A unified approach to type theory through a refined lambda-calculus

Eindhoven University of Technology

Department of Mathematics and Computing Science

A unified approach to Type Theory through a refined λ-calculus

by

Rob Nederpelt Fairouz Kamareddine

92/18

Computing Science Note 92/18
Eindhoven, September 1992

COMPUTING SCIENCE NOTES

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

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

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

A unified approach to Type Theory through a refined λ-calculus

Rob Nederpelt
and
Fairouz Kamareddine

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

August 21, 1992

Abstract

λ-calculus is at the heart of type theory while type theory has been the most stimulating part of the theory of Computation. The presence however of various type systems provokes attempts at finding a unified way of describing these systems. Barendregt's cube for example, is such an attempt. Based on these observations, we will devise a new λ-notation which enables categorising most of the known systems in a unified way. More precisely, we will sketch the general structure of a system of typed lambda calculus and show that this system has enough expressive power for the description of various existing systems, ranging from Automath-like systems to singly-typed Pure Type Systems.

Keywords: *Lambda Calculus, Pure Type Systems, Barendregt's Cube, Automath and the Calculus of Constructions.*

1 Introduction

Terms of the lambda calculus are constructed by two principles: *abstraction*, by means of which free variables are bound, thus generating some sort of functions; and *application*, being in a sense the opposite operation, formalising the application of a function to an argument. We will introduce a slight change to the λ-notation to enable us to construct lambda terms in a modular way, in accordance with the demands and needs of a mathematical entourage. This new notation will be based on abstraction and application and, as an alternative to the use of variables, will assume de Bruijn-indices. These are natural numbers that do not suffer from the usual problems with variable names (the danger of "clash of variables", the need for appropriate renaming, etc.).

Our notation is very advantageous and should be seen as an alternative to the usual λ-calculus notation. We claim that this new formulation can avoid many of the complications associated with the old formulation. In fact, in [NeK92a], we showed the usefulness of the new notation for variable and term manipulation and for typing. In particular, we showed in that paper, that the restriction of a term to a variable x is obtained by simply taking the substring of string t from the beginning of t until x and then deleting all unmatched opening parentheses. Moreover, we showed in the same paper that accounting for bound and free variables in a term is only a matter of a very simple calculation and demonstrated that term construction can be done via trees which are at the same time proofs of the well-typedness of the term. In [NeK92b], we embedded stepwise substitution in the new calculus showing how the new notation facilitates the introduction of substitution as an object level notation in the λ-calculus resulting in a system which can accommodate most substitution strategies. All this points towards the advantages of the new notation but this is not all. In this paper, we will show how various existing systems ranging from Authomath-like systems to singly-typed pure type systems could be expressed in a uniform way in our proposed setting.

In particular, after introducing in Sections 2 and 3 the new notation and all the formal machinery needed for the paper, we concentrate in Section 4 on the typing relation. 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 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. In fact, not only the type of a λ- or a Π-abstraction is found but also Π-application (and not only λ-application) is allowed.

In Section 5, 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.

In Section 6, we describe a number of Automath-systems in our setting. One of these possibilities is a de Bruijn's system $\Delta\Lambda$, which is a version of Automath in the format of typed lambda calculus.

Finally, in Section 7, we demonstrate the features of typing and term construction, through a short example. This example is a system that we propose and that has in principle similar power to that of Coquand and Huet's *Calculus of Constructions* (or λC, see [CoH88]). We work out the proof of a theorem taken from logic in our system.

2 The new notation

We assume the reader familiar with De Bruijn's indices and of why they were introduced. If not, the reader is referred to [deB72], and we hope that the following examples give him an idea of what these indices are.

Example 2.1 Terms such as $\lambda_x.x$ and $\lambda_y.y$ are the "same", and the use of x, y or any other variable does not change the semantic meaning of the function denoted by this term (the identity function). The identity function using de Bruijn's indices will be denoted by $\lambda.1$. The bond between the bound variable x and the operator λ is expressed by the number 1; the position of this number in the term is that of the bound variable x, and the value of the number ("*one*") tells us how many lambda's we have to count, going leftwards in the term, starting from the mentioned position, to find the binding place (in this case: the *first* λ to the left is the binding place).

Example 2.2 The identity function above could have been identity over a particular type y (let us say) written as $\lambda_{x:y}.x$. In such a case y is a free variable and the function is denoted by: $(\lambda 1.1)$. The free variable y in the typed lambda term is translated into the first number 1. Such a number refers in this case to an "invisible" lambda that is not present in the term, but may be thought of to *preceed* the term, binding the free variable. Note here that if we had more than one free variable, we have to know which one comes before the other. For this, we assume an arbitrary, but fixed order so that these invisible lambda's form a **free variable list**. The number 1 next to the λ tells us how many λs we have to count from (and excluding[1]) this λ. (The variable x, as before, is translated in the second number 1.)

Example 2.3 To demonstrate how β-reduction works with de Bruijn's indices, we consider the term $(\lambda_{x:z}.(xy))u$ which β-reduces to uy. Under the assumption that the free variable list is $\lambda_y, \lambda_z, \lambda_u$, this reduction using de Bruijn's indices can be represented as: $(\lambda 2.1\,4)1$ reduces to $1\,3$. Here the contents of the subterm $1\,4$ changes: 4 becomes 3. This is due to the fact that $\lambda 2$ disappeared (together with the argument 1). The first variable 1 did not change;

[1] This technical peculiarity disappears in the new notation.

note, however, that the λ binding this variable has changed "after" the reduction; it is the last λ in the free variable list ("λ_u") and no longer the λ inside the original term ("λ_x"). The reference changed, but the number stayed (by chance) the same.

Now take the type free λ-calculus, with the following three ways of forming terms:
$t ::= x \mid (\lambda_x.t) \mid (t_1 t_2)$.
If we forget variables (as we shall when we use de Bruijn's indices), then we begin with natural numbers and all that remains is *abstraction* and *application*. We shall consider these to be the basic *operations* on terms and shall use λ to refer to the first and δ to refer to the second. Note that when we work with the typed λ-calculus, these two operators can be considered to be binary. In fact, λ links a type to a term, (think of $\lambda_{x:y}.x$ which is $\lambda 1.1$) and δ links a function to an argument. As we are trying to give a general notation which can be used to describe the other ones, we will use a typed λ-calculus notation which is also suitable to write type free terms. This will be done via our special index ε below.

Notation 2.4 *(Abstraction and Application operators)*
As we are trying to devise a system which will be general enough to represent a whole variety of type systems, we shall not assume the uniqueness of the λ and the δ operators. Rather we consider $\lambda, \lambda_1, \lambda_2, \ldots$ for abstraction, and $\delta, \delta_1, \delta_2, \ldots$ for application and use $\omega, \omega_1, \omega_2, \ldots$ as meta-variables for both kinds of operators. Moreover, we refer to the set of λ-**operators** by Ω_λ and to the set of δ-**operators** by Ω_δ. We assume that Ω_λ and Ω_δ are disjoint and finite and write Ω (or $\Omega_{\lambda\delta}$) for their union.

