EINDHOVEN UNIVERSITY OF TECHNOLOGY

The fine-structure of lambda calculus

Citation for published version (APA):
Nederpelt, R. P. (1992). *The fine-structure of lambda calculus*. (Computing science notes; Vol. 9207). 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

The fine-structure of lambda calculus

by

R.P.Nederpelt

92/07

Computing Science Note 92/07
Eindhoven, April 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.

The fine-structure of lambda calculus

R.P. Nederpelt
Department of Mathematics and Computing Science
Eindhoven University of Technology
Eindhoven, the Netherlands

April 15, 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. Such notation singles out those parts of a term, called items in the report, that are added during abstraction and application. It turns out that this item-based notation offers many advantages for various notions of the lambda calculus. It allows a linear representation of terms and makes it, for example, straightforward to locate free and bound variables and to find the binding λ-operators relevant to particular occurrences. Furthermore, step-wise explicit substitution is easily embeddable in the lambda calculus using the new notation. The item notation proves to be a powerful device for the representation of basic substitution steps, giving rise to different versions of β-reduction. Last but not least, the new notation allows for segment abbreviations, enabling one to prevent a lot of duplications in λ-terms, while remaining in the general proposed frame.

In this paper we don't stop at the advantages of the new notation, but go further to accommodate important notions of the lambda calculus in the new framework. We discuss the role of types in the presented setting and provide a type operator which gives a representative type for a typeable term. Moreover, in accommodating types in our system, it turns out that a general framework for many typed lambda calculi can be obtained in the presented setting. Another attraction of our new approach is that by specifying a number of parameters, one defines one system of typed lambda calculus or another. In fact, it turns out that many known systems of typed lambda calculus fit in the proposed setting, in particular the ones connected with "Barendregt's cube" [Barendregt 9x]. The general framework leads naturally to a number of generalizations. It gives much freedom and is at the same time simple and perspicuous. It allows theorists to compare the different systems as to important properties and enables practical users to make their choices at the relevant place.

Contents

1 Introduction **5**

2 Term formation **9**
- 2.1 The item notation 9
- 2.2 The inner structure of terms 14
- 2.3 The restriction of a term 17
- 2.4 Bound and free variables 20
- 2.5 Limiting the set of terms with a view to the types 23

3 Reduction **26**
- 3.1 Global vs. local β-reduction 26
- 3.2 Step-wise substitution 28
- 3.3 A general step-wise substitution 31
- 3.4 Substitution and β-reduction 33
- 3.5 Matching δ-λ-couples 38
- 3.6 Strategies for β-reduction 44

4 The typing relation **47**
- 4.1 Degrees 47
- 4.2 Canonical types 49
- 4.3 A context-free type reduction 54
- 4.4 The typing relation in PTS's 55
- 4.5 The typing relation in Automath-systems 59
- 4.6 Remarks on the conditions in term construction 63
- 4.7 Higher degrees 66

5 Abbreviations for segments **69**
- 5.1 The use of segment variables 69
- 5.2 Referencing in relation with segment variables 71
- 5.3 Segments and stepwise substitution 72

6 Parameters for different systems **75**

7 Conclusions **79**

8 Acknowledgements **83**

A An example 84

B An abstract grammar for terms 93

C An alternative step-wise substitution 94

D A comparison with the explicit substitution of Abadi, Cardelli, Curien and Lévy 98

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) lamda calculus: "The lambda calculus – its syntax and semantics"([Barendregt 84]). 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 benefitted 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 [de Bruijn 70] or [de Bruijn 80]).

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.

The *typing* may come in as an additional feature, restricting the set of terms in a natural manner, in correspondence with the "function and application"-intuition: types are the coding of a sort of "domains" for functions, and function applications are restricted to cases were the argument fits in the domain. Using figurative language, one might say that typing is intended to domesticate the species of the lambda terms, which contains very wild specimens by nature. (For this last observation, again see [Barendregt 84], which treats in the first place the untyped lambda calculus.)

The relations (reductions) in lambda calculus are meant to formalize 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.

In this report, we start in Chapter 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 report, is fundamental for the *fine-structure* of lambda calculus which we pursue as the main theme of the present work.

As an alternative to the use of variables, in this report 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 α-equivalence classes, etc.).

The analysis of the structure of a term leads to a number of syntactical concepts which identify the different components in an obvious manner. The notions of *item* and *segment*, both acting as modules in term construction, play an important role. We discuss the ordinary way of term construction and we develop an alternative way of term construction, based on three rules: one for adding variables, one for constructing an abstraction and one for constructing an application. In the latter option, we take account of the types, which limit the set of obtainable terms.

En passant, we introduce the process of *stepwise shifting*, which is meant to refine reduction procedures. That is to say, in order to establish that two terms are related by some reduction relation, we develop a method to transform one term into the other by shifting a certain module through the term. This step by step process thus builds a chain of intermediate terms (members of an extended collection of terms) between the original ones. Since these steps are, in a sense, "minimal", we thus obtain an ultimate fine-structure of the reduction process.

This stepwise process is first used to "mark" all variables in a term

which are bound by a certain abstraction-λ. In Chapter 3 we use a similar process for substitution. Since substitution is the fundamental operation in β-reduction, which is, 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 in this detailed form.

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. Our intention is to treat stepwise substitution on a par with the usual notions of lambda calculus. In making stepwise substitution explicit, we try to treat its syntactic components as "first class citizens" of the lambda calculus world. In fact, we succeed to a great extent, as we will show. The final "rehabilitation" of substitution still is a subject for further study.

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, we show in Section 3.6 how well-known strategies for compound β-reduction can be expressed in our setting.

In Chapter 4 we concentrate on the typing relation in typed lambda calculus. We introduce a canonical type operator, suited for the "calculation" of one canonical type in the class of all types of a certain (typeable) term. The typing relation connected with this type operator is presented again by means of a stepwise "process", which can be described in different manners. Again, we claim to give the fine-structure of a central subject in lambda calculus, this time being the typing relation.

We compare the canonical type operator with the usual typing relation $t_1 : t_2$. We discuss the relation between our approach and certain Pure Type Systems (*PTS's*), which make use of this typing relation ":". An important subclass of this class of typed lambda calculi, systematized and

studied by Barendregt and others, is relatively easy to embed in our setting. Again, a number of obvious extensions can be made for different purposes. We describe a number of Automath-systems in this setting. One of these possibilities is de Bruijn's system $\Delta\Lambda$, which is a version of Automath in the format of typed lambda calculus .

In Chapter 5 we propose a way of abbreviating segments. Since segments are a generalization of contexts, playing an important role in type systems, it turns out to be useful to consider these segment abbreviations. Several arguments are given in favour of such a feature (cf.[Balsters 86]). We can embody segment abbreviations easily, since our setting deals with segments in the same way as with terms. This is due to the fact that for terms an (uncontracted) β-redex may act as an abbreviation mechanism, which is not too hard to extend to segments.

There are, however, a number of complications as to the referencing of variables and to the stepwise substitution introduced before. These complications are discussed and mended, to such an extent that segment abbreviations become incorporable in the fine-structure of typed lambda calculus which we developed before.

Chapter 6 gives an overview of the various parameters which may be used to make a choice between systems, with references to the other chapters of this report. Chapter 7 contains a number of conclusions and results.

The report ends with four appendices. In the first one we give a small example, which is treated in detail. The second and third appendix give alternatives for subjects discussed in the report. In the fourth appendix we compare our approach regarding the substitution operator with the explicit substitution of Abadi, Cardelli, Curien and Lévy.

2 Term formation

2.1 The item notation

We 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 $\lambda_1, \lambda_2, \ldots$ for abstraction and $\delta_1, \delta_2, \ldots$ for application.

Variable names can be avoided by means of de Bruijn-indices, as was already demonstrated in ([de Bruijn 72]). We describe these de Bruijn-indices in the above-given setting.

We consider abstraction as a *binary* operation, linking types to terms (*in this order*; see Example 2.2). 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. (This observation is due to de Bruijn, see [de Bruijn 70] or [de Bruijn 80].)

Definition 2.1 *(terms)*

Ω (or $\Omega_{\lambda\delta}$) is a finite set of binary **operators** *which is the union of two disjoint sets: Ω_λ (the set of* λ-**operators***) and Ω_δ (the set of* δ-**operators***).*

We write the operators in infix-*notation.*

Ξ is the set of **variables***: $\Xi = \{\varepsilon, 1, 2, \ldots\}$.*

$F_\Omega(\Xi)$ 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***.*

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

Meta-variables for operators are ω, ω_1, etc., for λ-operators: λ, λ_1, etc., for δ-operators: δ, δ_1, etc., for variables: x, x_1, y, etc. and for terms: t, t_1, t_i, etc.

As said before, we take numbers $1, 2, \ldots$ (de Bruijn-indices) for expressing variables. The variable ε is a special one, to be described later. Because of its role, we may think of ε as being a "constant". In order to keep close to tradition, we call all of $1, 2, \ldots$ and ε "variables".

Note: There can be different (finitely many) λ's and/or δ's in terms. For the time being we shall consider only one of each, denoted λ and δ, respectively.

Since the operators are binary, we use infix notation. However, we 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, and
- the application tt' (in "classical" notation) is obtained by postfixing the argument-item t' to the term t.

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.

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 $(2\delta)(\varepsilon\lambda)1$.)

More generally, the following holds. Instead of the "binary product" $(\ldots)\omega(\ldots)$ in ordinary notation, we put the parentheses thus in item notation: $(\ldots\omega)\ldots$. Otherwise said: let $s_1, \ldots, s_n$ be items, let t' be a term, x a variable and ω an operator. Then the term $(t'\omega)s_1 \ldots s_n x$ in item-notation, should be read as the binary product $(t')\omega(s_1 \ldots s_n x)$. And the item $(s_1 \ldots s_n x\omega)$ is a denotation for $(s_1 \ldots s_n x)\omega$, i.e. only the "left hand side" and the operator of such a product.

(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 $<t'>$.)

A convention for the use of parentheses, like the item-notation, is of course only appropriate for linearly written terms. 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

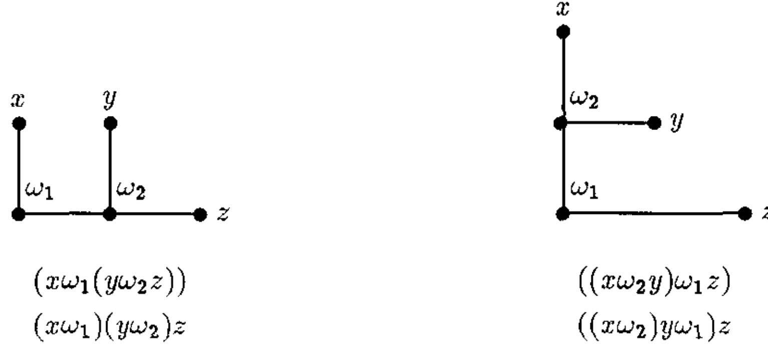


Figure 1: Term trees, with normal linear notation and item-notation

resemblance with the linear term. This has also advantages in the chapters to come.

Note that the item-notation suggests a partitioning of the term tree in vertical layers. For the first of the two terms mentioned above, 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 the latter example these layers are: the part of the tree corresponding with $((x\omega_2)y\omega_1)$ and the one corresponding with z.

Example 2.2 We give two examples of terms in typed lambda calculus and how these terms are denoted in our item-notation. We also explain the use of de Bruijn-indices.

Consider the typed lambda term $(\lambda_{x:y}.x)u$ (in "classical" notation). In this term, the subscript of λ contains the information that x has type y. The x following the subscripted operator $\lambda_{x:y}$ is a variable bound by this operator.

In item-notation with name-carrying variables (we use x, y, $z\ldots$ for variable-names) this term becomes $(u\delta)(y\lambda_x)x$. Note that the argument u has moved to the front of the term, in the argument-item $(u\delta)$. Note also that the type y of x *precedes* the λ having this x as a subscript.

In the item-notation with de Bruijn-indices, the above term is denoted as $(1\delta)(2\lambda)1$. Now the argument u has become the number 1. The "type" y has become the number 2 in front of the λ. The bond between the bound variable x and the operator λ is expressed by the final number 1. The position of this number in the term is that of the bound variable x. The

value of the number ("*one*") tells us how to find the binding place, in a manner which we explain below.

However strange it may look, in the de Bruijn-notation it is the case that *free* variables have a binding place, as well. For example, the free variables u and y in the typed lambda term given above, are translated into the number 1 occurring before the δ 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**).

The described way of omitting binding variables, and rendering bound and free variables by means of so-called **reference numbers**, is characteristic for de Bruijn-indices. How the reference can be detected, will be shown presently.

As a second example we take the typed lambda term $u(\lambda_{x:y}.x)$, which 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 term trees of the mentioned terms are given in Figure 2.

We will now explain how the link between variables (i.e. reference numbers) and λ's takes place. As noted before, these λ's may be "visible" (binding bound variables in the original term) or "invisible" (binding free variables).

In each of the two pictures in Figure 2, 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 *on this path* do count, the other λ's and all δ's do not. The reference for a variable x (i.e. the λ which binds x) is the x'th λ that one comes across following this path.

The free variable list, in the name-carrying version, is λ_y, λ_u, in both examples.

The variable ε, member of Ξ, acts as a "supertype", comparable to $\Box$ in [Barendregt 9x] and other papers. With the help of ε we can construct terms without free variables, e.g. $(\varepsilon\lambda)(1\lambda)(1\delta)((2\lambda)(1\lambda)1\lambda)3$. 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 especially in the case of an item $(\varepsilon\omega)$, which we render as (ω), for various operators ω.

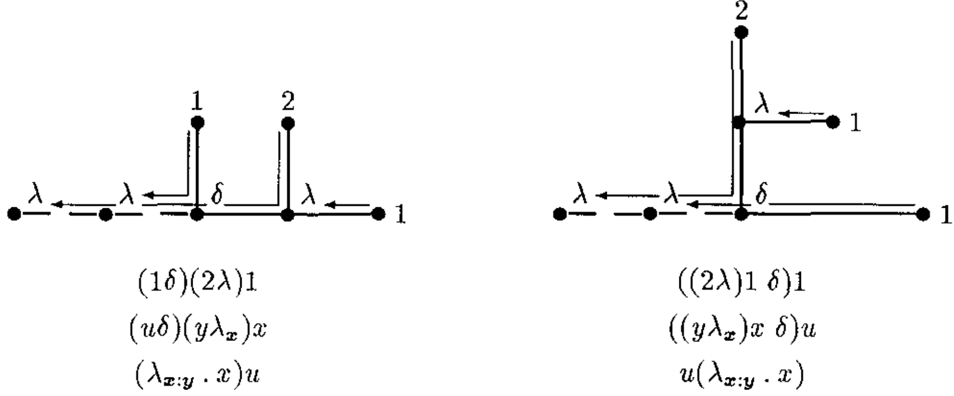


Figure 2: Term trees with free variable lists and reference numbers

Notation 2.3 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 2.2. 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.

Concatenation of strings is denoted by juxtaposition.

Remark 2.4 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.

(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 $(\beta', \xi') \in S$.) The notions to be defined in the following chapters 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 intended *occurrence* of a variable or a subterm. 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 report, we use terms and not term trees. This causes some inconveniences, especially as regards these "occurrences". Nevertheless, we prefer ordinary terms because they are easier to read than sets of pairs (β, ξ).

2.2 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 2.5 *(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**.

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

We use $s, s_1, s_i, \ldots$ as meta-variables for items and $\overline{s}, \overline{s}_1, \overline{s}_i, \ldots$ as meta-variables for segments.

We define a number of concepts connected with terms, items and segments. (A number of these definitions are rephrased in Appendix B on the basis of an abstract grammar.)

Definition 2.6 *(main items, main segments, empty segments, ω-items, ω_1-$\ldots$-ω_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** *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-$\ldots$-ω_n-***segment**.

*(An important case is that of a δ-λ-***segment**, *being a δ-item immediately followed by a λ-item.)*

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.*

Example 2.7 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 $\overline{s}$. This main segment 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 2.8 *(body, end variable, end operator)*

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

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

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

Example 2.9 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 2.10 *(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 $\overline{s}$.*

Analogously, $\mathtt{sieveseg}_{\omega_1,\ldots,\omega_n}(\overline{s})$ *or* $\mathtt{sieveseg}_{\omega_1,\ldots,\omega_n}(t)$ *"sieves" all main ω-items from $\overline{s}$, where ω is one of $\omega_1, \ldots, \omega_n$, and combines them into a segment.*

Example 2.11 In the term t of Example 2.7, $\mathtt{sieveseg}_\lambda(t) \equiv (\varepsilon\lambda)(2\lambda)$ and $\mathtt{sieveseg}_\delta(t) \equiv ((1\delta)(\varepsilon\lambda)1\delta)$.

If $t \equiv (\varepsilon\lambda_1)((1\delta)(\varepsilon\lambda_1)1\delta)(2\lambda_2)1$ (a term with two different λ's, viz. λ_1 and λ_2), then $\mathtt{sieveseg}_{\lambda_1,\lambda_2}(t) \equiv \mathtt{sieveseg}_{\lambda_2,\lambda_1}(t) \equiv (\varepsilon\lambda_1)(2\lambda_2)$.

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

Definition 2.12 *(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 $\mathsf{body}(t)$.

Analogously, the $\omega_1, \ldots, \omega_n$-weight of a segment $\overline{s}$ and a term t are defined as the weight of $\mathtt{sieveseg}_{\omega_1,\ldots,\omega_n}(\overline{s})$ *or* $\mathtt{sieveseg}_{\omega_1,\ldots,\omega_n}(t)$, *respectively.*

Next, we define the relations *direct subterm* and *subterm*, denoted by the relation symbols $\subset$ and $\Subset$:

Definition 2.13 *(subterms, direct 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 $t_1 \Subset t$.*

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 2.4). See the following example.

Example 2.14 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 $(y\lambda_x)x \Subset_{01} t$.

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

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 2.14) 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 $\underline{(y\lambda_x)x}$.

(In [de Bruijn 9x], the occurrence of a subterm is called a *positioned* subterm.)

We can also mark an occurrence of an operator: $(y\lambda_x^\circ)x$.

In Section 2.3 we need a notion that relates (left and right) subterms to an operator:

Definition 2.15 *(arguments)*
Let $(t'\omega^\circ)t'' \subset\!\!\subset t$. *Then* t' *is the* **left argument** *of* ω° *in* t*, denoted by* $\texttt{leftarg}(\omega^\circ)$*, and* t'' *is the right argument of* ω° *in* t*, or* $\texttt{rightarg}(\omega^\circ)$.

It follows from this definition that $\texttt{leftarg}(\omega^\circ)$ is the left direct subterm of $(t'\omega^\circ)t''$ and $\texttt{rightarg}(\omega^\circ)$ 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 for mathematical reasoning, λ-items may be used for assumptions or variable introductions and a δ-λ-segment may express a definition or a theorem. We come back to this in Sections 3.5 and 4.6. See also [Nederpelt 90].

2.3 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. This section is rather technical and may be skipped by a reader who is not interested in details.

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 which 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 formalize this intuition we give the following definition.

Definition 2.16 *(envelope, dominator, one-step restriction, full restriction)*
Let x° *be an occurrence of variable* x *in term* t *such that* $x^\circ \not\equiv \texttt{endvar}(t)$. *Then there is an* ω° *in* t *such that* $x^\circ \equiv \texttt{endvar}(\texttt{leftarg}(\omega^\circ))$. *For this* ω°,

the term $\texttt{leftarg}(\omega^\circ)$ *is called the* **envelope** *of* x° *or* $\texttt{env}(x^\circ)$*. The term* $(\texttt{leftarg}(\omega^\circ)\,\omega^\circ)\ \texttt{rightarg}(\omega^\circ)$ *is called the* **dominator** *of* x° *or* $\texttt{dom}(x^\circ)$*.*

(Note that the tree of $\texttt{dom}(x^\circ)$ *is the subtree with* ω° *as its root and that* $\texttt{env}(x^\circ)$ *is, in its turn, the "left direct subtree" of this subtree. See the example below.)*

The **one-step restriction** *of* t *to* x°*, denoted* $t \upharpoonright x^\circ$*, is:*
(1) in case $x^\circ \not\equiv \texttt{endvar}(t)$*: the term obtained from* t *by replacing* $\texttt{dom}(x^\circ)$ *by* $\texttt{env}(x^\circ)$*;*
(2) in case $x^\circ \equiv \texttt{endvar}(t)$*:* $t \upharpoonright x^\circ \equiv t$*.*

The **(full) restriction** *of* t *to* x°*, denoted* $t \upharpoonright\!\upharpoonright x^\circ$*, is the limit of the sequence* $t_1, t_2, \ldots$*, where* $t_1 \equiv t$ *and* $t_{i+1} \equiv t_i \upharpoonright x^\circ$*.*

Example 2.17 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 $t_1 \equiv \texttt{leftarg}(\lambda_y)$ and $x^\circ \equiv \texttt{endvar}(t_1)$. Moreover, $\texttt{rightarg}(\lambda_y) \equiv (u\lambda_z)y$, 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 equation 2:

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

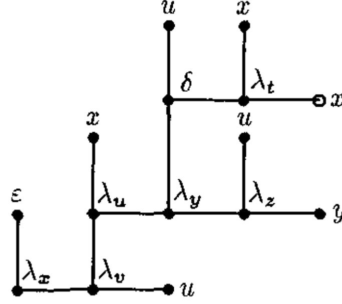
The replacement of t_2 ($\equiv \texttt{dom}(x^\circ)$) by t_1 ($\equiv \texttt{env}(x^\circ)$) gives the one-step restriction $t \upharpoonright x^\circ$:

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

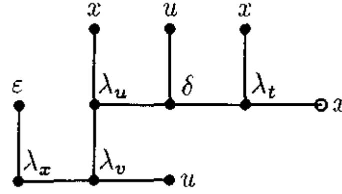
The full restriction $t \upharpoonright\!\upharpoonright x$ of the same x°, obtained after another one-step restriction, is:

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

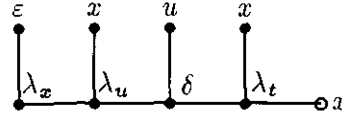
Now it will be clear that it is very easy to obtain the full restriction $t \upharpoonright\!\upharpoonright x^\circ$ using our item-notation: just take the substring of string t from the beginning of t until x° and delete all unmatched opening parentheses. This is an advantage of our new notation.



