\Omega_{\lambda\delta\sigma\varphi}$-terms and the *general* step-wise substitution as introduced in this section, unless otherwise stated.

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 reduceable segment is not removed. We still have to supply the tools for eliminating useless reduceable segments. In this section we explain how reduceable segments are removed and how local and global β-reduction are obtained.

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 3.16 Take the term $(\lambda_{x:y}.xx)u$. 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
$(1\delta)(2\lambda)(1\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)((\varphi)1\,\sigma^{(1)})(1\delta)1 \rightarrow_\sigma$

$(1\delta)(2\lambda)(((\varphi)1\,\sigma^{(1)})1\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)((\varphi)1\delta)1 \rightarrow_\varphi$
$(1\delta)(2\lambda)(2\delta)1$
case 2
$(1\delta)(2\lambda)(1\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)((\varphi)1\,\sigma^{(1)})(1\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(1\delta)((\varphi)1\,\sigma^{(1)})1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(1\delta)(\varphi)1 \rightarrow_\varphi$
$(1\delta)(2\lambda)(1\delta)2.$
case 3
$(1\delta)(2\lambda)(1\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)((\varphi)1\,\sigma^{(1)})(1\delta)1 \rightarrow_\sigma$
$(1\delta)(2\lambda)(((\varphi)1\,\sigma^{(1)})1\delta)((\varphi)1\,\sigma^{(1)})1 \twoheadrightarrow_{\sigma,\varphi}$
$(1\delta)(2\lambda)(2\delta)2$
Case one comes from using σ_{0_ω}-transition and is the local substitution for the second x in xx resulting in $(\lambda_{x:y}xu)u$. Case 2 comes from using σ_{1_ω}-transition and is the local substitution of the first x resulting in $(\lambda_{x:y}ux)u$. The third case comes from using σ_{01_ω}-transition and is the global substitution resulting in $(\lambda_{x:y}.uu)u$ which should of course be rewritten as uu (we still have not removed useless segments). That is: the reduceable 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.

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. 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. Let us hence distinguish four kinds of step-wise β-reduction:

local, minimal: Choose at each step the appropriate σ_0- or σ_1-rule.

local, maximal: Always take σ_{01}; restrict the σ-destruction rules.

global, minimal: Choose at each step the appropriate σ_0-, σ_1- or σ_{01}-rule.

global, maximal: Always take σ_{01}.

Of course, there are many intermediate possibilities between minimal and maximal reduction, both for local and for global β-reduction. There also 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 an $\Omega_{\lambda\delta}$-term consists of one σ-generation and a local reduction as described above, with either minimal or maximal strategy, executed until the σ in question (and the corresponding φ's) have disappeared. Cases 1 and 2 of example 3.16 are instances of a one-step local β-reduction. A one-step global β-reduction is defined analogously. Case 3 of example 3.16 is an instance of a one-step global reduction. Note that, in both cases, the reduceable 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. This possibility of different λ's and/or δ's was incorporated in our definition of terms in Subsection 2.2; it was, however, not yet used.

For example, we could use $\lambda_{\mathtt{loc}}$ for a future destination in local reductions and $\lambda_{\mathtt{glo}}$ for global reductions. A "definition" then could be rendered as a δ-λ-segment $(t_1\delta_{\mathtt{loc}})(t_2\lambda_{\mathtt{loc}})$, ready for local reduction. A "function" could start with a λ-item $(t_2\lambda_{\mathtt{glo}})$, whereas an "argument" for this function could have the form of a δ-item $(t_1\delta_{\mathtt{glo}})$.[11]

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

Definition 3.17 *(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* $i = \mathtt{loc}, \mathtt{glo}$.

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

Now we come to the issue of how to remove the useless reduceable 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 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 reduceable segment and an occurrence of one goal-variable in view, connected by means of a path in the tree. Hence we know that the reduceable 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 reduceable segment may be "without customers". Then σ-generation is undesirable (especially in the "maximal"-versions) 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 reduceable 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 3.18 *(void β-reduction)*
Assume that a δ-λ-segment $\overline{s}$ occurs in an $\Omega_{\lambda\delta}$-term t, where the final operator λ of $\overline{s}$ does not bind any variable in t. Let t_1 be the scope of $\overline{s}$. Then t reduces to the term t', obtained from t by removing $\overline{s}$ and replacing t_1 by $(\varphi^{(-1)})t_1$.

[11]See [Nederpelt 90] for an explanation of these notions "definition", "function" and " argument" with respect to typed lambda calculus.

