MATHEMATICS

LAMBDA CALCULUS NOTATION WITH NAMELESS DUMMIES, A TOOL FOR AUTOMATIC FORMULA MANIPULATION, WITH APPLICATION TO THE CHURCH-ROSSER THEOREM

BY

N. G. DE BRUIJN

(Communicated at the meeting of June 24, 1972)

ABSTRACT

In ordinary lambda calculus the occurrences of a bound variable are made recognizable by the use of one and the same (otherwise irrelevant) name at all occurrences. This convention is known to cause considerable trouble in cases of substitution. In the present paper a different notational system is developed, where occurrences of variables are indicated by integers giving the "distance" to the binding λ instead of a name attached to that λ. The system is claimed to be efficient for automatic formula manipulation as well as for metalingual discussion. As an example the most essential part of a proof of the Church-Rosser theorem is presented in this namefree calculus.

1. INTRODUCTION

For what lambda calculus is about, we refer to [1], [3] or [4], although no specific knowledge will be required for the reading of the present paper.

Manipulations in the lambda calculus are often troublesome because of the need for re-naming bound variables. For example, if a free variable in an expesssion has to be replaced by a second expression, the danger arises that some free variable of the second expression bears the same name as a bound variable in the first one, with the effect that binding is introduced where it is not intended. Another case of re-naming arises if we want to establish the equivalence of two expressions in those situations where the only difference lies in the names of the bound variables (i.e. when the equivalence is so-called α-equivalence).

In particular in machine-manipulated lambda calculus this re-naming activity involves a great deal of labour, both in machine time as in programming effort. It seems to be worth-while to try to get rid of the re-naming, or, rather, to get rid of names altogether.

Consider the following three criteria for a good notation:

(i) easy to write and easy to read for the human reader;
(ii) easy to handle in metalingual discussion;
(iii) easy for the computer and for the computer programmer.

The system we shall develop here is claimed to be good for (ii) and

good for (iii). It is not claimed to be very good for (i); this means that for computer work we shall want automatic translation from one of the usual systems to our present system at the input stage, and backwards at the output stage.

An example showing that our method is adequate for (ii) can be found in sections 10–12, which present the kernel of a proof for the Church-Rosser theorem. This proof is essentially the one that was given in [1], where it was attributed to P. Martin-Löf (1971). Later private information by Mr. H. P. Barendregt disclosed that the idea is due to W. W. Tait. For a survey of proofs of the Church-Rosser theorem see [1] p. 16–17. An elaborate treatment of the theorem can also be found in [4].

What is said about lambda calculus in this paper can be applied directly to other kinds of dummy-binding in mathematics. For example, if we have an expression like the product $\Pi^{q}_{k=p} f(k, m)$, we can write it as $\Pi(p, q, \lambda_k f(k, m))$. For any new quantifier we wish to use (like Π here) we have to take a particular symbol that is treated as an element of the alphabet of constants (see section 3).

Application to AUTOMATH is explained in section 13.

2. Notation in metalingual discussion

If we want to denote a string of symbols by a single ("metalingual") symbol, we have to be very careful, in particular if this procedure is repeated, e.g. if we form mixed strings of lingual and metalingual symbols, represent these by a new symbol, etc.

We shall use parentheses $\langle\ \rangle$ for this purpose. If Φ denotes a string, then Φ is not the string itself. For the string itself we shall use $\langle \Phi \rangle$. We shall say that Φ *denotes* the string and that $\langle \Phi \rangle$ *is* the string. Let us consider some examples, where the basic lingual symbols are all Latin letters as well as the hyphen. (These examples will show the use of $\langle\ \rangle$ in nested form, and therefore show that the simple device of using Greek letters on the metalingual level is definitely insufficient.) We shall use the symbol ϱ for reversing the order of a string. That is, $\varrho(pqra)$ denotes $arqp$, whence $\langle \varrho(pqra) \rangle = arqp$. Now let Φ denote the word phi, and let Σ denote the word sigma. Then $\langle \Phi \rangle \langle \Sigma \rangle$ is the word phisigma, $\langle \Phi \rangle$-$\langle \Sigma \rangle$ = phi-sigma, $\langle \varrho(\langle \Sigma \rangle) \rangle = \text{amgis}$, and

$$\langle \varrho(\langle \varrho(\langle \Phi \rangle) \rangle \langle \varrho(\text{sigma}) \rangle) \rangle =$$

$$= \langle \varrho(\text{ihpamgis}) \rangle = \text{sigmaphi} = \langle \Sigma \rangle \langle \Phi \rangle.$$

The $\langle\ \rangle$'s of this section are not to be confused with the similar symbols we use in Backus' normal form of a syntax (e.g. in section 5).