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



$$t \lceil x^\circ \equiv (\varepsilon\lambda_x)\overline{(\underline{(x\lambda_u)(u\delta)(x\lambda_t)x^\circ}\lambda_v)}u$$



$$t \upharpoonright x^\circ \equiv (\varepsilon\lambda_x)(x\lambda_u)(u\delta)(x\lambda_t)x^\circ$$

Figure 3: A term and its restriction to a variable

It is illustrative to draw the tree of t (see Figure 3) 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 3, 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, then the λ binding this x can be found in $t \upharpoonright x$; the same holds for the type of this x. (Note that 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). 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 generalization: 'restriction of a term to a *subterm*':

Definition 2.18 *(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 $\mathtt{dom}(x^\circ)$.

Finally, we note the following. We motivated the introduction of the restriction of a term to a variable or a term, by observing that $t \upharpoonright x^\circ$ or $t \upharpoonright \underline{t_0}$ contain all "information" necessary for x° or $\underline{t_0}$, respectively. This is the case, but in a sense we are over-general. In particular, some of the "bachelor δ-items" (as defined in the forthcoming Section 3.5) which are present in the restriction, can never play a role. This matter is explained and formalized in [de Bruijn 9x].

2.4 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 2.19 *(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 $\mathtt{endop}(s_x)$. The* **type** *of x° in t is $\mathtt{body}(s_x)$. Furthermore, x° is* **free** *in t if $x > m$.*

The variable ε is neither bound nor free in a term.

Term t is **closed** *when all occurrences of variables in t different from ε are bound in t. Otherwise t is* **open** *or* **has free variables**.

Example 2.20 The term $(\varepsilon\lambda_x)(x\lambda_v)(x\delta)(\varepsilon\lambda_y)((x\lambda_z)y^\circ\delta)(y\lambda_u)u$ 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 $t\lceil 2^\circ \equiv (\lambda)(1\lambda)(2\delta)(\lambda)(3\lambda)2^\circ$. So $\mathtt{sieveseg}_\lambda(t\lceil 2^\circ) \equiv s_4s_3s_2s_1 \equiv (\lambda)(1\lambda)(\lambda)(3\lambda)$. Hence, 2° is bound in t since $2 \leq \mathtt{weight}_\lambda(t\lceil 2^\circ) = 4$. Moreover, the type of 2° in t is $\mathsf{body}(s_2) \equiv \varepsilon$.

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 a variable occurrence in a restriction is free or bound if and only if the corresponding occurrence in the original term is free or bound, respectively.

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, as will become the basic approach in this report.

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. Note that the body of a search item is the empty term.

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 this $\overline{s}$ by $\overline{s'}$ in t. (It is assumed that rules of so-called "compatibility" or "monotonicity" have been added.)

Definition 2.21 *(ζ-reduction)*

(ζ-generation rule:)
$(t_1\lambda) \to_\zeta (t_1\lambda)(\zeta^{(1)})$
(ζ-transition rules:)
$(\zeta^{(i)})(t'\lambda) \to_\zeta ((\zeta^{(i)})t'\lambda)(\zeta^{(i+1)})$
$(\zeta^{(i)})(t'\delta) \to_\zeta ((\zeta^{(i)})t'\delta)(\zeta^{(i)})$
(ζ-destruction rules:)
$(\zeta^{(i)})i \to_\zeta \hat{i}$
$(\zeta^{(i)})x \to_\zeta x$ *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 2.22 If we want to find all variables bound by the third λ in $t \equiv (\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2\delta)(1\lambda)1$, we can apply the following sequence of ζ-reductions:

$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2\delta)(1\lambda)1 \to_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)(\zeta^{(1)})((3\lambda)2\delta)(1\lambda)1 \to_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((\zeta^{(1)})(3\lambda)2\delta)(\zeta^{(1)})(1\lambda)1 \to_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)(((\zeta^{(1)})3\lambda)(\zeta^{(2)})2\delta)(\zeta^{(1)})(1\lambda)1 \to_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)(\zeta^{(2)})2\delta)(\zeta^{(1)})(1\lambda)1 \to_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)\hat{2}\delta)(\zeta^{(1)})(1\lambda)1 \to_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)\hat{2}\delta)((\zeta^{(1)})1\lambda)(\zeta^{(2)})1 \to_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)\hat{2}\delta)(\hat{1}\lambda)(\zeta^{(2)})1 \to_\zeta$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)\hat{2}\delta)(\hat{1}\lambda)1$

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 2.23 *($\zeta_\star$-reduction)*

($\zeta_\star$-generation rule:)
$x^\circ \to_{\zeta_\star} (\zeta_\star^{(x)})x^\circ$

($\zeta_\star$-transition rules:)
$(t'\lambda)(\zeta_\star^{(i)}) \to_{\zeta_\star} (\zeta_\star^{(i-1)})(t'\lambda)$ *if* $i > 1$
$((\zeta_\star^{(i)})t'\lambda) \to_{\zeta_\star} (\zeta_\star^{(i)})(t'\lambda)$
$(t'\delta)(\zeta_\star^{(i)}) \to_{\zeta_\star} (\zeta_\star^{(i)})(t'\delta)$
$((\zeta_\star^{(i)})t'\delta) \to_{\zeta_\star} (\zeta_\star^{(i)})(t'\delta)$
(ζ-destruction rule:)
$(t'\lambda)(\zeta_\star^{(1)}) \to_{\zeta_\star} (t'\hat{\lambda})$

Example 2.24 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:
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2^\circ\delta)(1\lambda)1 \to_{\zeta_\star}$
$(\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)(\zeta_\star^{(2)})2^\circ\delta)(1\lambda)1 \to_{\zeta_\star}$
$(\lambda)(1\lambda)(2\delta)(\lambda)((\zeta_\star^{(1)})(3\lambda)2^\circ\delta)(1\lambda)1 \to_{\zeta_\star}$
$(\lambda)(1\lambda)(2\delta)(\lambda)(\zeta_\star^{(1)})((3\lambda)2^\circ\delta)(1\lambda)1 \to_{\zeta_\star}$
$(\lambda)(1\lambda)(2\delta)(\hat{\lambda})((3\lambda)2^\circ\delta)(1\lambda)1$

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.

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

In the previous chapters, 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 chapters, 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 an arbitrary segment, 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 **contextual segment**, because of its similarity with contexts in Pure Type Systems (see also Section 4.4). However, (pure) contexts only consist of λ-items, whereas contextual segments may also contain δ-items. We note that there is a second difference, viz. that the contexts in Pure Type Systems are not arbitrary, but in agreement with certain rules.

The variable condition is optional. In case one wishes to obtain only closed terms, this condition should read: $x \leq \texttt{weight}_\lambda(\overline{s})$ (count ε as zero, in case $x \equiv \varepsilon$). Later we shall discuss other possibilities. The abstraction condition and the application condition vary from system to system, or may even be absent. We shall discuss different options for these conditions in

Section 4.6. One example: with abstraction condition $t \equiv \varepsilon$, $t' \not\equiv \varepsilon$ and no variable condition or 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 2.25 The lowest part of the proof tree of term 1 (see Example 2.17), based on these rules, is the following:

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

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 2.1). 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 $\overline{s}\,x \equiv t \restriction x$.

3 Reduction

3.1 Global vs. local β-reduction

The relation called β-reduction is the formalization of the application of a function to an argument. In common parlance, "reduction of a term" is considered to be an *operation*, changing a "function-argument pair" into a "function value". We have used, and shall use, this colloquial way of speech throughout this report, in order to explain informally the background and the "effect" of certain reduction procedures.

On the other hand, at a purely formal level, β-reduction is nothing more than a relation between terms, hence with *no operational meaning whatsoever.* Being fully aware of this fact, we allow ourselves to mix the operational and the relational approach at pleasure.

Let us give an example in "classical" lambda calculus, in the "operational" mood. The term $(\lambda_{x:z}.(xy))u$ β-reduces to uy, i.e. the result of substituting "argument" u for x in xy. In our sugared item-notation this becomes: $(u\delta)(z\lambda_x)(y\delta)x$ reduces to $(y\delta)u$. Note that the presence of a δ-λ-segment $\overline{s} \equiv (u\delta)(z\lambda_x)$ is the signal for a possible β-reduction. "Execution" of this reduction amounts to the following syntactic procedure: replace all variables x that are bound by the end operator of the λ-item of $\overline{s}$, by the body of the δ-item of $\overline{s}$.

The "unsugared" version reads — under the assumption that the free variable list is $\lambda_y, \lambda_z, \lambda_u$: the term $(1\delta)(2\lambda)(4\delta)1$ reduces to $(3\delta)1$. Here the body of the δ-item (4δ) changes: 4 becomes 3. This is due to the fact that a λ-item, viz. (2λ), disappeared (together with the δ-item (1δ)). The end variable 1 did not change; note, however, that the λ binding this end 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.

In more complicated examples, there are more cases in which variables must be "updated". This updating of variables is an unpleasant consequence of the use of de Bruijn-indices. We shall come to this matter again later on in this chapter. It is the price we have to pay for the banishing of actual variable names (taking reference numbers instead).

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 "bulk" 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 desirable for legibility that the coded definiendum and definiens occur very close to each other in the term.

(Note: the distance between a λ-item and a corresponding δ-item can even *grow* in the traditional notation, for example when the term is extended with new information, which occurs frequently when the translation of a text is carried out! See e.g. [Nederpelt 90].)

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 just described, when a δ-λ-segment codes a definition, it is clear that this kind of β-reduction is too radical: one sometimes desires 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.

That is the reason for admitting another kind of β-reduction, called *local* β-reduction, where only one bound variable can be replaced. (See also [de Bruijn 87].) We will discuss this later. 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.

Another approach is to code substitution in such a manner that the actual "copying" of terms is never executed. There exist different settings for this kind of substitution. One of these was used as early as 1972 in the Automath-project by I. Zandleven, for the construction of a verifying program. A description of this implementation can be found in [van Benthem Jutting 88]. Another well-known method is 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 (we do not introduce, however, an analogue of graph reduction). 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 Step-wise substitution

In order to be able to push substitutions ahead, step by step, we shall introduce a new kind of items, called **substitution items** (or σ-items). These σ-items can move through the branches of the term, step-wise, from one node to an adjacent one, until they reach a leaf of the tree. At the leaf, if appropriate, a σ-item can cause the desired substitution effect.

In this manner these substitution items can bring about different kinds of β-reductions. Note that we have chosen to make substitution a part of the formal language for the terms; we do not treat it as a meta-operation, as is usually done.

Definition 3.1 *(σ-operators, $\Omega_{\lambda\delta\sigma}$-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 $\Omega_{\lambda\delta\sigma}$.*

For the time being, we take Ω_σ to have only one element: σ. We use σ as an indexed *operator, numbered with upper indices: $\sigma^{(1)}, \sigma^{(2)}, \ldots$.*

The notions: term, item, segment etc. take this extended Ω into account. 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 occurrence of a certain variable; the index i selects the appropriate occurrences.

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 $\twoheadrightarrow_\sigma$.

One-step σ-reduction is given below as a relation between segments (cf. the ζ-reduction relation of Section 2.4), 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 $\overline{s'}$.

Definition 3.2 *(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)*

$(t_1\sigma^{(i)})(t_2\lambda) \to_\sigma (t_2\lambda)(t_1\sigma^{(i+1)})$ *($\sigma_{1\lambda}$-transition)*
$(t_1\sigma^{(i)})(t_2\lambda) \to_\sigma ((t_1\sigma^{(i)})t_2\lambda)(t_1\sigma^{(i+1)})$ *($\sigma_{01\lambda}$-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)*
(σ-destruction rules:)
$(t_1\sigma^{(i)})i \to_\sigma \mathrm{ud}^{(i)}(t_1)$
$(t_1\sigma^{(i)})x \to_\sigma x$ *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 later 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 where 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** $\text{ud}^{(i)}$ 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.

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 substitution is not presented as a step-wise process. Moreover, 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. Only after application of the σ-destruction rule, the updating restores the proper value of such variables.

In Appendix C we give another approach, that only works with global β-reduction in mind. There a $\overline{\sigma}$-item travels step-wise through the term tree in order to accomplish the desired substitutions. However, here again one has to take care of the correct variable bindings.

In Section 3.3 we discuss yet another approach, where the updating can take place at any stage of a step-wise substitution process.

Again, we shall describe the effect of the update function by means of a 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 $\text{ud}^{(i)}(t_1)$, 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 3.3 *(φ-reduction)*

(σ-destruction/φ-generation rule:)
$(t_1\sigma^{(i)})i \rightarrow_\varphi (\varphi^{(0,i)})t_1$
(φ-transition rules:)
$(\varphi^{(k,i)})(t'\lambda) \rightarrow_\varphi ((\varphi^{(k,i)})t'\lambda)(\varphi^{(k+1,i)})$
$(\varphi^{(k,i)})(t'\delta) \rightarrow_\varphi ((\varphi^{(k,i)})t'\delta)(\varphi^{(k,i)})$
(φ-destruction rules:)
$(\varphi^{(k,i)})x \rightarrow_\varphi x + i$ *if* $x > k$
$(\varphi^{(k,i)})x \rightarrow_\varphi x$ *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. We come back to this matter in the following section.

3.3 A general step-wise substitution

In Section 3.2 we mentioned a drawback in our step-wise substitution: variables inside a σ-segment are shut off from the "outer world", meaning that their value need not reflect the exact binding place. Only by the updating after application of the σ-destruction rule, these variables regain their correct value.

In order to repair the drawback mentioned, there is an easy solution. We just have to add a φ-item $(\varphi^{(k,i)})$ inside the σ-item, as follows: $((\varphi^{(k,i)})t\sigma^{(j)})$. The φ-item registrates the necessary updating during the process of σ-transition.

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

Definition 3.4 *(φ-abbreviation)*

For all $k \in \mathcal{N}$, $\varphi^{(k)}$ *denotes* $\varphi^{(0,k)}$. *Moreover,* φ *denotes* $\varphi^{(1)}$ *(hence* $= \varphi^{(0,1)}$*).*

Now the rules for σ-items can be adapted as follows (cf. Definition 3.2):

Definition 3.5 *(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 3.6 (In this example we only use $\sigma_{01\lambda}$- and $\sigma_{01\delta}$- transitions):
$(1\delta)(2\lambda)(4\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)((\varphi)1\ \sigma^{(1)})(4\lambda)2 \to_\sigma$
$(1\delta)(2\lambda)(((\varphi)1\ \sigma^{(1)})4\lambda)((\varphi)(\varphi)1\ \sigma^{(2)})2 \to_\sigma$
$(1\delta)(2\lambda)(4\lambda)((\varphi)(\varphi)1\ \sigma^{(2)})2 \to_\sigma$
$(1\delta)(2\lambda)(4\lambda)(\varphi)(\varphi)1 \to_\varphi$
$(1\delta)(2\lambda)(4\lambda)(\varphi)2 \to_\varphi$
$(1\delta)(2\lambda)(4\lambda)3$.

It is not hard to see that this definition gives the same results as Definition 3.2 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 $(\varphi)\ldots(\varphi)$ (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 $(\varphi^{(l,m)})$, with $k \neq l$, in one new update function. We do not consider these matters, in order to maintain a simple system.

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

Definition 3.7 *(φ-transition rule for σ-items:)*
$(\varphi^{(k,i)})(t'\sigma^{(l)}) \to_\varphi ((\varphi^{(k,i)})t'\sigma^{(l)})(\varphi^{(k,i)})$ *if* $l \leq k$ *and*
$(\varphi^{(k,i)})(t'\sigma^{(l)}) \to_\varphi ((\varphi^{(k,i)})t'\sigma^{(l+i)})(\varphi^{(k,i)})$ *if* $l > k$.

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

A final remark in the same field is that we have now a feasible possibility at our disposal for the addition of a $\sigma_{01\sigma}$-transition. We can allow that σ-items intrude and step over other σ-items:

Definition 3.8 *($\sigma_{01\sigma}$-transition)*
$(t_1\sigma^{(i)})(t_2\sigma^{(k)}) \to_\sigma ((t_1\sigma^{(i)})t_2\sigma^{(k)})(t_1\sigma^{(i)})$ *if* $i \neq k$

3.4 Substitution and β-reduction

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

We can execute local and global β-reduction with the help of this step-wise substitution, but with certain limitations. Firstly, as mentioned before, the reduceable segment is not removed. We have to supply the tools for eliminating useless reduceable segments.

Secondly, the choice we have in the σ-transition rules has to be restricted. For local β-reduction we have to make (repeatedly) 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. It will be clear that the generation and maintenance of this information has its price as well.

Hence, we may 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. A one-step global β-reduction is defined analogously. Note that, in both cases, the reduceable segment is not yet removed.

A one-step local β-reduction of an $\Omega_{\lambda\delta\sigma}$-term is defined analogously; here it is understood that the originally present σ-items remain undisturbed. For

$\Omega_{\lambda\delta\sigma\varphi}$-terms we have to decide upon the removal of φ-items, because thes items blockade σ-items in their transition (cf. Section 3.3).

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 Section 2.1; it was, however, not yet used.)

For example, we could use $\lambda_{\texttt{loc}}$ for a future destination in local reductions and $\lambda_{\texttt{glo}}$ for global reductions. A "definition" then could be rendered as a δ-λ-segment $(t_1\delta_{\texttt{loc}})(t_2\lambda_{\texttt{loc}})$, ready for local reduction. A "function" could start with a λ-item $(t_2\lambda_{\texttt{glo}})$, whereas an "argument" for this function could have the form of a δ-item $(t_1\delta_{\texttt{glo}})$. (See [Nederpelt 90] for an explanation of these notions "definition", "function" and " argument" with respect to typed lambda calculus.)

Now, for example, the general σ-generation rule of Definition 3.5 obtains two versions:

Definition 3.9 *(local vs. global σ-generation)*
$(t_1\delta_i)(t_2\lambda_i) \rightarrow_\sigma (t_1\delta_i)(t_2\lambda_i)(t_1\sigma_i^{(1)})$, *for* $i = \texttt{loc}$.
$(t_1\delta_i)(t_2\lambda_i) \rightarrow_\sigma (t_1\delta_i)(t_2\lambda_i)(t_1\sigma_i^{(1)})$, *for* $i = \texttt{glo}$.

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

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 end 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 3.10 *(void β-reduction)*

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

Notation: $t \rightarrow_\emptyset t'$.

(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].)

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

Definition 3.11 *(δ-λ-destruction rule)*

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

Note the fact that updating here occurs with a *negative* amount of -1. This does not corrupt the term, precisely because there are no customers for the λ in the void segment. The reason is that the disappearance of the λ has to be compensated.

We note that this negative updating is not without complications. For example:

- The second φ-transition rule of Definition 3.7 is no longer valid, unless σ-items with index 0 are permitted.
- Additivity of φ-items (see Section 3.3) does not hold for negative indices. E.g. $(\varphi^{(1,1)})(\varphi^{(1,-1)})$ is not equal to $(\varphi^{(1,0)})$ (the identity), for example:

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

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

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

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

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

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

Definition 3.12 *(one-step β-reduction)*
One-step β-reduction of an $\Omega_{\lambda\delta}$-term is the combination of the following steps: 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 one void β-reduction for the disposal of $\overline{s}$. Finally, there follow a number of destructions, until again an $\Omega_{\lambda\delta}$-term is obtained (hence without σ- or φ-items).
(A one-step β-reduction of an $\Omega_{\lambda\delta\sigma}$-term is defined analogously; cf. one-step local β-reduction.)

Notation 3.13 As usual, we denote one-step β-reduction by $t \rightarrow_\beta t'$, and (ordinary) **β-reduction** – its reflexive and transitive closure – by $t \twoheadrightarrow_\beta t'$. The relation **$\beta$-equality** or **conversion** is the equivalence relation generated by $\twoheadrightarrow_\beta$; it is denoted by $t =_\beta t'$.

Remark 3.14 About the normalisation properties of our system (concerning the termination of β-reduction sequences) we note the following.
We first recall some well-known concepts:
A *redex* in a term is a subterm which starts with a δ-λ-segment.
A *normal form* is a term without a redex (hence without a δ-λ-segment).
A term t is *strongly normalizing* if *all* β-reduction sequences, starting from t, terminate (in a normal form).
A term t is *weakly normalizing* if *some* β-reduction sequence, starting from t, does terminate (in a normal form).
In general: the property *strong normalization* refers to the *necessary* termination, for each term, of all β-reduction sequences starting from that term, and the property *weak normalization* refers to the *possible* termination, for each term, of a β-reduction sequence starting from that term.

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

The σ-generation rule, as given in Definition 3.2, can be applied indefinitely many times. A similar remark holds for the $\sigma_{01\sigma}$-transition rule, which

permits an eternal reshuffling between adjacent σ-items. Hence, strong normalization is not guaranteed without extra provisions.

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

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

3.5 Matching δ-λ-couples

The step-wise substitution procedures as described, can have an undesired effect on β-reduction, especially when using lazy evaluation. In that case one likes to start another reduction before the old ones are fully done with, that is to say: while some σ-items are still present in the term as "relicts" of a previous reduction.

In such a case one may be caught. When, for example, a σ-item $(t'\sigma)$ has come to a standstill between a δ- and a λ-item, then the original δ-λ-segment has been split up and cannot be activated by the σ-generation rule.

A similar situation arises when a void β-reduction has not been executed. For example, let $(t_1\delta)(t_2\delta)(t_3\lambda)(t_4\lambda)$ be a segment of term t, and suppose that $(t_2\delta)(t_3\lambda)$ is void. Then we desire to start a β-reduction with $(t_1\delta)(t_4\lambda)$, but this is up to now only allowed after the removal of the void segment.

This observation can also be made in a more general setting: we need not require that an intermediate segment like $(t_2\delta)(t_3\lambda)$ is void. It is enough that such a segment is "well-balanced", a notion to be explained below.

