TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

A useful lambda notation

Citation for published version (APA):
Nederpelt, R. P., & Kamareddine, F. (1992). *A useful lambda notation*. (Computing science notes; Vol. 9222). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1992

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 16. Nov. 2023

Eindhoven University of Technology

Department of Mathematics and Computing Science

A useful lambda notation

by

Rob Nederpelt Fairouz Kamareddine

92/22

Computing Science Note 92/22
Eindhoven, September 1992

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

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

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

A useful lambda notation

Rob Nederpelt
and
Fairouz Kamareddine

Department of Mathematics and Computing Science
Eindhoven University of Technology
Eindhoven, the Netherlands

September 15, 1992

Abstract

Lambda Calculus is fundamental for the foundation of Logic, Mathematics, Computer and Cognitive Science. This makes it indispensable to formalise the Lambda Calculus in a way which avoids most of the complications associated with, among others, substitution, variable renaming, the search of bound and free occurrences of variables. Combinatory Logic could be seen as an attempt to do so, but is not as intuitive as the lambda calculus. This paper provides a new formulation of the λ-calculus. Such a formulation is shown to be useful for term and variable manipulation and for locating the type of a term. This will have advantages on all areas of the λ-calculus including substitution (global and local), unification and typing.

Keywords: *Lambda Calculus, Term Restriction, Types, Bound and Free Variables*

1 Introduction

As a discipline, lambda calculus started with Church in the forties, when he tried to give a foundation for mathematics. In the following decades, the development of lambda calculus was in the hands of a few specialists, such as Curry, Hindley, Seldin and Böhm. Despite the important work that was carried out, lambda calculus remained a rather isolated branch of logic. Major results were only valued at their true worth within a small community.

In the beginning of the sixties, there arose a new interest in lambda calculus from the side of computer science, where functional programming techniques like McCarthy's LISP borrowed lambda calculus concepts. Since that time lambda calculus inspired theoretical computer science and vice versa. The breakthrough became permanent when, in 1981, Barendregt published the standard work on the (untyped) lambda calculus ([Bar84]). This presentation of an extensive and impressive amount of knowledge was very influential.

In the present time, there is a remarkable revival of lambda calculus, especially in the versions which use types. Recently, both logicians and computer scientists have developed several branches of typed and untyped lambda calculus. Also mathematics has benefited from lambda calculus, especially since the time (around 1970) where de Bruijn used his lambda calculus-based Automath for the analysis and checking of mathematical texts (see [deB70] or [deB80]).

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, formalising the application of a function to an argument.

The relations (reductions) in lambda calculus are meant to formalise a connection between certain lambda terms that are calculationally comparable. "Calculus" is here meant to be an abstract form of function application, just as the function "plus" applied to the numbers 12 and 17 gives 29 as a calculational result.

Based on these observations, we start in Section 2 with the investigation of the basic construction principles of lambda terms, by comparing these principles with general term

construction methods in logic and mathematics. In a natural manner, we find a close correspondence with well-known tree structures. A slight change in notation enables us to construct lambda terms in a modular way, in accordance with the demands and needs of a mathematical entourage. That is to say, in our approach it is easy to develop a lambda term step by step, thereby accurately reflecting the construction of some text in mathematics, logic or computer science.

This step-by-step approach, employed throughout this paper, is fundamental for the *fine-structure* of lambda calculus which we pursue.

As an alternative to the use of variables, in this paper we will be using de Bruijn-indices. These are natural numbers that do not suffer from the usual problems with variable names (the danger of "clash of variables", the need for appropriate renaming, etc.).

The new notation introduced in section 2 is very advantageous and should be seen as an alternative to the usual λ-calculus notation. We claim that this new formulation can avoid many of the complications associated with the old formulation. For the purpose of this paper, we aim to show the usefulness of the new notation for variable and term manipulation and for typing. This will be done in section 3.

2 Term formation

2.1 De Bruijn's indices

In the type free λ-calculus, we have the following three ways of forming terms:

$t ::= x \mid (\lambda_x.t) \mid (t_1 t_2)$.[1]

The basic axiom of such a calculus is the following:

(β) $\qquad (\lambda_x.t_1)t_2 = t_1[x := t_2]$

where substitution has been defined in a way which deals with the problem of variable clashes. For example, $(\lambda_x.\lambda_y.xy)y = (\lambda_y.xy)[x := y] = \lambda_z.xy[y := z][x := y] = \lambda_z.xz[x := y] = \lambda_z.yz$. As it can be seen from this example, the y in $\lambda_y.xy$ had to be renamed to z before we could substitute y for the free occurrences of x in $\lambda_y.xy$; otherwise the y replacing x would have become bound when it should not be. This process gets more complicated and cumbersome when we work with more involved λ-terms.

De Bruijn in [deB72] proposes a solution to such a problem by the use of *indices* instead of variables. Moreover, in this manner he avoids α-conversion, where α-conversion is given by the axiom:

(α) $\qquad \lambda_x.t = \lambda_y.t[x := y]$ for y not free in t.

That is terms such as $\lambda_x.x$ and $\lambda_y.y$ are the "same", and the use of x, y or any other variable does not change the semantic meaning of the function denoted by this term (the identity function). The identity function using indices (called de Bruijn's indices) will be denoted by $\lambda.1$. 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).

De Bruijn's notation can also be used for the typed λ-calculus. For instance the identity function above could have been identity over a particular type y (let us say) written as $\lambda_{x:y}.x$.

[1] Parenthesis are omitted if no confusion can arise.

