Algorithmic definition of lambda-typed lambda calculus

Algorithmic definition of lambda-typed lambda calculus

N. G. de Bruijn
Technological University Eindhoven, Department of Mathematics and Computing Science, PO Box 513, 5600MB Eindhoven, The Netherlands

1 Introduction

1.1. The system $\Delta\Lambda$. The typed lambda calculus $\Delta\Lambda$ was defined in an earlier paper [2], also briefly discussed in [4], section 3. The present paper will deal essentially with the same system. The new version has exactly the same set of correct terms but allows a few more reductions. This has the effect of narrowing the gap between the definition of correctness and efficient correctness-checking algorithms.

It is not necessary to read [2] in order to understand the present paper. The material will be presented here independently. We only refer to [2] (section 6) for the reason why $\Delta\Lambda$ is to be preferred over other systems as a basic structure underlying Automath-like languages.

In the next sub-sections we use the notation explained in section 2.1 and some further terminology explained later in this paper.

1.2. The essence of $\Delta\Lambda$. The essential difference between $\Delta\Lambda$ and systems like Nederpelt's Λ (see [6]) lies in the matter of correctness of applicators. In the usual systems the correctness of a term $\langle t_1\rangle t_2$ requires in the first place that *both* t_1 and t_2 are correct. In $\Delta\Lambda$ the correctness of t_2 is not required. Whether t_2 is accepted here, depends on what we have for t_1. This dependence is connected to the usage of applicator-abstractor pairs for the role of definitions in a mathematical language (see [2], section 6.4).

Another feature of $\Delta\Lambda$, connected to elimination of definitions in a mathematical text, is the idea to split β-reduction into a number of local reductions plus one final AT-removal: the removal of an applicator-abstractor pair in the case that nothing refers to it any more. In a mathematical text the elimination of a definition at a certain place is a local affair too, since it does not involve the elimination of the definition at all other places. And AT-removal corresponds to discarding a definition that is nowhere used.

Nederpelt had also η-reduction in his system Λ. We shall not consider that in this paper, although it would not be very hard to add it.

Finally $\Delta\Lambda$ is what we may call a *pure* lambda calculus in the sense that both application and abstraction commute with typing (cf. [4], section 1).

1.3. Comparing the present paper to [2].
(i) The metalanguage in [2] discusses terms as trees, with references (arrows) from end-points to T-nodes (nodes representing abstractors). The present paper considers terms as strings of symbols, and the references are made in the conventional way by means of named variables.

(ii) The present system admits more reductions. The local reductions are not for AT-pairs only, but also for AT-couples. These reductions for couples were introduced by Nederpelt (see [6], Ch.II, Definitions 6.5 and 6.7). The couples were also mentioned in [2], section 4.3, but there they were not used for reductions.

A few words about couples here. In an applicator-abstractor sequence like $\langle t_1\rangle[t_2]_x\langle t_3\rangle[t_4]_y$ the pair $\langle t_1\rangle[t_2]_x$ can play its role in β-reduction. But in $\langle t_1\rangle\langle t_3\rangle[t_4]_y[t_2]_x$ the $\langle t_1\rangle$ and $[t_2]_x$ no longer form a pair but a couple. In our present version of $\Delta\Lambda$ we will allow reduction with respect to such couples.

(iii) The definition of correctness in [2], section 5.3 somehow suffered from the absence of couple reductions. In that definition one can sense an implicit re-ordering of couples to pairs. That made the definition quite heavy. And the step from the definition of correctness to the correctness-checking algorithm (described in some detail in [4], section 3) is not entirely trivial. In the present version these things are more direct, and the correctness definition is almost the same as the checking algorithm. The only thing that we have to do when turning the definition into an efficient algorithm is to cut out some obvious duplication of work.

(iv) The scopes of the old and new versions are the same, in the sense that the extra reductions by means of AT-couples do not lead to extra equivalences. This is clear in the example mentioned under (ii). If both before and after reduction with respect to the couple formed by $\langle t_1\rangle$ and $[t_2]_x$ in $\langle t_1\rangle\langle t_3\rangle[t_4]_y[t_2]_x$ we apply full β-reduction on $\langle t_3\rangle[t_4]_y$, we get terms that are related by β-reduction on the *pair* $\langle t_1\rangle[t_2]_x$.

(v) A central role in the present paper is played by the idea of *trimming* (section 3.6), which did not occur in [2]. If we consider a particular instance of a variable in a term t, then trimming means taking away certain applicators that can have no influence on that instance of that variable. The effect of trimming is again a term. One might say that the trimmed term is the variable preceded by its context ("context" being taken in the sense of what was called "knowledge frame" already in [2]). Talking about such trimmed terms simplifies the discussion of properties of t itself.

2 Notation

2.1. Applicators and abstractors. We deviate from standard lambda calculus notation in the sense that we follow the Automath tradition of writing applicators on the left instead of on the right. If f is a function and v a value

of the argument, we write $\langle v\rangle f$ instead of the more usual $f\ v$ or $f(v)$. This way of writing the argument in front is more than a matter of taste: it is absolutely essential for the present paper.

The delimiters $\langle\ \rangle$ should not be confused with the $<\ >$ we shall use for syntactic categories in BNF grammar.

Abstractors (or typed lambdas) are usually denoted with a notation like $\lambda_{x:A}$, denoting quantification over a variable of type A. In Automath it is written as $[x : A]$, but here we shall write $[A]_x$. As usual, the abstractors are written in front of the terms they act on.