In this respect we note the following. When one desires to start a β-reduction on the basis of a certain δ-item and a λ-item occurring in one segment, the *matching* of a the δ and the λ in question is the important thing, even when they are separated by other items. I.e., the relevant question is whether they *may* together become a δ-λ-segment after a number of (β-,

σ- or φ-) steps. This depends solely on the structure of the intermediate segment:

Definition 3.15 *(well-balanced segments)*

An empty segment is a well-balanced segment;

A σ-item $(t'\sigma)$ and a φ-item $(\varphi^{(k,i)})$ is a well- balanced segment;

If $\overline{s}$ is a well-balanced segment, then $(t'\delta)\overline{s}(t''\lambda)$ is a well-balanced segment.

The concatenation of well-balanced segments is a well-balanced segment;

A well-balanced segment has the same structure as a matching composite of opening and closing parentheses, each δ- (or λ-)item corresponding with an opening (resp. closing) parenthesis. For example, the well-balanced segment $(t_1\delta)(t_2\delta)(t_3\lambda)(t_4\delta)(t_5\lambda)(t_6\lambda)$ has the same "bracketing structure" as the string '(()())'. (Well-balanced segments in a typed lambda calculus without explicit substitution were introduced in [Nederpelt 73].)

Now we can easily define what matching δ-λ-couples are:

Definition 3.16 *(match, δ-λ- or reduceable couple, partner, partnered item, bachelor item, non-bachelor segment, bachelor segment)*

Let t be an $\Omega_{\lambda\delta\sigma\varphi}$-term (hence possibly containing σ- and/or φ-items). Let $\overline{s} \equiv s_1, \ldots, s_n$ be a segment occurring in t.

Now s_i and s_j **match**, *when $i < j$, s_i is a δ-item, s_j a λ-item, and the sequence $s_{i+1}, \ldots, s_{j-1}$ forms a well-balanced segment.*

*Such matching s_i and s_j are called a δ-λ-***couple** *or* **reduceable couple**. *Both s_i and s_j are called the* **partners** *in the δ-λ-couple. We also say that s_i and s_j are* **partnered** *items.*

All λ- (or δ-) items s_k in t that are not partnered, are called **bachelor** *λ- (resp. δ-)items.*

A segment consisting of partnered items only, is called a **non-bachelor segment**; *a non-empty, well-balanced segment is a special case of a non-bachelor segment, in which the partnered items occur* pair-wise.

A segment consisting of bachelor items only, is called a **bachelor segment**. *We define λ-bachelor segments and δ-bachelor segments similarly.*

(De Bruijn uses another terminology; see e.g. [de Bruijn 9x]. In his phrasing, δ-items are *applicators* or A's, and λ-items are *abstractors* or T's. For δ-λ-segments he uses the word *AT-pair* and for δ-λ-couples he uses *AT-couples.* Void β-reduction he calls *AT-removal.*)

It follows that each δ-λ-*segment* is a particular case of a δ-λ-couple (with an empty, hence well-balanced segment between the δ- and the λ-item). On the other hand, the δ-item and the λ-item in a δ-λ-*couple* may also be separated by σ- items, φ-items and other δ-λ-couples.

Definition 3.17 *(sieveseg for bachelor items)*

Let $t \in \Omega_{\lambda\delta}$.

The segment $\overline{s}$ consisting of all bachelor λ- (or δ-)items of $\mathtt{body}(t)$*, concatenated in the order in which they appear in t, is called the* **bachelor** *λ- (or δ-)***segment** *belonging to t, and will be abbreviated as* $\mathtt{sieveseg}_{\mathtt{bac}\lambda}(t)$ *(or* $\mathtt{sieveseg}_{\mathtt{bac}\delta}(t)$*).*

Similarly, we define $\mathtt{sieveseg}_{\mathtt{bac}\lambda}(\overline{s})$ *and* $\mathtt{sieveseg}_{\mathtt{bac}\delta}(\overline{s})$ *for a segment $\overline{s}$.*

Example 3.18 Consider the segment
$\overline{s} \equiv (t_1\lambda)(t_2\lambda)(t_3\delta)(t_4\lambda)(t_5\lambda)(t_6\delta)(t_7\delta)(t_8\delta)(t_9\lambda)(t_{10}\lambda)(t_{11}\delta)$.

Then $(t_3\delta)$ matches with $(t_4\lambda)$; moreover, $(t_8\delta)$ matches with $(t_9\lambda)$ and $(t_7\delta)$ with $(t_{10}\lambda)$. The segments $(t_3\delta)(t_4\lambda)$ and $(t_8\delta)(t_9\lambda)$ are δ-λ-segments. The items in these segments also form δ-λ-couples, and there is another δ-λ-couple in $\overline{s}$, viz. the couple of $(t_7\delta)$ and $(t_{10}\lambda)$.

The items mentioned just now ($(t_3\delta)$, $(t_4\lambda)$, $(t_7\delta)$, $(t_8\delta)$, $(t_9\lambda)$ and $(t_{10}\lambda)$) are the partnered main items of $\overline{s}$. The remaining main items of $\overline{s}$, viz. $(t_1\lambda)$, $(t_2\lambda)$, $(t_5\lambda)$, $(t_6\delta)$ and $(t_{11}\delta)$, are bachelor items.

The segments $(t_1\lambda)(t_2\lambda)$ and $(t_5\lambda)(t_6\delta)$ are examples of bachelor segments, the former being also a λ-bachelor segment. Examples of non-bachelor segments are $(t_7\delta)(t_8\delta)(t_9\lambda)$ and $(t_7\delta)(t_8\delta)(t_9\lambda)(t_{10}\lambda)$, the latter also being a well-balanced segment.

Moreover, $\mathtt{sieveseg}_{\mathtt{bac}\lambda}(\overline{s}) \equiv (t_1\lambda)(t_2\lambda)(t_5\lambda)$ and $\mathtt{sieveseg}_{\mathtt{bac}\delta}(\overline{s}) \equiv (t_6\delta)(t_{11}\delta)$.

Now we can make the important observation that the main items of a term t may be divided into different classes: the "partnered" items (i.e. the δ- and λ-items which are partners, hence "coupled" to a matching one), the "bachelors" (i.e. the bachelor λ-items and bachelor δ-items) and the "offspring" (i.e. the σ- and the φ-items).

Now let $\overline{s}$ be the body of a ("childless") term $t \in \Omega_{\lambda\delta}$. Then the following holds:

- Each bachelor main λ-item in $\overline{s}$ precedes each bachelor main δ-item in $\overline{s}$.

- The removal from $\overline{s}$ of all bachelor main items, leaves behind a *well-balanced* segment $\overline{s'}$ (which we denote by $\texttt{sieveseg}_{\texttt{bal}\lambda\delta}$).
- The removal from $\overline{s}$ of all main δ-λ-couples, leaves behind a λ^n-δ^m-segment, consisting of all bachelor main λ- and δ-items (which we denote by $\texttt{sieveseg}_{\texttt{bac}\lambda\delta}$).

From the above, we may conclude the following. For each non-empty segment $\overline{s} \in \Omega_{\lambda\delta}$ there is a unique *partitioning* in segments $\overline{s_1}, \ldots, \overline{s_n}$, such that
(1) $\overline{s} \equiv \overline{s_1} \ldots \overline{s_n}$,
(2) each $\overline{s_i}$ is either a bachelor λ- or δ-segment or a (non-bachelor) well-balanced segment,
(3) if $\overline{s_i}$ $(i < n)$ is a bachelor segment, then $\overline{s_{i+1}}$ is a well-balanced segment, and vice versa (hence, bachelor and well-balanced segments *alternate* in $\overline{s_1}, \ldots, \overline{s_n}$),
(4) each bachelor λ-segment $\overline{s_j}$ precedes each bachelor δ-segment $\overline{s_k}$ in $\overline{s}$.

(The mentioned partitioning of the sequence of main items of a term t is called the *canonical dissection* in [de Bruijn 9x]. In that paper, the sequence $\texttt{sieveseg}_{\texttt{bac}\delta}(t)$ is called the *waiting list* of t; the segment $\overline{s}$ obtained from t by removing all δ-items in $\texttt{sieveseg}_{\texttt{bac}\delta}(t)$ from t and omitting $\texttt{endvar}(t)$ is called the *knowledge frame* of t.)

Example 3.19 The segment $\overline{s}$ of Example 3.18 has the following partitioning:
bachelor λ-segment $(t_1\lambda)(t_2\lambda)$,
well-balanced segment $(t_3\delta)(t_4\lambda)$,
bachelor λ-segment $(t_5\lambda)$,
bachelor δ-segment $(t_6\delta)$,
well-balanced segment $(t_7\delta)(t_8\delta)(t_9\lambda)(t_{10}\lambda)$,
bachelor δ-segment $(t_{11}\delta)$.

Moreover,
$\texttt{sieveseg}_{\texttt{bac}\lambda} \equiv (t_1\lambda)(t_2\lambda)(t_5\lambda)$,
$\texttt{sieveseg}_{\texttt{bac}\delta} \equiv (t_6\delta)(t_{11}\delta)$,
$\texttt{sieveseg}_{\texttt{bac}\lambda\delta} \equiv (t_1\lambda)(t_2\lambda)(t_5\lambda)(t_6\delta)(t_{11}\delta)$,
$\texttt{sieveseg}_{\texttt{bal}\lambda\delta} \equiv (t_3\delta)(t_4\lambda)(t_7\delta)(t_8\delta)(t_9\lambda)(t_{10}\lambda)$.

We can extend our notions of β-reduction in this vein. That is to say, we may allow *δ-λ-couples* to have the same "reduction rights" as *δ-λ-segments.*

All we need for that, is to give a new generation rule for the step-wise substitution described in the Section 3.3; the transition and destruction rules can stay unchanged.

Definition 3.20 *(generalized β-reduction)*
For all well-balanced segments $\overline{s}$*:*
(general σ-generation rule:)
$(t_1\delta)\overline{s}(t_2\lambda) \to_\sigma (t_1\delta)\overline{s}(t_2\lambda)((\varphi^{(k+1)})t_1\sigma^{(1)})$, *where* $k = \mathtt{weight}_\lambda(\overline{s})$.

The new notion of β-reduction on the basis of these new generation rules, is called **generalized** β-reduction. (The notion of a well-balanced segment and a notion of generalized β-reduction as defined above were introduced in [Nederpelt 73]. The generalized β-reduction, called β_1-reduction, was used as an expedient for the proof of strong normalization in a uniformly typed lambda calculus.)

With this generalized reduction we can choose for delayed substitution and yet perform reductions in the same order as with ordinary β-reduction. Note, however, that our generation rules give more possibilities than before. Firstly, as said before, the intermediate segment only needs to be well-balanced; δ-λ-segments need not to be void. Secondly, with generalized β-reduction we may have more possible reduction paths.

For example, in the term $t \equiv (t_1\delta)(t_2\delta)(t_3\lambda)(t_4\delta)(t_5\lambda)(t_6\lambda)u$ there are *three* reduceable couples ready for a generalized β-reduction, but only *two* reduceable segments. (The three redexes in t corresponding to the reduceable couples are: t as a whole, $(t_2\delta)(t_3\lambda)(t_4\delta)(t_5\lambda)(t_6\lambda)u$ and $(t_4\delta)(t_5\lambda)(t_6\lambda)u$; only the last two redexes are "traditional" redexes, corresponding with reduceable *segments.*)

As said before, the pairwise matching δ- and λ-items occur in a nested pattern. If, however, one chooses for the traditional notation, where arguments *follow* the corresponding functions, this nesting property is lost.

We shall explain this by means of an example. The term t mentioned above looks like $((\lambda_{x:t_3}.(\lambda_{y:t_5}.\lambda_{z:t_6}.u)t_4)t_2)t_1$ in the traditional notation. Now the reducable segments $(t_2\delta)(t_3\lambda)$ and $(t_4\delta)(t_5\lambda)$ correspond to the redexes $(\lambda_{x:t_3}.(\lambda_{y:t_5}.\lambda_{z:t_6}.u)t_4)t_2$ and $(\lambda_{y:t_5}.\lambda_{z:t_6}.u)t_4$. These redexes occur "nested". The reduceable *couple* $(t_1\delta)(t_6\lambda)$ also has a corresponding ("generalized") redex in the traditional notation, which is, however, not so easy to find. The corresponding "traditional" redex will appear after two one-step β-reductions, leading to $(\lambda_{z:t_6}.u)t_1$. (We assume x, y, z and u to be different variables.)

Let us look deeper into this correspondence. Each δ-item $(t_i\delta)$ corresponds with a subterm t_i (in the traditional notation), each λ-item $(t_j\lambda)$ with a "term head" $\lambda_{w:t_j}$. Now we see that the matching in the traditional notation is no longer nested. The "bracketing structure" of the maximal main segment of t is not compatible with '(( ) ( ))' (see above), but with '$(_1\ (_2\ (_3\ )_2\ )_1\ )_3$', where '$(_i$' and '$)_i$' match.

Since the notion of δ-λ-couple is a natural extension in the light of delayed substitution or local β-reduction, the above gives an extra argument in favour of the convention "argument precedes function".

We may observe that δ-items of δ-λ-couples can occupy different positions in a term, without disturbing the meaning of the term, both semantically and procedurally. For example, the last-mentioned term t is equivalent, in both aspects, to the term t' defined as $t' \equiv (t_2\delta)(t_3\lambda)(t_4\delta)(t_5\lambda)(t_1\delta)(t_6\lambda)u$ (when we disregard updating). So, in principle there is no objection against a "re-shuffling" of a well-balanced segment $\overline{s}$, in the sense that all δ-items in $\overline{s}$ are shifted to the right until they meet their λ-partners. If one chooses for this option, formalizing it into a "δ-shift-transition" or "-reduction" in the obvious manner, then all δ-λ-couples become δ-λ-segments, and the general generation rules are no longer required. (A well-balanced segment then is the concatenation of δ-λ-segments.)

This "re-shuffling" leads in the traditional notation from, e.g. the term $((\lambda_{x:t_3}.(\lambda_{y:t_5}.\lambda_{z:t_6}.u)t_4)t_2)t_1$ to the term $(\lambda_{x:t_3}.(\lambda_{y:t_5}.(\lambda_{z:t_6}.u)t_1)t_4)t_2$. Here the action clearly is more difficult to describe.

Note that the proposed δ-shift does not disturb the bonds between λ's and variables (when defined properly, i.e. with appropriate updating φ-items in the shifted δ-items). However, undesired effects as to binding may arise in case one would define a comparable δ-shift in the opposite direction, e.g. when starting with term t' and transforming it into term t as above. For example, in t' the subterm t_1 may contain variables bound by the λ in $(t_5\lambda)$; this binding would be lost in t.

Also, a "swap" between adjacent δ-λ-segments can, in general, lead to undesired situations.

Finally, we can specify two different kinds of bound variables:

Definition 3.21 *(weakly bound, strongly bound)*

Let variable x occur bound in term t.

Then we say that (this occurrence of) x is **weakly bound** *if the binding λ-item of this x is a bachelor item (hence has no matching δ-item in t), and*

we call x **strongly bound** *if this λ-item is partnered (hence does have a matching δ-item).*

In the latter case, when t' is the body of the matching δ-item, we say that (the occurrence of) x is **bound to** *this t'.*

Another description of local β-reduction now is the following: select a strongly bound variable x in t and replace x by the term to which x is bound. And global β-reduction amounts to: select all x's bound to a certain term t' and replace all these x's by t'.

Semantically, we know "nothing" about a free variable, "a little bit" about a weakly bound variable (namely its type), and "everything" about a strongly bound variable (namely its type *and* its "meaning", being the term to which it is bound). The words "a little bit" and "everything" should not be taken too seriously; e.g.: in the term $t \equiv (u\delta)(v\lambda_x)x$ the x is strongly bound but neither its type (v) nor the term to which it is bound (u) are very informative, since both are free in t.

By selecting a number of strongly bound variables in a term t, we can execute all sorts of **compound** reductions in t by simultaneously replacing the selected variables by the terms to which they are bound. For example, when all the selected x's are bound to the same subterm t', we have an intermediate form between local and global β-reduction. When *all* strongly bound variables are selected, we have a generalization of the so-called Gross-Knuth-reduction (see the following section).

3.6 Strategies for β-reduction

Different strategies are known for applying consecutive β-reductions. A strategy, in this respect, is a procedure which prescribes the order of the reductions which have to be "executed" consecutively. Hence, such a strategy tells us, at any moment in a more-step reduction, which reduceable segment(s) is/are selected for "firing", i.e. for performing one or more β-reductions.

We shall describe three of these strategies in our uniform notation. For further details we refer to [Barendregt 84].

(For the definition of concepts like normal form, redex and normalization, see Remark 3.14.)

The following strategies will be described: head reduction, Gross-Knuth-reduction and normal order reduction. Each of these strategies has certain advantages. For this we refer again to Barendregt's book.

1. Head reduction

Definition 3.22 *(head normal form, head segment, head redex, head reduction)*

An $\Omega_{\lambda\delta}$*-term* t *is in* **head normal form** *if none of the main segments of* t *is a* δ-λ*-segment (hence, all main items are bachelors). Otherwise, the leftmost main* δ-λ*-segment is called a* **head (δ-λ-)segment**. *This segment concatenated with the right argument of its* λ *is the* **head redex**.

A **one-step head reduction** *is a* β*-reduction generated by the head segment.*

A **(more-step) head reduction** *is a sequence of one-step head reductions.*

Note that a head reduction path is uniquely determined, since each term has at most one head segment. A terminating head reduction ends in a head normal form. Such a head normal form is a concatenation of λ-items, followed by δ-items, followed by a variable. Not all head reductions terminate.

2. Gross-Knuth reduction

Definition 3.23 *(indexed segments, indexed terms, Gross-Knuth reduction)*

An **indexed** δ-λ*-segment is a* δ-λ*-segment in which the final operator* (λ) *has been indexed. A (one-step)* **β'-reduction** *is a* β*-reduction generated by an indexed* δ-λ*-segment. An* **indexed** *term is a term in which one or more* δ-λ*-segments are indexed. A* **fully indexed** *term is a term in which* all δ-λ*-segments are indexed.*

A **Gross-Knuth reduction** *of an* $\Omega_{\lambda\delta}$*-term* t *is a sequence of* β'*-reductions according to the following procedure. First index* all δ-λ*-segments in* t *with different indices; then apply* β'*-reductions until all indices have disappeared.*

Every one-step Gross-Knuth reduction actually terminates, since the described process of *completely developing* a term t is always finite, and is independent of the order of application of the β'-reductions (also in typed lambda calculus). See [Barendregt 84, pp. 330 ff].

3. Normal order reduction

Definition 3.24 *(precede, normal order reduction, quasi normal order reduction)*

Let $\overline{s_1}^\circ$ *and* $\overline{s_2}^\circ$ *be occurrences of nonempty segments in term t. We say that* $\overline{s_1}^\circ$ **precedes** $\overline{s_2}^\circ$ *(or* $\overline{s_1}^\circ \prec \overline{s_2}^\circ$*), if* $\mathbf{env}\overline{s_1}^\circ$ *is a subterm of* $\mathbf{env}\overline{s_2}^\circ$.

A δ-λ*-segment in t is called a* **normal order** δ-λ**-segment** *if it is maximal in t with respect to* $\prec$.

A **one-step normal order reduction** *is a one-step* β*-reduction which is generated by a normal order* δ-λ*-segment. A* **normal-order reduction** *is a sequence of one-step normal order reductions. A* **quasi normal order reduction** *is a sequence of one-step* β*-reductions such that each reduction in the sequence is followed (somewhere in the sequence) by a one-step normal order reduction.*

Now it can be shown, as in [Barendregt 84, pp. 326 ff], that if a term has a normal form, then every (quasi) normal order reduction is finite.

4 The typing relation

4.1 Degrees

Until now, typing did not play a role in the construction of lambda-terms. In each λ-item there is "room" for information about the type of all variables bound by the λ under consideration. But this information has not yet been explicitly used. We have only provided a possibility for using this type information. For example, the types can be taken into account in the abstraction condition and the application condition, which are part of the term construction process (see Section 2.5).

In this chapter we give examples of how these conditions may actually restrict the collection of terms obtained. Hence, the type information will play an essential role. Moreover, in this chapter it turns out to be advantageous to use different λ's and δ's, a possibility that was present from the beginning, but until now not really employed.

Before going deeper into the matter, we emphasize that there is no fundamental difference between types and terms. A term may act as a *type of* another term, a term may even have different types or it may happen that a term has no type at all. But each type is itself a term. Hence, typing is a relation between terms, and a *statement* $t_1 : t_2$ expresses that term t_1 has term t_2 as (one of its) types.

By the typing relation, there is a kind of hierarchy between terms. A notion expressing this hierarchy, is the notion of *degree*. It will turn out, in the system that we present, that the degree of a "type" is *one less* than the degree of a term of this type.

We start with the definition of the degree of a variable and thereupon we define the degree of a term.

Definition 4.1 *(degree of a variable)*

The **degree** *of a variable x that is free in term t, is undefined.*

The degree $\mathsf{deg}(\varepsilon)$ *of every ε occurring in t, is zero.*

Assume that (the occurrence of) x is bound in t and let t' be the type of x. Further, let y be the end variable of this type t' and assume that $\mathsf{deg}(y)$ *is defined. Then* $\mathsf{deg}(x) = \mathsf{deg}(y) + 1$.

Note that each variable in a closed term has a degree. The set of the degrees of variables occurring in a term, is always a set $\{0, \ldots, n\}$ for some $n \geq 0$.

Definition 4.2 *(degree of a term)*

The **degree of a term** *is the degree of its end variable, if this degree is defined; otherwise it is undefined. The* **maximal degree** *of a term is the maximal number that occurs as a degree of a variable occurring in the term; if there is no such number, then the maximal degree of such a term is undefined.*

Remark 4.3 Many existing definitions of the notion 'degree' count "the other way round", with the result that the degree of a "type" is one *more* than the degree of a term of this type. Our degrees 0, 1, 2, 3 then change into (e.g.) 3, 2, 1, 0. In our approach we start with a "top level" having degree zero, and lower levels are numbered upwards, *without restriction.* This makes it easier to discuss the subject of "more degrees". Cf. Section 4.7.