Notation: $t \to_\emptyset t'$.[12]

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. Moreover, φ-items lose the property of additivity in general because of this negative exponent. For example, $(\varphi^{(1,1)})(\varphi^{(1,-1)})$ is not equal to $(\varphi^{(1,0)})$ = identity, since e.g. $(\varphi^{(1,1)})(\varphi^{(1,-1)})(1\delta)2 \to_\varphi (\varphi^{(1,1)})(1\delta)1 \to_\varphi (1\delta)1 \not\equiv (1\delta)2$. However, by the voidness of the δ-λ-segment generating a φ-item with negative exponent, such a situation cannot occur.

This discussion might still be a bit unclear. We shall give an example which demonstrates how void segments can disappear.

Example 3.19 Let us take example 3.9. After σ-reduction we obtained $(1\delta)(2\lambda)(4\delta)2$. 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 $(3\delta)1$.

Now we can describe the usual one-step β-reduction:

Definition 3.20 *(one-step β-reduction)*
One-step β-reduction of an $\Omega_{\lambda\delta}$-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 (either minimal or maximal), followed by a number of destructions, until again an $\Omega_{\lambda\delta}$-term is obtained (hence without σ- or φ-items).
Finally, there follows one void β-reduction for the disposal of $\overline{s}$.

Notation 3.21 As usual, we denote one-step β-reduction by $t \to_\beta t'$, and (ordinary) β-reduction — its reflexive and transitive closure — by $t \twoheadrightarrow_\beta t'$.

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

In [Abadi et al. 90], 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

[12]This reduction was introduced in [Nederpelt 73], 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 [de Bruijn 87].

number of 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 $\{1/1, 2/2, \ldots\}$ — [Abadi et al. 90] introduces three other index manipulations:[13]

- $\uparrow$ (*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:
  $\uparrow \circ (a \cdot s) = \{(a \cdot s)(\uparrow(i))/i\} = \{(a \cdot s)(i+1)/i\} = \{s(i)/i\} = s$.

With the help of our system, we can give a soundness proof for the equality axioms in [Abadi et al. 90]. 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 $(t_i\sigma^{(\vec{\imath})})$. The "vector" upper index $(\vec{\imath})$ abbreviates a universal quantification. By $(t_i\sigma^{(\vec{\imath})})$ 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, $(t_i\sigma^{(\vec{\imath}>1)})$ denotes the same as $\{t_2/2, t_3/3, \ldots\}$, and so on.

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

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

Let a be a term, $\lfloor s \rfloor = (t_i\sigma^{(\vec{\imath})})$ and $\lfloor s' \rfloor = (t'_i\sigma^{(\vec{\imath})})$. Then:

- $\lfloor id \rfloor = (i\sigma^{(\vec{\imath})})$,
- $\lfloor \uparrow \rfloor = ((i+1)\sigma^{(\vec{\imath})})$,
- $\lfloor a \cdot s \rfloor = (a\sigma^{(1)}) \oplus (t_{i-1}\sigma^{(\vec{\imath}>1)})$ and
- $\lfloor s \circ s' \rfloor = (t'_j\sigma^{(\vec{\jmath})})(t_i\sigma^{(\vec{\imath})})$.

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

Moreover, it will be clear that (φ) — recall definition 3.12 — and $\uparrow$ (or $(\, i+1\ \sigma^{(\vec{\imath})})$) have the same effect. The same holds, in general, for $(\varphi^{(k,l)})$ and $(\, i+l\ \sigma^{(\vec{\imath}>k)})$.

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

[13]The examples are taken from [Abadi et al. 90]. Note how the operations can be used for algebraic manipulations.

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 [Abadi et al. 90], 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 ans 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 [Abadi et al. 90] 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 as follows:
$(\varphi^{(-1)})((\varphi)t_1\sigma^{(1)}) =$
$(\ i-1\ \sigma^{(\bar{\imath})})((\ j+1\ \sigma^{(\bar{\jmath})})t_1\sigma^{(1)}) =$
(differentiate between the effect of this substitution on index 1 and on indices > 1, respectively)
$(\ i-1\ \sigma^{(\bar{\imath})})((\ j+1\ \sigma^{(\bar{\jmath})})t_1\sigma^{(1)}) \oplus (\ i-1\ \sigma^{(\bar{\imath}>1)}) =$
(since, as noted above: $(t_i\sigma^{(\bar{\imath})})(t'_j\sigma^{(\bar{\jmath})}) = ((t_i\sigma^{(\bar{\imath})})t'_j\sigma^{(\bar{\jmath})}))$
$((\ i-1\ \sigma^{(\bar{\imath})})(\ j+1\ \sigma^{(\bar{\jmath})})t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)}) =$
(by additivity, which holds in this case)

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

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

- *VarCons:*
  $\boldsymbol{|}1[a \cdot s]\boldsymbol{|} = \boldsymbol{|}(a \cdot s)1\boldsymbol{|} = ((a\sigma^{(1)}) \oplus (t_{i-1}\sigma^{(\bar{\imath})}))1 \rightarrow_\sigma a$.