In typescript and in handwriting it is convenient to *underline* a formula instead of putting it in $\langle\ \rangle$'s. In print, however, underlining, and in particular multi-level underlining, is awkward.

3. Name-carrying expressions

We explain the kind of lambda calculus expressions which we want to turn into namefree expressions. We have a set of "constants" $(a, b, c, f, g, \ldots)$ and a set of "variables" $(s, t, u, v, w, x, \ldots)$. And there is the symbol λ that can have any variable as a suffix. Moreover we admit *application*, of which the following is the interpretation. If Φ denotes a function, and Γ a value of the variable, then $\langle\Phi\rangle(\langle\Gamma\rangle)$ is the value of the function at $\langle\Gamma\rangle$. We shall use a different notation instead: we add a symbol A to the list of constants, and we write $A(\langle\Phi\rangle, \langle\Gamma\rangle)$ instead of $\langle\Phi\rangle(\langle\Gamma\rangle$. This puts it on a par with another kind of expression we are going to admit, viz. things like $f(, ,)$, where f is any constant. In the interpretations the latter kind of expression can be very close to what we have just called application, but that does not bother us at the moment.

We shall not go into a formal definition of the syntax; the following example (that accidentally does not contain the symbol A at all) will be clear enough. We take the expression

$$\lambda_x a(\lambda_t b(x, t, f(\lambda_u a(u, t, z), \lambda_s w)), w, y). \tag{3.1}$$

4. Getting rid of the names of variables

In order to facilitate the discussion, we represent the expression as a planar tree which is easier to read than (3.1) itself.

If in (3.1) we change the names of the bound variables, e.g. x, t, u, s into p, u, s, x, we get an expression that is what is usually called α-equivalent to (3.1):

$$\lambda_p a(\lambda_u b(p, u, f(\lambda_s a(s, u, z), \lambda_x w)), w, y).$$

We shall take the simplistic point of view that α-equivalent expressions are the same. Formula (3.1) contains bound variables x, t, u, s and free variables z, w, y. We shall keep a list of letters from which the free variables are to be taken. Let that list be, in this order, z, v, w, y; we draw the points $\lambda_z, \lambda_v, \lambda_w, \lambda_y$ under the tree.

The variables in the tree (fig. 1) are encircled (unless they occur as a suffix of λ).

For every encircled letter we evaluate two integers which are indicated in the figure, viz. the reference depth and the level. The *reference depth* of an encircled variable at a certain spot, x say, is the number of λ's we encounter when running down until we meet λ_x (this λ_x is counted as one of the encountered λ's). It is agreed that the $\lambda_y, \lambda_w, \lambda_v, \lambda_z$ (which do not belong to the tree itself) can also be encountered on our way down, e.g. if we run down to λ_z we encounter $\lambda_y, \lambda_w, \lambda_v, \lambda_z$.

The *level* of an encircled variable at a certain spot counts the total number of λ's we encounter when running down the tree until we get to the root (if the root is a λ, like λ_x here, this one is also included in the count; the loose $\lambda_y, \lambda_w, \lambda_v, \lambda_z$ are not counted this time.

Let us now erase the variables and the integers indicating the levels; we keep the reference depth. No information is lost: the erased letters and numbers can be easily reconstructed. If we are not interested in the names of the bound variables (and honestly we should not be) we can erase the suffix in λ_x, λ_t, λ_u, λ_s. In those cases where we are interested in the names of the free variables we have to keep the ordered list z, v, w, y in order to be able to reconstruct our expression. Note that a point of the tree refers to a free variable if and only if the reference depth exceeds the level.

