Generalizing Automath by Means of a Lambda-Typed Lambda Calculus*

N.G. de Bruijn

SUMMARY

The calculus $\Delta\Lambda$ developed in this paper is claimed to be able to embrace all the essential aspects of Automath, apart from the feature of type inclusion, which will not be considered in this paper. The calculus deals with a correctness notion for lambda-typed lambda formulas (which are presented in the form of what will be called lambda trees). To an Automath book there corresponds a single lambda tree, and the correctness of the book is equivalent to the correctness of the tree. The algorithmic definition of correctness of lambda trees corresponds to an efficient checking algorithm for Automath books.

1. INTRODUCTION

1.1. Automath and lambda calculus

We are not going to explain Automath in this paper; for references and a few remarks we refer to Section 6.1. The basic common feature of the languages of the Automath family is lambda-typed lambda calculus. Nevertheless Automath has various aspects of a different nature, of which we mention the context administration and the mechanism of instantiation. Moreover there is the notion of degree, and the rules of the languages, in particular those regarding abstractors, are different for different degrees. But a large part of what can be said about Automath, in particular as far as language theory is concerned, can be said about the bare lambda-typed lambda calculus already.

In [*de Bruijn 71 (B.2)*] it was described how a complete Automath book can be considered as a single lambda calculus formula, and that idea gave rise

*Reprinted from: Kueker, D.W., Lopez-Escobar, E.G.K. and Smith, C.H., eds., *Mathematical Logic and Theoretical Computer Science*, p. 71–92, by courtesy of Marcel Dekker Inc., New York.

to work on language theory ([*Nederpelt 73 (C.3)*], [*van Daalen 80*]) about the lambda-typed lambda calculus system called Λ. This system of condensation of an Automath book into a single formula (AUT-SL: single line Automath book) had a disadvantage, however. In order to put the book into the lambda calculus framework it was necessary to first eliminate all definitional lines of the book. Considering the fact that the description of a mathematical subject may involve a large number of definitions, the exponential growth in length we get by eliminating them is prohibitive in practice: it can serve a theoretical purpose only.

The kind of lambda-typed lambda calculus to be developed in the present paper may be better in this respect. It makes it possible to keep the full abbreviational power of Automath books within the framework of a lambda calculus. In this framework a number of features of Automath can be explained in a unifying way. Lines, contexts and instantiations all vanish from the scene. They find their natural expression in the lambda calculus, like in AUT-SL, but now without loosing the relation with the original Automath book. In particular the way we actually check the correctness of an Automath book is directly related to an efficient way to check the correctness of a lambda formula.

Therefore the checking algorithm described in this paper can be expected to become a basis of all checkers of Automath-like languages. The little differences between the various members of the Automath family lead to rather superficial modifications of that basic program. It can be expected that most of these modifications will be felt at the input stage only.

The paper is restricted to the Automath languages without type inclusion. The feature of type inclusion (which is used in AUT-68 and AUT-QE) requires modifications in the correctness definition and the checking algorithm. We shall not discuss such modifications here.

1.2. Trees

The paper has another feature, not strongly related to the main theme. That feature is the predominant place given to the description in terms of trees rather than to the one in terms of character strings. Of course, this may be considered as just a matter of taste. Nevertheless it may have an advantage to have a coherent description in terms of trees, in particular for future reference.

The author believes that if it ever comes to treating the theory of Automath in an Automath book, the trees may stand a better chance than the character strings.

2. LAMBDA TREES

2.1. What to take as fundamental, character strings or trees

Syntax is closely connected to trees. Formulas, and other syntactic structures, are given as strings of characters, but can be represented by means of trees. On the other hand, treeshaped structures can be coded in the form of strings of characters. One might say that the trees and the character strings are two faces of one and the same subject. The trees are usually closer to the nature of things, the character strings are usually better for communication. Or, to put it in the superficial form of a slogan, the trees are what we mean, the strings are what we say.

Discussing syntax we have to choose which one of the two points of view, trees of strings, is to be taken as the point of departure. Usually one seems to prefer the character strings, but we shall take the less traditional view to start from the trees. One can have various reasons for this preference, but here we mention the following two as relevant for the present paper:

(i) It seems to be easier to talk about the various points of a tree than about the various "places" in a character string.

(ii) The trees make it easier to discuss the matter of bound variables.

We shall use the character strings as a kind of shorthand in cases where the trees become inconvenient. This shorthand is quite often easier to write, to print and to read, but the reader should know all the time that the trees are the mathematical structures we really intend to describe. In Section 2.7 we shall display the shorthand rules.

2.2. The infinite binary tree

We start from a set with two elements, l and r (mnemonic for "left" and "right"). W is to be the set of all words over $\{l, r\}$, including the empty word ε; in standard notation $W = \{l, r\}^*$. This set W will be called the *infinite binary tree.*

We consider the mappings

$$\begin{aligned} \text{father} &: (W\backslash\{\varepsilon\}) \to W\ , \\ \text{leftson} &: W \to W\ , \\ \text{rightson} &: W \to W\ . \end{aligned}$$

The father of a word is obtained by omitting its rightmost letter, the leftson is obtained by adding an l on the right, the rightson is obtained by adding an r on the right.

Examples:

$$\text{father}(l) = \varepsilon\,, \quad \text{father}(r) = \varepsilon\,, \quad \text{father}(lrl) = lr\,,$$
$$\text{leftson}(\varepsilon) = l\,, \quad \text{leftson}(lrrl) = lrrll\,,$$
$$\text{rightson}(\varepsilon) = r\,, \quad \text{rightson}(rll) = rllr\,.$$

In these examples we have followed the usual sloppy way to write words as concatenated sequences of letters, and to make no distinction between a one-letter word and the letter it consists of.

We define the binary infix relation $<$ by agreeing that $u < v$ (with $u \in W$, $v \in W$) means that the word u is obtained from the word v by omitting one or more letters on the right. So $lrr < lrrrl$, and $\varepsilon < u$ for all $u \in W\backslash\{\varepsilon\}$. The relation is obviously transitive. As usual, $u \leq v$ means that either $u < v$ or $u = v$. And $v > u$ ($v \geq u$) will mean the same thing as $u < v$ ($u \leq v$).

2.3. Binary trees

We shall consider all binary trees to be finite subtrees of the infinite binary tree. A *binary tree* is a finite subset V of W with the following properties:

(i) $\varepsilon \in V$,

(ii) for all $u \in V$ with $u \neq \varepsilon$ we have $\text{father}(u) \in V$,

(iii) if $u \in V$ then $\text{leftson}(u) \in V$ if and only if $\text{rightson}(u) \in V$.

Elements of V are called *points* of the binary tree.

If $u \in V$ and $\text{leftson}(u) \notin V$, $\text{rightson}(u) \notin V$, then u is called an *end-point* of V. The set of all end-points is denoted V_e. The point ε is called the *root* of V.

There are two popular ways to draw two-dimensional pictures of a binary tree. The way we follow in this paper is to draw sons above their fathers. The other one has the fathers above their sons (such pictures can be called weeping willows). In both cases $\text{leftson}(u)$ is drawn to the left of $\text{rightson}(u)$, for all u. Readers who prefer to draw weeping willows instead of upright trees will not have any trouble, since for their benefit we shall avoid the use of terms like "up", "down", "above", "below" for describing vertical orientation. The inequality $<$ is neutral in this respect.

2.4. Labels

We consider three different objects outside W. They will be called A, T and τ. Elements of the set $W \cup \{A, T, \tau\}$ will be called *labels*. Points with label A or T will be called A-nodes and T-nodes, respectively.

If V is a binary tree then any mapping of V into the set of labels is called a *labeling* of V. If f is a labeling, and $u \in V$, then $f(u)$ is called *the label* of u.