Example 4.4 In Example 2.17 we considered the following $\Omega_{\lambda\delta}$-term t: $(\varepsilon\lambda_x)((x\lambda_u)((u\delta)(x\lambda_w)x\lambda_y)(u\lambda_z)y\lambda_v)u$.

The degrees for the variables occurring in this term are: $\mathsf{deg}(\varepsilon) = 0$; $\mathsf{deg}(x) = 1$; $\mathsf{deg}(u) = 2$, except for the free u which is the end variable of the term: this u has no degree; $\mathsf{deg}(y) = 2$; $\mathsf{deg}(z) = 3$. If w would have occurred as a variable, then its degree would have been 2. The term itself has no degree (since its end variable is free). The maximal degree of the term is 3.

The restriction of a term to a variable (see Section 2.3) does not change any degree.

Finally, we define two desirable "consistency" properties of typing relations:

Definition 4.5 *(degree-consistency)*

We call a typing relation *on a set of terms* **degree-consistent** *if for all terms t_1 and t_2 we have:*

if $t_1 : t_2$ and if both $\mathsf{deg}(t_1)$ and $\mathsf{deg}(t_2)$ are defined, then $\mathsf{deg}(t_1) = \mathsf{deg}(t_2) + 1$.

A reduction relation $\rightarrow_\rho$ *on a set of terms is* **degree-consistent** *if the following holds:*
for all t_1 and t_2 such that $t_1 \rightarrow_\rho t_2$, if $\mathsf{deg}(t_1)$ is defined, then also $\mathsf{deg}(t_2)$ is defined and $\mathsf{deg}(t_1) = \mathsf{deg}(t_2)$.

Note: In [Barendregt 9x], a typing relation which is degree-consistent is called *ok.*

4.2 Canonical types

Variables occurring bound in a term in typed lambda calculus have a "natural" type, as expressed in Definition 2.19. This type is the body of the λ-item ending in the λ which binds the variable. We extend this process of typing to (general) *terms* by means of a *canonical typing function* `typ`, acting on arbitrary subterms t' of a term t.

Definition 4.6 *(canonical type)*
The **canonical type** $\mathtt{typ}(t')$ *of a subterm* t' *of a term* t*, with* $x \equiv \mathtt{endvar}(t')$ *and* x *bound in* t*, is defined as follows:*
$\mathtt{typ}(t') \equiv \mathtt{body}(t')(\varphi^{(x)})t''$,
where t'' *is the type of* x *in* t *as defined in Definition 2.19.*

Clearly, this definition is not very complicated. The canonical type $\mathtt{typ}(t')$ of a (sub-)term t' is obtained by replacing the end variable of t' by its type t'' (together with some updating of free variables in t'').

Following the general style of this report, we can also use a *type item* (τ) and a type reduction operator $\rightarrow_\tau$ instead of the typing function `typ`. Hence, we extend our set of terms in order to incorporate these τ-items (we now have $\Omega_{\lambda\delta\sigma\varphi\tau}$-terms).

The search for the canonical type of a subterm t' of t starts with $(\tau)t'$; this term may be transformed to $\mathtt{typ}(t')$ by using the following τ-reduction rules for $\Omega_{\lambda\delta\tau}$-terms (so we assume that the term under consideration contains no σ- or φ-items):

Definition 4.7 *(τ-reduction)*
(τ-transition rules:)
$(\tau)(t_1\omega) \rightarrow_\tau (t_1\omega)(\tau)$
(τ-destruction rule:)
$(\tau)x \rightarrow_\tau (\varphi^{(x)})t''$, *if* t'' *is the type in* t *of the* x *under consideration.*

Note that the search for the type t'' of x is not yet formalized. The replacement of this x° by t'' can also be described by means of step-wise operations, for example as follows. First send an inverse search item from x° "backwards" into t until the λ binding x° has been found; then collect $t'' \equiv \mathtt{body}(s)$ from the corresponding λ-item s and transport it stepwise "forwards" to x°. Finally, replace x° by t''. This procedure can easily be formalized. (In a given implementation, there will possibly be a more direct way to collect the type of x°.)

In Section 4.3 we give another, more direct solution for this matter.

1. The type of an abstraction

Things become more interesting when we go further into the matter. Firstly, we regard the type of an abstraction. We recall that we have the possibility to use different λ's. For example, λ_1 could be used for dependent product formation (usually denoted as Π), and λ_2 for the — ordinary — function operator λ. This enables us to make a difference between the type of a Π-abstraction on the one hand, and the type of a λ-abstraction on the other hand, as we shall now explain.

Usually, given that the term t' has type t'', one defines the type of a Π-abstraction $\Pi x : t_1 . t'$ to be t'', as well. One could say that "the Π-item is not accounted for in the type". On the other hand, the type of a λ-abstraction $\lambda x : t_1 . t'$ is the corresponding Π-abstraction $\Pi x : t_1 . t''$. Hence, in calculating the type of a λ-abstraction, the λ-item changes into the corresponding Π-item, and the rest of the term is replaced by its type.

As a consequence, one may refine the transition rules for λ-items as follows, replacing those of Definition 4.7 for the case that $\omega \equiv \lambda$:

Definition 4.8 *(τ-transition rules for indexed λ-items:)*
$(\tau)(t_1\lambda_1) \rightarrow_\tau (\tau)$
$(\tau)(t_1\lambda_2) \rightarrow_\tau (t_1\lambda_1)(\tau)$

Note that λ_1 and λ_2 behave differently with respect to a τ-item.

Here we used λ_1 for Π and λ_2 for the — ordinary — λ. This is not a mere symbol renaming; it also points at a possible generalization. In fact, we can extend this kind of systems by incorporating more different λ's. For example, with an infinity of λ's, viz. λ_0, λ_1, λ_2, λ_3 ..., we can give the following type rules:

Definition 4.9 *(τ-transition rule for arbitrarily many indexed λ-items)*
$(\tau)(t_1\lambda_{i+1}) \rightarrow_\tau (t_1\lambda_i)(\tau)$, *for* $i = 0, 1, 2, \ldots$

This is a generalization of Definition 4.8, if we add a reduction rule stating that $(t_1\lambda_0)$ reduces to the empty segment.

We may use as many of these λ_i's as we like. For example, in Definition 4.8 we only use λ_1, replacing Π, and λ_2, replacing λ. However, there may be circumstances in which one desires to have more "layers" of λ's. Cf. Section 4.7.

2. The type of an application

As regards the type of an application, one usually employs a rule of the following form: given a "function" F of type $\Pi x : t'' . t_1$ and an "argument" t of the appropriate type t'' (this is the type or domain which is associated with this function), then the application term $F\,t$ (in our notation: $(t\delta)F$) has type $t_1[x := t]$. Here $[x := t]$ is a postfix meta-operator standing for the substitution of t for all free occurrences of x.

In our approach, where substitution is treated as a "first class citizen" (an operation *inside* the system), the above is not satisfactory. Instead, we may express the substitution by means of a σ-item. Consequently, the type $t_1[x := t]$ is not given immediately, but is the result of a sequence of σ- and φ-reductions. (In fact, this only holds when t_1 is an $\Omega_{\lambda\delta\sigma}$-term, hence originally without φ-items. The reason is, that σ-items are not allowed to "pass" φ-items, as we already noted in Section 3.3.)

For this purpose we maintain Definition 4.8 as regards the λ-items, and we employ the following τ-transition rule for δ-items (as in Definition 4.7):

Definition 4.10 *(τ-transition rule for δ-items)*
$(\tau)(t_1\delta)t_2 \to_\tau (t_1\delta)(\tau)$.

However, we make demands to rule 7 (see Section 2.5), which we repeat for convenience' sake:

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

The requirement now is that the following application condition does hold in this rule:
$(\tau)t' =_{\tau,\beta} (t''\lambda_1)t_1$ and $(\tau)t =_{\tau,\beta} t''$.

Now it follows that

$$(\tau)(t\delta)t' \to_\tau (t\delta)(\tau)t' =_{\tau,\beta} (t\delta)(t''\lambda_1)t_1 \twoheadrightarrow_{\sigma,\varphi,\emptyset} t_1[x := t] \tag{8}$$

where the x's are the variables in t_1 bound by the mentioned λ_1. Hence, we obtain the desired result that $(t\delta)t'$ "has type" $t_1[x := t]$.

Note that we see the λ_1 (i.e., the Π) indeed as a kind of λ, hence eligible for an application. This is a quite natural approach. In the usual notation, this would amount to the introduction of a β-reduction caused by a Π-application:

$(\Pi x : A \,.\, B)a \to_\beta B[x := a]$.
Here one may interpret $(\Pi x : A \,.\, B)a$ as the wish to select the "axis" $B(a)$ in the Cartesian product $\Pi x : A \,.\, B$.

In our notation, a Π-application is characterized by a δ-Π-segment of the form $(t_1\delta)(t_2\Pi)$ (i.e., $(t_1\delta)(t_2\lambda_1)$). We speak about a *$\beta_{\delta\Pi}$-reduction* when referring to a β-reduction generated by such a δ-Π-segment. Similarly, a *$\beta_{\delta\lambda}$-reduction* is an "ordinary" β-reduction, generated by a δ-λ-segment (or δ-λ_2-segment).

Summarizing, we note that there are two possible approaches regarding Π-application:

- *Implicit* or *compulsory* $\beta_{\delta\Pi}$-reduction, i.e. for F of type $(\Pi x : A \,.\, B)$ and a of type A we immediately have that Fa is of type $B[x := a]$, without intermediate steps. Here Π-application is *not allowed.*

  This is the case in PTS's (see Section 4.4).

- *Explicit* $\beta_{\delta\Pi}$-reduction, where Π-application is allowed. Now we have, for F and a as above, that Fa has type $(\Pi x : A.B)a$, which $\beta_{\delta\Pi}$-reduces to $B[x := a]$.

The latter option is an extension of the former one. With *explicit* $\beta_{\delta\Pi}$-reduction one may simulate the effects of *implicit* $\beta_{\delta\Pi}$-reduction, as we explained above. One might argue that implicit $\beta_{\delta\Pi}$-reduction is closer to the intuition in the most usual applications. However, experiences with the Automath-languages, containing explicit $\beta_{\delta\Pi}$-reduction, demonstrated that there exists no formal or informal objection against the use of this explicit $\beta_{\delta\Pi}$-reduction in natural applications of type systems.

The two options can also be described in our step-wise structure. Our description of *explicit* $\beta_{\delta\Pi}$-reduction has been given above. If one desires to have *implicit* $\beta_{\delta\Pi}$-reduction as a formalized notion, then we can make use of the possibility to have different δ's at our disposal. In that case, a δ_1-item $(t\delta_1)$ can be used as a signal for *forced* priority for certain operations which execute the desired implicit $\beta_{\delta\Pi}$-reduction.

For example, the δ_1's in the chain

$$(\tau)(t\delta_1)t' \to_\tau (t\delta_1)(\tau)t' =_{\tau,\beta} (t\delta_1)(t''\lambda_1)t_1 \twoheadrightarrow_{\sigma,\varphi,\emptyset} t_1[x := t]$$

(cf. equation 8) can be used to enforce with highest priority, i.e. before the execution of any other "operation" on the term:
(1) the "calculation" of the type $\mathbf{typ}(t')$ obtained by τ-reduction of $(\tau)t'$,

(2) the search for a term of the form $(t''\lambda_1)t_1$ which is β-convertible to (or a β-reduct of) $\mathtt{typ}(t')$,
(3) and the reduction $(t\delta_1)(t''\lambda_1)t_1 \twoheadrightarrow_{\sigma,\varphi,\emptyset} t_1[x := t]$, which actually is a β-reduction.

By this process we obtain the term $t_1[x := t]$ as a *necessary* and *immediate* result of a τ-reduction on $(\tau)(t\delta_1)t'$. For ordinary, non-compulsory $\beta_{\delta\lambda}$-reductions, we may employ another δ, e.g. δ_2.

For simplicity, however, we shall not use these different δ's in the following sections and chapters of this report.

Remark 4.11 In a now commonly accepted setting (see [Barendregt 9x] or [Barendregt and Hemerik 90]), the typing relation is expressed in the format $\Gamma \vdash t_1 : t_2$. Here Γ is a context, and the *statement* $t_1 : t_2$ expresses that t_1 has type t_2 *relative to* this context Γ. Such a context can be considered as a segment consisting of main λ-items, meant to bind all free variables occurring in t_1 and t_2.

For example, in $(\varepsilon\lambda_x)(x\lambda_y) \vdash y : x$ it is stated that y has type x in the context $(\varepsilon\lambda_x)(x\lambda_y)$, which is indeed the case, as is visible in the context-item $(x\lambda_y)$. Also, $(\varepsilon\lambda_x)(x\lambda_y) \vdash x : \varepsilon$ holds.

In order to define the typing relation, one usually starts with the types of variables, as in the above example. Subsequently one deduces other statements of the form $\Gamma \vdash t_1 : t_2$, by regarding more complex terms t_1 and their types t_2. Finally, a conversion rule expresses that the types of terms are given *modulo* conversion; i.e., if $t_1 : t_2$ and $t_2 =_\beta t_3$, then $t_1 : t_3$. The typing relation is the smallest relation satisfying these rules.

In our opinion, the sketched approach is, in a sense, not consistent. Note that with *each variable* there is associated a preference type, as given either in the context or in the λ- or Π-item in which this variable is bound. For *terms in general* no preference type has been given, but a whole collection of types, which are typeable by themselves and linked by means of β-reduction. We think, however, that the canonical type as we introduced above can easily play the role of a preference type. The typing relation $t_1 : t_2$ is then no longer necessary. The β-conversion finds its place in the application condition, where it naturally belongs. Hence, we need not mix typing with reduction as in the conversion rule of PTS's, which is a simplification ("separation of concerns") that may be advantageous.

We note, however, that this matter has not yet been completely worked out. In particular, the claims above and the precise relation with Pure Type Systems (see also Section 4.4) are subject for further investigation.

4.3 A context-free type reduction

We already noted in the previous section that the τ-destruction rule as given in Definition 4.7 is *context sensitive*, in the sense that the type of x has to be found "somewhere else" in the full term t. This we consider an imperfection. We show below that one can solve this matter by incorporating in a τ-item a segment $\overline{s}$, which contains all the types of the free variables in the rest of the term. The idea is, that this *generalized* τ-item travels through the term (from left to right) in the direction of $x = \mathbf{endvar}(t)$, collecting all possible binders for this x. Finally, the correct type is selected in the segment $\overline{s}$.

In this approach, τ-items have the form $(\overline{s}\tau)$. When using the term construction as described in Section 2.5, there is a natural $\overline{s}$ at hand, as we shall show.

The general τ-transition rules, as given below, will be obvious. In order to be able to select the correct type out of $\overline{s}$, we introduce ϑ-items $(\overline{s}\vartheta^{(i)})$. (Formally, we now have $\Omega_{\lambda\delta\sigma\varphi\tau\vartheta}$-terms.)

The application condition as given in the previous section, now looks as follows:
$(\overline{s}\tau)t' =_{\tau,\vartheta,\beta} (t''\lambda_1)t_1$ and $(\overline{s}\tau)t =_{\tau,\vartheta,\beta} t''$.

It will be clear that each check of this application condition *generates* two τ-items. Therefore, the application condition can be considered to represent the *general τ-generation rule.* The other general ("context-free") rules for τ-items and ϑ-items are the following (cf. Definition 4.9; also compare the ϑ-rules with the $\zeta_\star$-rules of Definition 2.23):

Definition 4.12 *(general τ-reduction)*

(general τ-transition rules:)
$(\overline{s}\tau)(t_1\lambda_{i+1}) \to_\tau (t_1\lambda_i)(\overline{s}(t_1\lambda_{i+1})\tau)$
$(\overline{s}\tau)(t_1\delta) \to_\tau (t_1\delta)(\overline{s}\tau)$
(general τ-destruction/ϑ-generation rule:)
$(\overline{s}\tau)x \to_\vartheta (\overline{s}\vartheta^{(x)})x$
(general ϑ-transition rules:)
$(\overline{s}(t_1\lambda_i)\vartheta^{(k)})x \to_\vartheta (\overline{s}\vartheta^{(k-1)})x$ *if* $k > 1$
$(\overline{s}(t_1\delta)\vartheta^{(k)})x \to_\vartheta (\overline{s}\vartheta^{(k)})x$
(general ϑ-destruction rule:)
$(\overline{s}(t_1\lambda_i)\vartheta^{(1)})x \to_\vartheta (\varphi^{(x)})t_1$

Note that the proper type for x is selected by erasing main items, one by one, at the end of the segment $\overline{s}$. The upper index k of the ϑ is counting the number of λ-items that have been erased (plus one).

Remark 4.13 (1) The rules above can be extended with rules for the transition of τ- and ϑ-items over σ- and φ-items, if one desires so.

(2) Note that the context-free rules as given above miss the property that all values of variables reflect the proper binding place: the free variables in the body $\overline{s}$ of the traveling τ-item are not updated during the step-wise transition process, but only after the application of the general ϑ-destruction rule.

In this sense, the transition rules are not general enough. (Cf. the "general" step-wise substitution of Section 3.3, which *do* have this desirable property as regards the binding structure.)

This imperfection can be mended in a similar manner as in Definition 3.5, by changing the first general τ-transition rule and the general φ-destruction rule into the following rules:

(general τ-transition rule for λ, context-free version:)
$(\overline{s}\tau)(t_1\lambda_{i+1}) \to_\tau (t_1\lambda_i)((\varphi)\overline{s}(t_1\lambda_{i+1})\tau)$
(general ϑ-destruction rule, context-free version:)
$(\overline{s}(t_1\lambda_i)\vartheta^{(1)})x \to_\vartheta \mathtt{sieveseg}_\varphi(\overline{s})t_1$
(For $\mathtt{sieveseg}$: see Definition 2.10.)

In the rest of this report, we shall use the *context-sensitive* τ-reduction as introduced before; that is, we only use τ-items of the form (τ), and not τ-items of the form $(\overline{s}\tau)$. However, all that follows can also be expressed by means of *context-free* τ-reduction as defined in the present section.

4.4 The typing relation in PTS's

In this section and the following ones we discuss how the canonical type operator of Section 4.2 is related to the usual typing relation in different systems. We also show how this typing relation can be expressed with the help of the τ-"operator". The systems that we investigate in these respects are Pure Type Systems and a few systems which are members of the Automath-family.

We start with a short summary of so-called Pure Type Systems (*PTS's*), as described in [Barendregt and Hemerik 90]; see also [Barendregt 9x]. We are only interested in the *singly sorted* PTS's, where different types of a given term are always β-convertible; hence, typable terms are *uniquely typed* (but for β-conversion). Moreover, we require that the typing relation is degree-consistent, thus preventing "impredicative typing" like $* : *$.

PTS's do not have the ultimate fine-structure which we described in the previous chapters. The smallest reduction-"steps" are β-reduction steps. There is no explicit substitution in PTS's, hence there is no counterpart of σ-items. Moreover, PTS's employ ordinary variables, and not de Bruijn-indices or another referential variable denotation. So also φ-items and updating are not incorporated. Finally we note that PTS's have a typing relation $t_1 : t_2$ (i.e. term t_1 has type t_2), and no canonical type operator as the one explained in Section 4.2.

First we give the Π-formation and Π-introduction rules of Pure Type Systems. These Π-rules give the conditions which must be obeyed for the construction of (λ- or Π-) abstraction terms. The type information plays an important role. One can consider the Π-rules to embody the *abstraction conditions* for PTS's.

Definition 4.14 *(Π-rules)*

(Π-formation rule:)

$$\frac{\Gamma \vdash t_1 : s_1 \quad \Gamma,\ x : t_1 \vdash t_2 : s_2}{\Gamma \vdash (\Pi x : t_1 \,.\, t_2) : s_3}$$

(Π-introduction rule:)

$$\frac{\Gamma \vdash t_1 : s_1 \quad \Gamma,\ x : t_1 \vdash t_2 : s_2 \quad \Gamma,\ x : t_1 \vdash t' : t_2}{\Gamma \vdash (\lambda x : t_1 \,.\, t') : (\Pi x : t_1 \,.\, t_2)}$$

In these rules, Γ denotes a context, t_1, t_2 and t' are terms and s_1, s_2 and s_3 are so-called sorts. For convenience' sake, we only regard the case that $s_2 \equiv s_3$; these PTS's contain the ones of Barendregt's λ-cube (to be explained below).

The above rules may only be applied for a given set of triples (s_1, s_2, s_3). See below for a further explanation of these matters.

Note: Do not confuse these sorts s_1 and s_2 with meta-variables s_1 and s_2 for items.

The Π-formation and Π-introduction rules as given above can be condensed into one Π-rule:

Definition 4.15 *(combined Π-rule)*

$$\frac{\Gamma, [x :]t_1 : s_1 \vdash [t' :]t_2 : s_2}{\Gamma \vdash [(\lambda x : t_1 \,.\, t')] : (\Pi x : t_1 \,.\, t_2) : s_2}$$

As already noted in Section 4.2 under the heading "The type of an abstraction", Π-items are "not counted" when the type of a Π-term is established. We see this in the Π-formation rule: the type of $\Pi x : t_1 \,.\, t_2$ is the same as the type of t_2 by itself (viz. s_2).

When, however, the type of a λ-term is desired, then the Π-introduction rule makes us change the λ-item into the corresponding Π-item: the type of $\lambda x : t_1 \,.\, t'$ is $\Pi x : t_1 \,.\, t_2$, given that the type of t' is t_2.

This is also expressed in Definition 4.8, where $(\tau)(t_1\lambda_1)$ τ-reduces to (τ) by itself (the λ_1-item — i.e. the Π-item — is erased). On the other hand, $(\tau)(t_1\lambda_2)$ τ-reduces to $(t_1\lambda_1)(\tau)$, so the λ_2-item (an ordinary λ-item) changes into the corresponding λ_1-item (a Π-item).

Hence, Definition 4.8 encorporates the essential part of both Π-rules, translated in our setting.

In the case of "Barendregt's cube" both s_1 and s_2 can be either $*$ or $\Box$ (again, see [Barendregt 9x] or [Barendregt and Hemerik 90]). These two are related by the *axiom* statement: $* : \Box$.