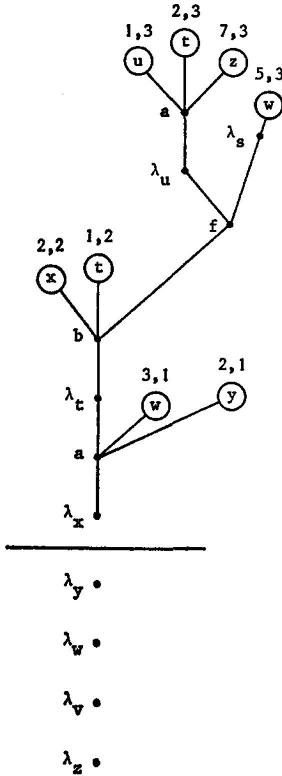

Fig. 1.

Thus the information contained in our name-carrying expression can be presented as

(4.1) $$\lambda a(\lambda b(2, 1, f(\lambda a(1, 2, 7), \lambda 5)), 3, 2)$$

with the free variable list z, v, w, y. This expression is called *namefree.*

Note that (3.1) can be represented differently if we take a different free variable list. Any sequence of distinct variables may serve as a free variable list provided it contains z, w, y in any order. Conversely, every namefree expression can be decoded into a name-carrying one if we provide a free variable list that is long enough. This determines the name-carrying expression up to name-changing of the bound variables.

Instead of providing a finite free variable list we can take an infinite one (with the effect that we need not bother whether the list is long enough). The reference depths refer to a count in the reference list from right to left, corresponding to the fact that λ's are written *in front* of the formula they act on. Therefore, such infinite free variable lists have to be written as $\ldots, x_3, x_2, x_1$ instead of the usual left-to-right notation of an infinite sequence.

5. The syntax for namefree expressions

We present the syntax in Backus' normal form:

$$\langle\text{constant}\rangle ::= a \mid b \mid c \mid d \mid \ldots$$

$$\langle\text{NF expression string}\rangle ::= \langle\text{NF expression}\rangle \mid \langle\text{NF expression string}\rangle, \langle\text{NF expression}\rangle$$

$$\langle\text{NF expression}\rangle ::= \langle\text{constant}\rangle \mid \langle\text{constant}\rangle(\langle\text{NF expression string}\rangle) \mid \langle\text{positive integer}\rangle \mid \lambda\langle\text{NF expression}\rangle$$

In the next sections we shall use, in an informal way, the notions "level" of an integer in an NF expression, in the sense of section 4. (The "reference depth" of an integer is, of course, the integer itself).

6. Substitution

We shall define the effect of a substitution of a sequence of NF expressions into a single NF expression denoted by Ω.

What we intend to describe is the following. Let $\ldots, \Sigma_3, \Sigma_2, \Sigma_1$ denote the sequence (in right-to-left notation). (In practice only finitely many Σ_k's are relevant; whence we need not always give the full infinite sequence). We attach a free variable list $\ldots, x_3, x_2, x_1$ to Ω, and one and the same free variable list $\ldots, y_3, y_2, y_1$ to every Σ_i. That determines name-carrying expressions to be denoted by Ω^* and Σ_i^*. Now replace any free x_i in $\langle\Omega^*\rangle$ by the corresponding $\langle\Sigma_i^*\rangle$. Thus we get an expression, to be denoted by Γ^*, with possible free variables $\ldots, y_3, y_2, y_1$. With respect to this free variable list $\ldots, y_3, y_2, y_1$ this Γ^* corresponds to the NF expression $S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)$ we shall define presently. The definition will be recursive with respect to the structure of $\langle\Omega\rangle$; Ω may denote either an NF expression string or an NF expression. We follow the syntactic classification of section 5.

(i) If $\langle\Omega\rangle = \langle\Omega_1\rangle, \langle\Omega_2\rangle$ then

$$\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)\rangle = \langle\Gamma_1\rangle, \langle\Gamma_2\rangle$$

where Γ_i denotes

$$\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega_i\rangle)\rangle.$$

(ii) If $\langle\Omega\rangle$ is a constant then

$$\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)\rangle = \langle\Omega\rangle.$$

(iii) If $\langle\Omega\rangle = \langle\gamma\rangle(\langle\Omega_1\rangle)$ (where γ denotes a constant and Ω_1 an NF expression string) then

$$\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)\rangle = \langle\gamma\rangle(\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega_1\rangle)\rangle).$$

(iv) If $\langle\Omega\rangle$ is the positive integer k then

$$\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)\rangle = \langle\Sigma_k\rangle.$$