In such a case y is a free variable and the function is denoted by: (λ1.1). The free variable y in the typed lambda term is translated into the first number 1. Such a number refers in this case to an "invisible" lambda that is not present in the term, but may be thought of to *proceed* the term, binding the free variable.[2] The number 1 next to the λ tells us how many λs we have to count from (and excluding[3]) this λ. (The variable x, as before, is translated in the second number 1.)

To demonstrate how β-reduction works in this notation we consider the term $(\lambda_{x:z}.(xy))u$ which β-reduces to uy. Under the assumption that the free variable list is $\lambda_y, \lambda_z, \lambda_u$, this reduction using de Bruijn's indices can be represented as: (λ2.1 4)1 reduces to 1 3. Here the contents of the subterm 1 4 changes: 4 becomes 3. This is due to the fact that λ2, 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.

The notation that we will introduce in the following section makes use of de Bruijn's indices but assumes a linear representation of terms 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 term and variable manipulation, and types in detail in this paper.

2.2 The new notation

Let us look again at the syntax of λ-terms given in 2.1. If we forget variables (as we shall when we use de Bruijn's indices), then we begin with natural numbers and all that remains is *abstraction* and *application*. We shall consider these to be the basic *operations* on terms and shall use λ to refer to the first and δ to refer to the second. Note that both operators are binary. That is, in the typed λ-calculus, λ links a type to a term, (think of $\lambda_{x:y}.x$ which is λ1.1) and application links a function to an argument. We will use a typed λ-calculus notation which is also suitable to write type free terms. This will be done via our special index ε below.

Now our terms are application terms such as t applied to t' and abstraction terms such as $\lambda t.t'$. We shall not assume the uniqueness of the λ and the δ operators.[4] Rather we consider $\lambda, \lambda_1, \lambda_2, \ldots$ for abstraction, and $\delta, \delta_1, \delta_2, \ldots$ for application. We use $\omega, \omega_1, \omega_2, \ldots$ as meta-variables 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. As we decided to use indices instead of variables, we take Ξ the set of **variables** to be $\Xi = \{\varepsilon, 1, 2, \ldots\}$. Sometimes we will need to use actual variables, but this is not a part of our syntax. It is only a matter of simplifying the conversation. We use $x, x_1, y, \ldots$ to denote variables.[5] Using Ω and Ξ we define our terms (which we denote $t, t_1, \ldots$)

[2] If we had more than one free variable, we have to know which one comes before the other. For this, we assume an arbitrary, but fixed order so that these invisible lambda's form a **free variable list**.

[3] This technical peculiarity disappears in the new notation of section 2.2.

[4] This is to enable our system to be general enough to represent a whole variety of type systems. For example to accommodate second-order theories, we use λ_2 for λ and λ_1 for Λ. To accommodate Pure Type Systems we use λ_2 for Π and λ_1 for the ordinary λ. Moreover, these various λ's and δ's will be useful for stepwise substitution and lazy evaluation, which we pursue in another article.

[5] ε is a special variable that denotes the "empty term". It can be used for rendering ordinary (untyped)

to be those symbol strings obtained in the usual manner on the basis of Ξ, the operators in Ω and parentheses. That is our terms are the elements of $F_\Omega(\Xi)$, the free Ω-structure generated by Ξ. We call these terms $\Omega_{\lambda\delta}$-**terms** or simply **terms**.

We will defer from usual practice and use the operators in Ω as *infix* ones. That is we write $(t\delta t')$ for the *function* t' applied to the *argument* t (note the *reversed* order!) and write $(t\lambda t')$ for $(\lambda t.t')$. We go even further by using what we call **item**-notation where we place parentheses in an unorthodox manner: we write $(t_1\omega)t_2$ instead of $(t_1\omega t_2)$.

Examples of terms are: ε, 3, $(2\delta)(\varepsilon\lambda)1$, in item notation or $(2\delta(\varepsilon\lambda 1))$ in the original infix notation. (We assume that $\lambda \in \Omega_\lambda$ and $\delta \in \Omega_\delta$.)

Notation 2.1 *(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.

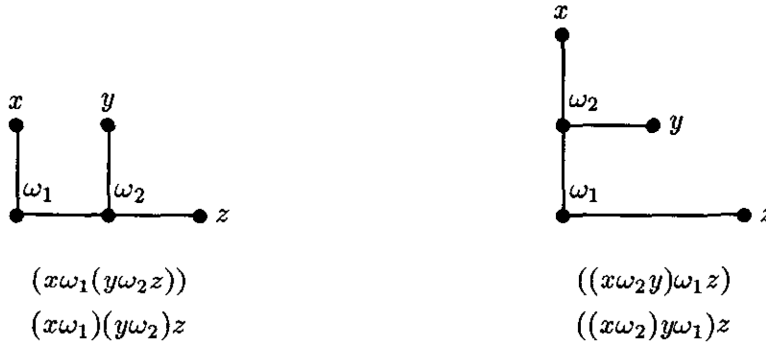

Figure 1: Term trees, with normal linear notation and item-notation

Notation 2.2 *(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 2.3. 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 2.3 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.

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)

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 upper-left side* of the branches of the tree. Only the λ's met do count, the δ's do not.