2.2. Variables. We shall be a bit informal about how variables are represented. Anyway we take the point of view that the names of variables are inessential. One may use depth references (nowadays often called de Bruijn indices) to indicate to which lambda any particular instant of a variable is referring (see [1]), but one may also think of more direct ways. In [2] terms were represented as trees, and a reference from an instance of a variable to its lambda was an arrow from an end-point to a lambda lower in the tree. We shall not use that tree language in the present paper, however. Instead of it, we just consider terms as strings of symbols, and we can say that references are not made to a point in a tree but to a closing bracket] in a term. In order to simplify our presentation we just fall back on the use of named variables, not bothering about how variables are dealt with in implementations. Therefore we provide each closing] with the name of a variable as a subscript. The variables act as addresses of those brackets. So references to a variable x of type t will be made like this:

$$\cdots[t]_x\cdots x\cdots x\cdots x\cdots. \tag{2.2.1}$$

In the presentation of this paper we shall occasionally omit the subscripts of the]'s.

2.3. Metavariables. We shall use w, v (sometimes with subscripts) as metavariables for trains (see section 3.1), t (sometimes with subscripts) for terms, q for terminals (variables or τ). And we use $\sqsubset$ as a metavariable representing either $\langle$ or [.

3 Terms

3.1. Syntax. We have a set of variables and a basic symbol τ, different from the variables. And we have four delimiters: $\langle$, $\rangle$, [,], where the last one has to be seen as being subscripted by a variable, like $]_x$. With these delimiters, variables and τ we build *terms*; as auxiliary syntactic categories we handle *terminals*, *wagons* and *trains*. A terminal is a variable or τ, a wagon is a term surrounded by $\langle\ \rangle$ or by [ ]. In the first case the wagon is called an *applicator*,

in the second case an *abstractor*. A train is a (possibly empty) sequence of wagons.

In Backus-Naur form (with the empty string being denoted by <empty>) the syntax can be given by

<terminal> ::= $\tau|x|y|\cdots|\eta|\theta|\cdots$
<term> ::= <train><terminal>
<train> ::= <empty> | <wagon><train>
<applicator> ::= $\langle$<term>$\rangle$
<abstractor> ::= [<term>]
<wagon> ::= <applicator> | <abstractor>

The following is an example of a term:

$$[[\tau]_\alpha[\tau]_\eta\tau]_\beta\langle[\tau]_\delta[\tau]_\varepsilon\langle\delta\rangle\langle\varepsilon\rangle p\rangle[[\tau]_\zeta[\tau]_\mu\tau]_\eta[[\tau]_\sigma[\tau]_\varphi[\langle\varphi\rangle\langle\sigma\rangle p]_\xi\langle\varphi\rangle\langle\sigma\rangle q]_\vartheta[\tau]_\psi[\tau]_\eta[\langle\eta\rangle\langle\psi\rangle p]_\theta\langle\theta\rangle\langle\eta\rangle\langle\psi\rangle s.$$

3.2. Positioned subterms. We have to be careful about the word "subterm" since there are two different possible meanings. In order to stress the difference, we shall denote these notions by "subterm" and "positioned subterm", respectively. It is the same situation with the idea of a substring of a string. We consider that first.

Let us consider words as strings of letters. The two-letter string "in" is a substring of the string "invincible". But there are two positioned substrings in "invincible" which show the string "in", one consisting of the first and second letter, the other one of the fourth and fifth (we do not count the first and fifth as a substring). A *positioned substring* of a string of length L can be described as a pair of integers p, q with $1 \leq p \leq q \leq L$. The *content* of that positioned substring is the string we get from the string of length L by cancelling the first $p-1$ as well as the last $L-q$ letters.

A term t_1 is called a *subterm* of a term t_2 if t_1 is one of the terms used in building up t_2 by means of the syntax of section 3.1.

A subterm t_2 of t_1 can occur more than once in the production of t_1. Each one of these occurrences gives rise to a positioned substring of t_1 of which the content is equal to t_2. Such a positioned substring can now be called a *positioned subterm* of t_1.

As an example we mention the term $[\tau]_a[[\tau]_a\tau]_b[\langle a\rangle b]_v\langle\langle a\rangle b\rangle b$ which has two different positioned subterms which both have the content $\langle a\rangle b$.

3.3. Structure of trains. According to section 3.1 a train is a (possibly empty) sequence of applicators and abstractors. In this subsection we are interested only in the question: To which one of the two categories do entries of the sequence belong. Let us replace each applicator by the letter A and each abstractor by the letter T. Now the train is reduced to a word of A's and T's, possibly empty. Such a word will be called an *AT-word*.

We introduce the notion *coupled word* by the following grammar:

<coupled word>::= <empty> |A <coupled word> T|
<coupled word><coupled word>

There is exactly one way to group the A's and T's in a coupled word into *AT-couples*, such that in each couple the A comes before the T and such that any two couples have the property that one is entirely inside the other. Actually we get the couples from the above grammar if we always couple the A and T in A <coupled word> T. The situation is well-known, of course, from the coupling of parentheses in strings like ((()())((()()))).