Example 2.5 To accommodate second-order theories, we use λ_2 for λ and λ_1 for Λ. To accommodate Pure Type Systems we use λ_1 for Π and λ_2 for the ordinary λ.

Notation 2.6 *(Variables)*
As we decided to use indices instead of variables, we take Ξ the set of **variables** to be $\Xi = \{\varepsilon, 1, 2, \ldots\}$. Sometimes we will need to use actual variables, but this is not a part of our syntax. It is only a matter of simplifying the conversation. We use $x, x_1, y, \ldots$ to denote variables. ε is a special variable that denotes the "empty term". It can be used for rendering ordinary (untyped) lambda calculus, by taking all types to be ε. Another use is as a "final type", like $\Box$ in Barendregt's cube.

Using Ω and Ξ we define our terms (which we denote $t, t_1, \ldots$) to be those symbol strings obtained in the usual manner on the basis of Ξ, the operators in Ω and parentheses. That is:

Definition 2.7 *(Terms)*
Terms are the elements of $F_\Omega(\Xi)$, the free Ω-structure generated by Ξ. We call these terms $\Omega_{\lambda\delta}$-**terms** *or simply* **terms**.

Notation 2.8 *(Item Notation)*
We will defer from usual practice and use the operators in Ω as *infix* ones. That is we write $(t\delta t')$ for the *function* t' applied to the *argument* t (note the *reversed* order!) and write $(t\lambda t')$ for $(\lambda t.t')$. We go even further by using what we call **item**-notation where we place parentheses in an unorthodox manner: we write $(t_1\omega)t_2$ instead of $(t_1\omega t_2)$.

Example 2.9 The following are terms: ε, 3, $(2\delta)(\varepsilon\lambda)1$, in item notation or $(2\delta(\varepsilon\lambda 1))$ in the original infix notation. (We assume that $\lambda \in \Omega_\lambda$ and $\delta \in \Omega_\delta$.)

Notation 2.10 *(Tree notation)*
One can also consider terms as trees, in the usual manner (in this case we shall speak of **term trees**). In term trees, parentheses are superfluous (see figure 1). In this figure, we deviate from the normal way to depict a tree; for example: we position the root of the tree in the lower left hand corner. We have chosen this manner of depicting a tree in order to maintain a close resemblance with the linear term. This has also advantages in the sections to come. The item-notation suggests a partitioning of the term tree in vertical layers. For $(x\omega_1)(y\omega_2)z$, these layers are: the parts of the tree corresponding with $(x\omega_1)$, $(y\omega_2)$ and z (connected in the tree with two edges). For $((x\omega_2)y\omega_1)z$ these layers are: the part of the tree corresponding with $((x\omega_2)y\omega_1)$ and the one corresponding with z.

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

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

Notation 2.11 *(Name carrying terms)*
For ease of reading, we occasionally use customary variable names like x, y, z and u instead of reference numbers. Thus creating name-carrying terms in item-notation, such as $(u\delta)(y\lambda_x)x$ in Example 2.12. The symbols used as subscripts for λ in this notation are only necessary for establishing the place of reference; they do not "occur" as variables in the term.

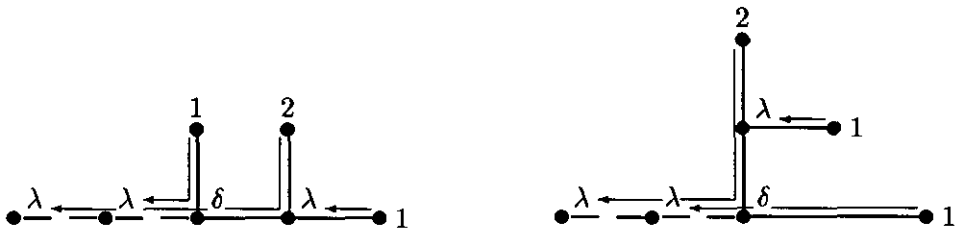
Example 2.12 Let the free variable list, in the name-carrying version, be λ_y, λ_u.

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

2. The typed lambda term $u(\lambda_{x:y}.x)$ is denoted as $((y\lambda_x)x\delta)u$ in our name-carrying item-notation and as $((2\lambda)1\delta)1$ in item-notation with de Bruijn-indices.

The term trees of these lambda terms are given in figure 2. In each of the two pictures, the references of the three variables in the term have been indicated: thin lines, ending in arrows, point at the λ's binding the variables in question. Note that these lines follow the path which leads from the variable to the root following *the upper-left side* of the branches of the tree. Only the λ's met do count, the δ's do not.

Example 2.13 Now for β-reduction, the term $(\lambda_{x:z}.(xy))u$ β-reduces to uy. In our sugared item-notation this becomes: $(u\delta)(z\lambda_x)(y\delta)x$ reduces to $(y\delta)u$ (see figure 3). Note that the



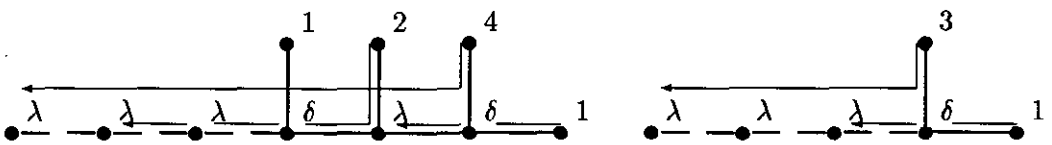
$(1\delta)(2\lambda)1$ $\quad$ $((2\lambda)1\ \delta)1$

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

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

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

presence of a so-called δ-λ-segment (i.e. a δ-item immediately followed by a λ-item, in this example: $(u\delta)(z\lambda_x)$) is the signal for a possible β-reduction. The "unsugared" version reads: the term $(1\delta)(2\lambda)(4\delta)1$ reduces to $(3\delta)1$.



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

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

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

Figure 3: β-reduction in our notation

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

Remark 2.14 With the help of ε we can construct terms without free variables, for example we can construct $(\varepsilon\lambda)(1\lambda)(1\delta)((2\lambda)(1\lambda)1\lambda)3$. We note that it may be profitable to use the empty term instead of ε, which allows us to write terms like $(\lambda)(1\lambda)2$ or even $(\lambda)(1\lambda)$, representing the typed lambda terms $\lambda_{y:\varepsilon}.\lambda_{x:y}.y$ and $\lambda_{y:\varepsilon}.\lambda_{x:y}.\varepsilon$, respectively. We shall use this convention in the case of an item $(\varepsilon\omega)$, which we render as (ω), for different operators ω.