In Automath-like interpretations (cf. [de Bruijn 80], [Nederpelt 80] or [Nederpelt 90]), $*$ can be interpreted as the class of sets and/or the class of propositions, and $\Box$ as the "superclass" of these two classes.

In Barendregt's cube, there are eight systems of typed lambda calculus. They differ in whether $*$ and/or $\Box$ may be taken for s_1 and s_2, respectively. (We recall that we take $s_2 \equiv s_3$.) The basic system is the one where $(s_1, s_2) = (*, *)$ is the only possible choice. All other systems have this version of the two Π-rules, plus one or more other combinations of $(*, \Box)$, $(\Box, *)$ and $(\Box, \Box)$ for (s_1, s_2). The four possible versions of the Π-rule can be listed as follows:

degree	3		2		1		0
$(*,*)$	x	:	t_1	:	$*$	:	$\Box$
	t'	:	t_2	:	$*$	:	$\Box$
$(*,\Box)$	x	:	t_1	:	$*$	:	$\Box$
			t'	:	t_2	:	$\Box$
$(\Box,*)$			x	:	t_1	:	$\Box$
	t'	:	t_2	:	$*$	:	$\Box$
$(\Box,\Box)$			x	:	t_1	:	$\Box$
			t'	:	t_2	:	$\Box$

The system with only $(*,*)$ for (s_1, s_2) is known as λ-Church or $\lambda\rightarrow$ (this is essentially the Automath-system AUT-68). The addition of $(*,\Box)$ gives λP, which is a system that is rather close to another variant of the Automath-family, AUT-QE (see [de Bruijn 80]). The addition of $(\Box,*)$ to $(*,*)$ gives the second order typed lambda calculus, also called $\lambda 2$. Adding $(\Box,\Box)$ to $(*,*)$, we obtain $\lambda\underline{\omega}$. There are three systems that are defined by adding a combination of two of the three last-mentioned possibilities to $(*,*)$. When all mentioned (s_1, s_2)-combinations are permitted, we have a version of the Calculus of Constructions (λC) (see [Coquand and Huet 88]).

In our system, we may identify $\Box$ with ε (see Section 2.1). Subsequently, the axiom $* : \Box$ may be rendered as the λ-item $(\varepsilon\lambda_*)$. Thus we can express all eight systems of Barendregt's cube (and, in fact, many other PTS's) by adding the appropriate abstraction conditions. (See Section 4.6 below.)

Just as the Π-formation and -introduction rules incorporate the PTS-version of the abstraction conditions, the following Π-elimination rule contains the *application condition* for PTS's:

Definition 4.16 *(Π-elimination rule)*

$$\frac{\Gamma \vdash F : (\Pi x : A \,.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$$

Here $[x := a]$ stands for the substitution of a for (free occurrences of) x. We already discussed the contents and the implications of this rule under the heading "The type of an application" in Section 4.2.

Summarizing, it is our opinion that the main rules for term construction in many PTS's have a natural rendering in our setting. The construction of abstraction terms can be simulated with the use of λ_1- and λ_2-items. Application terms can be constructed with an appropriate application condition, which mirrors the Π-elimination rule but for the difference between implicit (compulsory) and explicit $\beta_{\delta\Pi}$-reduction. However, the latter kind of $\beta_{\delta\Pi}$-reduction, being more general, and fitting naturally in our setting, can be used to establish the same effects as the former one.

Remark 4.17 The fact that systems with explicit $\beta_{\delta\Pi}$-reduction are conservative over systems with implicit $\beta_{\delta\Pi}$-reduction, has been proven by van Benthem Jutting (private communication). Hence, there is no technical objection against the definition of PTS's by means of a canonical type operator. Note, however, that Jutting did not define types for Π-application terms like $(\Pi x : A \,.\, B)a$.

4.5 The typing relation in Automath-systems

In this section we describe the definitions of three Automath-systems in our setting. These systems do not have the fine-structure which we gave in previous chapters. Reduction is the ordinary β-reduction, substitution is a meta-operation; hence, we do not take σ-items and φ-items into account.

The systems *do* have a canonical type operator, albeit not as part of its language. Consequently, we have $\Omega_{\lambda\delta}$-terms in the language. Moreover, there is just one δ and one λ, this λ taking the role both of the ordinary functional operator λ and the product constructor Π.

In the present section we discuss the systems AUT-68, AUT-QE and Λ. All these systems have been developed around 1970. The oldest of the three is AUT-68, the more powerful variant AUT-QE followed soon. These two systems are invented by de Bruijn.

The system Λ was meant to be a simplified and more uniform version of the two other systems. It was developed slightly later. Nederpelt started studying these systems in [Nederpelt 71] by putting contexts as λ-segments *in front of* the AUT-terms and, moreover, identifying the concept "instantiation" (of a defined notion with respect to a chain of arguments) with repeated "λ-application" (of a function to a number of arguments). De Bruijn simplified the system still further by omitting definitions, obtaining a typed lambda calculus (AUT-SL, called Λ in [Nederpelt 73]) with lambda-term-like types. Recently, de Groote studied a variant of Λ in his thesis (see [de Groote 91]).

A much more recent Automath-system, meant to give Λ the same power — especially as regards definitions — as e.g. AUT-QE, is the system $\Delta\Lambda$, which will be discussed in Section 4.6.

All Automath-systems have the property of degree-consistency (see Section 4.1), both for the typing relation and for β-reduction. (The same observation holds for the systems in Barendregt's cube, but *not* for general PTS's.)

1. The system AUT-68

The system AUT-68 was meant as a formal system suitable for expressing large parts of mathematics, including all the usual features like:

- the possibility to use it for reasoning, in a logic chosen by the user (e.g. classical predicate logic, intuitionistic logic); but note that no logical constants have been built in, the lambda calculus frame itself can be used for expressing most of the logical notions,
- the possibility (again without built-in constants) of a step-wise development of a mathematical theory by means of axioms and primitive notions; lemma's, theorems, corrolaries and their proofs; definitions and abbreviations,
- an explicit treatment of contexts (assumptions, variable introductions) for theorem-like and definition-like notions.

(For more details about these matters, see [Nederpelt 90].)

If we *disregard the definition mechanism* of AUT-68 (otherwise said: if all definitions are "unfolded"), then we can give a simple, straightforward description of AUT-68 in our setting by choosing the appropriate parameters[1] (in this simple version it is also not possible to use primitive notions in a certain context).

Beforehand, we note that only degrees 1, 2 and 3 are permitted. Hence, ε (of degree 0) is not an Automath-term. As a consequence, the λ-item $(\varepsilon\lambda_*)$, expressing that $*$ is of type ε, is a "meta-axiom", which cannot be rendered inside one of the described Automath-systems.

The remaining parameters for AUT-68 can be listed as follows:

- The **canonical type** $\mathtt{typ}(t')$ of a term t' can be calculated by means of the following τ-transition rules:

[1] This version of AUT-68, together with the following description of AUT-QE, have been elaborated by Bert van Benthem Jutting

$$(\tau)(t\lambda)t' \to_\tau \begin{cases} * & \text{if } \mathtt{deg}(t') = 2 \\ (t\lambda)(\tau)t' & \text{if } \mathtt{deg}(t') = 3 \end{cases}$$

$$(\tau)(t\delta)t' \to_\tau (t\delta)(\tau)t'$$

- As regards the **variable construction** rule of Section 2.5, namely:

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

  we have as variable condition that the only variable of degree 1 is $*$.

- In the **abstraction construction** rule of Section 2.5, namely:

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

  we have as abstraction conditions:
  (1) Either $\mathtt{deg}(t) = 2$, or $\mathtt{deg}(t) = 1$ and $\overline{s}$ is a context (i.e. a segment consisting only of λ-items), and (2) $2 \leq \mathtt{deg}(t') \leq 3$.

- In the **application construction** of Section 2.5, namely

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

  we take the following application condition:

  $$\mathtt{deg}(t') = 3 \textit{ and } \mathtt{typ}(t') =_\beta (\mathtt{typ}(t)\lambda)t'' \textit{ for some } t''$$

(Remark. The given τ-transition rules for AUT-68 can be justified by the following observation. In AUT-68 the main λ-items in any (sub-)term of degree 3 play the role of the ordinary functional operator λ and in terms of degree 2 they can be interpreted as the product constructor Π.)

2. The system AUT-QE

The system AUT-QE has so-called Quasi Expressions: abstractions over $*$, functioning as types of dependent products. This extra feature facilitates the applicability of the system in a mathematical environment.

AUT-QE has, like AUT-68, only terms of degree 1, 2 and 3.

The other parameters for (again) a *definition-free* version of AUT-QE are:

- (Canonical type) As for AUT-68.
- (Variable condition) As for AUT-68.
- (Abstraction condition 1) As for AUT-68.
- (Abstraction condition 2) Absent.
- (Application condition)
  Either $\mathtt{deg}(t') = 3$ and $\bar{s} \vdash (t\delta)\mathtt{typ}(t')$,
  or $\mathtt{deg}(t') = 2$ and $\mathtt{typ}(t') =_\beta (\mathtt{typ}(t)\lambda)t''$ for some term t''.

Remark 4.18 The uniform behaviour of the type operator in the various Aut-systems, together with the non-distinction of λ and Π, causes difficulties in practical applications: a consequence is, that the type of a Π-abstraction is again an abstraction, which is not what one usually desires (the type of a universally quantified predicate is a proposition, and not a *function* over the type of all propositions).

In the original system AUT-QE (*with* definitions and primitive notions in a context), there are so-called *type-inclusion* rules which are meant to undo these undesired effects caused by the "overloading" of the single λ. The rule enables one to "strike out" certain strings of λ-items and/or segments which immediately precede an x with $\mathtt{typ}(x) = *$.

This is in accordance with the natural definition of types for Π- and λ-abstractions, which we discussed before (see e.g. the paragraph "The type of an abstraction" in Section 4.2). We recall that, for a term t' of type $t'' = *$, we have that the type of $\lambda x : t_1 . t'$ is $\Pi x : t_1 . *$, but the type of $\Pi x : t_1 . t'$ is only $*$. When identifying λ and Π, as is the case in many Automath-languages, we have that the type of $\lambda x : t_1 . t'$ is *either* $\lambda x : t_1 . *$ *or* $*$. Hence, technically spoken, one desires the possibility to strike out the λ-item $\lambda x : t_1$ in the type.

Note that the unicity of types is no longer valid in such a system. Moreover, there is a kind of *non-determinism* since there may be different types for a certain type.

However, in the simplified AUT-QE version as given above, these problems do not play a role.

3. The system Λ.

In view of the sketched development of Λ as a uniform system (however maintaining most of the possibilities for practical applications in logic

and mathematics), it will be no surprise that Λ is the system closest to the approach that we follow in this report. Moreover, Λ does not have the simplifications that we applied above: abbreviations (i.e. definitions) are possible, and so are primitive notions in a context. Abbreviations, however, cannot be used at full capacity. This is partly due to the absence of type- inclusion (see Remark 4.18), partly to a syntactic feature. The latter restriction can easily be removed (as has been done in the system $\Delta\Lambda$, see Section 4.6).

As a matter of fact, Λ is contained in our description as given before, with the following parameters:

- There is no restriction on degrees, all degrees ≥ 0 are possible.
- There is only one abstraction operator λ (hence, there is no Π, or $\lambda_0, \lambda_1, \lambda_2, \ldots$). This restriction complicates the practical use of Λ, since Π-abstractions cannot be easily distinguished from λ-abstractions (cf. Remark 4.18).
- Application is only restricted in the sense that the general application condition of Section 4.2 must hold, albeit in a generalized version (due to the unlimited degrees) which we will discuss in Section 4.6. Application is allowed for terms of all degrees, so that Π-application (see again Section 4.2) is one of the features: β-reduction is treated similarly for all degrees, in the form $(t_1\delta)(t_2\lambda_x)t_3 \rightarrow_\beta t_3[x := t_1]$.
- The type operator behaves uniformly, as in Definition 4.7: we have that $(\tau)(t_1\omega) \rightarrow_\tau (t_1\omega)(\tau)$, for $\tau \equiv \lambda$ or $\tau \equiv \delta$. Hence, Λ has explicit, and not implicit (compulsory) $\beta_{\delta\Pi}$-reduction.

For more details about the behaviour of Λ (Church-Rosser, normalization, etc.), see [Nederpelt 73].

4.6 Remarks on the conditions in term construction

In this section we give more details about a number of possibilities for the conditions used in the construction of terms, as introduced in Section 2.5. We shall go into variable conditions, abstraction conditions and application conditions. The most commonly used versions of the lastmentioned two conditions have been discussed in Section 4.2. Examples of all three conditions have been given in Section 4.5. One extended application condition that we describe will lead to an adaptation of the system Λ, called $\Delta\Lambda$.

In this section we will not take σ- and φ-items into account. Hence, we only regard $\Omega_{\lambda\delta\tau}$-terms.

1. Variable conditions

A variable condition that is often used, is the one that has been explained in Section 2.5, serving to restrict the collection of terms to closed ones. Another restriction is often imposed on the degree of a term. For example, one may require that the degree is always ≥ 1 and ≤ 3 (as it is the case in AUT-QE and AUT-68; see the preceding section). The reasonableness of such a requirement is shown in practical applications. For example, large pieces of mathematical texts have been coded in AUT-QE, thereby demonstrating its utility. (An exception is AUT-4, where degrees range over 1 to 4; cf. Section 4.7.)

2. Abstraction conditions

As already mentioned in Section 4.4, both the Π-formation and Π-introduction rules for PTS's (see Definition 4.14) can be expressed by means of appropriate abstraction conditions. For ease of reference, we repeat the construction rule under consideration, as stated in Section 2.5:

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

Now the Π-formation rule can be obtained by reading λ_1 for λ and taking the following abstraction condition (see also Section 4.4):
$(\tau)t \twoheadrightarrow_{\tau,\beta} s_1$ and $(\tau)t' \twoheadrightarrow_{\tau,\beta} s_2$, for $s_1, s_2 \in \{*, \Box\}$.
For the Π-introduction rule we take λ_2 for λ and the abstraction condition $(\tau)t \twoheadrightarrow_{\tau,\beta} s_1$ and $(\tau)^2 t' \twoheadrightarrow_{\tau,\beta} s_2$.
Here $(\tau)^2$ is an abbreviation for $(\tau)(\tau)$.

The Π-formation and Π-introduction rule also give information about the type (via the statements $(\Pi x : t_1 \,.\, t_2) : s_2$ and $(\lambda x : t_1 \,.\, u) : (\Pi x : t_1 \,.\, t_2)$, respectively). This is no longer necessary, since we have the canonical type operator τ at our disposal (cf. Definition 4.8 and Remark 4.11).

3. Application conditions

As regards the Π-elimination rule for PTS's, we can use the appropriate construction rule from Section 2.5:

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

The application condition for the PTS-case should read:
there are t'' and t_1 such that $(\tau)t' =_{\tau,\beta} (t''\lambda_1)t_1$ and $(\tau)t =_{\tau,\beta} t''$.
In words: the type of t' must be β-convertible to some λ_1-abstraction (i.e., a Π-abstraction); the domain t'' of this λ_1-abstraction must be such that t fits into it, i.e., this domain t'' should be β-equivalent to the type of t. (We explained this before in Section 4.2.)

Again, the type information $Fa : B[x := a]$ can be extracted from the definitions concerning the canonical type operator τ. As we noted before, the substitution *result* $B[x := a]$ is rendered as a Π-application $(t\delta)(t''\lambda_1)t_1$, which is able to yield a similar substitution result, but only *after* a $\beta_{\delta\Pi}$-reduction generated by the δ-λ-segment $(t\delta)(t''\lambda_1)$.

This version of the application condition only concerns "ordinary functions", i.e. F such that $F : (\Pi x : A . B)$. A more general application condition, comparable with the general typing rule for λ-items of Definition 4.9, is based on an iterated version of τ. For example, when $t : t'$, $t' : t''$ and $t'' : (t_1\lambda_1)t_2$, then t'' is some kind of a function, but also t' and t can be regarded to be "functions", all having "domain" t_1. Hence, one might agree to permit that each of the terms t'', t' and t may be applied to a term having type t_1.

In systems without a distinction between λ and Π (e.g. Λ), this can be solved as follows. Extend the application condition to higher degrees:

There are t_2 and j such that $(\tau)^j t' =_{\tau,\beta} ((\tau)t\lambda_1)t_2$

The iterated τ-item here has the usual meaning: $(\tau)^0 \equiv$ identity, $(\tau)^1 \equiv (\tau)$, $(\tau)^2 \equiv (\tau)(\tau)$, and so forth. Instead of $(\tau)^j$ we may also take $(\tau)^\infty$, being the "highest $(\tau)^j$ possible in the given circumstances".

Note: This maximal j is indeed well-defined, as can easily be seen. The general application condition in this form can already be found in [Nederpelt 73]. The τ-free term to which $(\tau)^\infty t'$ τ-reduces is called the *final type* of t' in [de Bruijn 9x].

An application condition similar to those described above, is present in the system Λ. This kind of application condition may look very general, but it misses a feature that is really desirable.

The shortcomings of this kind of application condition are a result of the β-reduction, as present in the application condition $(\tau)^j t' =_{\tau,\beta} ((\tau)t\lambda_1)t_2$. Here one does not take into account that the construction rule under consideration contains a context $\overline{s}$, which may play an essential role in two respects:

- for *typing*, because $\overline{s}$ may contain λ-items $(t_1\lambda)$ which contain a type for all free variables bound by that λ, also in the part of the term after the $\vdash$;
- and for *reduction*, since a segment $\overline{s}$ may contain definitional δ-λ-segments which may affect the rest of the term.

Hence, segments may cause both *weak* and *strong* bindings in the rest of the term (cf. Section 3.5).

To make things clearer, we discuss the latter case. In $\overline{s}$ there may occur δ-λ-segments, or even δ-λ-couples, as main subsegments. Such a δ-λ-segment or -couple may contain a definition: $(t_1\delta)\ldots(t_2\lambda_x)$, with matching $(t_1\delta)$ and $(t_2\lambda_x)$, which can encode the definition of x as being (an abbreviation for) t_1. Now in the terms behind the symbol $\vdash$, both x and t_1 can occur, being *not* β-convertible as long as $\overline{s}$ is not taken into account. In meaning, however, x and t_1 are "the same", which cannot be established syntactically.

For this reason, de Bruijn always regarded Λ as being a typed lambda calculus without definitions, or: a typed lambda calculus for mathematical theories in which all definitions have been elaborated. This is a correct observation. In theory this is harmless. In practice, however, it is highly undesirable to have no definitions at one's disposal, since texts then grow exponentially in size.

To cope with this inconvenience, de Bruijn proposed $\Delta\Lambda$. The difference is, that the above-mentioned β-reduction in the application condition $((\tau)^j t' =_{\tau,\beta} ((\tau)t\lambda_1)t_2)$ must be adapted to $(\tau)^j \overline{s}t' =_{\tau,\beta} \overline{s}((\tau)t\lambda_1)t_2$. This simple intervention permits us to use definitional δ-λ-segments or -couples in $\overline{s}$ in the quest for τ-β-convertibility. See also [de Bruijn 9x].

(Note that the use of the context-free τ-reduction of Section 4.3 does not help here. To be precize: a condition like $(\overline{s}\tau)^j t' =_{\tau,\beta} ((\overline{s}\tau)t\lambda_1)t_2$ would only cause the desired effect if σ-items generated as a main item in $\overline{s}$ could *"pass"* the mentioned τ-items.)

4.7 Higher degrees

As already mentioned in Section 4.6, under circumstances there may be a desire for higher degrees of terms. As an example we discuss the essential features of AUT-4, a member of the Automath-family.

The idea is the following. In the propositions-as-types conception (see e.g. [Howard 80]), propositions are coded as lambda terms. When t is a term which is regarded as a proposition, then any "inhabitant" of t — i.e., a term

t' such that $t' : t$ — serves as an assertion (a "proof") of that proposition. There clearly is a strong parallel with sets and elements: when t codes a set, and when t' is again an inhabitant of t, then t' represents an element of the set t.

When a term t is considered to represent a set and when it turns out that t is *not* inhabited, then t may be considered to represent an empty set. Analogously, when t codes a proposition and t is *not* inhabited, then the proposition t may be considered to be not provable.

(Note: the logical counterpart of this propositions-as-types notion is the Brouwer-Heyting-Kolmogorov interpretation; for further explanation: see [Troelstra and van Dalen 88].)

When t *is* inhabited, then the comparison does not hold completely. A set can have many elements, and a proposition can have many proofs. The elements of a set are considered to be different, but it may be useful to *identify* all proofs of a certain proposition. This is because — from the point of view of classical logic — the important thing is often whether there *is* a proof of a proposition, and not so much what the exact content of the proof is.

In most of the systems which have been studied up to now, sets and props occupy the same level in the degree-hierarchy. One presupposes, for example, a class of sets ($*_s$) and a class of propositions ($*_p$), both inhabitants of the "super-class" $\Box$. The situation then is as follows:

degree	3	2	1	0
term	a :	A :	$*_s$:	$\Box$
interpr.	element	set	class of sets	
term	P :	Q :	$*_p$:	$\Box$
interpr.	proof of Q	prop	class of props	

In this schema it is possible to treat proofs and elements in a different manner. For example, one could define an equivalence $=_i$ for proofs, viz. for those terms t of degree 3 for which $(\tau)^2 t =_\beta *_p$.

Another way to identify proofs is the following. In the previous diagram one shifts the proof-prop row one column to the left, adding a class Δ between $*_p$ and $\Box$. Now proofs become the only terms of degree 4:

degree	4	3	2	1	0
term		a :	A :	$*_s$:	$\Box$
interpr.		element	set	class of sets	
term	P :	Q :	$*_p$:	$\triangle$:	$\Box$
interpr.	proof of Q	prop	class of props		

This is the AUT-4 interpretation (see [de Bruijn 74]). "Irrelevance of proofs" can now be implemented by a rule of the following form, where $=_i$ is some equivalence:

$$\frac{\Gamma \vdash P : Q : *_p : \triangle \qquad \Gamma \vdash P' : Q' : *_p : \triangle \qquad Q =_\beta Q'}{P =_i P'}$$

The use of the equivalence $=_i$ can best be demonstrated by means of an example. The natural logarithm ln, considered on the domain of the real numbers, depends essentially on *two* arguments: a real number and a proof that this real number is positive. So one has to consider $ln(x,p)$ instead of $ln(x)$, identifier p being a (reference to) a proof that $x > 0$.