2.5. Definition of lambda trees

A lambda tree is a pair (V, lab), where V is a binary tree, and lab is a labeling of V that satisfies the following conditions (i), (ii), (iii):

(i) If $u \in V \backslash V_e$ then $\mathrm{lab}(u) \in \{A, T\}$.

(ii) If $u \in V_e$ then $\mathrm{lab}(u) \in V \cup \{\tau\}$.

(iii) If $u \in V_e$ and $\mathrm{lab}(u) \in V$ then $\mathrm{lab}(\mathrm{lab}(u)) = T$ and $\mathrm{rightson}(\mathrm{lab}(u)) \leq u$.

2.6. An example

We give an example with 17 points. These points and their labels are specified as follows:

$$\mathrm{lab}(\varepsilon) = T,\ \mathrm{lab}(l) = \tau,\ \mathrm{lab}(r) = T,\ \mathrm{lab}(rl) = T,$$
$$\mathrm{lab}(rll) = \tau,\ \mathrm{lab}(rlr) = A,\ \mathrm{lab}(rlrl) = rl,$$
$$\mathrm{lab}(rlrr) = \varepsilon,\ \mathrm{lab}(rr) = A,\ \mathrm{lab}(rrl) = T,$$
$$\mathrm{lab}(rrll) = r,\ \mathrm{lab}(rrlr) = \tau,\ \mathrm{lab}(rrr) = T,$$
$$\mathrm{lab}(rrrl) = \tau,\ \mathrm{lab}(rrrr) = T,\ \mathrm{lab}(rrrrl) = rrr,$$
$$\mathrm{lab}(rrrrr) = r.$$

This lambda tree is pictured in Figure 1.

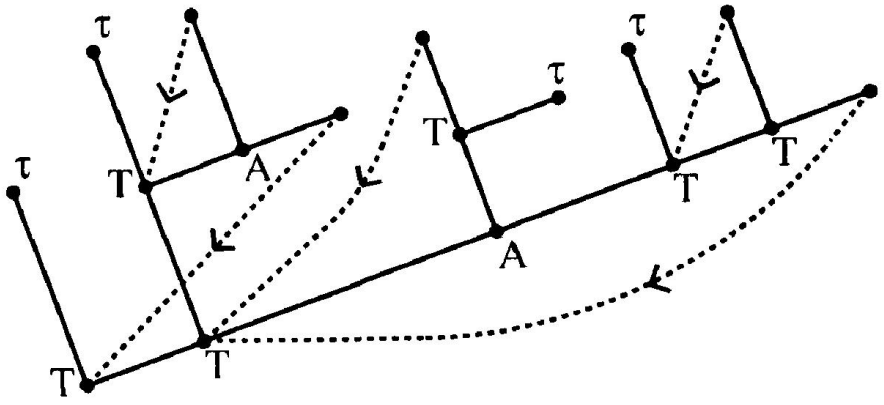

Figure 1, A lambda tree

The picture does not show the names of the points, but it does show their labels as far as they are A, T or τ. In the cases of points u with labels in V we have indicated $\mathrm{lab}(u)$ by means of a dotted arrow from u (which is always an

end-point, according to 2.5 (i)) to lab(u) (which is a point on the path from u to the root of the tree). Indeed the arrows always go to points with label T, and at such points the arrows always come from the right, according to 2.5 (iii).

2.7. Representation of a lambda tree as a character string

We begin by taking a set of identifiers to be called dummies. They are no elements of the set of labels. Next in some arbitrary way we attach a dummy to every point of the tree that has label T, and different points get different dummies. In the example of 2.6 we attach x_1 to ε, x_2 to r, x_3 to rl, x_4 to rrl, x_5 to rrr, x_6 to $rrrr$.

We can now also attach dummies to the end-points as far as their label is not τ. To the end-point u (with label lab(u) $\in V$) we attach the same dummy as we attached to lab(u). The point lab(u) with its dummy is called the *binding instance* of the dummy, the point u with its dummy is called a *bound instance*. In Figure 2 the dummies are shown. The arrows could be omitted since their information is provided by the dummies: the arrows run from the bound instances of a dummy to its binding instance.

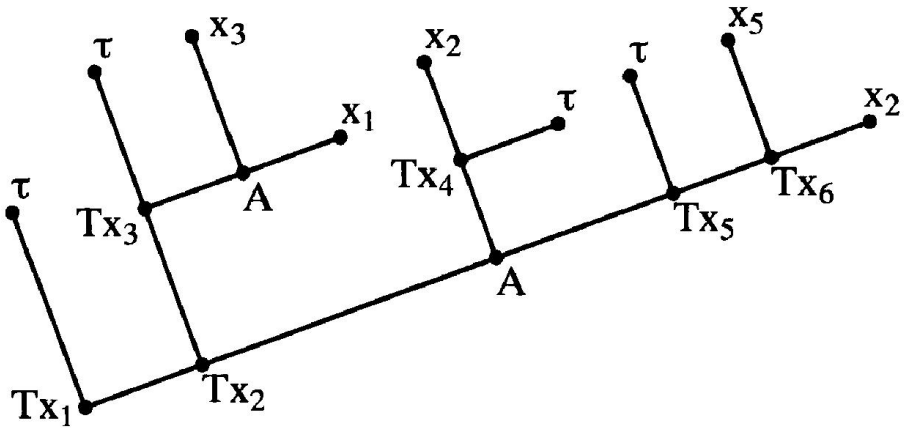

Figure 2, Tree with named dummies

We now produce the character string representation by the following algorithm that attaches character strings to all subtrees:

(i) A subtree consisting of a single point is represented by τ if its label is τ and by its dummy if it is a bound instance of that dummy.

(ii) A subtree whose root is labeled by A, to which there is attached a left-hand subtree (with character string P) and a right-hand subtree (with character string Q), gets the character string $\langle P \rangle Q$.

(iii) A subtree whose root is labeled by T, with dummy x_i, say, and with P and Q as under (ii), gets the character string $[x_i : P]\, Q$.

If we apply the algorithm to the tree of Figure 2 we get

$$[x_1 : \tau]\,[x_2 : [x_3 : \tau]\,\langle x_3\rangle\, x_1]\,\langle [x_4 : x_2]\,\tau\rangle\,[x_5 : \tau]\,[x_6 : x_5]\,x_2\ .$$

The way back from character string to lambda tree is easy, and we omit its description.

2.8. Remarks

The following remarks might give some background to our definitions and notations.

(i) The notation $[x : P]\,Q$ is the notation in Automath for the typed lambda abstraction. Here the binding dummy x is declared as being of type P. In untyped lambda calculus one might write $[x]\,Q$, but it is usually written with a lambda: $\lambda x.Q$ or $\lambda_x\,Q$.

(ii) In standard lambda calculus there is the construct called "application", usually written as a concatenation QP. The interpretation is that Q is a function, P a value of the argument, and that QP is the value of the function Q at the point P.

The Automath notation puts the argument in front of the function: it has $\langle P\rangle\,Q$ instead of QP. The decision to put the "applicator" $\langle P\rangle$ in front of the function Q, is in harmony with the convention to put abstractors (like the $[x : P]$ above) in front of what they operate on.

Older Automath publications had $\{P\}\,Q$ instead of $\langle P\rangle\,Q$.

(iii) The τ has about the same role that is played in Automath by '**type**' and '**prop**', the basic expressions of degree 1.

(iv) The labels A and T in the lambda tree are mnemonic for "application" and "typing".

(v) The typing nodes are at the same time lambda nodes. This is different from what we had in [*de Bruijn 72b (C.2)*], Section 13; there the lambda was a separate node in the right-hand subtree of the node with label T. Taking them together has the effect that the arrows in the lambda tree lead to nodes labeled T instead of λ, and that the provision has to be made that arrows leading to a T-node always arrive from the right (see 2.5 (iii)).