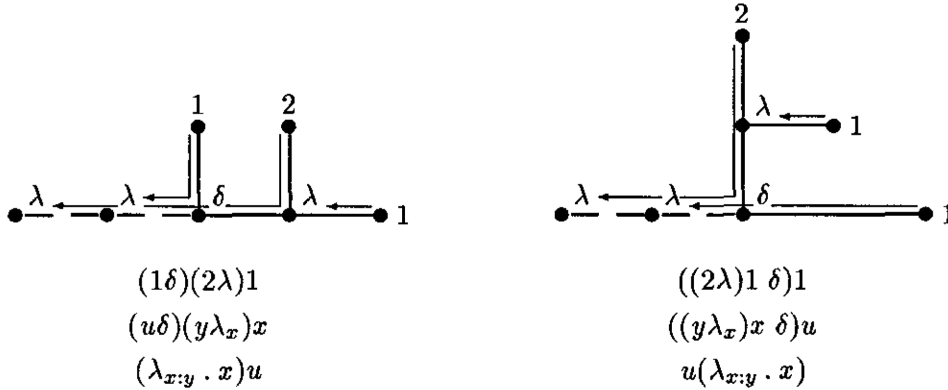

Figure 2: Term trees with explicit free variable lists and reference numbers

Example 2.4 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: $(u\delta)(z\lambda_x)$) is the signal for a possible β-reduction. The "unsugared" version reads: the term $(1\delta)(2\lambda)(4\delta)1$ reduces to $(3\delta)1$.

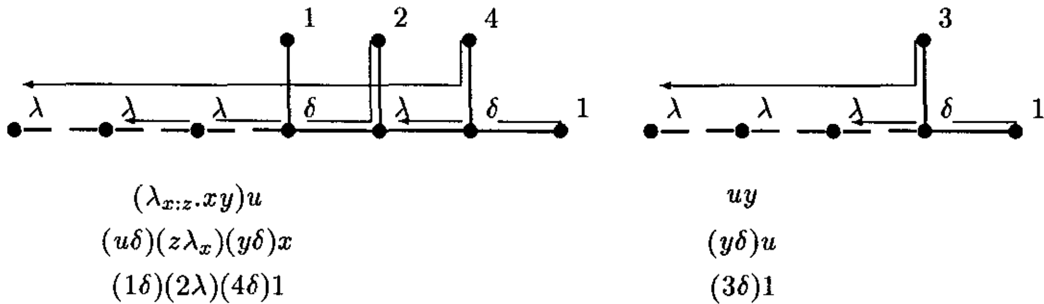

Figure 3: β-reduction in our notation

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.

Remark 2.5 With the help of ε we can construct terms without free variables, for example we can construct $(\varepsilon\lambda)(1\lambda)(1\delta)((2\lambda)(1\lambda)1\lambda)3$. We note that it may be profitable to use the empty term instead of ε, which allows us to write terms like $(\lambda)(1\lambda)2$ or even $(\lambda)(1\lambda)$, representing the typed lambda terms $\lambda_{y:\varepsilon}.\lambda_{x:y}.y$ and $\lambda_{y:\varepsilon}.\lambda_{x:y}.\varepsilon$, respectively. We shall use this convention in section 3.2, especially in the case of an item $(\varepsilon\omega)$, which we render as (ω), for different operators ω.

Remark 2.6 The presented way of describing typed lambda calculus is relatively easy to read. Another approach is to define a term in tree format, e.g. as a set S of pairs (β, ξ), where β is a finite sequence of zeros and ones and $\xi \in \Xi \cup \Omega$. The string β codes a root path in the binary tree, starting at the root. Each 'zero' in the string means: "go upwards and follow an edge until the next node", each 'one' in the string means: "go to the right and do the same". The ξ is the label connected with the final node of this path.[6] The notions to be defined in the following sections can also be expressed in this tree language.

There is one important advantage in using this kind of term *trees* instead of terms: one needs not bother about the *occurrences* of a variable or a subterm which are meant. In fact, the β of the pair (β, ξ) gives the exact location of ξ in the tree. Hence, in the case that ξ is a variable, the β fixes the occurrence of ξ; in the case that ξ is an operator, the β fixes the location of a subterm (subtree) with the mentioned ξ as its main operator.

In the rest of this paper, we use terms and not term trees. This causes some inconveniences, especially as regards these "occurrences". Apart from that, we prefer ordinary terms because they are easier to read than sets of pairs (β, ξ).

2.3 The inner structure of terms

In this section we give a number of definitions regarding certain substrings of terms.

First, we give a formal definition of *items* and *segments*.

Definition 2.7 *(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**.[7]

We use $\overline{s}, \overline{s}_1, \overline{s}_i, \ldots$ as meta-variables for segments.

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

Definition 2.8 *(main items, main segments, ω-items) 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 $\overline{s}$.

[6] The set S should have some obvious additional properties, such as prefix-closedness: if $(\beta, \xi) \in S$, then for all prefixes β' of β there must exist a ξ' such that $(\beta', \xi') \in S$.

[7] In [deB9x] an item is called a *wagon* and a segment is called a *train*.

An item $(t\ \omega)$ is called an ω-**item**. *Hence, we may speak about* λ-**items** *and* δ-**items**.

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**.[8]

A **context** *is a segment consisting of only* λ*-items.*

Example 2.9 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 2.10 *(body, end variable, end operator)*

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

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

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.

Example 2.11 In the previous example, $\overline{s}$ is the body of t and 1 is the end variable of t. Let s be the item $((1\delta)(\varepsilon\lambda)1\delta)$. Then $(1\delta)(\varepsilon\lambda)1$ is the body of s and δ the end operator of s.

By means of the following definition one can *sieve* the main items with certain end operator(s) from a given segment or term, forming a (new) segment:

Definition 2.12 *(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 2.13 In the term t of Example 2.9, $\mathtt{sieveseg}_\lambda(t) \equiv (\varepsilon\lambda)(2\lambda)$ and $\mathtt{sieveseg}_\delta(t) \equiv ((1\delta)(\varepsilon\lambda)1\delta)$.