Now assume one has two proofs of the positiveness of the number 3, coded as t_1 and t_2, respectively. Then it is desirable to have that $ln(3,t_1)$ and $ln(3,t_2)$ are "equal". This can be achieved by declaring t_1 and t_2 to be equivalent and adapting the overall equivalence of terms accordingly.

5 Abbreviations for segments

5.1 The use of segment variables

In the previous chapter, contexts Γ (see Section 4.4) and segments $\overline{s}$ (see Sections 2.5 and 4.6) played a role in the construction of terms. These notions are also of value in their own right. For example, mathematical phrases like "Let x be a natural number" or "Assume that P is true" can be expressed as λ-items: $(\mathcal{N}\lambda_x)$ and $(P\lambda_t)$ (cf. Section 4.7). A more complicated phrase like "Let $(G, \cdot, e, ^{-1})$ be a group" is basically a segment-like concept: it is the composition (product) of "Let G be a set, let $\cdot$ be a function $G \times G \to G$, assume $\cdot$ to be associative, let e be an element of G, assume that e obeys the properties for a unit element, let $^{-1}$ be a function $G \to G, \ldots$".

Essentially, syntactic meta-constructs like $\Gamma \vdash t : t'$ or $\overline{s} \vdash t : t'$ express that the statement $t : t'$ "holds" in *context* Γ or with *context segment* $\overline{s}$. Here Γ (or $\overline{s}$) is related to both t and t'. In fact, $\Gamma \vdash t : t'$ expresses that t *relative to* Γ has type t' *relative to* Γ. So there is a twofold reference to Γ. Without the contextual part "$\Gamma \vdash$" one would need two copies of Γ. Hence, the syntactic constructs $\Gamma \vdash t : t'$ and $\overline{s} \vdash t : t'$ contain a device which prevents us from unnecessary copying a string.

But even with this more concise notation, repeated duplications of segments are unavoidable. For example, when Γ expresses "Let $(G, \cdot, e, ^{-1})$ be a group", then in ordinary circumstances many statements follow in this context Γ. For every such statement $t_i : t'_i$ we have to mention the context: $\Gamma \vdash t_i : t'_i$ for $i = 1, 2, \ldots$.

In practice, this is a nuisance. Hence, some abbreviation mechanism for contexts is very welcome. De Bruijn devised a nice context administration for the main languages of the Automath-family (see [de Bruijn 70]).

When taking the line that typed lambda calculus is an appropriate language for many purposes concerning the formalization of logical and mathematical notions and also for concepts from programming languages — as we do in this report, cf. [Nederpelt 90] — it would be nice to adopt some kind of context abbreviation *in this typed lambda calculus.* (Of course, this is a matter of taste. One is not obliged to do so. Context administration can be dealt with at a meta-level, or with a language extension.)

In our conception of what the structure of a typed lambda calculus really is, there is, at least in principle, no problem in this respect. In fact, contexts and segments can be regarded as special terms in the calculus, viz. those terms ending in ε. Now terms can be abbreviated in a definition, as we saw

before. Hence, in particular, contexts and segments can be abbreviated. All this holds under the condition that we consider $\overline{s}\varepsilon$ to be the same as $\overline{s}$ itself. (Remember that we already used the empty term instead of ε in Section 2.1.)

In this conception, segment $\overline{s}$ can be called a by adding the "definitional segment" $(\overline{s}\delta)(\lambda_a)$.

There are two remarks in this respect. Firstly, in order to reap full benefit from the abbreviations, we should allow that segment-abbreviating variables may occur in the place of actual segments everywhere in a term. For example, with the above definition, the term $(t\lambda_x)a(t'\lambda_y)z$ is an abbreviation for $(t\lambda_x)\overline{s}(t'\lambda_y)z$, with $\overline{s}$ completely copied out (but for the final ε, which is omitted!). Of course, adjustments are necessary in order to keep the referencing by means of de Bruijn-indices in order (see Section 5.2 below).

Secondly, the corresponding application condition in the term construction should say something about the type that is expected for $\overline{s}$ when $(\overline{s}\delta)$ is added. Above, we chose the empty term ε, rendered invisibly (cf. Section 2.1).

The approach described above will be elaborated in the following sections.

Remark 5.1 In a sense, the introduction of segment-abbreviating variables, as the a in the term $(t\lambda_x)a(t'\lambda_y)z$ discussed above, disturbs our uniform item-notation, since variables now not only occur at the end of a term, but also on other places. Otherwise said, it does no longer hold that each term is the concatenation of a segment and a variable.

A technical solution to this (technical) inconvenience is to introduce a new kind of items, namely *variable items*, denoted e.g. as $(a\psi)$. The term above is then written as $(t\lambda_x)(a\psi)(t'\lambda_y)z$, or, if we also "itemize" ordinary variables: $(t\lambda_x)(a\psi)(t'\lambda_y)(z\psi)$. With this convention, a term is again a segment followed by a variable, or, in the second case, nothing more than a segment ending in a variable item $(x\psi)$ (which can be omitted if $x \equiv \varepsilon$).

Hence, there are three possiblities:

- to use (ordinary) variables and segment-abbreviating variables;
- to use ordinary variables and segment-variable-*items*;
- to use variable-items for both categories of variables.

Since we regard it to be a matter of minor importance which of these options is chosen, we will, with a view to the foregoing part of this report, continue employing the first option, hence without variable-items.

5.2 Referencing in relation with segment variables

Things are, however, not so simple as suggested in the previous section. The main problem is the question of the right referencing. For example, in the term $(t\lambda_x)a(t'\lambda_y)z$, where a abbreviates a segment $\overline{s}$, the binding λ of the variable z may be found "inside" a, e.g. when $\overline{s} \equiv (t_1\lambda_u)(t_2\lambda_z)(t_3\delta)$. But neither λ_u nor λ_z is "visible" in a. Hence, using de Bruijn-index 2 for z would connect this variable with the wrong λ (viz. λ_x).

It will be clear from this example that the *λ-weight* of the abbreviated segment, i.e. the number of main λ-items in the segment, plays an important role. This number can always be recovered by inspecting the abbreviated segment. One can imagine, however, that it is more practical to register this number together with the segment variable. (A circumstance that facilitates this choice is, that *the λ-weight of a segment does not change by the local and global reductions which we described before*, provided that one avoids void reductions; see Section 3.4. This observation can easily be verified.)

Therefore, we add a collection of segment variables to our set of variables, which are pairs of numbers:

Definition 5.2 *(segment variables)*

We add to Ξ a new set Σ of **segment variables***:*

$\Sigma = \{(n;m)|n = 1,2,\ldots;m = 0,1,\ldots\}$.

Moreover, we distinguish the λ-operator $\lambda_{\mathbf{sg}}$ as being a binding λ for segment abbreviations. We do not allow that $\lambda_{\mathbf{sg}}$-items occur "on their own". They should always be a part of a δ-λ-segment of the form $(\overline{s}\delta)(\lambda_{\mathbf{sg}})$, coding the abbreviation of a segment $\overline{s}$.

In $(n;m)$, a **segment variable item***, the index n gives a reference to the binding $\lambda_{\mathbf{sg}}$ and m is the λ-weight of the abbreviated segment.*

Just as local β-reduction could be used to undo a definition, we can use local β-reduction for undoing segment abbreviations. For this purpose we have to extend our rules for β-reduction. We use the same σ's as before for substitution items originating from a segment abbreviation. This can give no confusion, as long as "ordinary" λ's bind "ordinary" variables and $\lambda_{\mathbf{sg}}$'s bind segment variables.

In our step-wise style, this leads to the following extension of the σ-rules in Definition 3.5 (note that the term t_1 in the rules below is a segment, so a term with $\texttt{endvar}(t_1) \equiv \varepsilon$):

Definition 5.3 *(general σ-reduction for segments)*
(general σ-generation rule for segments:)
$(t_1\delta)(\lambda_{\mathbf{sg}}) \rightarrow_\sigma (t_1\delta)(\lambda_{\mathbf{sg}})((\varphi)t_1\sigma^{(1)})$
(general σ-transition rule for segment variables:)
$(t_1\sigma^{(i)})(x;n) \rightarrow_\sigma (x;n)((\varphi^{(n)})t_1\sigma^{(i+n)})$ *($\sigma_{1\mathbf{sg}}$-transition)*
(general σ-destruction rule for segment variables:)
$(t_1\sigma^{(i)})(i;n) \rightarrow_\sigma t_1$, *provided that t_1 contains no main φ-items.*

(In the last-mentioned rule, the end variable ε in the right-hand side term t_1 should be suppressed, as explained before.)

The proviso in the destruction rule is necessary in order to prevent that the updating of the free variables in t_1 has undesired side-effects. To be precise: a main φ-item in t_1 does affect all free variables which occur in its scope. Since $(i;n)$ is embedded in a term like $(i;n)t'$, such an undesired situation may actually happen as soon as t_1 replaces $(i;n)$: some free variables of t' may then (erroneously) be updated, as well.

The rules above should be added to Definition 3.5 for *local* substitutions of segments for segment variables. For *global* substitutions we also need the following rule, since there may exist more segment variables $(j;n)$ which are bound by the $\lambda_{\mathbf{sg}}$ in the original δ-$\lambda_{\mathbf{sg}}$-segment:

Definition 5.4 *(second general σ-transition rule for segment variables:)*
$(t_1\sigma^{(i)})(i;n) \rightarrow_\sigma t_1((\varphi^{(n)})t_1\ \sigma^{(i+n)})$ *($\sigma_{01\mathbf{sg}}$-transition)*

(Here the same proviso as above must be taken into account.)

Note: we can replace the first destruction rule of Definition 3.5 by the one in Definition 5.3, by conceiving an ordinary variable x as a special case of a segment variable, with irrelevant weight: $x \equiv (x;\bullet)$.

5.3 Segments and stepwise substitution

In the previous section we saw how segment abbreviations in a δ-$\lambda_{\mathbf{sg}}$-segment may give rise to a σ-item (σ-generation rule for segments). This σ-item acts as a stepwise substitution operator, just as with ordinary δ-λ-segments, with only one difference: the σ-items originating from segment abbreviations "aim" at segment variables of the form $(x;n)$ and not at ordinary variables.

It may happen that σ-items occur in an abbreviated segment, either by being "shifted inside" due to the use of the transition rules, or due to being generated in that segment. In the case that such a σ-item is a *main* item of t_1, the stepwise shift of this σ-item through t_1 may come to a standstill at the end of the segment.

With the ordinary general σ-destruction rules, as introduced in Definition 3.5, the σ-item disappears when meeting an ε, as implicitly is the case at the end of a segment. However, this is not what we expect. The substitution should also influence all subterms occurring after a segment variable $(n; m)$ which refers to the segment under consideration, as can easily be seen.

In order to treat such a substitution instance properly, we differentiate between terms ending in ε, and abbreviated segments. For the latter constructs we add a special transition rule. Furthermore, we restrict the σ-destruction rules of Definition 3.5 to "proper" variables; these being either a natural number ≥ 1 or an ε which is intended to be the end variable of a term (and not of a segment). Or, using ψ-items: restrict the rules to variable-items $(x; \bullet)$, with $x \in \mathcal{N}$ or $x \equiv \varepsilon$. An ε marking the end of a segment then should be denoted differently, e.g., $(\varepsilon; \circ)$

This leads to the following extra rule:

Definition 5.5 *(segment-σ-generation rule)*
$(t'(t_1\sigma^{(k)})\delta)(\lambda_{\mathbf{sg}}) \to_\sigma (t'\delta)(\lambda_{\mathbf{sg}})(t_1\sigma^{(1,k)})$

As can be seen in this definition, the σ-item now obtains *two* arguments. The second, k, is meant to find the right substitution place for t_1. This index is, however, "frozen" until an appropriate segment variable has been found. This is because the segment abbreviation must be thought of as being apt for being inserted at the place of a corresponding segment variable. Hence, the search outside t' for the proper place where t_1 should be substituted, starts at the place of the occurrence of such a segment variable.

The first argument of this $\sigma^{(l,k)}$-item, the l, initialized on 1, is a counter which locates such an appropriate segment variable. The corresponding transition rules are quite similar to the σ-transition rules of Definition 3.5:

Definition 5.6 *(segment-σ-transition rules)*
$(t_1\sigma^{(l,k)})(t_2\lambda) \to_\sigma ((t_1\sigma^{(l,k)})t_2\lambda)$
$(t_1\sigma^{(l,k)})(t_2\lambda) \to_\sigma (t_2\lambda)(t_1\sigma^{(l+1,k)})$
$(t_1\sigma^{(l,k)})(t_2\lambda) \to_\sigma ((t_1\sigma^{(l,k)})t_2\lambda)(t_1\sigma^{(l+1,k)})$

$(t_1\sigma^{(l,k)})(t_2\delta) \to_\sigma ((t_1\sigma^{(l,k)})t_2\delta)$
$(t_1\sigma^{(l,k)})(t_2\delta) \to_\sigma (t_2\delta)(t_1\sigma^{(l,k)})$
$(t_1\sigma^{(l,k)})(t_2\delta) \to_\sigma ((t_1\sigma^{(l,k)})t_2\delta)(t_1\sigma^{(l,k)})$

Note that the $\sigma^{(l,k)}$-item $(t_1\sigma^{(l,k)})$ is isolated from the rest of the term as regards the correct referencing of free variables inside t_1. This is necessary, since some λ's binding these variables in the full term may not be present along the path leading to the root, viz. the λ's in the abbreviated segment.

We also need the following transition rule (cf. Definition 5.4):

Definition 5.7 *(special segment-σ-transition rule for segment variables)*
$(t_1\sigma^{(l,k)})\ (m;n) \to_\sigma (m;n)(t_1\sigma^{(l+n,k)})$

This rule expresses that a $\sigma^{(l,k)}$-item may pass a segment variable, provided that the weight of that variable is taken into account. Note that this also holds when $(m;n)$ is a segment variable bound by the segment abbreviation where the $\sigma^{(l,k)}$-item came into being! There may be another segment variable, also bound by this segment abbreviation, that is really intended as the place where the segment abbreviation should be undone, and it may be the case that *this other segment variable is still to come.*

The destruction of a $\sigma^{(l,k)}$-item, and its replacement by a $\sigma^{(k)}$- item, can be performed by the following rule.

Definition 5.8 *(segment-σ-destruction rule)*
$(t_1\sigma^{(l,k)})\ (l;n) \to_\sigma (l;n)((\varphi^{(n,l)})t_1\sigma^{(k)})$ *if* $k \leq n$,
$(t_1\sigma^{(l,k)})\ (l;n) \to_\sigma (l;n)((\varphi^{(n,l)})t_1\sigma^{(k+l)})$ *if* $k > n$

The first of these rules covers the cases that the $\sigma^{(k)}$-item was generated inside the δ-item of the δ-$\lambda_{\mathbf{sg}}$-item where the $\sigma^{(k,l)}$-item was generated. The second rule applies when the σ-item has been generated earlier. In case of a global reduction, we can simplify the latter rule. This is the case since the $\sigma^{(k)}$-item will meet the $(l;n)$ anyhow after following another, more direct path. The segment-σ-destruction rules then becomes:

Definition 5.9 *(segment-σ-destruction rule for global reduction:)*
$(t_1\sigma^{(l,k)})\ (l;n) \to_\sigma (l;n)$ *if* $k > n$.

6 Parameters for different systems

In the preceding chapters we sketched the general structure of a system of typed lambda calculus. We showed that this system has enough expressive power for the description of various existing system, ranging from Automath-like systems to singly-typed Pure Type Systems.

In order to be able to use our system in different applications, it is required that a number of *parameters* is adjusted. In the present chapter we give an overview of the different parameters, with references to the section where they are introduced or where a certain choice is made.

1. Operators

The "parameters" regarding operators establish the admitted collection and the appearance of these operators.

In Section 2.1 we already opened the possibility to use different operators. Both the abstraction operator λ and its mirror image, the application operator δ, can be indexed, if one desires so. In Section 3.4 we used this possibility for the first time, in distinguishing local and global reduction. For this purpose we suggested a difference between $\lambda_{\mathbf{loc}}$ and $\lambda_{\mathbf{glo}}$.

In Sections 4.2, 4.4 and 4.6 we actually employed this possible variation; firstly, by rendering the Π and the λ of Pure Type Systems by λ_1 and λ_2, respectively, and secondly by regarding an infinity of λ's: $\lambda_1, \lambda_2, \ldots$. In the same section, we mentioned a possible difference between application operators δ_1 and δ_2.

But apart from the λ('s) and the δ('s), we introduced other operators. A minor extension was caused by the introduction of search items ζ and ζ_*, as was done in Section 2.4. A rather central notion, however, is the σ-operator, introduced in Section 3.2. We used this operator with upper indices: $\sigma^{(i)}$. Another operator, related to the σ-operator, is the update operator φ, used with a double upper index: $\varphi^{(k,l)}$.

Moreover, in Section 4.2 we employed the type operator τ, as an expedient for the construction of the canonical type of a given term. In the "context free" version of τ-reduction (Section 4.3) we also introduced the operator ϑ.

Finally, in Chapter 5 we used the operator $\lambda_{\mathbf{sg}}$ as a binding λ-operator for segment abbreviations. In Segment 5.3 a variant of the σ-operator appeared, this time with *two* arguments.

By each choice concerning these operators, the system changes. In the report we mentioned $\Omega_{\lambda\delta}$-terms, $\Omega_{\lambda\delta\sigma\varphi}$-terms and $\Omega_{\lambda\delta\sigma\varphi\tau\vartheta}$-terms. But other

combination are possible, as well.

As soon as an operator like σ is incorporated in the system, it becomes to some extent a *first class citizen.* The operators λ and δ, possibly indexed, are first-class citizens in every system. But for the other operators one has to decide to consider them as such. Such a choice will also affect the formation conditions. For example, for a "first class" σ one should add a rule like the following:

$$\frac{\overline{s} \vdash t \qquad \overline{s}(t\sigma^{(i)}) \vdash t' \qquad \textit{substitution condition}}{\overline{s} \vdash (t\sigma^{(i)})t'}$$

2. Variables

As to the set of variables, there are also some parameters which have to be chosen. First, one has to decide whether one desires to adhere to name-carrying variables like x and y, or to use de Bruijn-indices, being natural numbers (see Section 2.1). Another choice concerns the variable (or is it a constant?) ε: is ε a special symbol or is it rendered invisibly? In the latter case one has to be careful in distinguishing *terms* ending in ε and *segments*, as noted in Section 5.1.

Finally, in the same section we suggested the use of variable *items* instead of ordinary variables and/or segment variables.

3. Degrees

Different choices are possible as to the degrees permitted. As we demonstrated in Sections 4.4, 4.5, 4.6 and 4.7, one often restricts the degrees. This subject can be considered a sub-topic of term construction (viz. in the variable condition), hence we may skip this paragraph and continue with the next one.

4. Term construction

Important parameters are the ones required for the selection of the "correct" (or admissible) terms out of the set of all terms. The "correctness rules" concerning these choices have been introduced in Section 2.5. Here the variable condition, the abstraction condition and the application condition were meant to select the terms that "behave well" in certain respects, e.g. as regards the types: is every "argument" (δ-item) related to a "function" (λ-item) and does the argument *match* the domain of that function?

Many examples of different versions of these three kinds of conditions have been discussed in the report. See Sections 2.5, 4.2, 4.4 and 4.5.

As noted just now in Subsection 1, there is a possibility to add term construction rules for e.g. σ- and φ-items.

5. Reductions

β-reduction

An important choice regarding the relation of β-reduction is, whether one chooses the standard β-steps $(t_1\delta)(t_2\lambda_x)t_3 \to t_3[x := t_1]$, or the fine-structure induced by simple σ-steps. In the latter case, one-step β-reduction is a *composed* reduction (see Section 3.4).

Another choice is the one between global and local β-reduction, as explained in Section 3.1. A combination is possible, as well, as is shown is Section 3.4. In the same section we introduced void β-reduction.

Finally, an important decision is whether one desires to use the ordinary β- reduction, starting from *δ-λ-pairs*, or the generalized β- reduction, which also allows reductions on the basis of a *δ-λ-couple.* For this subject, see Section 3.5.

η-reduction

In this report, we did not consider *extensional* or *η-reduction.* The main rule for one-step η-reduction is:
$(t_1\lambda_x)(x\delta)t_2 \to_\eta t_2$ if x does not occur free in t_2.

Hence, η-reduction amounts syntactically to a kind of *void λ-δ-removal*, where the δ-item concerned has a special form: its body only consists of a variable, which is bound by the λ-item under consideration.

(The name-free version of this reduction rule is:
$(t_1\lambda)(1\delta)t_2 \to_\eta (\varphi^{(-1)})t_2$ if t_2 contains no reference to the mentioned λ.)

It may be considered to be a parameter of the system whether one incorporates η-reduction or not.

σ-reduction

As to σ-reduction, one has to decide whether the "update-neglecting" form of Section 3.2 is chosen, or the general, "update-providing" one of Section 3.3. Yet another version is the $\overline{\sigma}$-reduction described in Appendix C.

In making this choice, one also has to consider whether σ-items are first-class or second-class citizens, as remarked in Subsection 1 above.

τ-reduction

For τ-reductions there are many possiblities:

- whether to use canonical types or not, with important consequences for the conditions in term construction (see Sections 4.2, 4.4, 4.5 and 4.6);

- whether to allow Π-application (and the corresponding $\beta_{\delta\Pi}$-reduction), or not, a decision which again has major consequences (see Section 4.2 ff);

- and, less important, whether to employ ordinary ("context-sensitive") or general ("context-free") type reduction (see Sections 4.2, 4.3 ff);

-finally, type-inclusion may or may not play a role (see Remark 4.18).

Again, one has to decide whether τ-items are first-class or second-class. In the former case, they are part of the language and should be treated consistently as such. In the latter case, τ-items are in a sense "meta", which has consequences for the formulation of the theory.