In the character string representation this provision means that in the case of $[x : P]\,Q$ the dummy x does not occur in P.

(vi) The tree of Figure 2 can be presented in namefree form by means of the reference depth system of [*de Bruijn 72b (C.2)*]. We explain it here: If

there is an arrow from an end-point u to $\mathrm{lab}(u)$ then the reference depth of u is the number of v with $\mathrm{lab}(v) = T$ and $\mathrm{lab}(u) \leq v < \mathrm{rightson}(v) \leq u$. We can replace the information contained in $\mathrm{lab}(u)$ by the reference depth of u. If that depth is 3, say, then we find $\mathrm{lab}(u)$ by proceeding from u to the root of the tree; the point we want is the third T-node we meet, provided that we only count T-nodes we approach from the right.

For the tree of Figure 1 this is carried out in Figure 3.

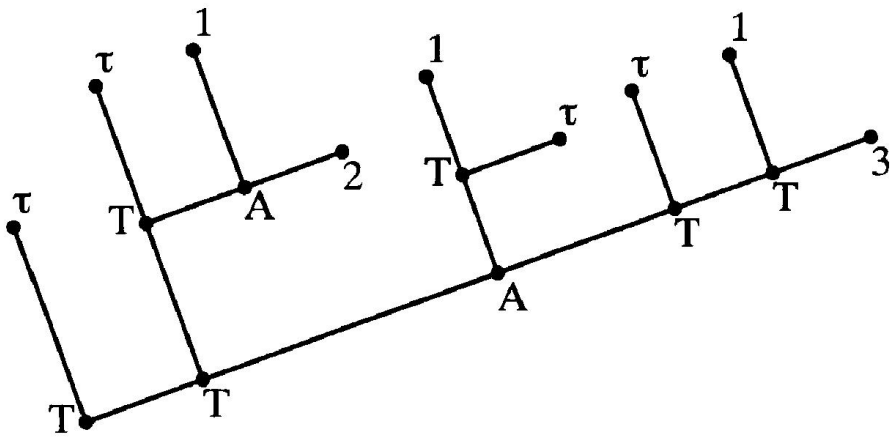

Figure 3, Tree with reference depths

Comparing Figure 3 to Figure 2 we note that the three ones in Figure 3 lead to three different dummies (x_3, x_2, x_5) in Figure 2, and that the two bound instances of x_2 have the different reference depths 1 and 3.

If we pass from the tree to the character string representation, we can omit the names of the dummies. We can write the namefree form of the example of Figure 2 as

$$[\tau]\,[[\tau]\,\langle 1\rangle\,2]\,\langle[1]\,\tau\rangle\,[\tau]\,[1]\,3\ .$$

This simple example demonstrates that the depth reference system was designed for other purposes than for easy reading.

3. DEGREE AND TYPE

3.1. Introduction

To every lambda tree we shall assign a non-negative number, to be called its degree. And we shall even assign a degree to every end-point of a lambda tree.

If a lambda tree has degree > 1 we shall define its type, which is again a lambda tree. The degree of the new tree is 1 less than the one of the original one.

As a preparation we need the notion of the lexicographical order in a binary tree. Moreover, for the definition of the type of a lambda tree we need the notion of implantation.

3.2. Lexicographical order

In a lambda tree the points are words consisting of l's and r's. We can order them as in a dictionary, starting with the empty word. For the tree of Figure 1 (Section 2.6) the dictionary is:

$$\varepsilon, l, r, rl, rll, rlr, rlrl, rlrr, rr, rrl, rrll, rrlr, rrr, rrrl, rrrr, rrrrl, rrrrr\ .$$

A word u is said to be *lexicographically lower* than the word v if u comes before v in the dictionary.

Note that if $u < v$ (in the sense of Section 2.2) then u is lexicographically lower than v, but the converse need not be true.

3.3. Ascendants

Let (V, lab) be a lambda tree. If u is an end-point with $\mathrm{lab}(u) \neq \tau$ then we shall define the *ascendant* of u, to be denoted by $\mathrm{asc}(u)$; it will be again an end-point. We note that $\mathrm{lab}(u)$ is not an end-point (see 2.5 (iii)), and therefore there exist points of V which are lexicographically higher than $\mathrm{lab}(u)$ and lower than $\mathrm{rightson}(\mathrm{lab}(u))$. The lexicographically highest of these is an end-point of V, and it is this end-point that we take as the definition of $\mathrm{asc}(u)$.

Let us take Figure 1 as an example. The end-points are, in lexicographic order: l, rll, $rlrl$, $rlrr$, $rrll$, $rrlr$, $rrrl$, $rrrrl$, $rrrrr$. Of these, l, rll, $rrlr$ and $rrrl$ have no ascendants, but $\mathrm{asc}(rlrl) = rll$, $\mathrm{asc}(rlrr) = l$, $\mathrm{asc}(rrll) = rlrr$, $\mathrm{asc}(rrrrl) = rrrl$, $\mathrm{asc}(rrrrr) = rlrr$.

3.4. Degree of an end-point

If u is an end-point of (V, lab), and $\mathrm{lab}(u) \neq \tau$, then $\mathrm{asc}(u)$ is lexicographically lower than u. This is obvious since

(i) $\mathrm{asc}(u)$ is lexicographically lower than $\mathrm{rightson}(\mathrm{lab}(u))$,

(ii) $\mathrm{rightson}(\mathrm{lab}(u)) \leq u$ by Section 2.5, so

(iii) $\mathrm{rightson}(\mathrm{lab}(u))$ is either equal to or lexicographically lower than u.

We can now define the *degree* $\deg(u)$ of the end-points one by one, proceeding through the lexicographically ordered sequence of end-points. We define

$$\deg(u) = 1 \qquad \text{if } \mathrm{lab}(u) = \tau \ ,$$
$$\deg(u) = 1 + \deg(\mathrm{asc}(u)) \qquad \text{if } \mathrm{lab}(u) \neq \tau \ .$$

This defines deg as a function:

$$\deg \ : \ V_e \to \{1, 2, 3, ...\} \ .$$

In the example of Figure 1, we have l, rll, $rrlr$, $rrrl$ of degree 1, $rlrl$, $rlrr$, $rrrrl$ of degree 2, and $rrll$, $rrrrr$ of degree 3.

3.5. Degree of a lambda tree

As the *degree* of a lambda tree (V, lab) we define $\deg(w)$, where w is the lexicographically highest point of V. This w is a word without l's.

Note that a lambda tree can have end-points whose degree exceeds the degree of the tree. We get an example if in the tree of Figure 1 we replace the label of $rrrrr$ by τ: then the tree has degree 1 but its point $rrll$ has degree 3.

3.6. Implantation

Let (V, lab) be a lambda tree, and u be a point of V, not necessarily an end-point. And let S be a set of end-points of V. We assume that the following *implantation condition* holds: for every $w \in S$ and for every $v \in V$ with $v \geq u$, $\mathrm{lab}(v) \in V$, $\mathrm{lab}(v) < u$ we have $\mathrm{rightson}(\mathrm{lab}(v)) \leq w$.

In this situation we shall describe a new lambda tree obtained by implanting at every point of S a copy of the subtree whose root is u. This new tree (V', lab') will be denoted as

$$(V', \mathrm{lab}') = \mathrm{impl}(V, \mathrm{lab}, u, S) \ .$$

First we form the subtree at u, to be denoted as $\mathrm{sub}(u)$. It is the set of all words $p \in \{l, r\}^*$ such that the concatenation up belongs to V. Next we define V' as

$$V' = V \cup \bigcup_{w \in S} \ \bigcup_{p \in \mathrm{sub}(u)} wp$$

where wp is the concatenation of w and p.