3 The formal machinery

In this section, we will introduce most of the machinery needed for the paper. We start by the two basic concepts *item* and *segment.*

Definition 3.1 *(items, segments)*

1. *If ω is an operator and t a term, then $(t\omega)$ is an* **item.**
2. *A concatenation of zero or more items is a* **segment.**

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

Definition 3.2 *(main items, main segments, ω-items, ω_1-...-ω_n-segments, (non)empty segments, contexts)*

1. *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.*
2. *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 $\overline{s}$.*
3. *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 $\overline{s}$).*
4. *An item $(t\ \omega)$ is called an* **ω-item.** *Hence, we may speak about* **λ-items** *and* **δ-items.**
5. *If a segment consists of a concatenation of an ω_1-item up to an ω_n-item, $\omega_i \in \Omega$, this segment may be referred to as being an* **ω_1-...-ω_n-segment.** *(An important case is that of a* **δ-λ-segment,** *being a δ-item immediately followed by a λ-item.)*
6. *A segment $\overline{s}$ such that $\overline{s} \equiv \emptyset$ is called an* **empty segment;** *other segments are* **non-empty.**
7. *A* **context** *is a segment consisting of only λ-items.*

Example 3.3 Let the term t be defined as $(\varepsilon\lambda)((1\delta)(\varepsilon\lambda)1\delta)(2\lambda)1$ and let the segment $\overline{s}$ be $(\varepsilon\lambda)((1\delta)(\varepsilon\lambda)1\delta)(2\lambda)$. Then the main items of both t and $\overline{s}$ are $(\varepsilon\lambda)$, $((1\delta)(\varepsilon\lambda)1\delta)$ and (2λ), being a λ-item, a δ-item, and another λ-item. Moreover, $((1\delta)(\varepsilon\lambda)1\delta)(2\lambda)$ is an example of a main segment of both t and $\overline{s}$, which is not a context, but a δ-λ-segment. Also, $\overline{s}$ is a λ-δ-λ-segment, which is a main segment of t.

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.

Definition 3.4 *(Segment abbreviation)*
Segment $\overline{s}$ can be called "a" by adding the "definitional segment" $(\overline{s}\delta)(\lambda_a)$.

Example 3.5 In this example we use two λ's which we denote Π and λ respectively. Now the following introduces $*$ as a term of type ε, $\perp$ as a term of type $*$ and defines $\Rightarrow$ as the product $(*\lambda_a)(*\lambda_b)(a\Pi_x)b$. This states that, given c and d of type $*$, the term $(d\delta)(c\delta)\Rightarrow$ β-reduces to the dependent product which sends 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 type $*$ to a "new" element of type $*$.

1. (λ_*)
2. $(*\lambda_\perp)$
3. $((*\lambda_a)(*\lambda_b)(a\Pi_x)b\,\delta)\ ((*\Pi_a)(*\Pi_b)*\ \ \lambda_\Rightarrow)$

Remark 3.6 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!).

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

1. *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 $t \equiv \mathtt{body}(t)\ \mathtt{endvar}(t)$.*
2. *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 meanings: the body of a term is a segment, and the body of an item is a term.

Example 3.8 In Example 3.3, $\overline{s}$ 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 3.9 *(sieveseg)*
Let $\overline{s}$ be a segment, or let t be a term with body $\overline{s}$, then $\mathtt{sieveseg}_\omega(\overline{s}) = \mathtt{sieveseg}_\omega(t) =$ *the segment consisting of all main ω-items of $\overline{s}$, concatenated in the same order in which they appear in $\overline{s}$.*

Example 3.10 In the term t of Example 3.3, $\mathtt{sieveseg}_\lambda(t) \equiv (\varepsilon\lambda)(2\lambda)$ and $\mathtt{sieveseg}_\delta(t) \equiv ((1\delta)(\varepsilon\lambda)1\delta)$.

Definition 3.11 *(ω-weight)*

1. *The* **weight** *of a segment $\overline{s}$,* $\mathtt{weight}(\overline{s})$*, is the number of main items that compose the segment.*
2. *The weight of a term t is the weight of* $\mathtt{body}(t)$*.*
3. *The ω-weight* $\mathtt{weight}_\omega(\overline{s})$ *of a segment $\overline{s}$ is the weight of* $\mathtt{sieveseg}_\omega(\overline{s})$*.*

4. *The ω-weight of a term t is the ω-weight of* $\mathtt{body}(t)$.

Example 3.12 For the term $t \equiv (\varepsilon\lambda_x)(x\lambda_v)(x\delta)(\varepsilon\lambda_y)((x\lambda_z)y\delta)(y\lambda_u)u$ and the segment $\overline{s} \equiv (\varepsilon\lambda_x)(x\lambda_v)(x\delta)(\varepsilon\lambda_y)((x\lambda_z)y\delta)(y\lambda_u)$, $\mathtt{weight}(t) = \mathtt{weight}(\overline{s}) = 6$ and $\mathtt{weight}_\lambda(t) = \mathtt{weight}_\lambda(\overline{s}) = 4$.

Definition 3.13 *(direct subterms, subterms)*

1. *If* $\mathtt{body}(t) \neq \emptyset$, *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* $t'' \subset t$.

2. *The relation* $\Subset$ *is the reflexive and transitive closure of* $\subset$. *We say that* t_1 *is a* **subterm** *of* t *iff* $t_1 \Subset t$.

Example 3.14 Let t be the term $((1\delta)2\lambda)(1\lambda)3$. The left direct subterm of t is $(1\delta)2$, the right direct subterm of t is $(1\lambda)3$. The subterms of t are $t, (1\delta)2, (1\lambda)3, 1$ (twice), 2 and 3.

Notation 3.15 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.) 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 $t \equiv ((x\delta)(y\lambda_x)x\lambda_u)(z\delta)y$ 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 $\underline{(y\lambda_x)x}$. We can also mark the occurrence of an operator: $(y\lambda_x^\circ)x$.

Definition 3.16 *(arguments)*
Let $(t'\omega^\circ)t'' \Subset t$. *Then* t' *is the* **left argument** *of* ω° *in* t, *or* $\mathtt{leftarg}(\omega^\circ)$, *and* t'' *is the* **right argument** *of* ω° *in* t, *or* $\mathtt{rightarg}(\omega^\circ)$.

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

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

Definition 3.17 *(degree of a variable)*

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

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

3. *Assume that (the occurrence of) x is bound*[2] *in t and let t' be the type of x. Further, let y be the end variable of this type t' and assume that* $\mathtt{deg}(y)$ *is defined. Then* $\mathtt{deg}(x) = \mathtt{deg}(y) + 1$.

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 3.18 *(degree of a term)*

[2]The notions "bound" (for a variable) and "type" (of a term) are formally defined in Definition 3.26.

1. *The* **degree of a term** *is the degree of its end variable, if this degree is defined; otherwise it is undefined.*

2. *The* **maximal degree** *of a term is the maximal number (if any) 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.*

Example 3.19 Take the $\Omega_{\lambda\delta}$-term t: $(\varepsilon\lambda_x)((x\lambda_u)((u\delta)(x\lambda_t)x\lambda_y)(u\lambda_z)y\lambda_v)u$. The degrees for the variables occurring in this term are: $\mathbf{deg}(\varepsilon) = 0$; $\mathbf{deg}(x) = 1$; $\mathbf{deg}(u) = 2$, except for the free u which is the end variable of the term: this u has no degree; $\mathbf{deg}(y) = 2$; $\mathbf{deg}(z) = 3$. If t occurred, 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.

Remark 3.20 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". See Example 3.21 which has also for aim to show the usefulness of more degrees.

Example 3.21 In the propositions-as-types conception (see e.g. [How80]), 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.

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 many systems, sets and propositions 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 some "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 the type of the type of $t =_\beta *_p$.