(v) If $\langle\Omega\rangle = \lambda\langle\Gamma\rangle$ then

$$\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)\rangle = \lambda\langle S(\ldots, \langle\Lambda_3\rangle, \langle\Lambda_2\rangle, \langle\Lambda_1\rangle, 1; \langle\Gamma\rangle)\rangle \tag{6.1}$$

where Λ_i denotes

$$\langle S(\ldots, 4, 3, 2; \langle\Sigma_i\rangle)\rangle. \tag{6.2}$$

Note that $\langle\Lambda_i\rangle$ is obtained from $\langle\Sigma_i\rangle$ by adding 1 to every integer in $\langle\Sigma_i\rangle$ that refers to a free variable.

7. The operators τ_h, and global description of substitution

It will be convenient to use the separate notation $\tau_h(\langle\Sigma\rangle)$ in order to abbreviate

$$S(\ldots, h+3, h+2, h+1; \langle\Sigma\rangle).$$

It means adding h (which is a positive integer) to every integer $\langle\Sigma\rangle$ that refers to a free variable. The special case $\tau_1(\langle\Sigma_i\rangle)$ occurs in (6.2). With the aid of this notation we can give a more global description of how $\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)\rangle$ is obtained: start from Ω, and in each case where an integer t in $\langle\Omega\rangle$ exceeds its level l, we replace that t by $\langle\tau_l(\langle\Sigma_{t-l}\rangle)\rangle$.

In automatic formula manipulation it may be a good strategy to refrain from evaluating such $\tau_l(\langle\Sigma\rangle)$'s, but just to store them as pairs $l, \langle\Sigma\rangle$, and go into (full or partial) evaluation only if necessary. The following formulas may come in handy:

$$\tau_k\tau_l = \tau_{k+l}; \ \langle\tau_0(\langle\Omega\rangle)\rangle = \langle\Omega\rangle$$

$$\langle S(\ldots, \langle\Sigma_3\rangle, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\tau_k(\langle\Omega\rangle)\rangle)\rangle = \langle S(\ldots, \langle\Sigma_{k+3}\rangle, \langle\Sigma_{k+2}\rangle, \langle\Sigma_{k+1}\rangle; \langle\Omega\rangle)\rangle.$$

The latter formula is a special case of the following result on composite substitution:
If

$$\langle\Omega\rangle = \langle S(\ldots, \langle\Lambda_2\rangle, \langle\Lambda_1\rangle; \langle\Delta\rangle)\rangle,$$

then

$$S(\ldots, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle) = S(\ldots, \langle\Gamma_2\rangle, \langle\Gamma_1\rangle; \langle\Delta\rangle)$$

where

$$\langle\Gamma_i\rangle = \langle S(\ldots, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Lambda_i\rangle)\rangle \qquad (i = 1, 2, \ldots).$$

8. Beta reduction

If we have an applicational expression $A(\langle\Phi\rangle, \langle\Gamma\rangle)$ (cf. section 3), then the interpretation is that $\langle\Phi\rangle$ is a function, $\langle\Gamma\rangle$ a value of the variable

in that function, and $A(\langle\Phi\rangle, \langle\Gamma\rangle)$ is intended to represent the value of the function $\langle\Phi\rangle$ at the point $\langle\Gamma\rangle$. If $\langle\Phi\rangle$ happens to have the form $\lambda\langle\Omega\rangle$, then the function value can actually be evaluated. Roughly speaking, it comes down to substituting $\langle\Gamma\rangle$ in $\langle\Omega\rangle$ for all occurrences of the bound variable corresponding to the λ in front of $\langle\Omega\rangle$.

A precise definition in terms of NF expressions is easy to give: If Ω and Γ denote NF expressions, then $A(\lambda\langle\Omega\rangle, \langle\Gamma\rangle)$ is an NF expression to which beta reduction can be applied. The effect of the beta reduction is the NF expression

$$(8.1) \qquad \langle S(\ldots, 3, 2, 1, \langle\Gamma\rangle; \langle\Omega\rangle)\rangle.$$

The usual beta reduction for name-carrying expressions is obtained if we use one and the same free variable list for all four expressions $\lambda\langle\Omega\rangle$, $\langle\Gamma\rangle$, $A(\lambda\langle\Omega\rangle, \langle\Gamma\rangle)$ and (8.1).