In order to define the labeling lab' of V' we divide the set $\mathrm{sub}(u)$ into two categories: $\mathrm{sub1}(u)$ and $\mathrm{sub2}(u)$. The first one, $\mathrm{sub1}(u)$, is the set of all $p \in \mathrm{sub}(u)$ for which both $\mathrm{lab}(up) \in V$ and $\mathrm{lab}(up) \geq u$. Such a p has the property that $\mathrm{lab}(up) = uq$ with some $q \in \mathrm{sub}(u)$. We may call these p's points with internal reference. All other points of $\mathrm{sub}(u)$ are put into $\mathrm{sub2}(u)$. This consists of the p's such that $\mathrm{lab}(up)$ is A, T or τ and of all p's for which both $\mathrm{lab}(up) \in V$ and $\mathrm{lab}(up) < u$. The latter p's may be called points with external reference.

We are now ready to describe the labeling lab' of V'. For all $u \in V \backslash S$ we take $\mathrm{lab}'(u) = \mathrm{lab}(u)$. The other points of V' can be uniquely written as wp with $w \in S$, $p \in \mathrm{sub}(u)$. If $p \in \mathrm{sub2}(u)$ we simply take $\mathrm{lab}'(wp) = \mathrm{lab}(up)$. If

$p \in \mathrm{subl}(u)$, however, the label of the copied point is no longer the same as the original label, but the copy of the original label. To be precise, if q is such that $\mathrm{lab}(up) = uq$, then we take $\mathrm{lab}'(wp) = wq$.

Note that if $s \in S$ then s belongs to both V and V', and that $\mathrm{lab}'(s)$ can be different from $\mathrm{lab}(s)$.

It is not hard to show that (V', lab') is again a lambda tree.

In Figure 4 we show a case of implantation. The lambda tree on the left is (V, lab), the one on the right is (V', lab').
We have $(V', \mathrm{lab}') = \mathrm{impl}(V, \mathrm{lab}, rl, \{rrr\})$.

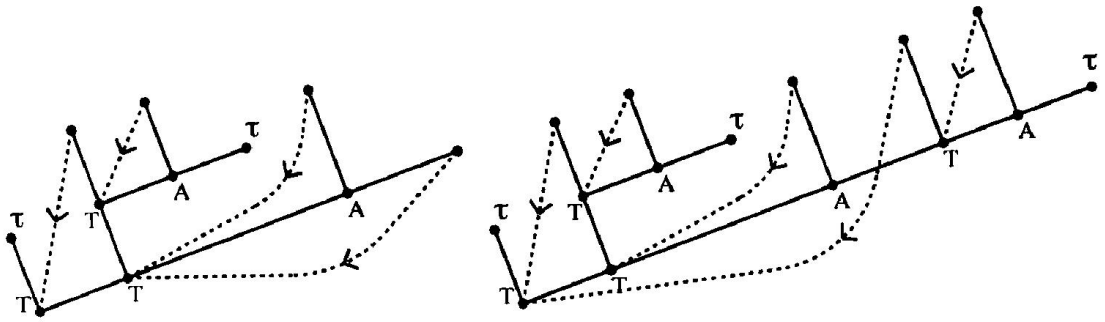

Figure 4, Implantation

3.7. Implantation and degree

We keep the notation of Section 3.6. Taking some fixed $w \in S$ we can consider the wp (with $p \in \mathrm{sub}(u)$) as copies of the corresponding up. We now claim that the degree of wp in (V', lab') is always equal to the degree of up in (V, lab). This can be proved by induction, letting p run through $\mathrm{sub}(u)$ in lexicographical order. If p is an external reference then the statement on the degrees is an easy consequence of the fact that the points of $V \backslash S$ have in (V, lab) the same degree as in (V', lab'). If p is an internal reference, however, we remark that the ascendants of up in (V, lab) and of wp in (V', lab') are corresponding points again, so that they have equal degree by the induction hypothesis. Since $\deg(up) = \deg(\mathrm{asc}(up)) + 1$, $\deg(wp) = \deg(\mathrm{asc}(wp)) + 1$, we also have $\deg(up) = \deg(wp)$.

3.8. Type of a lambda tree

If a lambda tree (V, lab) has degree > 1 we shall define the *type*, which is again a lambda tree.

Let w be the lexicographically highest point of w (see 3.5). If $\mathrm{lab}(w) = \tau$ then (V, lab) has degree 1 and its type will not be defined. The only other possibility is that $\mathrm{lab}(w) \in V$. Now $\mathrm{lab}(\mathrm{lab}(w)) = T$, whence $\mathrm{lab}(w)$ has a leftson. We now

define the type of (V, lab), to be denoted $\mathrm{typ}(V, \mathrm{lab})$, by implanting the subtree of $\mathrm{leftson}(\mathrm{lab}(w))$; here the set S consists of the single point w:

$$\mathrm{typ}(V, \mathrm{lab}) = \mathrm{impl}(V, \mathrm{lab}, \mathrm{leftson}(\mathrm{lab}(w)), \{w\}) .$$

An example of typing is already available in Figure 4: the tree on the right is the type of the one on the left.

3.9. Typing lowers the degree by 1

We shall show that if the degree of (V, lab) exceeds 1, then the degree of $\mathrm{typ}(V, \mathrm{lab})$ is one less than the degree of (V, lab).

Let again w be as in Sections 3.5 and 3.7. Since $\deg(w) > 1$, w has an ascendant: $v = \mathrm{asc}(w)$. Then $\deg(w) = \deg(v) + 1$. In the terminology of Section 3.8 we can now state that the lexicographically highest point in $\mathrm{typ}(V, \mathrm{lab})$ is the copy of v, and so, by the result of that section, its degree in $\mathrm{typ}(V, \mathrm{lab})$ equals the degree of v in (V, lab). So the degree of $\mathrm{type}(V, \mathrm{lab})$, i.e., the degree of its lexicographically highest point, is one less than $\deg(w)$, which is the degree of (V, lab).

4. REDUCTIONS

4.1. Beta reduction

We shall not present beta reduction directly. It will be introduced as the result of a set of more primitive reductions: local beta reductions and AT-removals. The reason for this is that the delta reductions of Automath can be considered as local beta reductions, and not as ordinary beta reductions.

4.2. AT-pairs

Let (V, lab) be a lambda tree. An AT-pair is a pair (u, v) where $u \in V$, $v \in V$, $\mathrm{lab}(u) = A$, $\mathrm{lab}(v) = T$, $v = \mathrm{rightson}(u)$.

Example: in Figure 1 (rr, rrr) is an AT-pair.

4.3. AT-couples

We mention that whatever we do with AT-pairs can be generalized to AT-couples. We shall not actually use AT-couples, but we give the definition for the sake of completeness. Let n be a positive integer, let $u_1, u_2, ..., u_n$ be points of V with $u_i = \mathrm{rightson}(u_j)$ for $1 < j + 1 = 1 \leq n$. Furthermore, whenever $1 \leq m < n$, the number of i with $1 \leq i \leq m$ and $\mathrm{lab}(u_i) = T$ is less than the number of i with $1 \leq i \leq m$ and $\mathrm{lab}(u_i) = A$. And finally the number of i with $1 \leq i \leq n$ and $\mathrm{lab}(u_i) = T$ is equal to the number of i with $1 \leq i \leq n$ and $\mathrm{lab}(u_i) = A$. Now (u_1, u_n) is called an AT-couple. It is easy to see that

$\mathrm{lab}(u_1) = A$, $\mathrm{lab}(u_n) = T$.