6. Segment abbreviations

In Chapter 5 we explained why segment abbreviations may be profitable. Hence, we have here another "parameter" for the system: incorporate segment abbreviations, yes or no. If we do, a differentiation between global and local substitutions for segments is possible (see Section 5.2).

7. Reduction strategies

A parameter which is not of importance for the system as such, but which *does* have consequences for an implementation, is the choice for the various reduction strategies.

In Section 3.6 we discussed a number of well-known strategies for β-reduction. However, there are many other strategies, for β-reduction, but also for each of the other reduction introduced in the report, and in particular as regards the order in which the different strategies (e.g. for σ-, φ- and τ-reduction) are intermingled.

We shall not discuss these matters here.

Remark 6.1 It will be clear that not all the parameters sketched above are *independent*. The consequences of the different choices and their interdependency is a subject for further research.

7 Conclusions

In this report it was our intention to investigate some structural aspects of term construction 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 Chapter 2 with a novel description of term formation, regarding abstraction and application as binary operations. 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. We also proposed a change in bracketing with respect to "operators" like λ and δ, writing e.g. $(t_1\lambda)t_2$ instead of $(t_1\lambda t_2)$. This gave rise to items like $(t_1\lambda)$, prefixed to a term t_2. In this manner we fostered the modularity of the notation.

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.

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) precisely the string of symbols that *precedes* this occurrence in the term.

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.

In Chapter 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.

It is our conviction that the step-wise substitution as introduced in this report 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.

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. The resulting reduction relations are clear and relatively simple. The relations "behave" nicely, both separately and in combination. (Note, however, that these matters are more complicated in the usual β-reduction setting, where δ-λ-pairs "disappear" immediately; in Appendix C we will mention a few difficulties occurring in this case.)

It can be a well-motivated wish to keep σ-items at a certain place in a term, without proceeding with the σ-reductions for which they are tailored. This can be the case with lazy evaluation. Moreover, there can occur a void δ-λ-segment which is not (yet) removed. In both cases new β-reduction may be hampered.

Both subjects can be treated by extending the notion of β-reduction in such a manner that not only *δ-λ-segments* can generate a σ-item (or a β-reduction), but such that also *δ-λ-couples*, consisting of two matching δ- and λ-items, can do so. The description of this generalized reduction can be given rather smoothly in item-notation, which is in contrast with the normal notation, where matching does not occur in a nested way. We demonstrated this with an example. It is in particular the "argument-before-function" notation that turns out to be advantageous in this respect.

The generalized form of β-reduction is worth to be studied separately. Part of this work has already been carried out before by the author in his Ph.D. thesis.

We also showed that the usual strategies for β-reduction can be expressed

concisely in our setting.

In Chapter 4 we looked at the role of the types. For typable terms we defined a canonical type, which can be effectively computed in a straightforward manner. The usual relation $t_1 : t_2$, i.e. term t_1 has as one of its types the term t_2, can also be expressed by means of this canonical type typ and β-reduction, viz. as $\mathtt{typ}(t_1) =_\beta t_2$.

We showed how type systems such as Barendregt's cube of Pure Type Systems can also be defined with this typ-operator in a rather uniform way. We also presented a number of Automath-systems in the proposed setting, which resulted in concise definitions for complicated systems. Next, we explained how the abstraction condition and the application condition, present in our alternative term construction rules, can be phrased in correspondence with the PTS-rules. (It seems, however, that only so-called *singly-sorted* PTS's fit in this frame. See [Barendregt 9x].)

It turned out that our approach is flexible enough for the expression of many type systems. Also, generalizations, for example leading to higher degrees, are straightforwardly attainable. A difference between functions (λ-terms) and dependent products (Π-terms) can be made by adapting the appropriate rules, whereas both kinds of abstractions still fit in the same framework, since they may be treated as two similar kinds of λ-abstraction. This turned out to hold to such an extent that application and β-reduction become also possible for Π-abstractions, thus simplifying and unifying the patterns. Moreover, generalizations are here possible as well, which may be advantageous.

We concluded Chapter 4 with a discussion in which we gave arguments why higher degrees may be useful in applications of typed lambda calculus.

In Chapter 5 we introduced segment variables and segment abbreviations, which are very useful in the development of coded mathematical texts. We explained use and meaning and gave examples about their usefulness.

The abbreviation facilities that we gave fit neatly in our setting. The result is a simple and direct mechanism, which, in our opinion, is more manageable than earlier proposals in this direction (see e.g. [de Bruijn 72] or [Balsters 86]). There are some complications as to referencing and stepwise substitution, but these can all be solved, again in a straightforward fashion. It appears that we do not need intricate reference transforming mappings, since a slight extension of the earlier given rules for φ-items can do the job as desired. A similar remark holds for the σ-items, which only

have to be temporarily "frozen" in order to attain the right place in the term (a segment variable).

Nevertheless, we think that this is one of the parts of the report that is not yet completely elaborated. We plan to give this subject more attention in the future.

Finally we demonstrated the power of the proposed setting by listing the possible "parameters" which enable the customer to adjust the system to his or her personal taste: everyone can compose one's favourite system.

Concluding, we note two things. The first is that, in our opinion, we showed that the drawbacks of de Bruijn-indices need not be greater than those of the usual name-carrying variables (think of name-clashes, α-reduction, the problems with non-unique binding variables, etc.). This becomes especially evident in our approach. The advantages of de Bruijn-indices, on the other hand, are obvious, and can be exploited.

As a second remark we emphasize that our report mainly deals with the exposition of a number of ideas about term construction, reduction, substitution, typing and segment abbreviation in typed lambda calculus. The report does not contain any formal justification of the soundness of its proposals. For example, it has not been investigated whether well-known theorems like the Church-Rosser theorem, the subject reduction theorem or the strong normalization theorem do hold for any of the systems under discussion. Especially for extensions like the system with explicit step-wise substitution and φ-items, or with segment variables and segment abbreviations, this justification work is necessary in order to obtain the conviction that all these new features are doing what they are supposed to do. A lot of work in this area still has to be done.

Finally, we refer the reader to the example in Appendix A for getting an impression of where all the above-described notions may lead to.

8 Acknowledgements

First of all, I have to pay my tribute to Dick de Bruijn, who devised the mathematical language Automath (see [de Bruijn 70]), now approximately 25 years ago. His scientific work in this area inspired a lot of research, especially in typed lambda calculus. Many ideas exposed in this report are due to de Bruijn, and my own contributions have benefitted from his influence.

Secondly, I am most grateful to Bert van Benthem Jutting, who read several draft versions of this reports very thoroughly, and gave suggestions for major improvements of various kinds. Moreover, I like to express my gratitude to my colleagues Kees Hemerik and Fairouz Kamareddine for their valuable remarks and their contributions to the final result.

Finally, I thank Jan Jaap van Horssen for many valuable remarks after a careful reading of parts of this report, which he did as a student in computer science.

A An example

In order to demonstrate some of the features discussed above, especially those regarding segment abbreviations, we give a short example. We propose a system that has in principle similar power as Coquand and Huet's *Calculus of Constructions* (or λC, see [Coquand and Huet 88]).

The system in this example has the following general features:

- In order to keep close to this system λC, we do not employ explicit (step-wise) substitution as explained in this report.
- Hence, we have ordinary β-reduction and β-conversion (we do not use the generalized version of β-reduction as defined in Section 3.5).
- Moreover, we use variable names like x, y, $\ldots$, and no de Bruijn-indices.
- However, we *do* incorporate segment abbreviations, as discussed in Chapter 5.
- There is a distinction between Π's and λ's, (i.e., λ_1's and λ_2's), respectively.
- The maximal degree is three.
- Deviating from λC, we use a canonical type operator $\mathtt{typ}$, with the usual notational convention that $\mathtt{typ}^2(t) \equiv \mathtt{typ}(\mathtt{typ}(t))$, etc.
- Again deviating from λC, we have Π-application and the corresponding $\beta_{\delta\Pi}$-reduction.

Hence, we deviate in several respects from the official λC.

Note that we use three λ's, viz. λ_1, λ_2 and $\lambda_{\mathbf{sg}}$. (In the second part of this example, we write Π for λ_1 and λ for λ_2.) Moreover, we have one δ, and as a consequence of what we said in the previous paragraph, there will be *no* σ's, *no* φ's and *no* τ's. The last three operators may only be used in the meta-language.

The rules of this system have all been given and explained before. We repeat the rules below, referring to the section of this report in which they have been described.

The construction rules for terms are the following. (When we use $\mathtt{deg}$ or $\mathtt{typ}$ in a condition, we implicitly require that these operations are indeed defined for the terms under consideration.)

variable construction:

$$\frac{1 \leq \mathtt{deg}(\overline{s}x) \leq 3}{\overline{s} \vdash x} \tag{1}$$

(Cf. Sections 2.5 and 4.6.)

abstraction construction:

$$\frac{\overline{s} \vdash t \qquad \overline{s}(t\lambda) \vdash t' \qquad \mathtt{abscon}}{\overline{s} \vdash (t\lambda)t'} \tag{2}$$

where, for $\lambda \equiv \lambda_k$ and $k = 1$ or 2, respectively,

$$\mathtt{abscon} \text{ is } \begin{cases} \mathtt{typ}^i(t) =_\beta \varepsilon & \text{for } i = 1 \vee i = 2; \\ \mathtt{typ}^j(t') =_\beta \varepsilon & \text{for } j = k \vee j = k+1 \end{cases}$$

This is the same abstraction condition as phrased in Section 4.6, for λC. However, we do not use s_1 and s_2. To be precise: in λC both s_1 and s_2 can be either $*$ or $\Box$. We identify $\Box$ with ε. Moreover, we assume that $* : \Box$, as in Section 4.4, and we assume that $*$ is the *only* inhabitant of $\Box$.

Hence, the condition "$t : s_1$" can be replaced by $\mathtt{typ}(t) =_\beta \varepsilon$ (in the case that $s_1 \equiv \Box$) or $\mathtt{typ}(t) =_\beta *$, which is equivalent to $\mathtt{typ}^2(t) =_\beta \varepsilon$ (in the case that $s_1 \equiv *$).

Analogously, in the case that $\lambda \equiv \lambda_1$ (i.e., Π), the condition "$t' : s_2$" becomes $\mathtt{typ}(t') =_\beta \varepsilon$ or $\mathtt{typ}^2(t') =_\beta \varepsilon$. In the case that $\lambda \equiv \lambda_2$ (i.e., the ordinary "functional" λ), the condition "$t' : t'' : s_2$ for some t''" becomes $\mathtt{typ}^2(t') =_\beta \varepsilon$ or $\mathtt{typ}^3(t') =_\beta \varepsilon$.

The calculation of $\mathtt{typ}(t)$ should be excuted via the τrules as given in Def. 4.7 and Def. 4.8 of Section 4.2.

application construction:

$$\frac{\overline{s} \vdash t \qquad \overline{s}(t\delta) \vdash t' \qquad \mathtt{appcon}}{\overline{s} \vdash (t\delta)t'} \tag{3}$$

where

appcon is : $\mathtt{typ}^j(t') =_\beta (\mathtt{typ}(t)\ \lambda_1)t_1$ *for some* t_1 *and* $j\epsilon\{0,1\}$.

(Again, the calculation of $\mathtt{typ}(\ldots)$ is based on the τ-rules. Cf. Section 4.6 for this version of the application condition.)

It is not hard to see that both the typing relation and the reduction relations in the presented system are degree-consistent.

The example that we give is very short and is taken from logic. The logic is based on the Curry-Howard-De Bruijn isomorphism, that is the notion of "propositions-as-types". (Cf. Section 4.7.)

The example only concerns the following subjects:

– a class $*$ of propositions is taken as primitive,

– in this class the notion *falsum* (= absurdity), denoted as $\perp$, is introduced as a primitive notion,

– the axiom scheme $\frac{\perp}{a}$ (for all propositions a) is stated (i.e. "ex falso sequitur quodlibet"; when absurdity holds, then every proposition holds),

– next, the notion of implication $a \Rightarrow b$ is defined as the class of all mappings from a to b, hence sending proofs of a to proofs of b,

– and the notion of negation $\neg a$ is defined as $a \Rightarrow \perp$,

– finally, the following logical theorem is expressed and proved:

$$\frac{a \quad \neg a}{b}.$$

Remark A.1 In this example, $\perp$ is introduced as a *primitive notion* by means of an axiom. This is, of course, unnecessary in λC, since the contradiction $\perp$ can easily be *defined* in λC, viz. as $(*\Pi_a)a$.

However, for the case of the example we introduce $\perp$ as above.

In a kind of "Mathematical Vernacular", adopted from the style of the Automath-family, this piece of logico-mathematical text can be expressed as follows. For the sake of clearness, we divide the text in three parts (although this is by no means necessary):

I. the axiomatic part,

II. the definitional part,

III. the theorem-and-proof part.

I.

<u>`let`</u> $*$ <u>`be`</u> <u>`by axiom`</u> the class of all propositions.

<u>`let`</u> $\perp$ <u>`be`</u> <u>`by axiom`</u> a proposition.

let a be a proposition
and let t be a proof of $\perp$;
then $\perp$-*el* of a and t is by axiom a proof of a.

II.
let a be a proposition
and let b be a proposition;
then '$\Rightarrow$' of a and b is by definition the class of all mappings from a to b.
let a be a proposition;
then '$\neg$' of a is by definition '$\Rightarrow$' of a and $\perp$.

III.
let a be a proposition
and let b be a proposition,
let x be a proof of a
and let y be a proof of '$\neg$' of a;
then *pr* of a, b, x and y is by definition $\perp$-*el* of b and y of x,
being a proof of b.

This text will be given, in its entirety, as one term in the system described above. For convenience' sake, we write this term as a concatenation of separate items, corresponding with the different axioms, definitions and theorems in the text. Moreover, we assume that the reader who is familiar with PTS's will be pleased when we write Π instead of λ_1 and the ordinary λ instead of λ_2.

Part I gives the following three λ-items:
(λ_*)
$(*\lambda_\perp)$
$((*\Pi_a)(\perp\Pi_t)a\ \lambda_{\perp-el})$

Its intuitive contents will be clear: $*$ is introduced as a term of type ε and $\perp$ as a term of type $*$; finally, $\perp$-*el* is presented as being a primitively given, fixed function, sending a of type $*$ to an element of the set of all functions from $\perp$ to a (this set is coded as $(\perp\Pi_t)a$). Otherwise said, $\perp$-*el* is a function sending a of type $*$ and t of type $\perp$ to a. This function causes any proposition a to be inhabited as soon as $\perp$, the absurdity, is inhabited.

Part II, coding the definitions of implication and negation, can be expressed by the following four items, being two pairs of ('definitional') δ-λ-

segments:
$((*\lambda_a)(*\lambda_b)(a\Pi_x)b\ \delta)\ ((*\Pi_a)(*\Pi_b)*\ \ \lambda_{\Rightarrow})$
$((*\lambda_a)(\bot\delta)(a\delta)\Rightarrow\ \ \delta)\ ((*\Pi_a)*\ \ \lambda_{\neg})$

Here $\Rightarrow$ is defined as the product $(*\lambda_a)(*\lambda_b)(a\Pi_x)b$; this product is 'polymorphic', in the sense that it only becomes a real product after application, in this case to two arguments. To be precise, for given c and d of type $*$, the term $(d\delta)(c\delta)\Rightarrow$ β-reduces to the dependent product (in this case, the set of all functions) $(c\Pi_x)d$, functions which send inhabitants of c to inhabitants of d. The type of $\Rightarrow$ is $(*\Pi_a)(*\Pi_b)*$, the class of all functions sending pairs (a,b) of 'propositions' to a "new" 'proposition' (in this case: $a\Rightarrow b$).

Similarly, $\neg$ is defined as the 'polymorphic' negation $(*\lambda_a)(\bot\delta)(a\delta)\Rightarrow$; thus, $(c\delta)\neg$ β-reduces to $(\bot\delta)(c\delta)\Rightarrow$. The type of $\neg$ is $(*\Pi_a)*$, the class of all functions sending a 'proposition' a to a "new" 'proposition' (in this case: $\neg a$).

The reader may check that the following chain of β-reductions is correct with respect to the contextual segment given so far:
$\neg\ \ \rightarrow_\beta$
$(*\lambda_a)(\bot\delta)(a\delta)\Rightarrow\ \ \rightarrow_\beta$
$(*\lambda_a)(\bot\delta)(a\delta)(*\lambda_a)(*\lambda_b)(a\Pi_x)b\ \ \twoheadrightarrow_\beta$
$(*\lambda_a)(a\Pi_x)\bot$.

Hence,
$(a\delta)\neg\ \ =_\beta$
$(a\Pi_x)\bot$.

So $(a\delta)\neg$ (or $\neg a$ in prefix-notation) is β-convertible to $(a\Pi_x)\bot$ (or, in infix-notation, $a\Rightarrow\bot$; it is easy to check that $(a\Pi_x)\bot$, in its turn, is β-convertible to $(\bot\delta)(a\delta)\Rightarrow$).

The final part III of the text can be translated into one δ-λ-segment:
$((*\lambda_a)(*\lambda_b)(a\lambda_x)((a\delta)\neg\ \lambda_y)((x\delta)y\ \delta)(b\delta)\bot{-}el\ \delta)$
$((*\Pi_a)(*\Pi_b)(a\Pi_x)((a\delta)\neg\ \Pi_y)b\ \lambda_{pr})$

The main λ-item of this segment contains the theorem:
$(*\Pi_a)(*\Pi_b)(a\Pi_x)((a\delta)\neg\ \Pi_y)b$.

The contents of this theorem are that any inhabitant of this term, being a proof for the theorem, must be a function which, for a and b of type $*$, for x of type a and y of type $(a\delta)\neg$, gives an inhabitant of (= a proof of) the type b. Translated in more customary phrasing: the desired function must be such that for any pair of 'propositions' a and b and for any pair of 'proofs' of a and $\neg(a)$, we have a 'proof' of b.

This theorem indeed has an inhabitant (and hence has a proof). This

inhabitant can be found in the main δ-item of the δ-λ-segment:
$(*\lambda_a)(*\lambda_b)(a\lambda_x)((a\delta)\neg\ \lambda_y)((x\delta)y\ \delta)(b\delta)\bot{-}el$.

In order to show that this term is indeed a proof of the theorem, we have to show that its type is β-equivalent to the term coding the theorem. Otherwise said: we have to demonstrate that this δ-λ-segment, in particular, obeys the application condition. See below.

The obtained coding of the text is, indeed, one long term. For the sake of completeness, we give the full term:

$$
\begin{array}{l}
(\lambda_*) \\
(*\lambda_\bot) \\
((*\Pi_a)(\bot\Pi_t)a\ \lambda_{\bot-el}) \\
((*\lambda_a)(*\lambda_b)(a\Pi_x)b\ \delta)\ ((*\Pi_a)(*\Pi_b)*\ \lambda_\Rightarrow) \\
((*\lambda_a)(\bot\delta)(a\delta)\Rightarrow\ \delta)\ ((*\Pi_a)*\ \lambda_\neg) \\
((*\lambda_a)(*\lambda_b)(a\lambda_x)((a\delta)\neg\ \lambda_y)((x\delta)y\ \delta)(b\delta)\bot{-}el\ \delta) \\
((*\Pi_a)(*\Pi_b)(a\Pi_x)((a\delta)\neg\ \Pi_y)b\ \lambda_{pr})
\end{array}
\qquad (4)
$$

It is not hard to check that this terms obeys the conditions for term construction as given above:

variable condition:

The term is closed and all degrees are ≤ 3.

abstraction condition:

Left to the reader.

application condition:

Examples are:

$\mathtt{typ}(*\lambda_a)(*\lambda_b)(a\Pi_x)b \equiv (*\Pi_a)(*\Pi_b)*$,

since:

$(\tau)(*\lambda_a)(*\lambda_b)(a\Pi_x)b$
$\to_\tau$ (by Def. 4.9)
$(*\Pi_a)(\tau)(*\lambda_b)(a\Pi_x)b$
$\to_\tau$ (by Def. 4.9)
$(*\Pi_a)(*\Pi_b)(\tau)(a\Pi_x)b$
$\to_\tau$ (by Def. 4.9; $(a\Pi_x)$ reduces to the empty segment)
$(*\Pi_a)(*\Pi_b)(\tau)b$
$\to_\tau$ (by Def. 4.7)

$(*\Pi_a)(*\Pi_b)*$

and

$\mathtt{typ}(*\lambda_a)(\bot\delta)(a\delta)\Rightarrow\equiv(*\Pi_a)*$,

since:

$(\tau)(*\lambda_a)(\bot\delta)(a\delta)\Rightarrow$

$\twoheadrightarrow_\tau$ (by Def. 4.9)

$(*\Pi_a)(\tau)(\bot\delta)(a\delta)\Rightarrow$

$\twoheadrightarrow_\tau$ (by Def. 4.7;

$(\tau)\Rightarrow \;=_\tau\; (*\Pi_{a'})(*\Pi_{b'})* \;=_{\tau,\beta}\; ((\tau)a\ \Pi_{a'})(*\Pi_{b'})*$, so

$(\tau)(a\delta)\Rightarrow \;=_\tau\; (*\Pi_{b'})* \;=_{\tau,\beta}\; ((\tau)\bot\ \Pi_{b'})*$)

$(*\Pi_a)*$

.

Other checks of the application condition, like the one mentioned above:

$\mathtt{typ}\ (*\lambda_a)(*\lambda_b)(a\lambda_x)((a\delta)\neg\ \lambda_y)((x\delta)y\ \delta)(b\delta)\bot{-}el\ =_\beta$

$(*\Pi_a)(*\Pi_b)(a\Pi_x)((a\delta)\neg\ \Pi_y)b$,

are left as an exercise for the reader.

The complete term 4, as given above, is not very complicated. Yet, there are already several segment duplications. For example, the segments $(*\lambda_a)$ and $(*\lambda_a)(*\lambda_b)$ occur repeatedly; the same is the case for their respective types: $(*\Pi_a)$ and $(*\Pi_a)(*\Pi_b)$.