For later use, we define different kinds of weight for segments and terms:

Definition 2.14 *(weight,* ω*-weight)*

The **weight** *of a segment* $\overline{s}$*,* $\mathtt{weight}(\overline{s})$*, is the number of main items that compose the segment.*

The weight of a term t *is the weight of* $\mathsf{body}(t)$*.*

The ω*-weight* $\mathtt{weight}_\omega(\overline{s})$ *of a segment* $\overline{s}$ *is the weight of* $\mathtt{sieveseg}_\omega(\overline{s})$*.*

Again, the ω*-weight of a term* t *is the* ω*-weight of* $\mathsf{body}(t)$*.*

Example 2.15 For the term $t \equiv (\varepsilon\lambda_x)(x\lambda_v)(x\delta)(\varepsilon\lambda_y)((x\lambda_z)y\delta)(y\lambda_u)u$ and the segment $\overline{s} \equiv (\varepsilon\lambda_x)(x\lambda_v)(x\delta)(\varepsilon\lambda_y)((x\lambda_z)y\delta)(y\lambda_u)$, $\mathtt{weight}(t) = \mathtt{weight}(\overline{s}) = 6$ and $\mathtt{weight}_\lambda(t) = \mathtt{weight}_\lambda(\overline{s}) = 4$.

[8] As noted before, an important case is that of a δ-λ-**segment**, being a δ-item immediately followed by a λ-item.

Next, we show how the relations *direct subterm* and *subterm*, denoted by the relation symbols $\subset$ and $\Subset$, can be concisely defined in our notation:

Definition 2.16 *(direct subterms, subterms)*

If $\mathsf{body}(t) \neq \emptyset$*, then* $t \equiv (t'\omega)t''$. *In this case we call* t' *and* t'' *the (left and right)* **direct subterms** *of* t. *We denote this by* $t' \subset t$ *and* $t'' \subset t$.

The relation $\Subset$ *is the reflexive and transitive closure of* $\subset$.

We say that t_1 *is a* **subterm** *of* t *iff* $t_1 \Subset t$.

Example 2.17 Let t be the term $((1\delta)2\lambda)(1\lambda)3$. The left direct subterm of t is $(1\delta)2$, the right direct subterm of t is $(1\lambda)3$. The subterms of t are $t, (1\delta)2, (1\lambda)3, 1$ (twice), 2 and 3.

When one says that t' is a subterm of t, one usually has a certain *occurrence* of t' in t in mind. (There can be more occurrences of t' in t.) The precise location of the occurrence meant has not been accounted for in the definition given above. This shortcoming can be mended by giving a third argument to $\subset$ and $\Subset$, being a code for the path leading from the root of t to the root of the t' meant (cf. Remark 2.6). See the following example.

Example 2.18 Let t be the term $((x\delta)(y\lambda_x)x\lambda_u)(z\delta)y$.

Then $x \Subset_{00} t$, $x \Subset_{011} t$ and $(y\lambda_x)x \Subset_{01} t$.[9]

However, we shall not use this way of describing the intended occurrence. If necessary, we shall "mark" an occurrence, e.g. with a small circle, $\circ$, or with under- or overlining. For example, the first occurrence of x in t (see Example 2.18) can be fixed by referring to it as x° in $((x^\circ\delta)(y\lambda_x)x\lambda_u)(z\delta)y$. And the occurrence of the subterm $(y\lambda_x)x$ in this t can be marked as $\underline{(y\lambda_x)x}$.[10] We can also mark the occurrence of an operator: $(y\lambda_x^\circ)x$.

In the following section we need a notion that relates (left and right) subterms to an operator:

Definition 2.19 *(arguments)*

Let $(t'\omega^\circ)t'' \Subset t$. *Then* t' *is the* **left argument** *of* ω° *in* t, *or* $\mathtt{leftarg}(\omega^\circ)$, *and* t'' *is the right argument of* ω° *in* t, *or* $\mathtt{rightarg}(\omega^\circ)$.

Hence, $\mathtt{leftarg}(\omega^\circ)$ is the left direct subterm of $(t'\omega^\circ)t''$ and $\mathtt{rightarg}(\omega^\circ)$ is the right direct subterm of $(t'\omega^\circ)t''$.

Note that a *maximal* subterm of a term t (i.e. a subterm that cannot be extended to the left in t) is either t itself or a *left* direct subterm of t and hence the left argument of some operator occurring in t.

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 (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 [Ned90]).

[9] Note that t_1 is a *direct* subterm of t if and only if $t_1 \Subset_a t$ for $a = 0$ or $a = 1$.

[10] In [deB9x], the occurrence of a subterm is called a *positioned* subterm.

3 The usefulness of the new term notation

The notation introduced in the previous section provides useful advantages related to many notions of the lambda calculus. In this section we study the usefulness of this notation to three notions; namely, term restriction, bound and free variables and term typing.

3.1 The restriction of a term

In the present section we explain how to derive the restriction $t \upharpoonright x$ of a term t to a variable occurrence x° in t. This restriction is itself a term, consisting of precisely those "parts" of t that may be relevant for this x°, especially as regards binding and typing.