The situation can be illustrated by replacing the sequence $u_1, ..., u_n$ by a sequence of opening and closing brackets: u_i is replaced by an opening or a closing bracket according to $\mathrm{lab}(u_i) = A$ or $\mathrm{lab}(u_i) = T$. The conditions mentioned above mean that the first and the last bracket form a matching pair of brackets, like in [[ ] [[ ][ ]]].

4.4. Local beta reduction

Let (V, lab) be a lambda tree, and let w be an end-point with $\mathrm{lab}(w) \neq \tau$. We assume that the point $\mathrm{lab}(w)$ is the rightson of a point u with label A. So $(u, \mathrm{lab}(w))$ is an AT-pair. We can now form the following implantation

$$(V', \mathrm{lab}') = \mathrm{impl}(V, \mathrm{lab}, \mathrm{leftson}(u), \{w\}) .$$

The passage from (V, lab) to (V', lab') is called *local beta reduction* at w.

We give an example in the language of character strings. Let (V, lab) correspond to

$$[w : \tau]\,[x : \langle [z : w]\, z\rangle\,[y : [p : \tau]\,\tau]\,\langle y\rangle\, y]\,\tau .$$

We apply local beta reduction to the second one of the two bound occurrences of y. It comes down to replacing that y by $[z : w]\, z$ (but we have to refresh the dummy z):

$$[w : \tau]\,[x : \langle [z : w]\, z\rangle\,[y : [p : \tau]\,\tau]\,\langle y\rangle\,[q : w]\, q]\,\tau .$$

4.5. AT-removal

Let (u, v) be an AT-pair in the lambda tree (V, lab), and assume that there is no $w \in V$ such that $\mathrm{lab}(w) = v$. Then we can define a new lambda tree (V', lab') that arises by omitting this AT-pair and everything that grows on u and v on the left. A formal definition of V' is the following one. We omit u and v from V, and furthermore all points which are $\geq ul$ and all points which are $\geq vl$. Next every point of the form $urrw$ is replaced by the corresponding uw. In the latter cases the labels are redefined: if in V we had $\mathrm{lab}(urrw) = urrz$ then we take $\mathrm{lab}'(uw) = uz$; if, however, $\mathrm{lab}(urrw)$ is not $\geq urr$ we just take $\mathrm{lab}'(uw) = \mathrm{lab}(urrw)$.

We given an example of AT-removal in the language of character strings. In

$$\langle\langle\tau\rangle\,[x : \tau]\,[y : \tau]\, y\rangle\,\tau$$

there are no bound instances of x, so the pair $\langle\tau\rangle\,[x : \tau]$ can be removed.

The result is

$$\langle [y : \tau]\, y \rangle\, \tau \ .$$

4.6. Mini-reductions

We shall use the word mini-reduction for what is either a local beta reduction or an AT-removal.

4.7. Beta reduction

Let (u, v) be an AT-pair in the lambda tree (V, lab). Then beta reduction of (V, lab) (with respect to (u, v)) is obtained in two steps:

(i) We pass from (V, lab) to

$$\mathrm{impl}(V, \mathrm{lab}, \mathrm{leftson}(u), S) \ ,$$

where S is the set of all $w \in V$ with $\mathrm{lab}(w) = v$.

(ii) This new lambda tree still has the AT-pair (u, v). To this pair we apply AT-removal.

Step (i) can also be described as a sequence of local beta reductions, applied one by one to the w with $\mathrm{lab}(w) = v$. The order in which these w's are taken is irrelevant.

4.8. The Church-Rosser property

In the following, R is a relation on the set of all lambda trees. For example, the relation R can be the one of mini-reduction: if A and B are lambda trees then $(A, B) \in R$ expresses that B is obtained from A by mini-reduction.

If A and B are lambda trees, we say that B is an *R-reduct* of A if either $B = A$ or there is a finite sequence $A = A_0, A_1, ..., A_n = B$ such that for every i $(0 \leq i < n)$ we have $(A_i, A_{i+1}) \in R$.

We say that lambda trees C and D are *R-equivalent* if there is an E which is an R-reduct of both C and D. Simple examples of this are

(i) the case $C = D$, and

(ii) the cases where D is an R-reduct of C.

We note that this equivalence notion is obviously reflexive and symmetric. If it is also transitive, we say that R has the Church-Rosser property.

4.9. Church-Rosser for beta reductions

The famous Church-Rosser theorem states that in untyped lambda calculus the set of all beta reductions has the Church-Rosser property (see [Barendregt 81]). The fact that we have lambda trees with T-nodes does not make it much harder. The left-hand subtrees of the T-nodes do not play an important role in the beta reductions, but nevertheless reductions take place in these subtrees too, so they cannot be ignored. For a treatment that inludes the case of lambda trees we refer to [*de Bruijn 72b (C.2)*].

4.10. Church-Rosser for mini-reductions

The Church-Rosser property for mini-reductions is a simple consequence of the one for beta reductions. Actually two lambda trees C and D are beta equivalent if and only if they are mini-equivalent. This follows from the transitivity of beta equivalence, combined with

(i) If A leads to B by a beta reduction then B is a mini-reduct of A. (This was already noted at the end of Section 4.7.)

(ii) If A leads to B by a mini-reduction then A and B are beta equivalent.

In order to show (ii) we note that if the mini-reduction is local beta reduction with the AT-pair (u, v), then beta reduction with respect to (u, v) can be applied both to A and B, and the results are identical. If the mini-reduction is AT-removal, it is just a case of beta reduction.

4.11. Equivalence

Now that we know that beta equivalence and mini-equivalence are the same, we just use the word *equivalence* for both.

5. CORRECTNESS

5.1. Introduction

The notion of correctness of a lambda tree is concerned with the type of the P's that occur in subtrees $\langle P \rangle Q$. Roughly speaking, we require that either Q, or the type of Q, or the type of the type of $Q, \ldots$, is equivalent to something of the form $[x : R] S$, where R is equivalent to the type of P.

The system of all correct lambda trees will be called *delta-lambda* (or $\Delta\Lambda$). It is different from the older system Λ (see [*Nederpelt 73 (C.3)*], [*van Daalen 80*]) in the following respect. In Λ we always require for the correctness of $\langle P \rangle Q$ that

both P and Q are correct themselves. In $\Delta\Lambda$ we do not: P should be correct, but in formulating the requirements for Q we may make use of P. For example, in $\langle P\rangle\,[x : R]\,S$ the $[x : R]\,S$ need not be correct. We may have to apply local beta reduction by means of the pair $\langle P\rangle\,[x : R]$, that transforms S into some S' such that $[x : R]\,S'$ is correct. We actually need this feature if we want to interpret an Automath book as a correct lambda tree.

5.2. Subdivided lambda trees

In order to facilitate the formulation of correctness, we introduce a particular kind of lambda trees, where the points are colored red, white and blue. We consider a quadruple (V, lab, p, q), where (V, lab) is a lambda tree, and p, q are non-negative integers. Every $u \in V$ is a word of r's and l's, and by $nr(u)$ we denote the number of r's it starts with. So $nr(\varepsilon) = 0$, $nr(rr) = 2$, $nr(rrlr) = 2$, etc. The points u with $nr(u) \leq p$ are called *red*, those with $p < nr(u) \leq q$ *white*, those with $nr(u) > q$ *blue*.

The points $\varepsilon, r, rr, rrr, \ldots$ are called *main line* points.

We shall call (V, lab, p, q) a *subdivided lambda tree* if (i) and (ii) hold:

(i) The white main line points all have label A.

(ii) Among the red main line points there are no two consecutive labels A, and the last one in the red sequence $\varepsilon, r, rr, \ldots$ has label T. In other words, the sequence can be partitioned into groups of length 1 and 2, those of length 1 have label T, and those of length 2 consist of two consecutive points with labels A and T, respectively.