In the following example the A's and T's are provided with subscripts in order to let us indicate the couples:

$$A_1A_2A_3T_4A_5T_6T_7A_8A_9A_{10}A_{11}T_{12}A_{13}T_{14}T_{15}T_{16}T_{17}T_{18}.$$

The couples are:

$$A_3T_4,\ A_5T_6,\ A_2T_7,\ A_{11}T_{12},\ A_{13}T_{14},\ A_{10}T_{15},\ A_9T_{16},\ A_8T_{17},\ A_1T_{18}.$$

3.4. Canonical dissection of an AT-word.

Theorem. For every AT-word w there are non-negative integers p and q such that

$$w = c_0Tc_1Tc_2\cdots Tc_{p-1}Tc_pAc_{p+1}Ac_{p+2}\cdots Ac_{p+q-1}Ac_{p+q} \tag{3.4.1}$$

where $c_0,\ldots,c_{p+q}$ are coupled words. (If $p=0$, the part $c_0Tc_1Tc_2\cdots Tc_{p-1}Tc_p$ has to be read as c_0, and if $q=0$, the part $Ac_{p+1}Ac_{p+2}\cdots Ac_{p+q-1}Ac_{p+q}$ has to be read as the empty string.)

The numbers p and q and the coupled words $c_0,\ldots,c_{p+q}$ are uniquely determined.

Proof. By induction, starting from the case that w is the empty word. The induction step is the addition of an A or T at the end.

This induction also provides an algorithm for obtaining (3.4.1).

The p T's mentioned in (3.4.1) can be called *bachelor* T's, and the q A's *waiting* A's. During the execution of the algorithm the bachelor T's stay bachelors, but some of the waiting A's can sometimes be coupled to later T's on the right.

In the following examples the dissection is made visible by underlining the bachelor T's and the waiting A's:

w_1=$AATTTTAATTATTAAATATAATTTATAATAATTAA$
w_2=$AATTATAAATA$

3.5. Canonical representation of a train. Since a train is an AT-word where the A's are replaced by applicators and the T's by abstractors, the canonical representation of a train follows at once from section 3.4. Accordingly, we shall use the terminology of couples, consisting of an applicator and an abstractor; and we shall use the phrase *AT-couple*. Moreover we have *bachelor abstractors* and *waiting applicators*.

From a train we make two new trains: the knowledge frame and the waiting list. The *knowledge frame* of a train w is the train w_1 we get by omitting all waiting applicators, and the *waiting list* of w is the train w_2 we get by removing all wagons belonging to the knowledge frame, so that we only keep the waiting applicators (in their original order). In all this, the word "wagon" refers to wagons of the train w and not to those of trains that possibly lie *inside* wagons of w.

An example is given by

$$\begin{aligned} w &= [\tau]_u[\tau]_v\langle u\rangle\langle v\rangle[[\tau]_r u]_p\langle w\rangle[\tau]_q\langle\langle u\rangle p\rangle, \\ w_1 &= [\tau]_u[\tau]_v\langle v\rangle[[\tau]_r u]_p\langle w\rangle[\tau]_q, \\ w_2 &= \langle u\rangle\langle\langle u\rangle p\rangle. \end{aligned}$$

Note that both the waiting list of a knowledge frame and the knowledge frame of a waiting list are empty.

3.6. Trimming a term with respect to a positioned terminal. Consider a term t and one of its positioned terminals. We can *truncate* that term t by omitting everything to the right of that positioned terminal. As an example we may take the seventh τ from the left in the term displayed in section 3.1. The truncation is

$$[[\tau]_\alpha[\tau]_\eta\tau]_p\langle[\tau]_\delta[\tau]_\varepsilon\langle\delta\rangle\langle\varepsilon\rangle p\rangle \underline{[}\ [\tau]_\zeta \underline{[}\ \tau.$$

The underlined opening brackets are those which do not have a matching closing bracket in the truncation.

Unless the positioned terminal is at the far end of the term, the truncation will not be a term. It looks like this

$$w_1 \sqsubset w_2 \sqsubset w_3 \sqsubset \cdots \sqsubset w_m q \tag{3.6.1}$$

where $w_1, w_2, \ldots, w_m$ are trains, q is a terminal, and every $\sqsubset$ is either $[$ or $\langle$. This representation (3.6.1) is unique.

For every w_i $(1 \le i \le m)$ we form the knowledge frame w_{i1} and the waiting list w_{i2} according to section 3.5.

We now define the *trimmed term*. It is the term obtained from the truncation (3.6.1) by omitting the $\sqsubset$'s and removing the waiting list from all w_i with $i < m$ but not from the last one. So the trimmed term is

$$w_{11}w_{21}\cdots w_{m-1,1}w_m q. \tag{3.6.2}$$

Note that the train $w_{11}w_{21}\cdots w_{m-1,1}w_m$ has $w_{11}w_{21}\cdots w_{m1}$ as its knowledge frame and w_{m2} as its waiting list.

Here is an example of a term t with a positioned terminal underlined:

$$\langle\tau\rangle[[\tau]_\zeta\tau]_\alpha\langle\alpha\rangle\langle[\alpha]_\eta\langle\eta\rangle[[\tau]_\delta\langle\alpha\rangle\underline{\eta}]_\varepsilon\rho\rangle\alpha.$$

The truncation is (with unmatched opening delimiters underlined)

$$\langle\tau\rangle[[\tau]_\zeta\tau]_\alpha\langle\alpha\rangle \underline{\rangle}\ [\alpha]_\eta\langle\eta\rangle \underline{[}\ [\tau]_\delta\langle\alpha\rangle\eta,$$

and the trimmed term is

$$\langle\tau\rangle[[\tau]_\zeta\tau]_\alpha[\alpha]_\eta\ [\tau]_\delta\langle\alpha\rangle\eta.$$

3.7. Proper terms. Let t be a term. It has the form wq where w is a train and q a terminal. The train w is a string of wagons $v_1\cdots v_k$. In the following cases we say that t is *proper at the far end*:

(i) if q stands for τ;

(ii) if q is a variable, θ, say, and if *exactly one* of $v_1,\ldots,v_k$ is an abstractor ending in $]_\theta$.