When a variable x occurs in term t, then it is not the case that all the "information" contained in t is necessarily relevant for a specific occurrence x° of x in t. For example, in the term $(\varepsilon\lambda_x)(x\lambda_v)(x\delta)(\varepsilon\lambda_y)((x\lambda_z)y^\circ\delta)(y\lambda_u)u$, only the items $(\varepsilon\lambda_x)$, $(x\lambda_v)$, $(x\delta)$, $(\varepsilon\lambda_y)$ and $(x\lambda_z)$ are of importance for the variable occurrence y°. These items are all the items that can be found *to the left* of y°. In the traditional notation this is not the case; cf. the same term as above in the usual notation: $\lambda_{x:\varepsilon}.\lambda_{v:x}.(\lambda_{y:\varepsilon}.(\lambda_{u:y}.u)\lambda_{z:x}.y^\circ)x$.

In order to formalise this intuition we give the following definition.

Definition 3.1 *(envelope, dominator, one-step restriction, full restriction)*
Let x° be an occurrence of variable x in term t such that $x^\circ \not\equiv$ `endvar`(t). Then there is an operator occurrence ω° in t such that $x^\circ \equiv$ `endvar(leftarg(`ω°`))`. The term `leftarg(`ω°`)` is called the **envelope** *of x° or `env`(x°). The term `(leftarg(`ω°`)` ω°`) rightarg(`ω°`)` is called the* **dominator** *of x° or `dom`(x°).*
(Note that the tree of `dom`(x^0) is the subtree with ω° as its root and that `env`(x^0) is, in its turn, the "left direct subtree" of this subtree. See the example below.)
The **one-step restriction** *of t to x°, denoted $t \lceil x^\circ$, is:*
1. in case $x^\circ \not\equiv$ `endvar`(t): the term obtained from t by replacing `dom`(x°) by `env`(x°);
2. in case $x^\circ \equiv$ `endvar`(t): $t \lceil x^\circ \equiv t$.
The **(full) restriction** *of t to x°, denoted $t \upharpoonright x^\circ$, is the limit of the sequence $t_1, t_2, \ldots$, where $t_1 \equiv t$ and $t_{i+1} \equiv t_i \lceil x^\circ$.*

Example 3.2 Let t be the following term:

$$(\varepsilon\lambda_x)((x\lambda_u)((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y\lambda_v)u. \tag{1}$$

Then the envelope of x° is $t_1 \equiv (u\delta)(x\lambda_t)x^\circ$, since $t_1 \equiv$ `leftarg`(λ_y). Moreover, `rightarg`$(\lambda_y) \equiv (u\lambda_z)y$, so the dominator of x° is $t_2 \equiv ((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y$. See the underlining and the overlining in (2):

$$(\varepsilon\lambda_x)((x\lambda_u)\overline{(\underline{(u\delta)(x\lambda_t)x^\circ}_{\text{env}}\lambda_y)(u\lambda_z)y}^{\text{dom}}\lambda_v)u. \tag{2}$$

The replacement of t_2 ($\equiv$ `dom`(x°)) by t_1 ($\equiv$ `env`(x°)) gives the one-step restriction $t \lceil x^\circ$:

$$(\varepsilon\lambda_x)((x\lambda_u)(u\delta)(x\lambda_t)x^\circ\lambda_v)u. \tag{3}$$

The full restriction $t \upharpoonright x$ of the same x°, obtained after another one-step restriction, is:

$$(\varepsilon\lambda_x)(x\lambda_u)(u\delta)(x\lambda_t)x^\circ. \tag{4}$$

$$t \equiv (\varepsilon\lambda_x)((x\lambda_u)\overline{(\underline{(u\delta)(x\lambda_t)x^\circ}\lambda_y)(u\lambda_z)y}\lambda_v)u \equiv (\varepsilon\lambda)((1\lambda)\overline{(\underline{(1\delta)(2\lambda)3^\circ}\lambda)(2\lambda)4}\lambda)3$$

$$t \upharpoonright x^\circ \equiv (\varepsilon\lambda_x)\overline{(\underline{(x\lambda_u)(u\delta)(x\lambda_t)x^\circ}\lambda_v)u} \equiv (\varepsilon\lambda)\overline{(\underline{(1\lambda)(1\delta)(2\lambda)3^\circ}\lambda)3}$$

$$t \Upharpoonright x^\circ \equiv (\varepsilon\lambda_x)(x\lambda_u)(u\delta)(x\lambda_t)x^\circ \equiv (\varepsilon\lambda)(1\lambda)(1\delta)(2\lambda)3^\circ$$

Figure 4: A term and its restriction to a variable

Now it will be clear that it is very easy to obtain the full restriction $t \Upharpoonright x^\circ$ using our item-notation: just take the substring of string t from the beginning of t until x° and delete all unmatched opening parentheses. This is an advantage of our new notation.

It is illustrative to draw the tree of t (see figure 4) and to see what happens when the restriction process is executed with this tree. In the first one-step restriction in the example given above, the subtree corresponding with the subterm $(u\delta)(x\lambda_t)x^\circ$ is "pushed down" to the node formerly labeled λ_y, annihilating the rest of the subtree rooting in this node. The full restriction is the result of a continuation of this process. In figure 4, the intended occurrence of x° in the trees is denoted with an *open* circle.

Intuitively, the body of $t \Upharpoonright x^\circ$ is the only thing that matters for x° in t; the rest of (the tree of) the term t may be neglected, as far as the x° is concerned. As said before, this is essentially the importance of the restriction: $t \Upharpoonright x$ is a term with x as its end variable, that contains all "information" relevant for x. For example, when x is bound (see the following subsection), then the bond between x and the λ binding this x does not change in the process of restriction; i.e. corresponding variables x in the described sequence $t_1, t_2, \ldots$ refer to corresponding λ's (the number x does not change). So the λ binding this x can be found in $t \Upharpoonright x$; the same holds for the type of this x. Moreover, when x is a candidate for a substitution caused by a reduction, then the δ-λ-segment connected with this reduction can be found, again, in $t \Upharpoonright x$.