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 [deB74]). "Irrelevance of proofs" can now be implemented by a rule of the following form, where $=_i$ is some equivalence:

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

Definition 3.22 *(degree-consistency)*

1. *A* typing relation *is* **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 $\mathsf{deg}(t_1) = \mathsf{deg}(t_2) + 1$.*

2. *A* reduction relation $\rightarrow_\rho$ *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 $\mathsf{deg}(t_1) = \mathsf{deg}(t_2)$.*[3]

Example 3.23 All Automath-systems have the property of degree-consistency, both for the typing relation and for β-reduction (see Section 6). The same observation holds for the systems in Barendregt's cube, but *not* for general PTS's (see Section 5).

Definition 3.24 *(term restriction)*
If t is a term, and $\underline{t'} \subset t$ (t' is underlined in order to identify a unique occurrence of t' in t), then $t \restriction t'$ (pronounced the restriction of t to t') is defined inductively as follows:

$$\underline{t} \restriction \underline{t} \equiv t$$

$$(t_1 \omega) t_2 \restriction \underline{t} \equiv \begin{cases} t_1 \restriction \underline{t} & \text{if } \underline{t} \subset t_1 \\ (t_1 \omega)(t_2 \restriction \underline{t}) & \text{if } \underline{t} \subset t_2 \end{cases}$$

Example 3.25 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 restriction $t \restriction x$ of t to x° is:

$$(\varepsilon\lambda_x)(x\lambda_u)(u\delta)(x\lambda_t)x^\circ. \tag{2}$$

Moreover, the restriction $t \restriction (x\lambda_t)x^\circ \equiv t \restriction x^\circ$.

Definition 3.26 *(Bound and free variables, type, open and closed terms)*

[3] A typing relation which is degree-consistent is called *ok* in [Bar84].

1. *Let x° be a variable occurrence in t such that $x \not\equiv \varepsilon$ and assume that $\mathtt{sieveseg}_\lambda(t\restriction x^\circ) \equiv s_m \ldots s_1$ (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 $x > m$.*

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

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

Example 3.27 The term $t \equiv (\varepsilon\lambda_x)(x\lambda_v)(x\delta)(\varepsilon\lambda_y)((x\lambda_z)y^\circ\delta)(y\lambda_u)u$ becomes, in the notation with de Bruijn- indices: $t \equiv (\lambda)(1\lambda)(2\delta)(\lambda)((3\lambda)2^\circ\delta)(1\lambda)1$. Now $t\restriction 2^\circ \equiv (\lambda)(1\lambda)(2\delta)(\lambda)(3\lambda)2^\circ$. So $\mathtt{sieveseg}_\lambda(t\restriction 2^\circ) \equiv s_4s_3s_2s_1 \equiv (\lambda)(1\lambda)(\lambda)(3\lambda)$. Hence, 2° is bound in t since $2 \leq \mathtt{weight}_\lambda(t\restriction 2^\circ) = 4$. Moreover, the type of 2° in t is $\mathtt{body}(s_2) \equiv \varepsilon$. There are no free variables in t, hence t is closed.

Things are, however, not so simple in the case that the term contains segment abbreviations.

Example 3.28 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. Therefore, we add a collection of segment variables to our set of variables, which are pairs of numbers:

Definition 3.29 *(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 $\lambda_{\mathtt{sg}}$ as being a binding λ for segment abbreviations. We do not allow that $\lambda_{\mathtt{sg}}$-items occur "on their own". They should always be a part of a δ-λ-segment of the form $(\overline{s}\delta)(\lambda_{\mathtt{sg}}a)$, coding the abbreviation of a segment $\overline{s}$.

In $(n;m)$, a **segment variable item**, *the index n gives a reference to the binding $\lambda_{\mathtt{sg}}$ and m is the λ-weight of the abbreviated segment. Section 7 will give many examples of such a phenomenon.*

Definition 3.30 *(Well-typedness of terms)*

We say that a term t is "well-typed" with respect to a particular system containing variable, abstraction and application conditions, if we can deduce $\vdash t$ where $\vdash$ is defined by the following three equations:

$$\frac{\textit{variable condition}}{\overline{s} \vdash x} \tag{3}$$

$$\frac{\overline{s} \vdash t \qquad \overline{s}(t\lambda) \vdash t' \qquad \textit{abstraction condition}}{\overline{s} \vdash (t\lambda)t'} \tag{4}$$

$$\frac{\overline{s} \vdash t \qquad \overline{s}(t\delta) \vdash t' \qquad \textit{application condition}}{\overline{s} \vdash (t\delta)t'} \tag{5}$$

Notation 3.31 *(Construction rules)*
We call equations 3, (respectively 4 and 5), a **variable** (respectively **abstraction** and **application**) **construction rule.**

Example 3.32 With abstraction condition $t \equiv \varepsilon$, $t' \not\equiv \varepsilon$, empty variable condition and application condition, we obtain the syntax of the *un*typed lambda calculus.

Remark 3.33 The variable condition is optional. Example 3.34 gives two variable conditions. The abstraction condition and the application condition vary from system to system, or may even be absent. In type systems for example, the type information plays a predominant role in the application condition: t may only be an "argument" of t' (i.e. $\overline{s} \vdash (t\delta)t'$) if t' is some kind of "function", with a "domain" in which t fits. This requirement must be expressed formally in the application condition. Sections 4, 5 and 6 give examples of the abstraction and application conditions. Example 3.36 gives a well-typed term.

Example 3.34 Here are some examples of variable conditions:

1. $x \leq \mathtt{weight}_\lambda(\overline{s})$ (Here count ε as zero, in case $x \equiv \varepsilon$).

   This variable condition restricts terms to the closed ones.

2. $1 \leq deg(x) \leq 3$.

   Hence the degree of any term is between 1 and 3. This is the case in AUT-QE and AUT-68; (see Section 6). 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.

Definition 3.35 *(Proof trees)*
For each "well-typed" term, we call the construction tree, which contains at the same time a proof for its "well-typedness", the **proof tree** *for the term.*

Example 3.36 The lowest part of the proof tree of $(\varepsilon\lambda_x)\,((x\lambda_u)\,((u\delta)(x\lambda_t)x\lambda_y)\,(u\lambda_z)y\;\lambda_v)u$, based on these rules, is the following:

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

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

We need a function which updates variables. This we do by extending our set $\Omega_{\lambda\delta}$ with a set of φ-operators Ω_φ. We use the φ's with a double index: $\varphi^{(k,i)}; k, i \in \mathcal{N}$ and call all $(\varphi^{(k,i)})$'s φ-**items**. Our terms are now $\Omega_{\lambda\delta\varphi}$-terms. The use of the φ-items is established in the following rules.

Definition 3.37
(φ-transition rules:)
$(\varphi^{(k,i)})(t'\lambda) \to_\varphi ((\varphi^{(k,i)})t'\lambda)(\varphi^{(k+1,i)})$
$(\varphi^{(k,i)})(t'\delta) \to_\varphi ((\varphi^{(k,i)})t'\delta)(\varphi^{(k,i)})$
(φ-destruction rules:)
For $k, i \in \mathcal{N}$, we have:
$(\varphi^{(k,i)})x \to_\varphi x + i$ *if* $x > k$
$(\varphi^{(k,l)})x \to_\varphi x$ *if* $x \leq k$ *or* $x \equiv \varepsilon$.