In the latter case we say that the terminal θ *refers to*, or *is bound by*, that last abstractor $[\ ]_\theta$. (Note that this is to be considered as a positioned abstractor.)

Next we look at terminals which are not necessarily in the last position of a term. If we have any positioned terminal in a term t, we can build the trimmed term (3.6.2), and we can investigate whether that trimmed term is proper at the far end. If that is the case for every terminal in t we say that t is *proper*.

There are no such things as free variables in a proper term: every positioned variable is bound by a uniquely determined positioned abstractor. Note that the positioned abstractors in the trimmed term are directly derived from positioned abstractors in t itself: in passing from (3.6.1) to (3.6.2) we did not omit any abstractors.

Rule (ii) allows many cases where abstractors in different positions have the same subscript at their closing bracket. As an example we mention that rule (ii) is still obeyed if we rewrite the term presented in section 3.1 as

$$[[\tau]_\delta[\tau]_\varepsilon\tau]_p\langle[\tau]_\delta[\tau]_\varepsilon\langle\delta\rangle\langle\varepsilon\rangle p\rangle[\ [\tau]_\delta[\tau]_\varepsilon\tau]_q[\ [\tau]_\delta[\tau]_\varepsilon[\langle\varepsilon\rangle\langle\delta\rangle p]_\theta\langle\varepsilon\rangle\langle\delta\rangle q]_s[\tau]_\delta[\tau]_\varepsilon[\langle\varepsilon\rangle\langle\delta\rangle p]_\theta\langle\theta\rangle\langle\varepsilon\rangle\langle\delta\rangle s.$$

This does not alter the references from positioned variables to positioned abstractors.

4 Reductions and equivalence

4.1. α-equivalence. Of course we want to consider names of variables to be irrelevant. The names serve to provide references from positioned terminals to closing brackets (]'s). If we replace the variables by dots, draw arrows from those dots to the respective]'s, and then omit the subscripts of the]'s, we get what we can call the *arrow representation* of the term. Two terms with the same arrow representation are called *α-equivalent*.

Here is an example of three α-equivalent terms:

$$[\tau]_\mu[[\mu]_\gamma\gamma]_\rho\mu,\quad [\tau]_\gamma[[\gamma]_\mu\mu]_\rho\gamma,\quad [\tau]_\mu[[\mu]_\rho\rho]_\rho\mu.$$

We take the point of view that the arrow representations are the essential things and that the names of variables are not.

Accordingly, two terms with the same arrow representation can be considered as equal, or, in other words, α-equivalence is taken as equality. So in particular, the term at the end of section 3.7 is the same as the one at the end of section 3.1.

Nevertheless we shall keep talking in terms of names of variables in this paper, since it makes the discussion on the metalevel somewhat easier. In implementations, however, other ways to describe the arrows may be more efficient.

4.2. Local reduction. Local reduction is a refinement of what Nederpelt called single-step β_1-reduction in [6], Ch. II, Definition 6.5.

Let t be a proper term, and q one of its positioned terminals. If q stands for τ there will be no reduction with respect to that terminal. If q is a variable, θ, say, we consider the abstractor ending in $]_\theta$ as mentioned in section 3.7. If this abstractor is a bachelor (section 3.5) there is no question of local reduction. But if it is not a bachelor, it is coupled to an applicator preceding it. Let that applicator be $\langle t_0 \rangle$. We now define the *local reduction* with respect to our positioned terminal. It leads to the term t_1 that we get by replacing our positioned terminal by t_0, provided that we take precautions to avoid clash of variables. It is certainly sufficient to replace all the *internal variables* of t_0 by entirely new ones. (Internal variables of t_0 are variables inside t_0 that refer to abstractors which are also inside t_0.)

Here is an example:

$$[\tau]_\rho[\underline{\langle t_0 \rangle}\langle t_1 \rangle[t_2]_\mu\underline{[t_3]_\theta}\langle[t_4]_\gamma\underline{\theta}\rangle\theta]_\gamma\tau. \tag{4.2.1}$$

The $\underline{\theta}$ indicates the positioned terminal we apply our reduction to. The abstractor to which this θ refers and the applicator coupled to it are also underlined. And $t_0, \ldots, t_4$ stand for terms. By local reduction (4.2.1) turns into

$$[\tau]_\rho[\langle t_0 \rangle\langle t_1 \rangle[t_2]_\mu[t_3]_\theta\langle[t_4]_\gamma t_0^*\rangle\theta]_\gamma\tau. \tag{4.2.2}$$

The t_0^* is obtained from t_0 by refreshing all internal variables. Note that γ might have been an internal variable of t_0, so without refreshing the variables of t_0 we might violate rule (ii) of section 3.7. The external variables in t_0 should *not* be refreshed. Note that possible ρ's in t_0 refer to the abstractor $[\tau]_\rho$ in front of (4.2.1) and that its copies in t_0^* have to refer to that same abstractor.

It is only our positioned terminal that is replaced by t_0, and not the possible further occurrences of that variable θ. That is why the reduction is called local. And note that, in contrast to what we have with ordinary β-reduction, the couple consisting of the applicator $\langle t_0 \rangle$ and the abstractor ending in $]_\theta$ are not necessarily adjacent. Moreover, that couple is not removed by the reduction, not even if there had been only one reference to that $]_\theta$.

It is easy to show that local reductions transform proper terms into proper terms.

4.3. AT-removal. AT-removal is essentially the same thing as what Nederpelt called single-step β_2-reduction in [6], Ch. II, Definition 6.7.