Full restriction is, of course, idempotent; more generally, the following holds: when y occurs in t, and x occurs in $t\upharpoonright y$, then $(t\upharpoonright y)\upharpoonright x \equiv t\upharpoonright x$.

The described notion 'restriction of a term to a variable' has an obvious generalisation: 'restriction of a term to a *subterm*':

Definition 3.3 *(restriction of a term to a subterm)*
Let $\underline{t_0}$ *be an occurrence of subterm* t_0 *in term* t. *Let* $x^\circ \equiv \mathtt{endvar}(\underline{t_0})$. *Then* $t\upharpoonright\underline{t_0}$, $t\upharpoonright\underline{t_0}$, $\mathtt{env}(\underline{t_0})$ *and* $\mathtt{dom}(\underline{t_0})$ *are defined as* $t\upharpoonright x^\circ$, $t\upharpoonright x^\circ$, $\mathtt{env}(x^\circ)$ *and* $\mathtt{dom}(x^\circ)$.

Note that a term $t\upharpoonright\underline{t_0}$ contains all "information" necessary for $\underline{t_0}$.

3.2 Bound and free variables

An important notion in lambda calculus is that of bound and free variables; for a bound variable the "binding place" is relevant. This can be defined as follows.

Definition 3.4 *(bound and free variables, type, open and closed terms)*
Let x° *be a variable occurrence in* t *such that* $x \not\equiv \varepsilon$ *and assume that* $\mathtt{sieveseg}_\lambda(t\upharpoonright x^\circ) \equiv s_m \ldots s_1$ *(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* $\mathrm{endop}(s_x)$. *The* **type** *of* x° *in* t *is* $\mathrm{body}(s_x)$. *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 3.5 The term $t \equiv (\varepsilon\lambda_x)(x\lambda_v)(x\delta)(\varepsilon\lambda_y)((x\lambda_z)y^\circ\delta)(y\lambda_u)u$ becomes, in the notation with de Bruijn- indices: $t \equiv (\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2^\circ\delta)(1\lambda)1$. Now $t\upharpoonright 2^\circ \equiv (\lambda)(1\lambda)(2\delta)(\lambda)(3\lambda)2^\circ$. So $\mathtt{sieveseg}_\lambda(t\upharpoonright 2^\circ) \equiv s_4 s_3 s_2 s_1 \equiv (\lambda)(1\lambda)(\lambda)(3\lambda)$. Hence, 2° is bound in t since $2 \leq \mathtt{weight}_\lambda(t\upharpoonright 2^\circ) = 4$. Moreover, the type of 2° in t is $\mathrm{body}(s_2) \equiv \varepsilon$.
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 (one-step or more-step) restriction does not affect whether a variable occurrence is free or bound.

There is a simple procedure for finding the variable occurrences bound by a certain λ in a term t. In the following definition this procedure is given as a step-by-step search.

For this purpose, we temporarily extend the language with a special **search item** or ζ-item and with a new relation, $\rightarrow_\zeta$, between (extended) terms. (We may speak of $\Omega_{\lambda\delta\zeta}$-terms when referring to these extended terms.)

We write the search item as an *indexed* item: $(\zeta^{(i)})$, with index i. This index serves for the identification of the proper variable occurrences, as turns out below. The ζ-operation is binary, just as λ and δ, but since a ζ-item always has ε as its body, one may also consider it to be a unary, prefix operator.

The search begins with the generation of a ζ-item, just behind the λ-item in question. Thereupon this ζ-item is pushed through all subterms of the term "in the scope of" the λ-item. The index (i), initialized on 1, increases with 1 whenever the ζ-item "passes" a λ. When ending at a variable x, the index i of the ζ-item decides whether x is bound by the λ

of the above-mentioned λ-item, or not. If this is the case, then the variable is capped with the symbol $\hat{}$.

The rules are given as a relation between terms, but in a compact format. Rule $\overline{s} \rightarrow_\zeta \overline{s'}$ states that the relation $t \rightarrow_\zeta t'$ holds for terms t and t' when a segment $\overline{s}$ occurs in t and t' is obtained by replacing $\overline{s}$ by $\overline{s'}$ in t. (Otherwise said: it is assumed that rules of so-called "compatibility" or "monotonicity" have been added.)

Definition 3.6 *(ζ-reduction)*
(ζ-generation rule:)
$(t_1\lambda) \rightarrow_\zeta (t_1\lambda)(\zeta^{(1)})$
(ζ-transition rules:)
$(\zeta^{(i)})(t'\lambda) \rightarrow_\zeta ((\zeta^{(i)})t'\lambda)(\zeta^{(i+1)})$
$(\zeta^{(i)})(t'\delta) \rightarrow_\zeta ((\zeta^{(i)})t'\delta)(\zeta^{(i)})$
(ζ-destruction rules:)
$(\zeta^{(i)})i \rightarrow_\zeta \hat{i}$
$(\zeta^{(i)})x \rightarrow_\zeta x$ *if* $x \neq i$.

(In order to prevent undesired effects, we only allow an application of the ζ-generation rule in a term t *when there is no other ζ-item present in t*).