We already mentioned in the beginning of Section 5.1, that there is no duplication of the last kind in the usual PTS-style, since types of segments are not often represented there. This is a consequence of the use of contexts Γ in the format $\Gamma \vdash t_1 : t_2$. Such a context contains segments like $(*\lambda_a)(*\lambda_b)$. One could say that such a Γ has a double function: it incorporates at the same time a segment like $(*\lambda_a)(*\lambda_b)$ and its type $(*\Pi_a)(*\Pi_b)$.

On the other hand, different contexts Γ_1, Γ_2, ..., used in a derivation, present similar duplications as in our system. Hence, for all such systems, in PTS-style or in our style, context abbreviations are remunerative.

The last-mentioned remark does apply *a fortiori* when we have terms translating longer texts than the very short one in the example above. Segments then can easily consist of many items. Moreover, in an average term translating a piece of mathematical text, the amount of duplications is very bothersome. In Section 5.1 we explained why duplications may occur so often.

Segments tend to be repeated almost literally. As a matter of fact, it turns out to be quite natural (as a consequence of the usual structure of a mathematical reasoning) that different segments occur stackwise in the complete term; that is to say, an occurrence of a segment $(t_1\lambda_{a_1})\dots(t_n\lambda_{a_n})$ may be followed rather closely by the same segment, or by a segment which is one item longer: $(t_1\lambda_{a_1})\dots(t_{n+1}\lambda_{a_{n+1}})$, or by a segment which is one item shorter: $(t_1\lambda_{a_1})\dots(t_{n-1}\lambda_{a_{n-1}})$, and this may happen again and again. (The same holds if some of the λ's are replaced by Π's.)

De Bruijn noted this very early and he took the above into consideration when he designed the mathematical language Automath. He invented a clever abbreviation mechanism that works nicely, as is e.g. shown in [van Benthem Jutting 77].

The segment abbreviations which we proposed in Chapter 5 can solve the same problem, to a great extend. We shall apply it to the example given above. Since that example is very simple, the gains, if any, are in this case only minimal. However, in other circumstances the profits may be great.

The term given below is the same as term 4, but with segment abbreviations.

We add one more abbreviation in this translation process: when, e.g. the λ-segment $(*\lambda_a)(*\lambda_b)$ is abbreviated by $(\mathsf{b};2)$, then we abbreviate the corresponding Π-segment $(*\Pi_a)(*\Pi_b)$ by $((\tau)\mathsf{b};2)$. This is quite natural, since the τ-transition rules are such that $(\tau)(*\lambda_a)(*\lambda_b)t' \twoheadrightarrow_\tau (*\Pi_a)(*\Pi_b)t''$ (see Definition 4.8).

We assume that more comments are unnecessary.

$$
\begin{array}{l}
(\lambda_*) \\
(*\lambda_\perp) \\
((*\lambda_a)\delta)\ (\lambda_{\mathtt{sg\ a}}) \\
(((\tau)a;1)(\perp\Pi_t)a\ \lambda_{\perp-el}) \\
((\mathsf{a};1)(*\lambda_b)\delta)\ (\lambda_{\mathtt{sg\ b}}) \\
((\mathsf{b};2)(a\Pi_x)b\ \delta)\ (((\tau)\mathsf{b};2)*\ \lambda_\Rightarrow) \\
((\mathsf{a};1)(\perp\delta)(a\delta)\Rightarrow\ \delta)\ (((\tau)\mathsf{a};1)*\ \lambda_\neg) \\
((\mathsf{b};2)(a\lambda_x)((a\delta)\neg\ \lambda_y)\ \delta)\ (\lambda_{\mathtt{sg\ c}}) \\
((\mathsf{c};4)((x\delta)y\ \delta)(b\delta)\perp-el\ \delta) \\
(((\tau)\mathsf{c};4)b\ \lambda_{pr})
\end{array}
\qquad (5)
$$

In a final step, we change the lay-out of this term in such a manner that it resembles an Automath-text. At the same time, for the sake of brevity we remove those variable items of the form $((\tau)\mathbf{x};n)$ for which the corresponding variable item $(\mathbf{x};n)$ figures in the same line. Instead, we shall use a horizontal stroke: $__$, which should be considered to refer to the segment variable $(\mathbf{x};n)$, with (τ) added in the left-hand side. This is again a way to avoid unnecessary duplications; the three horizontal strokes in the version below should read: $((\tau)\mathsf{b};2)$, $((\tau)\mathsf{a};1)$ and $((\tau)\mathsf{c};4)$, respectively.

Thus doing, we come closer to both Automath and to the general PTS-framework, which uses contexts Γ.

The following version will now speak for itself.

(					$\lambda_*)$
(				$*$	$\lambda_\bot)$
(		$(*\lambda_a)$	$\delta)$	(	$\lambda_{\mathtt{sg\ a}})$
(				$((\tau)\mathsf{a};1)(\bot\Pi_t)a$	$\lambda_{\bot-el})$
(	$(\mathsf{a};1)$	$(*\lambda_b)$	$\delta)$	(	$\lambda_{\mathtt{sg\ b}})$
(	$(\mathsf{b};2)$	$(a\Pi_x)b$	$\delta)$	$(__\ *$	$\lambda_\Rightarrow)$
(	$(\mathsf{a};1)$	$(\bot\delta)(a\delta)\Rightarrow$	$\delta)$	$(__\ *$	$\lambda_\neg)$
(	$(\mathsf{b};2)$	$(a\lambda_x)((a\delta)\neg\ \lambda_y)$	$\delta)$	(	$\lambda_{\mathtt{sg\ c}})$
(	$(\mathsf{c};4)$	$((x\delta)y\ \delta)(b\delta)\bot-el$	$\delta)$	$(__\ b$	$\lambda_{pr})$

B An abstract grammar for terms

We give an alternative definition of terms in the abstract grammar format as used e.g. by Barendregt (cf. [Barendregt and Hemerik 90]). Here $\mathcal{V}$ stands for the set of variables, $\mathcal{O}$ for the set of operators, $\mathcal{T}$ for the set of terms and $\mathcal{I}$ for the set of items. We use $\mathcal{S}$ for the set of segments.

Definition B.1 *(variables, operators, terms, items, segments)*

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

Next, we give an alternative definition of the body (`body`) and the end variable (`endvar`) of a term and the body (`body`) and the end operator (`endop`) of an item:

Definition B.2 *(body, end variable, end operator)*

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

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

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

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

C An alternative step-wise substitution

We mention another possibility for combining the general step-wise substitution described in Section 3.3, and β-reduction. In this case, we use $\overline{\sigma}$-items instead of σ-items. Instead of *generating* a σ-item from a δ-λ-segment, we *replace* such a δ-λ-segment by a $\overline{\sigma}$-item, which has a *global* substitutional character.

To be precise, we replace the segment $(t_1\delta)(t_2\lambda)$ in term t by the $\overline{\sigma}$-segment $(t_1\overline{\sigma}^{(1)})$. Here the symbol $\overline{\sigma}$ marks a term (viz. t_1) that has to be substituted for the appropriate variables.

(Note that the type of these variables, t_2, is no longer present in the term t, since the original δ-λ-segment has been destructed. This type information may, however, be useful. In case one desires to keep track of these types, one should choose another solution, for example the generation of a $\overline{\delta}$-$\overline{\lambda}$-segment instead of a $\overline{\sigma}$-item; the rules below then must be adapted in an obvious manner.)

The substitution in this form is immediately suited for the well-known version of β-reduction, in which also an abstraction-application pair in the redex (we would say: a δ-λ-segment) is destructed.

When choosing this type of step-wise substitution, one should take care of a number of things. Firstly, the bond that the end λ of the δ-λ-segment has with some bound variables, should be adopted by the $\overline{\sigma}$-item. Secondly, the other existing bonds between binding λ's and variables should not be disturbed in the transition of the new $\overline{\sigma}$-item through the branches of the tree. All this can be assured by the procedure described in the following two definitions.

Definition C.1 *($\overline{\sigma}$-reduction)*
(general $\overline{\sigma}$-generation rule:)
$(t_1\delta)(t_2\lambda) \rightarrow_{\overline{\sigma}} (\varphi^{(-1)})((\varphi)t_1\overline{\sigma}^{(1)})$
(general $\overline{\sigma}$-transition rules:)
$(t_1\overline{\sigma}^{(i)})(t_2\lambda) \rightarrow_{\overline{\sigma}} ((t_1\overline{\sigma}^{(i)})t_2\lambda)((\varphi)t_1\overline{\sigma}^{(i+1)})$
$(t_1\overline{\sigma}^{(i)})(t_2\delta) \rightarrow_{\overline{\sigma}} ((t_1\overline{\sigma}^{(i)})t_2\delta)(t_1\overline{\sigma}^{(i)})$
($\overline{\sigma}$-destruction rules:)
$(t_1\overline{\sigma}^{(i)})i \rightarrow_{\overline{\sigma}} t_1$
$(t_1\overline{\sigma}^{(i)})x \rightarrow_{\overline{\sigma}} x$ *if* $x \neq i$.

Note that these rules are adequate for global, maximal β-reduction of terms which originally are in $\Omega_{\lambda\delta}$. Moreover, $\overline{\sigma}$'s do not count as λ's in

the search for a binding λ of a variable occurrence, but note that σ's have binding "power" themselves.

The final problem that we have to consider is, how the bindings between λ's and variables change.

Definition C.2 *(the binding λ for a bound variable)*

In a term t with $\overline{\sigma}$-items and φ-items, the binding λ for a bound variable (occurrence) x can be found as follows.

First find $t' \equiv t \restriction x$.

Now x is bound by the end operator $\overline{\sigma}^{(i)}$ of a main $\overline{\sigma}$-item s' of t', if there are precisely $x - i$ λ-items between s' and x (intermediate $\overline{\sigma}$-items are not *counted; main $\varphi^{(k,i)}$-items count for $-i$ λ-items, but only in case the "counter" has a value greater than k at the moment that it "passes" this $\varphi^{(k,i)}$-item during its travel from right to left).*

When there is no $\overline{\sigma}^{(i)}$ binding x, then x can also be bound by the end λ of a main λ-item s of t', namely if there are precisely $x - 1$ λ-items between s and x (again, $\overline{\sigma}$-items are not counted and $\varphi^{(i)}$-items count for $-i$ λ-items).

If neither is the case, then x is free.

In the original term t, the variable x is bound by the λ or $\overline{\sigma}$ corresponding with the one in $t \restriction x$, or x is free in t if it is so in $t \restriction x$.

The described search for a λ binding an occurrence x° in a term t can also be described by means of an inverse search item, as in Section 2.4:

Definition C.3 *(alternative $\zeta_\star$-reduction)*

($\zeta_\star$-generation rule:)

$x^\circ \to_{\zeta_\star} (\zeta_\star^{(x)})x^\circ$

($\zeta_\star$-transition rules:)

$(t'\lambda)(\zeta_\star^{(j)}) \to_{\zeta_\star} (\zeta_\star^{(j-1)})(t'\lambda)$ *if* $j > 1$

$((\zeta_\star^{(j)})t'\lambda) \to_{\zeta_\star} (\zeta_\star^{(j)})(t'\lambda)$

$(t'\overline{\sigma}^{(i)})(\zeta_\star^{(j)}) \to_{\zeta_\star} (\zeta_\star^{(j)})(t'\overline{\sigma}^{(i)})$ *if* $j \neq i$

$((\zeta_\star^{(j)})t'\overline{\sigma}^{(i)}) \to_{\zeta_\star} (\zeta_\star^{(j)})(t'\overline{\sigma}^{(i)})$

$(t'\delta)(\zeta_\star^{(j)}) \to_{\zeta_\star} (\zeta_\star^{(j)})(t'\delta)$

$((\zeta_\star^{(j)})t'\delta) \to_{\zeta_\star} (\zeta_\star^{(j)})(t'\delta)$

$(\varphi^{(k,i)})(\zeta_\star^{(j)}) \to_{\zeta_\star} (\zeta_\star^{(j+k)})(\varphi^{(k,i)})$ *if* $j > k$,

$(\varphi^{(k,i)})(\zeta_\star^{(j)}) \to_{\zeta_\star} (\zeta_\star^{(j)})(\varphi^{(k,i)})$ *if* $j \leq k$,

($\zeta_\star$-destruction rules:)

$$(t'\lambda)(\zeta_\star^{(1)}) \rightarrow_{\zeta_\star} (t'\hat{\lambda})$$
$$(t'\overline{\sigma}^{(i)})(\zeta_\star^{(j)}) \rightarrow_{\zeta_\star} (t'\hat{\overline{\sigma}}^{(i)}) \text{ } \mathit{if}\ j = i.$$

In a term t with $\overline{\sigma}$-items originating from $\overline{\sigma}$-generations, every bound variable has precisely one binding place, which can be either a λ or a $\overline{\sigma}$.

Remark C.4 With this alternative step-wise substitution, things run not so smoothly as with the step-wise substitution as described in the body of this report. The reason is, that the φ-items with negative index play an unpleasant role.

As was shown in [van Horssen 92], there are two kinds of undesired effects. We give an example of both.

(1) Firstly, the transition of a $\varphi^{(-1)}$-item over a σ-item can give rise to ambiguity in binding. Consider the following φ-reduction:

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

The variable 2 in the last line but one has become (the second) 1 in the last line. The binder of this 1 should be the $\sigma^{(0)}$ in the σ-item. However, the binding has apparently been transferred to the λ, which is clearly undesired. The reason for this mismatch is the index 0 of the σ, caused by the transition of the φ-item. It will be clear that a variable 1 directly after a $\sigma^{(0)}$-λ-combination is always related to the λ, even if the $\sigma^{(0)}$ is meant.

Hence, it is undesirable to allow the transition of $(\varphi^{(-1)})$ over a $\sigma^{(1)}$-item. And this is not the only case where the transition of a φ-item over a σ-item causes trouble.

(2) There is a second disadvantage in this alternative step-wise substitution, namely that α-conversion returns "in disguise". This is caused by the fact that a generated $\sigma^{(i)}$-item takes over the bindings originally provided for by the λ of a λ-δ-segment. Consider the following two terms:

$$t_1 \equiv (t'\sigma^{(5)})5,$$

$$t_2 \equiv (t'\sigma^{(8)})8.$$

It will be clear that t_1 and t_2 are in a sense equivalent, both reducing in one step to t'. This equivalence is so strong that one is tempted to identify these terms. This resembles the strong desire to identify e.g. $\lambda_x.x$ and $\lambda_y.y$, which is actually the reason for the definition of *alpha*-conversion. It will be clear that this "return of alpha-conversion" is against the basic principles of this report, especially in combination with de Bruijn-indices.

These annoying matters concerning the alternative step-wise substitution still have to be investigated in greater detail. This is subject for further research.

A possible solution is to omit the $\varphi^{(-1)}$-items, and to give the σ-items the *"update power"* of the $\varphi^{(-1)}$-item. That is to say, the general σ-generation rule then becomes:

$$(t_1\delta)(t_2\lambda) \to_{\overline{\sigma}} (t_1\overline{\sigma}^{(1)})$$

and the $\zeta_\star$-transition rule for $\overline{\sigma}$-items changes into the following pair of rules:

$$(t'\overline{\sigma}^{(i)})(\zeta_\star^{(j)}) \to_{\zeta_\star} (\zeta_\star^{(j-1)})(t'\overline{\sigma}^{(i)}) \text{ if } j > i,$$

$$(t'\overline{\sigma}^{(i)})(\zeta_\star^{(j)}) \to_{\zeta_\star} (\zeta_\star^{(j)})(t'\overline{\sigma}^{(i)}) \text{ if } j < i.$$

The other $\zeta_\star$-transition rules and the $\zeta_\star$-destruction rule remain unchanged.

Note, however, that the sketched difficulties do only apply to the *alternative* substitution presented in this Appendix. Items of the form $(\varphi^{(-1)})$ were also present in the ordinary step-wise substitution, namely in the void β-reduction (see Section 3.4). However, there they were harmless, and exactly because of this "voidness": such a φ-item with negative exponent was only generated in case a δ-λ-segment was destructed *were the λ did not bind any variable.* That turns out to be crucial for the "decent behaviour" of these φ-items with exponent -1.

D A 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 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 $\{1/1, 2/2, \ldots\}$ — the paper [Abadi et al. 90] introduces three other index manipulations:

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

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

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 report. We have no direct means to render infinite substitutions, but we introduce *parallel σ-items* for this purpose. (As a matter of fact, we use the $\overline{\sigma}$-items of Appendix C; but for convenience' sake, we drop the "overline".)

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) 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. When we do not see our system as a *semantics* for the system of [Abadi et al. 90], but as a syntactic system in its own right, we may note the following: the equations that the authors of [Abadi et al. 90] give as an axiomatic basis for their equational theory, can all be *derived* in our approach.

We claim that the introduction of parallel σ-items is only apparently an extension of the system that we discussed in the present report:

— 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 examples of the rules in [Abadi et al. 90] and their justification in our setting.

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

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

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

The traditional rule of β-reduction has the following form in our system (see the $\overline{\sigma}$-generation rule in Appendix C):
$(t_1\delta)(t_2\lambda) \to_\sigma (\varphi^{(-1)})((\varphi)t_1\sigma^{(1)})$.
It is not very hard to check that

$(\varphi^{(-1)})((\varphi)t_1\sigma^{(1)}) = (t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)})$:
$(\varphi^{(-1)})((\varphi)t_1\sigma^{(1)}) =$
$(\ i-1\ \sigma^{(\bar{i})})((\ j+1\ \sigma^{(\bar{j})})t_1\sigma^{(1)}) =$
(compare the effect of this substitution on index 1 with the efffect on indices > 1)
$(\ i-1\ \sigma^{(\bar{i})})((\ j+1\ \sigma^{(\bar{j})})t_1\sigma^{(1)}) \oplus (\ i-1\ \sigma^{(\bar{i}>1)}) =$
(since, as noted above: $(t_i\sigma^{(\bar{i})})(t'_j\sigma^{(\bar{j})}) = ((t_i\sigma^{(\bar{i})})t'_j\sigma^{(\bar{j})})$)
$((\ i-1\ \sigma^{(\bar{i})})(\ j+1\ \sigma^{(\bar{j})})t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)}) =$
(by additivity, which holds in this case)
$((j\sigma^{(\bar{j})})t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)}) =$
$(t_1\sigma^{(1)}) \oplus (\varphi^{(1,-1)})$.

This enables us directly to derive the translation of the *Beta*-rule as given in [Abadi et al. 90]:
$(\lambda a)b = a[b \cdot id]$:
$[\![(\lambda a)b]\!] = (b\delta)(\lambda)a \rightarrow_\sigma (\varphi^{(-1)})((\varphi)b\sigma^{(1)})a$;
$[\![a[b \cdot id]]\!] = ((b\sigma^{(1)}) \oplus (\ i-1\ \sigma^{(\bar{i}>1)}))a =$
$((b\sigma^{(1)}) \oplus (\varphi^{(1,-1)}))a$.

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

Then the following correspondences hold:

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

In particular, $(t\sigma^{(1)}) = t \cdot \uparrow$.

Also, $(\varphi^{(n)}) = (\varphi^{(0,n)}) = (\uparrow)^n$, $(\varphi) = (\varphi^{(1)}) = \uparrow$ and $(\varphi^{(-1)}) = (\uparrow)^{-1} = 1 \cdot id$.

When we define $k!$ to be $1 \cdot 2 \cdot \ldots \cdot k$, then the above rules can be simplified to

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

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

Finally, we give the correspondence between our system and the $\Uparrow$-operator of [Hardin and Lévy 90]:

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

It is worth while to compare the latter term with the general $\overline{\sigma}$-rules of Definition C.1.

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.

[Balsters 86] Balsters, H., *Lambda calculus extended with segments*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, 1986.

[Barendregt 84] Barendregt, H.P., *The Lambda Calculus. Its Syntax and Semantics*, North Holland, Revised edition, 1984.

[Barendregt 9x] Barendregt, H.P., Lambda calculi with types, in: *Handbook of Logic in Computer Science*, Eds. S. Abramsky, D. Gabbay and T. Maibaum, Oxford University Press, Oxford, 199x. To appear.

[Barendregt and Hemerik 90] Barendregt, H. and Hemerik, K., Types in lambda calculi and programming languages, in *European Symposium on Programming, Copenhagen*, Ed. N. Jones, LNCS, 432, Springer, Berlin, 1990, pp. 1-36.

[van Benthem Jutting 77] Benthem Jutting, L.S. van, *Checking Landau's "Grundlagen" in the AUTOMATH system*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, 1977.

[van Benthem Jutting 88] Benthem Jutting, L.S. van, An implementation of substitution in a λ-calculus with dependent types. Philips Research Laboratories Eindhoven / Eindhoven University of Technology, 1988.

[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 74] Bruijn, N.G. de, Some extensions of AUTOMATH: the AUT-4 family, Dept. of Mathematics, Eindhoven University of Technology, 1974.

[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.

[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.

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

[Coquand and Huet 88] Coquand, T. and Huet, G., The calculus of constructions, *Information and Control* 76, 1988, pp. 95-120.

[de Groote 91] Groote, Ph. de, *Définition et propriétés d'un métacalcul de représentation de théories*, Ph.D. thesis, Université Catholique de Louvain, Louvain-la-Neuve, 1991.

[Hardin and Lévy 90] Hardin, Th. and Lévy, J.-J., A confluent calculus of substitutions, Lecture notes of the INRIA-ICOT symposium, Izu, Japan, November 1989.

[van Horssen 92] Horssen, J.J. van, *Explicit substitution in two versions of typed lambda calculus*, Master's thesis, Eindhoven University of Technology, 1992.

[Howard 80] Howard, W.A., The formulae-as-types notion of constructions, 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.

[Nederpelt 71] Nederpelt, R.P., *Lambda-Automath*, Internal Report 17, Eindhoven University of Technology, Dept. of Math., 1971

[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 80] Nederpelt, R.P., An approach to theorem proving on the basis of a typed lambda-calculus, in *5th Conference on Automated Deduction*, Les Arcs, France, 1980, Eds. W. Bibel and R. Kowalski, LCNS, 87, Springer, Berlin, 1980, pp.182-194.

[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.

[Troelstra and van Dalen 88] Troelstra, A.S. and van Dalen, D., *Constructivism in Mathematics, An Introduction*, Vol. 1, North-Holland, Amsterdam etc., 1988.

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

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.