Let t be a proper term, and t_1 one of its subterms. We put $t_1 = wq$, where w is a train and q a terminal. The train w is a sequence of positioned wagons. Let two of them be $\langle t_2\rangle$ and $[t_3]_\theta$, and assume that they form a couple. And assume that t_1 contains no references to this positioned abstractor $[t_3]_\theta$. Under these circumstances AT-removal can be carried out. It just consists of removing the positioned applicator $\langle t_2\rangle$ as well as the positioned abstractor $[t_3]_\theta$.

As an example we take

$$t = [\tau]_\rho[\langle t_2\rangle\langle t_3\rangle[t_4]_\mu[t_5]_\theta\langle[t_6]_\delta\mu\rangle\mu]_\gamma\tau.$$

In the subterm

$$t_1 = \underline{\langle t_2\rangle}\langle t_3\rangle[t_4]_\mu\underline{[t_5]_\theta}\langle[t_6]_\delta\mu\rangle\mu$$

the underlined couple is a candidate for AT-removal unless the term t_6 contains any references to θ. If that is not the case, AT-removal transforms t into

$$[\tau]_\rho[\langle t_3\rangle[t_4]_\mu\langle[t_6]_\delta\mu\rangle\mu]_\gamma\tau.$$

It is trivial that AT-removals transform proper terms into proper terms.

4.4. β-Equivalence. As in [2], we shall use the word *minireduction*: a minireduction is either a local reduction or an AT-removal.

If a term t_2 is obtained from the term t_1 by some minireduction, we write $t_1 >_\beta t_2$.

This $>_\beta$ is a relation between proper terms. Its symmetric, reflexive, transitive closure is called *β-equivalence*, and denoted by $=_\beta$. So two proper terms are β-equivalent if and only if they are the first and last element of a finite chain of proper terms $t_0, t_1, \ldots, t_m$ such that for all i with $0 \leq i < m$ at least one of the relations $t_i >_\beta t_{i+1}$, $t_{i+1} >_\beta t_i$ holds.

The notation $t_1 >_\beta t_2$ for minireductions is the same as the usual one for standard β-reduction. Nevertheless it is a convenient notation since the equivalence generated by it is the usual β-equivalence (cf. section 1.3 (iv)).

5 Type and final type of a term

5.1. Type of a proper term. Let t be a proper term (see section 3.7). Representing it as a train w followed by a terminal q we have

$$t = wq = v_1 \cdots v_k q$$

where $v_1, \ldots, v_k$ are wagons.

If q is τ, we shall say that t is a τ-term. Unless t is a τ-term, we shall define a new term to be called the *type* of t.

Let now q be the variable ρ. According to our assumption there is exactly one index i (with $1 \leq i \leq k$) such that

$$v_i \;=\; [t_1]_\rho$$

with some term t_1. We now define the type of t, denoted $\mathrm{typ}(t)$, by

$$\mathrm{typ}(t) \;=\; v_1 \cdots v_k t_1^*, \tag{5.1.1}$$

where t_1^* is obtained from t_1 by refreshing all internal variables (the reason for this is the same as in (4.2.2)).

As an example we present

$$\begin{array}{ll} t & = [\tau]_z[[z]_x x]_u[\tau]_x u, \\ \mathrm{typ}(t) & = [\tau]_z[[z]_x x]_u[\tau]_x[z]_y y. \end{array}$$

Note that in $[z]_x x$ the x is internal and z external.

5.2. Degree and final type of a term. In a proper term t we shall define the *degree* of every positioned terminal. We inspect the terminals in t one by one, running through t from left to right. If a terminal is τ, it gets degree 1. If it is a variable, x say, then it refers to some $]_x$ somewhere to the left of that terminal x. That $]_x$ is the closing bracket of an abstractor $[wq]_x$, where w is a train and q a terminal. The degree of q was defined before, so we can define the degree $\mathrm{degree}(x)$ of x by $\mathrm{degree}(x) = \mathrm{degree}(q)$.

We also define the degree $\mathrm{degree}(t)$ for any proper term t: it is just $\mathrm{degree}(q)$, where q is the terminal on the far right (so $t = wq$, where w is a train).

The degree of a proper term equals 1 if and only if t is a τ-term. It is easy to show that for all other proper terms we have $\mathrm{degree}(t) = \mathrm{degree}(\mathrm{typ}(t)) + 1$. So in the sequence

$$t,\ \mathrm{typ}(t),\ \mathrm{typ}(\mathrm{typ}(t)), \ldots$$

we sooner or later reach a τ-term, and then no further applications of typ are possible. This τ-term will be called the *final type* of t, and will be denoted by $\mathrm{ftyp}(t)$. So if $d = \mathrm{degree}(t)$ we have

$$\mathrm{ftyp}(t) \;=\; \mathrm{typ}^{d-1}(t),$$

where, of course, $\mathrm{typ}^2(t) = \mathrm{typ}(\mathrm{typ}(t))$, etc.

As an example we present a case with $\mathrm{degree}(t) = 4$

$$\begin{array}{ll} t & = [\tau]_z[\langle z\rangle[\langle\tau\rangle z]_y y]_x x, \\ \mathrm{typ}(t) & = [\tau]_z[\langle z\rangle[\langle\tau\rangle z]_y y]_x\langle z\rangle[\langle\tau\rangle z]_u u, \\ \mathrm{typ}(\mathrm{typ}(t)) & = [\tau]_z[\langle z\rangle[\langle\tau\rangle z]_y y]_x\langle z\rangle[\langle\tau\rangle z]_u\langle\tau\rangle z, \\ \mathrm{ftyp}(t) & = [\tau]_z[\langle z\rangle[\langle\tau\rangle z]_y y]_x\langle z\rangle[\langle\tau\rangle z]_u\langle\tau\rangle\tau. \end{array}$$

5.3. The typing relation. If t_1 and t_2 are proper terms, and if t_2 is β-equivalent to $\mathrm{typ}(t_1)$ then this is expressed in the metalanguage by the notation $t_1 : t_2$.