Example 3.7 Let $t \equiv (\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2\delta)(1\lambda)1$. If we want to find all variables bound by the third λ in t, we can apply the following sequence of ζ-reductions:
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2\delta)(1\lambda)1 \rightarrow_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)(\zeta^{(1)})((3\lambda)2\delta)(1\lambda)1 \rightarrow_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((\zeta^{(1)})(3\lambda)2\delta)(\zeta^{(1)})(1\lambda)1 \rightarrow_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)(((\zeta^{(1)})3\lambda)(\zeta^{(2)})2\delta)(\zeta^{(1)})(1\lambda)1 \rightarrow_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)(\zeta^{(2)})2\delta)(\zeta^{(1)})(1\lambda)1 \rightarrow_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)\hat{2}\delta)(\zeta^{(1)})(1\lambda)1 \rightarrow_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)\hat{2}\delta)((\zeta^{(1)})1\lambda)(\zeta^{(2)})1 \rightarrow_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)\hat{2}\delta)(\hat{1}\lambda)(\zeta^{(2)})1 \rightarrow_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)\hat{2}\delta)(\hat{1}\lambda)1$

A similar procedure can be given for searching the λ binding a certain occurrence x° of a variable x ($\neq \varepsilon$) in a term t. For this purpose we introduce an **inverse search item** or $\zeta_\star$-item. The inverse search item has to move in the opposite direction, while the index (i) *de*creases instead of *in*creases. A special provision has to be made for the case that the variable in question happens to be free; in that case the reverse search item becomes the initial item of the term, and must be destructed. This case is not provided for in the following definition:

Definition 3.8 *($\zeta_\star$-reduction)*
($\zeta_\star$-generation rule:)
$x^\circ \rightarrow_{\zeta_\star} (\zeta_\star^{(x)})x^\circ$
($\zeta_\star$-transition rules:)
$(t'\lambda)(\zeta_\star^{(i)}) \rightarrow_{\zeta_\star} (\zeta_\star^{(i-1)})(t'\lambda)$ *if* $i > 1$
$((\zeta_\star^{(i)})t'\lambda) \rightarrow_{\zeta_\star} (\zeta_\star^{(i)})(t'\lambda)$
$(t'\delta)(\zeta_\star^{(i)}) \rightarrow_{\zeta_\star} (\zeta_\star^{(i)})(t'\delta)$
$((\zeta_\star^{(i)})t'\delta) \rightarrow_{\zeta_\star} (\zeta_\star^{(i)})(t'\delta)$

(ζ-destruction rule:)
$(t'\lambda)(\zeta_\star^{(1)}) \rightarrow_{\zeta_\star} (t'\hat{\lambda})$

Example 3.9 Again, let $t \equiv (\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2^\circ\delta)(1\lambda)1$. The search for the λ binding 2° in t can be executed by the following sequence of $\zeta_\star$-reductions:
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2^\circ\delta)(1\lambda)1 \rightarrow_{\zeta_\star}$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)(\zeta_\star^{(2)})2^\circ\delta)(1\lambda)1 \rightarrow_{\zeta_\star}$
$(\lambda)(1\lambda)(2\delta)(\lambda)((\zeta_\star^{(1)})(3\lambda)2^\circ\delta)(1\lambda)1 \rightarrow_{\zeta_\star}$
$(\lambda)(1\lambda)(2\delta)(\lambda)(\zeta_\star^{(1)})((3\lambda)2^\circ\delta)(1\lambda)1 \rightarrow_{\zeta_\star}$
$(\lambda)(1\lambda)(2\delta)(\hat{\lambda})((3\lambda)2^\circ\delta)(1\lambda)1$

Note that the latter search (for a binding λ) is easier, since it follows only one path in the term tree, in the direction of the root; the former search (for all variables bound by a certain λ) disperses a ζ-item over all branches of the subtree with this λ as its root.

3.3 Limiting the set of terms with a view to the types

In the previous section, the types did not play any role of importance in the term construction. However, types are meant to restrict the class of terms in lambda calculus. When used properly, types can provide for some properties that are desirable in applications, e.g. termination of reductions ("calculations").

Given the class of typed lambda terms of the previous sections, one can follow two natural ways of using the type information for establishing the "well-typedness" or "correctness" of the term: firstly, to investigate for every term that one encounters, before using it, whether the term as a whole obeys certain type-conditions; secondly, to allow reduction of the term only when the argument and the function match (this, again, is dependent of some type-information, but this time only for a part of the term). In the first case one establishes the "well-typedness" or "correctness" of a full term before working with it; in the second case one aborts a calculation at the moment that the type-laws are infringed. In the latter case more terms are "usable", since improperly typed parts may disappear in the process of calculation before they are recognized as such.

A different approach is to reconsider term construction, allowing only those terms to be constructed which are "well-typed". This process is similar to the first option above, albeit that term construction and type checking are not performed subsequently, but intermingled. In this manner of term constructing, it is desirable that type checks occur as few as possible, in order to avoid unnecessary work. For this purpose we propose the following system of rules.

$$\frac{\textit{variable condition}}{\overline{s} \vdash x} \tag{5}$$

$$\frac{\overline{s} \vdash t \qquad \overline{s}(t\lambda) \vdash t' \qquad \textit{abstraction condition}}{\overline{s} \vdash (t\lambda)t'} \tag{6}$$

$$\frac{\overline{s} \vdash t \qquad \overline{s}(t\delta) \vdash t' \qquad \textit{application condition}}{\overline{s} \vdash (t\delta)t'} \tag{7}$$

These rules should be read as follows. The symbol '$\vdash$' separates an **antecedent** which is a segment that has not yet been approved and a **succedent**, being a term that is all right as far

as bound variables, abstraction and function application are concerned. For the establishment of this "all right-ness" the type-information will be used.