Definition 3.38 *(φ-abbreviation)*
For all $k \in \mathcal{N}$, $\varphi^{(k)}$ denotes $\varphi^{(0,k)}$. Moreover, φ denotes $\varphi^{(1)}$ (hence $\varphi^{(0,1)}$).

Definition 3.39 *(void β-reduction)*
Assume that a δ-λ-segment $\overline{s}$ occurs in an $\Omega_{\lambda\delta}$-term t, where the final operator λ of $\overline{s}$ does not bind any variable in t. Let t_1 be the scope of $\overline{s}$. Then t reduces to the term t', obtained from t by removing $\overline{s}$ and replacing t_1 by $(\varphi^{(-1)})t_1$.

Example 3.40 Let us take $(1\delta)(2\lambda)(4\delta)2$. In this term, call it t, the δ-λ-segment $(1\delta)(2\lambda)$ occurs and its λ does not bind any variable in t. Moreover, $(4\delta)2$ is the scope of $(1\delta)(2\lambda)$ and if in t we remove $(1\delta)(2\lambda)$ and replace $(4\delta)2$ by $(\varphi^{(-1)})(4\delta)2$ we get $(3\delta)1$. Hence t reduces to $(3\delta)1$.

Example 3.41

1. $(1\delta)(2\lambda)(2\delta)2 \to_\beta (1\delta)1$; this states that $(\lambda_{x:z}.uu)u$ reduces to uu.
2. $(1\delta)(2\lambda)(3\lambda)3 \to_\beta (2\lambda)2$; this states that $(\lambda_{u:y}.\lambda_{x:y}.z)z$ reduces to $\lambda_{x:y}.z$.

Notation 3.42 *(β-reduction)*
Note that void β-reduction is a β-reduction, so let us write $t \to_\beta t'$ when the reduction in the above definition takes place. β-reduction in general however, will not be explained and the reader is referred to [NeK92a]. It is not needed for this paper, further than saying that

- $(t\delta)(t'\lambda)t'' \twoheadrightarrow_\beta t''[x := t]$,
- the x's are the variables in t'' bound by the mentioned λ,
- $[x := t]$ is a postfix meta-operator standing for the substitution of t for all free occurrences of x.

4 Canonical types

Variables occurring bound in a term in typed lambda calculus have a "natural" type, as expressed in Definition 3.26. This type is the body of the λ-item 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 4.1 *(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 3.26.*

Example 4.2 Take the term $(1\delta)(2\lambda)1$ (or in sugared notation $(u\delta)(y\lambda_x)x$).

1. If $t' \equiv 1$ (the x), then $\mathtt{typ}(t') \equiv \varepsilon(\varphi^{(1)})2 \rightarrow_\varphi 3$. This is obvious, it says that the type of x is y (look at figure 2).
2. If $t'' \equiv (2\lambda)1$ then $\mathtt{typ}(t'') \rightarrow_\varphi (2\lambda)3$. This is intuitively correct. It states that the type of $\lambda_{x:y}.x$ is $\lambda_{x:y}.y$ (identifying λ's and Π's).
3. If $t''' \equiv (1\delta)(2\lambda)1$ then $\mathtt{typ}(t''') \rightarrow_\varphi (1\delta)(2\lambda)3 \rightarrow_\beta 2$. Again, this is intuitively correct. It states that the type of $(\lambda_{x:y}.x)u$ is y. In Section 4.2 we will see how to include an application condition stating that the type of u and the type y must be compatible. Recall moreover that types themselves are terms.

As we see, calculating the canonical type $\mathtt{typ}(t')$ of a (sub-)term t' is very straightforward. Just replace the end variable of t' by its type t'' (together with some updating of free variables in t'').

Following the general style of this paper, we can also use a *type item* (τ) and a type reduction operator $\rightarrow_\tau$ instead of the type function $\mathtt{typ}$. Hence, we extend our set of terms defined in Definition 2.7 in order to incorporate these τ-items (we now have $\Omega_{\lambda\delta\varphi\tau}$-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 φ-items):

Definition 4.3 *(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 here that a term t, φ-reduces (repectively τ-reduces) to another term t' if t' is obtained from t by φ-reducing (respectively τ-reducing) a subterm of t.

Example 4.4 Take again the term $(1\delta)(2\lambda)1$. Now

1. $(\tau)1 \rightarrow_\tau (\varphi^{(1)})2 \rightarrow_\varphi 3$.
2. $(\tau)(2\lambda)1 \rightarrow_\tau (2\lambda)(\tau)1 \rightarrow_\tau (2\lambda)(\varphi^{(1)})2 \rightarrow_\varphi (2\lambda)3$.
3. $(\tau)(1\delta)(2\lambda)1 \rightarrow_\tau (1\delta)(\tau)(2\lambda)1 \rightarrow_\tau (1\delta)(2\lambda)(\tau)1 \rightarrow_\tau (1\delta)(2\lambda)(\varphi^{(1)})2 \rightarrow_\varphi (1\delta)(2\lambda)3 \rightarrow_\beta 2$.

4.1 The type of an abstraction

In what follows, we use λ_1 for dependent product formation (usually denoted as Π), and λ_2 for the — ordinary — function operator λ. Now in Definition 4.3, we did not distinguish between the two operators. Usually, the following rule is employed:

Definition 4.5 *(Abstraction rule)*

1. *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.*
2. *The type of a λ-abstraction $\lambda x : t_1 \,.\, t'$ is the corresponding Π-abstraction $\Pi x : t_1 \,.\, t''$.*

As a consequence, one may refine the transition rules for λ-items as follows, replacing those of Definition 4.3 for the case that $\omega \equiv \lambda$:

Definition 4.6 *(τ-transition rules for indexed λ-items:)*
$(\tau)(t_1\lambda_1) \rightarrow_\tau (\tau)$
$(\tau)(t_1\lambda_2) \rightarrow_\tau (t_1\lambda_1)(\tau)$

Example 4.7

1. If $t \equiv (1\delta)(2\lambda_1)1$ then $(\tau)(2\lambda_1)1 \rightarrow_\tau (\tau)1 \rightarrow_\tau (\varphi^{(1)})2 \rightarrow_\varphi 3$. That is, the type of $\Pi_{x:y}.x$ is y.
2. If $t \equiv (1\delta)(2\lambda_2)1$ then $(\tau)(2\lambda_2)1 \rightarrow_\tau (2\lambda_1)(\tau)1 \rightarrow_\tau (2\lambda_1)(\varphi^{(1)})2 \rightarrow_\varphi (2\lambda_1)3$. That is, the type of $\lambda_{x:y}.x$ is $\Pi_{x:y}.y$.