5.4. Well-typedness of a train. Let w be a train and let its last wagon be an abstractor $[t_1]_x$. Assume that it is not a bachelor, whence it forms a couple with an applicator $\langle t_2\rangle$ which is a wagon of the train too. We shall formulate a condition of well-typedness of that couple with respect to the train w.

Let w_1 be the knowledge frame of w. This w_1 is obtained by removing all waiting applicators from w, so it still contains the couple. Let w_2 be the train we get from w_1 by removing the couple.

We consider the following two terms t_1^* and t_2^*. We obtain t_1^* by attaching t_1 to w_2, so $t_1^* = w_2 t_1$. And similarly $t_2^* = w_2 t_2$. The well-typedness condition for the couple can now be phrased by saying that t_2^* is not a τ-term and that

$$\mathrm{typ}(t_2^*) =_\beta t_1^*. \tag{5.4.1}$$

Next we formulate the condition of well-typedness for the whole train. For every wagon of the train which is a non-bachelor abstractor we form the truncation obtained by omitting everything to the right of it. We take condition (5.4.1), expressed for that abstractor (and the applicator with which it forms a couple) with respect to that truncated train. If that condition holds for all such wagons we say that the train is well-typed.

Note that this condition only refers to the wagons of the train itself and not to wagons of trains *inside* those wagons.

As an example we take the train

$$w = [\tau]_\xi[\xi]_x[[\tau]_\rho\tau]_\eta\langle\eta\rangle\langle\langle\xi\rangle\eta\rangle\langle x\rangle[\xi]_\gamma\langle[\tau]_\varepsilon\gamma\rangle[[\tau]_\theta\xi]_y[\tau]_\delta.$$

Well-typedness of this w means that the following three conditions are simultaneously satisfied:

$$\begin{aligned}
&\mathrm{typ}\ ([\tau]_\xi[\xi]_x[[\tau]_\rho\tau]_\eta x) =_\beta [\tau]_\xi[\xi]_x[[\tau]_\rho\tau]_\eta\xi,\\
&\mathrm{typ}\ ([\tau]_\xi[\xi]_x[[\tau]_\rho\tau]_\eta\langle x\rangle[\xi]_\gamma[\tau]_\varepsilon\gamma) =_\beta [\tau]_\xi[\xi]_x[[\tau]_\rho\tau]_\eta\langle x\rangle[\xi]_\gamma[\tau]_\theta\xi,\\
&\mathrm{typ}\ ([\tau]_\xi[\xi]_x[[\tau]_\rho\tau]_\eta\langle x\rangle[\xi]_\gamma\langle[\tau]_\varepsilon\gamma\rangle[[\tau]_\theta\xi]_y\langle\xi\rangle\eta) =_\beta [\tau]_\xi[\xi]_x\\
&\qquad [[\tau]_\rho\tau]_\eta\langle x\rangle[\xi]_\gamma\langle[\tau]_\varepsilon\gamma\rangle[[\tau]_\theta\xi]_y\tau.
\end{aligned}$$

6 Semi-correctness and correctness

6.1. Semi-correctness at the far end. Let t be a proper term (see section 3.7). According to section 5.2 we can form the final type $\mathrm{ftyp}(t)$, which is a τ-term. It has the form $\mathrm{ftyp}(t) = w\tau$ where w is a train. This train has a knowledge frame and a waiting list (section 3.5). We say that the term t is *semi-correct at the far end* if that waiting list is empty.

As an example we mention the term t presented in section 5.2. Its final type $\mathrm{ftyp}(t)$ has a nonempty waiting list, consisting of just one wagon $\langle\tau\rangle$. Therefore t is not semi-correct at the far end.

6.2. Semi-correctness. Let t be a proper term (see section 3.7). We say that t is *semi-correct* if for every positioned terminal in t the trimmed term (see section 3.6) is semi-correct at the far end.

6.3. Correctness at the far end. Let t be a proper term. Just like in section 6.1 we take the final type $\mathrm{ftyp}(t)$ and write $\mathrm{ftyp}(t) = w\tau$ where w is a train. We say that the term t is *correct at the far end* if the waiting list of w is empty and if moreover w is well-typed.

6.4. Correctness. Let t be proper term. We say that t is *correct* if for every positioned terminal in t the trimmed term (see section 3.6) is correct at the far end.

We leave it to the reader to check that the term presented at the end of section 3.1 is correct.

7 Algorithm for checking correctness

7.1. Making a task list. Checking correctness of a term t according to the definition of section 6.4 is at once reduced to checking well-typedness of certain trains, and this means that we have a number of *tasks* of the form (5.4.1). Each one of those tasks concerns two terms of which β-equivalence has to be shown. The tasks are independent in the sense that the complete list of tasks can be prepared before we start to carry any of them out.

We can prepare that list by following the definition of section 6.4, investigating all positioned terminals one by one from left to right. That algorithm will produce some duplication in the sense that we get a kind of repetition on the list: some tasks are trivial extensions of tasks we had before. Such repetitions are caused by the fact that we have to copy pieces of terms when evaluating the final types of section 5.