As can be seen from equations 6 and 7, the succedent grows at the front side at the cost of the antecedent, by taking over an item from the back side of the antecedent. The process is finished when the antecedent is empty; $\vdash t$ means that term t has been approved.

The segment forming the antecedent of a **statement** of the form $\overline{s} \vdash t$ is also called a **context segment**, because of its similarity with contexts in Pure Type Systems. However, (pure) contexts only consist of λ-items, whereas context segments may also contain δ-items.

The variable condition is optional. In case one wishes to obtain only closed terms, this condition should read: $x \leq \mathtt{weight}_\lambda(\overline{s})$ (count ε as zero, in case $x \equiv \varepsilon$). The abstraction condition and the application condition vary from system to system, or may even be absent.[11] One example: with abstraction condition $t \equiv \varepsilon$, $t' \not\equiv \varepsilon$, empty variable condition and application condition, we obtain the syntax of the *un*typed lambda calculus.

The variable condition, the abstraction condition and the application condition may each consist of different parts. The abstraction and application condition may also depend on the λ or δ in question (recall that, in principle, we allow more than one kind of λ and/or δ in a term).

With the use of these rules (provided with the appropriate conditions) we obtain for each "well-typed" term a construction tree, which contains at the same time a proof for its "well-typedness". We shall call such a tree a **proof tree** for the term.

Example 3.10 The lowest part of the proof tree of term (1) (see Example 3.2), based on these rules, is the following:

$$\dfrac{\dfrac{\tau_1}{\vdash \varepsilon} \qquad \dfrac{\dfrac{\tau_2}{(\varepsilon\lambda_x) \vdash (x\lambda_u)\ ((u\delta)(x\lambda_t)x\lambda_y)\ (u\lambda_z)y} \qquad \dfrac{\tau_3}{(\varepsilon\lambda_x)\ ((x\lambda_u)\ ((u\delta)(x\lambda_t)x\lambda_y)\ (u\lambda_z)y\ \lambda_v) \vdash u}}{(\varepsilon\lambda_x) \vdash ((x\lambda_u)\ ((u\delta)(x\lambda_t)(x\lambda_y)\ (u\lambda_z)y\ \lambda_v)u}}{\vdash (\varepsilon\lambda_x)\ ((x\lambda_u)\ ((u\delta)(x\lambda_t)x\lambda_y)\ (u\lambda_z)y\ \lambda_v)u}$$

Here τ_1 and τ_3 are only checks of the appropriate variable conditions (which we here assume to be empty) and τ_2 is a part of the tree that is not displayed.

The completion of the proof tree of the term in the above example will show a striking similarity with the usual term tree of this term (cf. Section 2.2). Formula's of the form $\overline{s} \vdash t$ in the above proof tree correspond with the labels at the nodes of the term tree.

This observation also holds in general. In particular, the following relation holds: the leaf in the proof tree of term t that corresponds with (the occurrence of) the variable x in the term tree of t, is labeled $\overline{s} \vdash x$, with $\overline{s}x \equiv t \upharpoonright x$.

[11] In type systems, the type information plays a predominant role in the application condition: t may only be an "argument" of t' (i.e. $\overline{s} \vdash (t\delta)t'$) if t' is some kind of "function", with a "domain" in which t fits. This requirement must be expressed formally in the application condition.

4 Conclusions

In this paper it was our intention to investigate some structural aspects of terms in typed lambda calculus and to identify a number of concepts that are of importance for the use of typed lambda calculus.

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. The notions of segment and subterm fit nicely in this pattern.

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 typed variable. The section on the restriction of a term to a variable shows that these notational changes have advantages, in particular in establishing which part of a linearly written term may be of influence for a given variable occurrence: this part is (but for some brackets) exactly the string of symbols that *precedes* this occurrence in the term. More precisely, we showed that it is very easy to obtain the full restriction $t \restriction x^\circ$ using our item-notation; we just had to take the substring of string t from the beginning of t until x° and delete all unmatched opening parentheses.

We showed in 3.2 that one can easily account for free and bound variables in our approach, it was all a matter of calculation. Moreover, we provided simple procedures for finding the variable occurrences bound by a certain λ in a term t and for finding the particular λ which binds a certain occurrence x° of a variable x ($\neq \varepsilon$) in a term t. All this points to the advantages of our new notation.

We also gave an alternative way of term construction, limiting the set of terms with a view to the types. This way of term construction was based on three rules, for variables, abstractions and applications, respectively. In each of these rules certain conditions can be specified in order to restrict the generation of terms, e.g. with a view to the "well-typedness" of a term. With these rules we obtain for each term a construction tree, which contains at the same time a proof for its "well-typedness". Such proof trees show a striking similarity with the usual term trees as provided in section 2.2. This can only be another advantage for our new notation.

References

[Bar84] Barendregt, H.P., *The Lambda Calculus. Its Syntax and Semantics*, North Holland, Revised edition, 1984.

[deB70] 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, pp. 29-61, 1970.

[deB72] 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*, pp. 381-392, 1972.

[deB80] 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, pp. 29-61, 1980.

[deB9x] Bruijn, N.G. de, Algorithmic definition of lambda-typed lambda calculus. In preparation.

[Ned90] Nederpelt, R.P., Type systems — basic ideas and applications, in: *CSN '90, Computing Science in the Netherlands 1990*, Stichting Mathematisch Centrum, Amsterdam, 1990.

In this series appeared:

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.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.

91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.

92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for $F\omega$, p. 15.