There may be circumstances in which one desires to have more "layers" of λ's. In such a case, we can extend this kind of systems by incorporating more different λ's. For example, with an infinity of λ's, viz. $\lambda_0, \lambda_1, \lambda_2, \lambda_3 \ldots$, we can generalize Definition 4.6, to the following, if we add a reduction rule stating that $(t_1\lambda_0)$ reduces to the empty segment:

Definition 4.8 *(τ-transition rule for arbitrarily many indexed λ-items)*
$(\tau)(t_1\lambda_{i+1}) \rightarrow_\tau (t_1\lambda_i)(\tau)$, *for* $i = 0, 1, 2, \ldots$

4.2 The type of an application

Recall from the third part of Example 4.2 that we might need to add an abstraction condition which states that the type of u and the type y are compatible. In fact, one usually employs a rule of the following form:

Definition 4.9 *(Application rule)*
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 $(t\delta)F$ has type $t_1[x := t]$.

For this purpose we maintain Definition 4.6 as regards the λ-items, and we employ the following τ-transition rule for δ-items (as in Definition 4.3):

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

However, we make demands to rule 5 (see Definition 3.30), 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:

Definition 4.11 *(General application condition)*

$(\tau)t' =_{\tau,\beta,\varphi} (t''\lambda_1)t_1$ *and* $(\tau)t =_{\tau,\beta,\varphi} t''$.

Now it follows that

$$(\tau)(t\delta)t' \to_\tau (t\delta)(\tau)t' =_{\tau,\beta,\varphi} (t\delta)(t''\lambda_1)t_1 \twoheadrightarrow_\beta t_1[x := t] \tag{6}$$

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" $t_1[x := t]$.

Example 4.12 Take the term $(1\lambda_2)(1\delta)(2\lambda_2)1$ (or in sugared notation $(y\lambda_u)(u\delta)(y\lambda_x)x$). From Example 4.7, $(\tau)(2\lambda_2)1 =_{\tau,\beta,\varphi} (2\lambda_1)3$. Moreover, the type of u is:
$(\tau)1 =_{\tau,\beta,\varphi} (\varphi^{(1)})1 =_{\tau,\beta,\varphi} 2$.
Hence the application condition for $(1\delta)(2\lambda_2)1$ is satisfied and
$(\tau)(1\delta)(2\lambda_2)1 =_{\tau,\beta,\varphi} \twoheadrightarrow_\beta 2$.

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 $(t_1\delta)(t_2\Pi)$. 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.

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 5).

- *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 is 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

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

(cf. equation 6) 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 $\mathtt{typ}(t')$ 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 β-reduction $(t\delta_1)(t''\lambda_1)t_1 \twoheadrightarrow_\beta t_1[x := t]$.

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 of this paper.

Remark 4.13 In a now commonly accepted setting (see [Bar84] or [BaH90]), 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.

Example 4.14 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.

5 The typing relation in PTS's

We start with a short summary of so-called Pure Type Systems (*PTS's*), as described in [BaH90]; see also [Bar84]. 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 employ ordinary variables, and not de Bruijn-indices or another referential variable denotation. So φ-items and updating are not incorporated. Moreover, 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. The following gives the conditions which must be obeyed for the construction of (λ- or Π-) abstraction terms in PTS's:

Definition 5.1 *(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:)

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

In these rules, Γ denotes a context, t_1, t_2 and u are terms and s_1, s_2 and s_3 are so-called sorts (these should not be confused with the meta-variable notation for items). 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). Note moreover that these rules are consistent with Definition 4.5.

Remark 5.2 The Π-formation and Π-introduction rules as given above can be condensed into one Π-rule (combined Π-rule):

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

Now it is obvious that Definition 4.6 encorporates the essential part of both Π-rules, translated in our setting. In fact,

- $(\tau)(t_1\lambda_1)$ τ-reduces to (τ) by itself (the λ_1-item — i.e. the Π-item — is erased).
- $(\tau)(t_1\lambda_2)$ 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).

Moreover, the type information given by the Π-formation and Π-introduction rules (via the statements $(\Pi x : t_1\ .\ t_2) : s_2$ and $(\lambda x : t_1\ .\ u) : (\Pi x : t_1\ .\ t_2)$, respectively) is no longer necessary, since we have the canonical type operator τ at our disposal (cf. Definition 4.6 and Remark 4.13).

Now we come to "Barendregt's cube" where both s_1 and s_2 can be either $*$ or $\Box$ (again, see [Bar84] or [BaH90]). These two are related by the *axiom* statement: $* : \Box$. In this 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	:	$*$	:	$\Box$
	u	:	t_2	:	$*$	:	$\Box$
$(*,\Box)$	x	:	t_1	:	$*$	:	$\Box$
			u	:	t_2	:	$\Box$
$(\Box,*)$			x	:	t_1	:	$\Box$
	u	:	t_2	:	$*$	:	$\Box$
$(\Box,\Box)$			x	:	t_1	:	$\Box$
			u	:	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 [deB80]). 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 [CoH88]).

In our system, we may identify $\Box$ with ε. 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. Let us repeat the construction rule under consideration, as stated in Definition 3.30:

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

Definition 5.3 *(Incorporating* Π*-formation)*
The Π*-formation rule is obtained by reading* λ_1 *for* λ *and taking the abstraction condition:* $(\tau)t\rightarrow_{\tau,\beta} s_1$ *and* $(\tau)t'\rightarrow_{\tau,\beta} s_2$*, for* $s_1,s_2\in\{*,\Box\}$.

Definition 5.4 *(Incorporating* Π*-introduction)*
For the Π*-introduction rule we take* λ_2 *for* λ *and the abstraction condition:* $(\tau)t\rightarrow_{\tau,\beta} s_1$ *and* $(\tau)^2t'\rightarrow_{\tau,\beta} s_2$*. Here* $(\tau)^2$ *is an abbreviation for* $(\tau)(\tau)$.

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 5.5 *(Π-elimination rule)*

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

Now we recall the appropriate construction rule from Definition 3.30:

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

and we incorporate Π-elimination as follows:

Definition 5.6 *(Incorporating* Π*-elimination)*
As regards the Π*-elimination rule for PTS's, we use the application condition: there are* t'' *and* t_1 *such that* $(\tau)t'=_{\tau,\beta}(t''\lambda_1)t_1$ *and* $(\tau)t=_{\tau,\beta}t''$.

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 5.7 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.

6 The typing relation in Automath-systems

In this section we describe the definitions of three of de Bruijn's Automath-systems in our setting. These systems *do* have a canonical type operator, albeit not as part of its language. Consequently, we only 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.

The systems that we discuss are AUT-68, AUT-QE and Λ.[4] All these systems have been developed around 1970. The oldest of the three is AUT-68, the more powerful variant AUT-QE followed soon. The system Λ was meant to be a simplified and more uniform version of the two other systems. It was developed slightly later.

6.1 The system AUT-68

The system AUT-68 ([vanD80]) was meant as a formal system suitable for expressing large parts of mathematics, some of its features include:

- An in-built logical frame for reasoning, in a logic chosen by the user (e.g. classical predicate logic, intuitionistic logic),
- The possibility of a step-wise development of a mathematical theory by means of axioms and primitive notions; lemma's, theorems, corollaries and their proofs; definitions and abbreviations,
- An explicit treatment of contexts (assumptions, variable introductions) for theorem-like and definition-like notions.
- 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.

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. The following definitions show what are the typing relation and construction rules that will describe AUT-68 in our setting.