The repetitions are of the following kind. The list already contained the task $wt_1 =_\beta wt_2$, and later we get the task $w^*t_1 =_\beta w^*t_2$ with the same t_1 and t_2, where the knowledge frame w^* is an extension of the knowledge frame w. It then follows that the truth of $wt_1 =_\beta wt_2$ has the truth of $w^*t_1 =_\beta w^*t_2$ as a trivial consequence. Therefore the latter task can be omitted from the list. Every task on the list is connected with a particular couple formed by an applicator and an abstractor (see section 5.4). That applicator ends with $\rangle$. Let us replace it by $\rangle^*$ directly after putting the corresponding task on the list. So at that moment a particular positioned $\rangle$ in the original term t (the term

we want to make the task list for) is replaced by)*. We keep that asterisk wherever we are making copies. This has the effect that some later task will be about some couple where the applicator ends in)*. It can be shown that in those cases the task is a trivial consequence of the old one. So we need not put it on the list.

This way we keep exactly one task corresponding to each) occurring in t. But it should be mentioned that in the algorithm they do not necessarily turn up in the same order as in the term t.

7.2. Carrying out the tasks. Building the task list is done practically in linear time. It is almost proportional to the length of the term. This is not entirely true, because of the fact that investigating the final types might multiply the work by a factor equal to the degree, and in our definition of $\Delta\Lambda$ we have not required the degrees to be bounded. But in applications to Automath-like books the degrees are bounded (in Automath the degrees are 1, 2 or 3). Another reason why we cannot have pure linearity is that handling very big terms will require longer addresses and longer knowledge frames.

But in practice all this trouble is small in comparison to what has to be done in *carrying out* the tasks. That requires β-equivalences to be checked. The amount of work it involves is hardly predictable in general. It can be almost hopeless in cases where the terms are *not* β-equivalent. In such cases, the fact that the question of β-equivalence is decidable in $\Delta\Lambda$ (see section 8.5) is of little practical use. On the other hand, the practical experience in verifying extensive Automath books has been definitely encouraging. If mathematics is written in such a way that human mathematicians can grasp it, the search for establishing β-equivalence will never have to go very deep.

8 Remarks on language theory

8.1. General remark. For our $\Delta\Lambda$ we can get theoretical results in the style of what was achieved for Λ (see [6] and [5]) and for Automath (see [5]). This is no surprise, since the features of $\Delta\Lambda$ which are not in Λ are essentially features of Automath.

We shall list a number of basic properties, without going into proofs. In some of the proofs the algorithmic nature of our definition of $\Delta\Lambda$ will be profitable.

8.2. Church-Rosser theorem. Let us write $t_0 >>_\beta t_k$ if there is a chain $t_0, \ldots, t_k$ such that $t_i >_\beta t_{i+1}$ for $0 \leq i < k$ (see section 4.4 for $>_\beta$). This includes the case $k = 0$, so we also write $t >>_\beta t$.

We can now phrase the Church-Rosser theorem. If t is a proper term, and if both $t >>_\beta t^*$ and $t >>_\beta t^{**}$, then there is a term t^{***} such that both $t^* >>_\beta t^{***}$ and $t^{**} >>_\beta t^{***}$.

It is easy to derive this from the standard Church-Rosser theorem. The Church-Rosser property for full β-reduction with couples was proved by Ne-

derpelt in [6], Ch. II, Theorem 6.43 (note that in that theorem and in its proof, the β should have been $\beta_1 + \beta_2$). From that result we get to the present situation by remarking that any minireduction $t >_\beta t^*$ uses a particular couple, and that the full β-reduction with respect to that couple can be obtained by successive application of all minireductions connected with that couple, starting with that particular step $t >_\beta t^*$.

8.3. The Nederpelt norm. For every proper term t we can define the Nederpelt norm (see [6] or [2]), to be denoted by norm(t). That norm is what we shall call a *norm-shaped* term: a term without variables and without ⟨'s or ⟩'s. It just consists of τ's, [′s and]′s (since there are no variables we can afford to omit the subscripts of the]′s).

One may think of the Nederpelt norm as the result of a "catch as catch can" process. We start with a term t, and wherever we see a positioned variable, we may replace it by its type. And wherever there is a case for AT-removal, we may do it. These operations are repeated in any order until finally all terminals are τ and no applicators are left.

But instead of building a definition on the basis of this intuitive idea, we shall present an algorithmic definition of the Nederpelt norm.

In order to produce norm(t) we read t from left to right. In most cases we do not take any action at all. If we read a τ or a [we do nothing. If we read a ⟨ we read on and do nothing until we have read the matching ⟩.

If we read a positioned variable, θ, say, we locate the $]_\theta$ it refers to, as well as the [that matches it. From this opening bracket until that $]_\theta$ we read a thing like $[t_1]_\theta$ where t_1 is a norm-shaped term (note that this lies to the left of our positioned variable, whence it has been processed already). We now replace our positioned variable by this t_1. From there we read on. That is, we do not read the t_1 we have just put there but we begin to read the part of the original t *after* the positioned variable we just replaced by t_1.

If we read a subscripted closing bracket, $]_\eta$, say, we locate the matching [. Between this [and the $]_\eta$ we have a term of the form $w\tau$, where w is a train. From this train we remove all waiting applicators and all couples, so that we only keep the bachelor abstractors. The insides of the remaining bachelors are norm-shaped, since they have been processed already. From the closing brackets of these abstractors we now remove the subscripts (note that nothing can refer to them any more).

After having read the whole term, there is nothing more to read, but we once more take the same action in the case of a $]_\eta$. The result of our processing thus far is of the form $w\tau$, so from w we skip all waiting applicators, all couples and all subscripts of]′s.