- *Abs:*
  $\boldsymbol{|}(\lambda a)[s]\boldsymbol{|} = (t_i\sigma^{(\bar{\imath})})(\lambda)a \rightarrow_\sigma (\lambda)((\varphi)t_{i-1}\sigma^{(\bar{\imath}>1)})a$,
  since $(t_i\sigma^{(i)})(\lambda)a \rightarrow_\sigma (\lambda)((\varphi)t_i\sigma^{(i+1)})a$ for each i;
  $\boldsymbol{|}\lambda(a[1 \cdot (s\circ \uparrow)]\boldsymbol{|} = (\lambda)((1\sigma^{(1)}) \oplus ((\varphi)t_{i-1}\sigma^{(\bar{\imath}>1)}))a =$
  $= (\lambda)((\varphi)t_{i-1}\sigma^{(\bar{\imath}>1)})a$,
  since $\boldsymbol{|}s\circ \uparrow\boldsymbol{|} = ((\ j+1\ \sigma^{(\bar{\jmath})})t_i\sigma^{(\bar{\imath})}) = ((\varphi)t_i\sigma^{(\bar{\imath})})$.

- *SCons:*
  $\boldsymbol{|}1[s] \cdot (\uparrow \circ s)\boldsymbol{|} = ((t_i\sigma^{(\bar{\imath})})1\sigma^{(1)}) \oplus ((t_j\sigma^{(\bar{\jmath})})i\sigma^{(\bar{\imath}>1)}) =$
  $= (t_1\sigma^{(1)}) \oplus (t_i\sigma^{(\bar{\imath}>1)}) = (t_i\sigma^{(\bar{\imath})}) = \boldsymbol{|}s\boldsymbol{|}$.

- *Beta:*
  The traditional rule of β-reduction has the following form in our system
  $(t_1\delta)(t_2\lambda) \rightarrow_\sigma (\varphi^{(-1)})((\varphi)t_1\sigma^{(1)})$.
  This enables us directly to derive the translation of the *Beta*-rule:
  $\boldsymbol{|}(\lambda a)b\boldsymbol{|} = (b\delta)(\lambda)a \rightarrow_\sigma (\varphi^{(-1)})((\varphi)b\sigma^{(1)})a$;
  $\boldsymbol{|}a[b \cdot id]\boldsymbol{|} = ((b\sigma^{(1)}) \oplus (\ i-1\ \sigma^{(\bar{\imath}>1)}))a =$
  $((b\sigma^{(1)}) \oplus (\varphi^{(1,-1)}))a$.
  Hence, $(\lambda a)b = a[b \cdot id]$ from the equation shown above.

5 Conclusions

In this paper 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 as 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.

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. The step-wise character of the corresponding reduction relation and of many other 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.

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 reduceable segments, we keep them present until they are no longer necessary, then we get rid of them using the notion of void β-reduction.

It is our conviction that the step-wise substitution as introduced in this paper is easier and more manegeable than proposals for explicit substitution that have recently been given in the literature (see e.g. [Abadi et al. 90]). Our approach is very close to intuition, yet the formulation remains simple.

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 and we interpreted [Abadi et al. 90] in this system. The advantages of the new notation do not stop at substitution, but extend to all branches of the λ-calculus. The linear 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.

References

[Abadi et al. 90] Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J., Explicit Substitutions, Rapports de Recherche no. 1176, INRIA, Le Chesnay, 1990 (Also appeared in the *Journal of Functional Programming 1 (4)*, pp. 375-416, 1991).

[de Bruijn 70] Bruijn, N.G. de, 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, pp. 29-61.

[de Bruijn 72] Bruijn, N.G. de, Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, *Indagationes Math. 34, No 5*, 1972, pp. 381-392.

[de Bruijn 80] Bruijn, N.G. de, 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, pp. 29-61.

[de Bruijn 87] Bruijn, N.G. de, 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, 1987, pp. 71-92.

[de Bruijn 9x] Bruijn, N.G. de, Algorithmic definition of lambda-typed lambda calculus. In preparation.

[Nederpelt 73] Nederpelt, R.P., *Strong normalisation in a typed lambda calculus with lambda structured types*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, 1973.

[Nederpelt 90] Nederpelt, R.P., Type systems — basic ideas and applications, in: *CSN '90, Computing Science in the Netherlands 1990*, Stichting Mathematisch Centrum, Amsterdam, 1990.

[Peyton Jones 87] Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice-Hall, Englewood Cliffs, 1987.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.