It is any easy consequence of (i) and (ii) that the set of blue points is non-empty.

Note that the conditions are automatically satisfied if $p = q = 0$. In other words, any lambda tree is a subdivided lambda tree if we color it all blue.

In the language of character strings a subdivided lambda tree looks like RWB, where W is a (possibly empty) string $\langle P_1\rangle \ldots \langle P_k\rangle$ (where $k = q - p$), and R is a string with entries either of the form $[x : Q]$ or of the form $\langle P\rangle\,[x : Q]$.

The red part R might be called a *knowledge frame*, the white part W a *waiting list*.

In order to clearly indicate the subdivision we write the character string as (R, W, B).

5.3. The definition of correctness

Let Slam3 be the set of all subdivided lambda trees. It can be presented as a set of triples (R, W, B).

We shall define a subset Corr3 of Slam3. The elements of Corr3 are called the *correct* elements of Slam3.

A lambda tree (V, lab) is called correct if $(V, \mathrm{lab}, 0, 0) \in$ Corr3. We note that $(V, \mathrm{lab}, 0, 0)$ equals $(\varepsilon, \varepsilon, B)$ if the character string B represents (V, lab). As always, ε stands for the empty string, and we shall use the obvious notations for concatenation of character strings.

We start by putting a set of triples (R, W, B) into Corr3, in rule (i); the other rules produce new triples on the basis of old ones.

(i) If $(R, \varepsilon, \tau) \in$ Slam3 then $(R, \varepsilon, \tau) \in$ Corr3.

(ii) If x is a dummy, if $(R, W, x) \in$ Slam3, and if $(R, W, \mathrm{typ}\, x) \in$ Corr3 then $(R, W, x) \in$ Corr3. We have not defined $\mathrm{typ}\, x$ separately in this paper (it would not be a lambda tree but part of a lambda tree). But we can define $(R, W, \mathrm{typ}\, x)$ as the subdivided lambda tree that represents $(V', \mathrm{lab}', p, q)$, where $(V', \mathrm{lab}') = \mathrm{typ}(V, \mathrm{lab})$, and (V, lab, p, q) is represented by (R, W, x).

(iii) If $(R, \varepsilon, K) \in$ Corr3 and $(R, W\,(K), B) \in$ Corr3 then $(R, W, (K)\,B) \in$ Corr3.

(iv) If $(R, \varepsilon, U) \in$ Corr3, $(R\,[x : U], \varepsilon, B) \in$ Corr3, then $(R, \varepsilon, [x : U]\,B) \in$ Corr3.

(v) If $(R, \varepsilon, U) \in$ Corr3, $(R\,(K)\,[x : U], W, B) \in$ Corr3, and if $TP(R, K, U)$ holds, then $(R, W\,(K), [x : U]\,B) \in$ Corr3. Here TP stands for "type property", and $TP(R, K, U)$ is the statement that if (R, ε, K) and (R, ε, U) represent (V, lab, p, p) and $(V', \mathrm{lab}', p, p)$, respectively, then (V', lab') is equivalent to $\mathrm{typ}(V, \mathrm{lab})$.

We remark that the conditions about Slam3 in rules (i) and (ii) guarantee that indeed Corr3 is a subset of Slam3.

It may seem strange that in rule (i) there is no correctness requirement on R. Therefore we cannot claim that the correctness of (R, W, B) implies the correctness of RWB. Nevertheless it can be shown that if we algorithmically check the correctness of a correct lambda tree (see Section 5.4), we will never enter into cases (R, W, B) where $(\varepsilon, \varepsilon, RWB)$ is not correct, and the conditions on Slam3 in (i) and (ii) will always be satisfied.

5.4. Algorithmic correctness check

For every $(R, W, B) \in$ Slam3 at most one of the rules (i)–(v) can be applied, and, apart from rule (i), these replace the question of the correctness by one or more uniquely defined other questions. If none of the rules can be applied

we conclude to incorrectness. Those "other questions" are all about correctness again, apart from the $TP(R, K, U)$ arising in (v).

This provides us with an algorithm for the task of the correctness check for a given lambda tree. We can think of the job as having been split into two parts:

(i) Preparing a type check list. This means that we do not answer the question about the $TP(R, K, U)$'s with the various R, K, U turning up, but just put them on a list of jobs that still have to be done. The fact that all degrees are finite (see Section 3.4) guarantees that this job list is made in a finite number of steps.

(ii) Establishing truth or falsity of the various $TP(R, K, U)$'s.

The work under (i) can already lead to the conclusion that our lambda tree is incorrect. If we forget about syntactic errors that arise if we are presented with a structure that is not a lambda tree at all, this only happens in cases where we get to (R, W, τ) with $W \neq \varepsilon$, where none of our rules apply.

5.5. Remarks about the type check list

The type check list can be prepared if we systematically apply the rules (i)–(v). In each one of the rules (iii), (iv), (v) there are two subgoals where something has to be shown to belong to Corr3. There are good reasons to tackle these subgoals in the order in which they are mentioned in the rules. This comes down to a lexicographical traversal of the lambda tree we have to investigate. This traversal can occasionally be interrupted by some application of rule (ii), which leads to an excursion in an extended tree.

The type check list prepared by the algorithm hinted at in 5.4 can lead to some duplication of work, by two causes:

(i) The given lambda tree can have one and the same substructure at various places. This will actually occur quite often if we represent an Automath book as a lambda tree.

(ii) Application of rule (iv) of Section 5.3 leads us into asking questions about $\mathrm{typ}\, x$ that have already been answered before.

The duplications mentioned in (ii) can be avoided to a large extent: see Section 5.8.

We mention a shortcut that reduces the work needed to prepare the type check list. It is obtained by splitting rule (ii) of Section 5.3 into (ii′) and (ii″): (ii′) is as (ii), but with the restriction $W \neq \varepsilon$, and

(ii′) if $(R, \varepsilon, x) \in$ Slam3 (where x is a dummy), then $(R, \varepsilon, x) \in$ Corr3.

5.6. Remarks about the type checks

The type checks $TP(R, K, U)$ were introduced in 5.3 (vi). Given R, K, U, we can consider the question to establish by means of an algorithm whether $TP(R, K, U)$ is true or false. The question comes down to establishing whether the (V, lab) of 5.3 (vi) has a type (which is simply a matter of degree) and whether (V', lab') and $\text{typ}(V, \text{lab})$ have a common reduct. It is quite easy to design an algorithm that does a tree search of all reducts of (V', lab') and $\text{typ}(V, \text{lab})$. If they do have a common reduct, that fact will be established in a finite time. But will "finite" be reasonably small here? And what if they do not have a common reduct? Are we able to establish that negative fact in a finite time, or at least in a reasonable time? And what if the tree search does not terminate?

From a theoretical point of view we can say that our questions about the correctness of a given lambda tree are decidable. For the system Δ this was already shown by R. Nederpelt ([*Nederpelt 73 (C.3)*], [*van Daalen 80*]), for $\Delta\Lambda$ by L.S. van Benthem Jutting (oral communication). It is done in two steps:

(i) Between the notion of "lambda tree" and "correct lambda tree" there is a notion "norm-correct lambda tree". For any given lambda tree it can be established in a finite time whether it is or is not norm-correct.

For the notion of norm-correctness we refer to Section 5.9. In [*Nederpelt 73 (C.3)*] the term "normable" was used instead of "norm-correct".

(ii) For every norm-correct lambda tree we have the strong normalization property: there exists a number N (depending on the tree) such that no sequence of reductions is longer than N.

As to (ii) we note that if we have reduced both (V', lab') and $\text{typ}(V, \text{lab})$ to a point where no further reductions are possible, then the question becomes trivial: in that case, having a common reduct just means being equal.