Definition 6.1 *(Canonical types for AUT-68)*
The **canonical type** $\mathbf{typ}(t')$ *of a term t' can be calculated by means of the following τ-transition rules:*

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

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

[4] We thank Bert van Benthem Jutting for the descriptions below of AUT-68 and AUT-QE.

Definition 6.2 *(Well-typedness of AUT-68)*
In Definition 3.30, we need the following variable, abstraction and application conditions:

- *Variable condition: The only variable of degree 1 is* $*$.
- *Abstraction conditions:*
  1. *Either* $\mathtt{deg}(t) = 2$, *or* $\mathtt{deg}(t) = 1$ *and* $\overline{s}$ *is a context (see Definition 3.2), and*
  2. $2 \leq \mathtt{deg}(t') \leq 3$.
- *Application condition:*

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

6.2 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. Moreover, AUT-QE has, like AUT-68, only terms of degree 1, 2 and 3. The following will show how we can incorporate a (again *definition-free*) version of AUT-QE in our setting:

- Canonical type: as for AUT-68 (see Definition 6.1).
- Variable condition: as for AUT-68 (see Definition 6.2).
- Abstraction condition 1: as for AUT-68 (see Definition 6.2).
- Abstraction condition 2: absent (see Definition 6.2).
- Application condition:
  either $\mathtt{deg}(t') = 3$ and $\overline{s} \vdash (t\delta)\mathtt{typ}(t')$,
  or $\mathtt{deg}(t') = 2$ and $\mathtt{typ}(t') =_\beta (\mathtt{typ}(t)\lambda)t''$ for some term t''.

6.3 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. 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 $\lambda_0, \lambda_1, \lambda_2, \ldots$).
- Application is only restricted in the sense that the general application condition (see Definition 4.11) must hold, albeit in a generalized version (due to the unlimited degrees). Application is allowed for terms of all degrees, so that Π-application (see again Section 4) is one of the features: β-reduction is treated similarly for all degrees, in the form $(t_1\delta)(t_2\lambda_x)t_3 \to_\beta t_3[x := t_1]$.
- The type operator behaves uniformly, as in Definition 4.3: we have that $(\tau)(t_1\omega) \to_\tau (t_1\omega)(\tau)$, for $\tau \equiv \lambda$ or $\tau \equiv \delta$. Hence, Λ has explicit, and not implicit (compulsory) $\beta_{\delta\Pi}$-reduction.

7 An example

In order to demonstrate some of the features discussed above, we propose a system λ_{C_1} that has in principle similar power as Coquand and Huet's *Calculus of Constructions* (or λC, see [CoH88]) and give the proof of a logic theorem in this setting.

7.1 The system λ_{C_1}

λ_{C_1} has the following general features:

- Variable names like x, y, ..., are used instead of de Bruijn-indices.
- Segment abbreviations, as discussed in Definitions 3.4 and 3.29 are incorporated.
- There is a distinction between Π's and λ's, (i.e., λ_1's and λ_2's), respectively.
- A canonical type operator typ, with the usual notational convention that $\mathtt{typ}^2(t) \equiv \mathtt{typ}(\mathtt{typ}(t))$, etc, is used.
- Π-application and the corresponding $\beta_{\delta\Pi}$-reduction are present.
- The maximal degree is 3.

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 Section 7.3, we write Π for λ_1 and λ for λ_2.) Moreover, we have one δ, and as a consequence of what we said above, there will be *no* φ's and *no* τ's. The last two operators may only be used in the meta-language.

Remark 7.1 When we use deg or typ in a condition, we implicitly require that these operations are indeed defined for the terms under consideration.

Definition 7.2 *(Construction rules for λ_{C_1})*
The construction rules for terms are the following:

variable construction:

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

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,

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

application construction:

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

where

appcon is : there are t_1 and $j \epsilon \{0,1\}$ such that $(\tau)^j t' =_{\tau,\beta} ((\tau)t\ \lambda_1)t_1$

Note that abscon is the same abstraction condition as the one for λC defined in Definitions 5.3 and 5.4. 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 5, and we assume that $*$ is the *only* inhabitant of $\Box$.

Hence, the condition "$t : s_1$" can be replaced by $\mathtt{typ}(t) \equiv \varepsilon$ (in the case that $s_1 \equiv \Box$) or $\mathtt{typ}(t) \equiv *$ (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 $(\tau)t' =_{\tau,\beta} \varepsilon$ or $(\tau)^2 t' =_{\tau,\beta} \varepsilon$. In the case that $\lambda \equiv \lambda_2$ (i.e., the ordinary "functional" λ), the condition "$t' : t'' : s_2$ for some t''" becomes $(\tau)^2 t' =_\beta \varepsilon$ or $(\tau)^3 t' =_\beta \varepsilon$. The rules for τ are given in Definitions 4.3 and 4.6.

Remark 7.3 It is not hard to see that both the typing relation and the reduction relations in the presented system are degree-consistent.

7.2 The environment of the theorem

The theorem 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. Example 3.21.) This environment that we work with 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. when absurdity holds, then every proposition holds),
- the notion of implication $a \Rightarrow b$ is defined as the class of all mappings of a to b, hence sending proofs of a to proofs of b,
- the notion of negation $\neg a$ is defined as $a \Rightarrow \perp$,
- the following logical theorem is expressed and proved:

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

In a kind of "Mathematical Vernacular", adopted from the style of the Automath-family, this piece of logico-mathematical text can be expressed by the following three definitions:

Definition 7.4 *(The axiomatic part)*

let $*$ be by axiom *the class of all propositions.*
let $\perp$ be by axiom *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*.

Definition 7.5 *(The definitional part)*

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* $\perp$.

Definition 7.6 *(The theorem-and-proof part)*

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*.

Remark 7.7 In the above text, $\perp$ 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.

7.3 Translating the environment in λ_{C_1}

The logico-mathematical text defined in the previous section, will be translated in its entirety, as one segment in λ_{C_1}. For convenience' sake, we write this segment 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.

Definition 7.8 *(Translating Definition 7.4)*
Definition 7.4 gives the following three λ-items:

(λ_*)
$(*\lambda_\perp)$
$((*\Pi_a)(\perp\Pi_t)a\ \lambda_{\perp-el})$

That is: $*$ 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.

Definition 7.9 *(Translating Definition 7.5)*
Definition 7.5, 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 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: $a \Rightarrow b$).

Analogously, $\neg$ 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: $\neg a$).

Example 7.10 The reader may check that the following chain of β-reductions is correct:

$\neg\ \rightarrow_\beta$
$(*\lambda_a)(\bot\delta)(a\delta)\Rightarrow\ \rightarrow_\beta$
$(*\lambda_a)(\bot\delta)(a\delta)(*\lambda_a)(*\lambda_b)(a\Pi_x)b \rightarrow_\beta$
$(*\lambda_a)(\bot\delta)(*\lambda_b)(a\Pi_x)b\ \rightarrow_\beta$
$(*\lambda_a)(a\Pi_x)\bot.$

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 $(\bot\delta)(a\delta)\Rightarrow$.