As an example we show the result of this algorithm when applied to the term of section 3.1: $[[\tau][\tau]\tau][[\tau][\tau][\tau]\tau][\tau][\tau][\tau]\tau$.

It is easy to see that we always have $\mathrm{norm}(\mathrm{typ}(t)) = \mathrm{norm}(t)$.

8.4. Norm-correctness. We get the definition of norm-correctness by a modification of the one of correctness (section 6.4). We still require semi-correctness, but condition (5.4.1) is weakened to

$$\mathrm{norm}(t_2^*) = \mathrm{norm}(t_1^*). \tag{8.4.1}$$

Norm-correctness is easy to check, since the norms can be evaluated along with the production of the task list. And (8.4.1) is just a matter of strict equality, not of β-equivalence.

If t is norm-correct, then $\mathrm{typ}(t)$ is norm-correct too, and $\mathrm{norm}(\mathrm{typ}(t)) = \mathrm{norm}(t)$.

If t is norm-correct and if $t >_\beta t_1$, then t_1 is norm-correct too, and $\mathrm{norm}(t_1) = \mathrm{norm}(t)$.

8.5. Strong normalization. In [6] Nederpelt showed the *strong normalization theorem* for the system Λ: For every norm-correct term t there exists a positive number N such that for all reduction chains

$$t = t_1 >_\beta t_2 >_\beta \cdots >_\beta t_k$$

we have $k \leq N$.

In [3] Nederpelt's method was carried out for $\Delta\Lambda$, with the same reductions (local couple reductions and AT-removals) as in the present paper.

The strong normalization theorem and the Church-Rosser theorem together imply that β-equivalence in $\Delta\Lambda$ is *decidable*. This is a theoretical result only, since in [3] the bounds (in terms of the length of the given term) are of the order of the diagonal of the Ackermann function.

8.6. Theorems on correct terms.

(i) If t is a correct term with $\mathrm{degree}(t) > 1$, then $\mathrm{typ}(t)$ is correct too.

(ii) If t is correct, $\mathrm{degree}(t) > 1$, and $t >_\beta t_1$, then $\mathrm{typ}(t) =_\beta \mathrm{typ}(t_1)$.

(iii) If t is correct and $t >_\beta t_1$, then t_1 is correct too.

Proofs for (i) and (ii) are simple. We can derive (iii) from the statement that if t is correct, $\mathrm{degree}(t) = j$ and $t >_\beta t_1$, then t_1 is correct and $\mathrm{ftyp}(t) =_\beta \mathrm{ftyp}(t_1)$. This can be proved by induction with respect to j. A crucial step in the induction proof is as follows. Let

$$t = \cdots \langle P \rangle [Q]_x \cdots \langle A_1 \rangle \cdots \langle A_h \rangle \cdots x,$$

where $\langle A_1 \rangle \cdots \langle A_h \rangle$ is the waiting list for the final terminal x. If it is given that t is correct, we know that this waiting list satisfies the well-typedness condition with respect to the first h bachelor abstractors in $\mathrm{ftyp}(Q)$. The question is whether it satisfies the same condition with respect to $\mathrm{ftyp}(P)$. From the

correctness of t we know that $\mathrm{typ}(P) =_\beta Q$. Since the degree of Q is less than the one of x we already know that $\mathrm{ftyp}(P)) = \mathrm{ftyp}(\mathrm{typ}(P)) =_\beta \mathrm{ftyp}(Q)$. We can now apply the fact that if two proper terms s_1 and s_2 are β-equivalent, then their sequences of batchelor abstractors are the same up to β-equivalence: if the j-th entries are $[B_j]$ and $[C_j]$ then $B_j =_\beta C_j$. It follows that the waiting list $\langle A_1 \rangle \cdots \langle A_h \rangle$ satisfies the well-typedness condition with respect to $\mathrm{ftyp}(P)$.

Using (ii), (iii) and the Church-Rosser theorem, we derive that if both t_1 and t_2 are correct, with degree > 1, and if $t_1 =_\beta t_2$, then $\mathrm{typ}(t_1) =_\beta \mathrm{typ}(t_2)$.

Correctness implies norm-correctness, and that implies strong normalization. So if both t_1 and t_2 are correct, it is decidable whether $t_1 =_\beta t_2$. In particular it is always decidable whether a given term is correct.

9 References

1. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Kon. Nederl. Akad. Wetensch. Proceedings Ser. A 75 (=Indagationes Math. 34) pp. 381–392 (1972).

2. N.G. de Bruijn. Generalizing Automath by means of a lambda-typed lambda calculus. In: *Mathematical Logic and Theoretical Computer Science*, Lecture Notes in Pure and Applied Mathematics, Vol. 106, (ed. D. W. Kueker, E.G.K. Lopez-Escobar, C.H. Smith) pp. 71–92. Marcel Dekker, New York 1987.

3. N.G. de Bruijn. Upper bounds for the length of normal forms and for the length of reduction sequences in lambda-typed lambda calculus. Report, Department of Mathematics and Computing Science, Eindhoven University of Technology.

4. N.G. de Bruijn. A plea for weaker frameworks. In: *Logical Frameworks* (ed. G. Huet and G. Plotkin), pp. 40–67. Cambridge University Press, Cambridge 1991.

5. D.T. van Daalen. The language theory of Automath. Ph.D. thesis, Eindhoven University of Technology, 1980.

6. R.P. Nederpelt. Strong normalization in a typed lambda calculus with lambda structured types. Ph.D. thesis, Eindhoven University of Technology, 1973.