The strong normalization property guarantees that the question whether a given lambda tree is or is not correct is a decidable question.

5.7. Practical standpoint

Apart from the cases of very small trees, the matter of decidability of correctness will not be of practical value: the number N mentioned in 5.6 (ii) will usually be prohibitively large. If a tree is incorrect, the finite time it takes to establish that fact may be hopelessly long. It is better to be more modest, and to try to design algorithms with efficient strategies, by means of which we can

show the correctness of the lambda trees we have to deal with in practice. If such algorithms are applied to an incorrect lambda tree, the fact that they have used an unreasonable amount of time without having reached a decision, may be considered as an indication that the tree is possibly incorrect.

Sometimes we can apply quite easy checks by means of which an incorrect tree can be rejected fast: it might fail to be norm-correct, or might be no lambda tree at all. Or we might run into cases where the type of some (V, lab) is required but where $\deg(V, \mathrm{lab}) = 1$.

5.8. Avoiding double work

We can rearrange the definition of correctness in such a way that it leads to an algorithm that gives just a single type check corresponding to each A-node in the lambda tree we have to check the correctness of.

If we just follow the algorithm sketched in Section 5.4, the cases where we have to treat $(R, W, \mathrm{typ}\, x)$ will cause double work: what is involved in $\mathrm{typ}\, x$ has been earlier dealt with in the execution of the algorithm. The only thing that deserves to be checked is whether the A nodes in W match with the T-nodes that arise from $\mathrm{typ}\ x$ (possibly after one or more further applications of rule (ii)).

Let us divide the waiting list into two consecutive parts. The first part is still called "white", the second part is called "yellow". For the yellow part the work load will be lighter than for the white part. A formal definition of these four-colored lambda trees is easily obtained by slight modification of Section 5.2. We have to consider $(V, \mathrm{lab}, p, s, q)$; the points with $p < nr(u) \leq s$ are white, those with $s < nr(u) \leq q$ yellow. And the yellow main line points are required to have label A, just like the white ones.

Let us denote the set of these four-colored lambda trees by Slam4. Its elements will be represented as (R, W, Y, B), just like those of Slam3 were represented by (R, W, B).

We now formulate a new definition of correctness of lambda trees, equivalent to the old one. The difference is that the new definition leads to an algorithm that avoids the duplication we hinted at. It involves both a subset of Corr3 of Slam3 and a subset Corr4 of Slam4. The final goal is as before: (V, lab) is called correct is its character string P is such that $(\varepsilon, \varepsilon, P) \in$ Corr3.

As rules we take (i), (iii), (iv), (v) as in Section 5.3, but we add new rules (vi)–(xii), where (iii) replaces the discarded rule (ii):

(vi) If $(R, \varepsilon, \varepsilon, \tau) \in$ Slam4 then $(R, \varepsilon, \varepsilon, \tau) \in$ Corr4.

(vii) If x is a dummy, and $(R, W, \varepsilon, x) \in$ Corr4 then $(R, W, x) \in$ Corr3.

(viii) If x is a dummy, if $(R, W, Y, x) \in$ Slam4, and if $(R, W, Y, \text{typ}\, x) \in$ Corr4 then $(R, W, Y, x) \in$ Corr4. The definition of $(R, W, Y, \text{typ}\, x)$ is similar to the one of $(R, W, \text{typ}\, x)$ in 5.3 (ii).

(ix) If $(R, W, Y\, \langle K \rangle, B) \in$ Corr4 then $(R, W, Y, \langle K \rangle\, B) \in$ Corr4.

(x) If $(R[x : U], \varepsilon, \varepsilon, B) \in$ Corr4 then $(R, \varepsilon, \varepsilon, [x : U]\, B) \in$ Corr4.

(xi) If $(R\langle K \rangle [x : U], W, \varepsilon, B) \in$ Corr4 and $TP(R, K, U)$ holds, then $(R, W\, \langle K \rangle, \varepsilon, [x : U]\, B) \in$ Corr4.

(xii) If $(R\langle K \rangle [x : U], W, Y, B) \in$ Corr4 then $(R, W, Y\, \langle K \rangle, [x : U]\, B) \in$ Corr4.

At the end of Section 5.5 we mentioned the shortcut rule (ii′). There is a similar shortcut here: it can replace (vi) and (x):

(x′) If $(R, \varepsilon, \varepsilon, B) \in$ Slam4 then $(R, \varepsilon, \varepsilon, B) \in$ Corr4.

However, the set of rules without shortcuts may be better for theoretical purposes.

5.9. Weaker notions of correctness

We can weaken the notion of correctness by weakening the requirement about $TP(R, K, U)$ in rule 5.3 (v).

If in rule 5.3 (v) we omit the requirement of $TP(R, K, U)$ altogether, we get what we can call *semicorrectness.*

For semicorrect lambda trees we can define a norm corresponding to Nederpelt's norm for the system Λ (see [*Nederpelt 73 (C.3)*]). A norm is a particular kind of lambda tree: it has no labels A and all end-point labels are τ. To every semicorrect lambda tree we attach such a norm. It can be defined algorithmically if we just follow the list of Section 5.8. First we define the norms of the (R, W, B)'s and (R, W, Y, B)'s:

(i) and (vi): as norms of (R, ε, τ) and $(R, \varepsilon, \varepsilon, \tau)$ we take the lambda tree consisting of just one node, labeled τ.

(iii): $\text{norm}(R, W, \langle K \rangle\, B) = \text{norm}(R, W\, \langle K \rangle, B)$.

(iv) and (x): as norm of $(R, \varepsilon, [x : U]\, B)$ (or of $(R, \varepsilon, \varepsilon, [x : U]\, B)$) we take the lambda tree with root labeled T, whose left-hand subtree is $\text{norm}(R, \varepsilon, U)$ (or $\text{norm}(R, \varepsilon, \varepsilon, U)$), and whose right-hand subtree is $\text{norm}(R[x : U], \varepsilon, B)$ (or $\text{norm}(R[x : U], \varepsilon, \varepsilon, B)$).

(v): $\text{norm}(R, W\, \langle K \rangle, [x : U]\, B) = \text{norm}(R\, \langle K \rangle [x : U], W, B)$.

(vii): $\text{norm}(R, W, x) = \text{norm}(R, W, \varepsilon, x)$.

(viii): $\text{norm}(R, W, Y, x) = \text{norm}(R, W, Y, \text{typ}\, x)$.

(ix): $\text{norm}(R, W, Y, \langle K \rangle\, B) = \text{norm}(R, W, Y\, \langle K \rangle, B)$.

(xi): $\mathrm{norm}(R, W\,\langle K\rangle, \varepsilon, [x:U]\,B) = \mathrm{norm}(R\,\langle K\rangle\,[x:U], W, \varepsilon, B)$.
(xii): $\mathrm{norm}(R, W, Y\,\langle K\rangle, [x:U]\,B) = \mathrm{norm}(R\,\langle K\rangle\,[x:U], W, Y, B)$.

The norms of (R, W, B) or (R, W, Y, B) are actually a kind of norm for WB or WYB; the role of R is only to provide the types of the dummies.

Finally the norm of a semicorrect lambda tree (V, lab) is defined as the norm of the all-blue lambda tree $(V, \mathrm{lab}, 0, 0)$ (which has the form $(\varepsilon, \varepsilon, B)$).

We can use a similar algorithm for finding the degree of a lambda tree: we just say in cases (i) and (vi) that the degree is 1, and in cases (ii) and (viii) that the degree is to be increased by 1.

Next we can define the notion of *norm-correct* lambda trees. We get that notion by replacing in rule 5.3 (v) the condition that the type of (V', lab') is equivalent to $\mathrm{typ}(V, \mathrm{lab})$ by the condition that (V', lab') has the same norm as (V, lab). This condition is weaker than $TP(R, K, U)$, and therefore every correct lambda tree is also norm-correct.