Definition 7.11 *(Translating Definition 7.6)*
Definition 7.6 of the text can be translated into one δ-λ-segment:

$((*\lambda_a)(*\lambda_b)(a\lambda_x)((a\delta)\neg\ \lambda_y)((x\delta)y\ \delta)(b\delta)\bot\text{-}el\ \delta)((*\Pi_a)(*\Pi_b)(a\Pi_x)((a\delta)\neg\ \Pi_y)b\ \lambda_{pr})$

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

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

It is not hard to check that this segment obeys the conditions for term construction as given above:[5]

[5]Note that this segment can be considered to be a term by adding ϵ to the segment.

variable condition:
The term is closed and all degrees are ≤ 3.

abstraction condition:
Left to the reader.

application condition:
Examples are:
$\mathtt{typ}(*\lambda_a)(*\lambda_b)(a\Pi_x)b \rightarrow_\tau$ (by Section 4)
$(\tau)(*\lambda_a)(*\lambda_b)(a\Pi_x)b \rightarrow_\tau$ (by Def. 4.8)
$(*\Pi_a)(\tau)(*\lambda_b)(a\Pi_x)b \rightarrow_\tau$ (by Def. 4.8)
$(*\Pi_a)(*\Pi_b)(\tau)(a\Pi_x)b \rightarrow_\tau$ (by Def. 4.8; $(a\Pi_x)$ reduces to the empty segment)
$(*\Pi_a)(*\Pi_b)(\tau)b \rightarrow_\tau$ (by Def. 4.3)
$(*\Pi_a)(*\Pi_b)*$
and
$\mathtt{typ}(*\lambda_a)(\bot\delta)(a\delta)\Rightarrow$
$\rightarrow_\tau$ (by Section 4)
$(\tau)(*\lambda_a)(\bot\delta)(a\delta)\Rightarrow$
$\rightarrow_\tau$ (by Def. 4.8)
$(*\Pi_a)(\tau)(\bot\delta)(a\delta)\Rightarrow$
$\rightarrow_\tau$ (since
$(\tau)\Rightarrow \ =_\tau \ (*\Pi_{a'})(*\Pi_{b'})* \ =_{\tau,\beta} \ ((\tau)a\ \Pi_{a'})(*\Pi_{b'})*$, so
$(\tau)(a\delta)\Rightarrow \ =_\tau \ (*\Pi_{b'})* \ =_{\tau,\beta} \ ((\tau)\bot\ \Pi_{b'})*$) and
$(\tau)(\bot\delta)(a\delta)\Rightarrow \ =_\tau \ *$)
$(*\Pi_a)*$.

Other checks of the application condition, such as:
$\mathtt{typ}\ (*\lambda_a)(*\lambda_b)(a\lambda_x)((a\delta)\neg\ \lambda_y)((x\delta)y\ \delta)(b\delta)\bot\text{-}el \ \twoheadrightarrow_{\tau,\beta}$
$(*\Pi_a)(*\Pi_b)(a\Pi_x)((a\delta)\neg\ \Pi_y)b$,
are left as an exercise for the reader.

7.4 The theorem and its proof

The main λ-item of the segment in definition 7.11 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 the theorem, 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 is true). This inhabitant can be found in the main δ-item of the δ-λ-segment:
$(*\lambda_a)(*\lambda_b)(a\lambda_x)((a\delta)\neg\ \lambda_y)((x\delta)y\ \delta)(b\delta)\bot\text{-}el$.

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. This is indeed the case, as the reader may check.

Finally, we show the usefulness of segment abbreviations for the same theorem and proof. (These abbreviations can also be of help for the check of the application condition.) Segment abbreviations add to the efficiency. There are already several segment duplications in term 4. 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 $(*\Pi_a)(*\Pi_b)$.

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. 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 mathematical reasoning) that different segments occur stackwise in the complete term; that is to say, an occurrence of a segment $(t_1\lambda_{a_1})\ldots(t_n\lambda_{a_n})$ may be followed rather closely by the same segment, or by a segment which is one item longer: $(t_1\lambda_{a_1})\ldots(t_{n+1}\lambda_{a_{n+1}})$ or shorter: $(t_1\lambda_{a_1})\ldots(t_{n-1}\lambda_{a_{n-1}})$, and this may happen again and again. (The same holds if some of the λ's are replaced by Π's.)

The segment abbreviations which we proposed can solve the problem. For this, we add one more abbreviation in this translation process: when, e.g. $(*\lambda_a)(*\lambda_b)$ is abbreviated by $(\mathtt{b};2)$, then we abbreviate $(*\Pi_a)(*\Pi_b)$ by $((\tau)\mathtt{b};2)$. 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 4.6).

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

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

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 $((\tau)\mathtt{x};n)$ for which the corresponding variable item $(\mathtt{x};n)$ figures in the same line. Instead, we shall use a horizontal stroke: $\_\!\_$, which should be considered to refer to the segment variable $(x;n)$, 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: $((\tau)\mathtt{b};2)$, $((\tau)\mathtt{a};1)$ and $((\tau)\mathtt{c};4)$, 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.

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

8 Conclusions

In this paper, we introduced an alternative λ-calculus notation which is flexible enough for the expression of many type systems. This notation allows many generalizations. For example higher degrees and segment abbreviations are straightforwardly attainable. Moreover, 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.

We looked at the role of the types in our setting. 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. Moreover, 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. We also presented a number of Automath-systems in the proposed setting, which resulted in concise definitions for complicated systems. Finally, we worked out the proof of a theorem taken from logic in our setting.

All the above is an evidence that our new framework is expressive, general and uniform. We believe that this framework deserves some attention in the ongoing research in λ-calculus and type theory.

9 Acknowledgments

We would like to thank Erik Poll for having read the paper very carefully and for his productive comments.

References

[Bar84] 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, 1992.

[BaH90] Barendregt, H.P., and Hemerik, C., Types in Lambda calculi and programming languages, *Proceedings of the ESOP conference*, Copenhagen 1990.

[Bej77] Benthem Jutting, L.S. van, *Checking landau's "Grundlagen" in the AUTHOMATH system*, Ph.D. thesis, Eindhoven university of Technology, Eindhoven, 1977.

[deB72] Bruijn, N.G. de, Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, *Indagationes Math. 34, No 5*, pp. 381-392, 1972.

[deB74] Bruijn, N.G. de, Some extensions of the AUTOMATH: the AUT-4 family, department of Mathematics, Eindhoven University of Technology, Eindhoven, 1974.

[deB80] Bruijn, N.G. de, A survey of the project AUTOMATH, in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Eds. J.R. Hindley and J.P. Seldin, Academic Press, New York/London, pp. 29-61, 1980.

[CoH88] Coquand, T. and Huet, G., The calculus of Constructions, *Information and Control 76*, pp. 95-120, 1988.

[vanD80] Daalen, D.T. van, *The language theory of Automath*, Ph.D. thesis, Eindhoven university of Technology, Eindhoven, 1980.

[How80] 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, 1980.

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

[NeK92a] Nederpelt, R.P., and Kamareddine, F.D., On stepwise substitution, submitted for publication.

[NeK92b] Nederpelt, R.P., and Kamareddine, F.D., A useful λ-notation, submitted for publication.

In this series appeared:

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

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

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

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

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