For norm-correct lambda trees we have the strong normalization property (see Section 5.6).

5.10. Norms for lambda trees which are not necessarily semicorrect

If (V, lab) is a lambda tree which is not semicorrect, that fact is established by the algorithm of Section 5.8 at some moment where we get to (R, W, τ) or (R, W, Y, τ) with W or Y non-empty.

For such lambda trees we can nevertheless still define the norm, by the procedure of Section 5.9, if we just extend the action in cases (i) and (vi) by saying that (R, W, τ) and (R, W, Y, τ) have the single-noded tree (labeled by τ) as their norm, also in cases where W or Y are not empty.

6. AUTOMATH BOOKS AS LAMBDA TREES

6.1. Some characteristics of Automath

We shall not explain Automath in detail here: we assume that the reader knows it from other sources (like [*de Bruijn 70a (A.2)*], [*de Bruijn 71 (B.2)*], [de Bruijn 73b],[*de Bruijn 80 (A.5)*],[*van Benthem Jutting 77*],[*van Daalen 80*]). In particular, we shall not try to be very precise in defining particular brands of Automath. Nevertheless we indicate a few characteristics, in order to get to the kind of Automath that corresponds to $\Delta\Lambda$.

For a discussion that compares various forms of Automath in the light of such characteristics we refer to [de Bruijn 74a].

(i) Automath books are written as sequences of lines: primitive lines, ordinary

lines (= definitional lines), and context lines that describe the contexts of the other lines.

(ii) We have the notion of typing, and that leads to the degrees. In standard Automath the only degrees are 1, 2 and 3, and it seems that for the description of mathematics no serious need for higher degrees ever turned up.

(iii) There are restrictions on abstraction. Contexts may be described as $[x_1 : A_1] \ldots [x_n : A_n]$ where the A_i may have degree 1 or 2, but in expressions (also in the A_i's of the contexts) we only admit abstractors $[x : A]$ where A has degree 2.

(iv) In Automath we have instantiation: if the identifier p is the identifier of a line in a context of length n, then the "instantiations" $p(E_1, \ldots, E_n)$, where the E_i are expressions, can be admitted in other contexts.

(v) In some of the Automath languages (like AUT-QE) we admit "quasi-expressions": expressions of degree 1 which are not just τ.

(vi) In some of the Automath languages we have type inclusion: if $E : [x_1 : A_1] \ldots [x_n : A_n] \tau$ then we admit that E is substituted at places where a typing $[x_1 : A_1] \ldots [x_k : A_k] \tau$ (with some $k < n$) is required.

6.2. Automath without type inclusion

We can take Automath with quasi-expressions but without type inclusion (AUT-QE-NTI). Both AUT-QE-NTI and AUT-68 are sublanguages of AUT-QE: we might say that in AUT-68 type inclusion is prescribed, in AUT-QE it is optional, in AUT-QE-NTI it is forbidden. In [*de Bruijn 78c (B.4)*] it was pointed out that AUT-QE-NTI can be used as a language for writing mathematics, somewhat lengthier than in AUT-QE. One might say that sacrificing type inclusion has to be paid by means of a number of extra axioms. But there is a disadvantage to type inclusion: type inclusion makes language theory considerably harder. The rules in AUT-QE-NTI are simple; we just mention that whenever $A : B$ in the context $[x : U]$, and U has degree 2, then $[x : U] A : [x : U] B$ in empty context.

6.3. AUT-LAMBDA

In AUT-QE-NTI we still had restrictions:

(i) the degrees are 1, 2 or 3, and

(ii) in expressions abstractors $[x : U]$ are allowed only if U has degree 2.

If we give up these restrictions, we get what we can call liberal AUT-QE-NTI. In liberal AUT-QE-NTI the role of instantiation can be taken over completely by abstractors and applicators. In order to make this clear we take a simple example: $f := A : B$ in context $[x : U]$. According to the liberal abstraction rules of AUT-LAMBDA we can write a new line $F := [x : U]\,A : [x : U]\,B$ in empty context. Next, the instantiation $f(E)$ is equivalent to $(E)\,F$, so we can just replace the line $f := A : B$ by the new one in empty context, and abolish the instantiation. Carrying on, we get books without instantiation, all written in empty context. Such books can be considered as having been written in a sublanguage of liberal AUT-QE-NTI; let us call it AUT-LAMBDA.

In AUT-LAMBDA there are just two kinds of lines:

(i) primitive lines

$$f := \text{'prim'} : P \quad \text{and}$$

(ii) definitional lines

$$g := Q : R\,.$$

(Note: 'prim' was written as PN in other Automath publications.)

6.4. Turning AUT-LAMBDA into $\Delta\Lambda$

We shall turn a book in AUT-LAMBDA into a lambda tree by a system that turns correct AUT-LAMBDA into correct lambda trees. It almost works in the opposite direction too, but it turns out that $\Delta\Lambda$ is a trifle stronger than AUT-LAMBDA. The difference lies in some cases of what was mentioned in Section 5.1, but the difference is so small and unimportant that it seems to be attractive to modify the definition of AUT-LAMBDA a tiny little bit, in order to make the correspondence complete.

The transition is simple. We turn the identifiers of a book in AUT-LAMBDA into dummies. To a line $f := \text{'prim'} : P$ we attach the abstractor $[f : P]$, to a line $g := Q : R$ we attach the applicator-abstractor pair $(Q)\,[g : R]$. We do this for each line of the book, and put the abstractors and applicator-abstractor pairs into a single string, and we close it off by τ. So to a book

$$\begin{aligned} f &:= \text{'prim'} : P \\ g &:= Q : R \\ k &:= V : W \\ h &:= \text{'prim'} : Z \end{aligned}$$

there corresponds the string

$$[f : P]\,\langle Q\rangle\,[g : R]\,\langle V\rangle\,[k : W]\,[h : Z]\,\tau\ ,$$

and this corresponds to a lambda tree.

6.5. Checking algorithms

If we start from an AUT-LAMBDA book, transform it into a lambda tree as in Section 6.4, and apply the checking algorithm of Section 5.4, then we have the advantage that the AUT-LAMBDA book is checked line by line. So even if the book is incorrect as a whole, the first k lines can still be correct, and the algorithm can establish that fact. The same thing holds if we take the weaker correctness notions discussed in Section 5.9.

6.6. Type inclusion

If we want to add the feature of type inclusion to AUT-LAMBDA, the transition of a book to a lambda tree can no longer be made in the same way. Moreover we need essential changes in the notion of typing in $\Delta\Lambda$.

6.6. A variation of $\Delta\Lambda$

We mention a modification of the definition of correctness of a lambda tree, obtained by considering different kinds of A-nodes.

Let us divide the set of all A-nodes of a lambda tree into two classes: strong ones and weak ones. We take it as a rule that whenever a part of a tree is copied (like in the definition of typing) the copies of weak nodes are weak again, and the copies of strong nodes are strong.

For the weak A-nodes the rules are as in Section 5.3, but for the strong ones we modify rule 5.3 (iii) by not just requiring that (R, ε, K) and $(R, W\,\langle K\rangle, B)$ are in Corr3, but also (R, ε, B). In connection to what was said in Section 5.1 we might say that Λ corresponds to the case where all A-nodes are taken to be strong, and that $\Delta\Lambda$ is the case where all A-nodes are weak. The case mentioned in Section 6.4 lies between these two: if we want to close the gap between AUT-LAMBDA and $\Delta\Lambda$ we have to make all main line A-nodes weak and all others strong.

If we replace weak A-nodes by strong ones, a correct lambda tree may turn into an incorrect one, but it can be expected to become correct again by reductions.