9. Eta reduction

In terms of name-carrying expressions, η-reduction means the following. If Σ denotes a name-carrying expression that does not contain the variable x, then $\lambda_x\langle\Sigma\rangle(x)$ (or in our notation $\lambda_x A(\langle\Sigma\rangle, x)$) has the same mathematical interpretation as $\langle\Sigma\rangle$ itself. The transfer from $\lambda_x\langle\Sigma\rangle(x)$ to $\langle\Sigma\rangle$ is called η-reduction. We shall define it for NF expressions:

For any NF expression $\langle\Lambda\rangle$ we define as η-reduction the transition of

$$(9.1) \qquad \lambda A(\langle\tau_1(\langle\Lambda\rangle)\rangle, 1) \text{ into } \langle\Lambda\rangle.$$

If we transform both expressions of (9.1) into name-carrying expressions by means of one and the same free variable list, the transition (9.1) becomes the η-reduction for name-carrying expressions.

10. Multiple beta reduction

In section 8 we considered beta reduction of an NF expression. It was reduction of the full expression and not the beta reduction of a subexpression (local beta reduction) which we shall consider presently. In order to be able to indicate where the β-reduction has to be carried out, we introduce a set of constants (applicational symbols) to be used instead of the single symbol A. By this same device we get the possibility of multiple local beta reduction: we indicate a subset of the set of applicational symbols and we carry out beta reduction for all symbols of that subset.

Let U be a subset of the set of constants. An NF expression $\langle\Sigma\rangle$ is called *U-correct* if every element of U that occurs in $\langle\Sigma\rangle$ is always followed by a string in parentheses with the form $(\lambda\langle\Omega\rangle, \langle\Gamma\rangle)$. In other words, each occurrence of each element of U is ready for local beta reduction. To be more precise, we indicate how the syntax of section 5 is to be changed in order to get the syntax of the U-correct NF expressions. We have to replace the entries

⟨constant⟩|⟨constant⟩(⟨NF expression string⟩)

by

⟨constant not in U⟩|⟨constant not in U⟩(⟨NF expression string⟩)|
⟨constant in U⟩(λ⟨NF expression⟩, ⟨NF expression⟩)

and, moreover, we have to write "U-correct NF" instead of "NF" throughout.

The following theorem is intuitively clear, and easily proved formally with the aid of the recursive definition of substitution (section 6).

THEOREM 10.1. If Ω, Σ_1, Σ_2, ... denote U-correct expressions, then

$$\langle S(\ldots, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)\rangle \text{ is } U\text{-correct.}$$

We shall now define the operator β_U on the set of U-correct NF expressions recursively:

(i) If $\langle\Sigma\rangle$ is a single constant or a positive integer, then

$$\langle\beta_U(\langle\Sigma\rangle)\rangle = \langle\Sigma\rangle.$$

(ii) If $\langle\Sigma\rangle = \langle\gamma\rangle(\langle\Sigma_1\rangle, \ldots, \langle\Sigma_k\rangle)$, where $\langle\gamma\rangle$ is a constant not in U, then

$$\langle\beta_U(\langle\Sigma\rangle)\rangle = \langle\gamma\rangle(\langle\beta_U(\langle\Sigma_1\rangle)\rangle, \ldots, \langle\beta_U(\langle\Sigma_k\rangle)\rangle).$$

(iii) If $\langle\Sigma\rangle = \lambda\langle\Sigma_1\rangle$ then

$$\langle\beta_U(\langle\Sigma\rangle)\rangle = \lambda\langle\beta_U(\langle\Sigma_1\rangle)\rangle.$$

(iv) If $\langle\Sigma\rangle = \langle\gamma\rangle(\lambda\langle\Omega\rangle, \langle\Gamma\rangle)$ and $\langle\gamma\rangle \in U$ then (cf. (8.1))

$$\langle\beta_U(\langle\Sigma\rangle)\rangle = \langle S(\ldots, 3, 2, 1, \langle\beta_U(\langle\Gamma\rangle)\rangle; \langle\beta_U(\langle\Omega\rangle)\rangle)\rangle.$$

Needless to say, the effect of β_U on an expression string $\langle\Sigma_1\rangle, \ldots, \langle\Sigma_k\rangle$ is to be defined by $\langle\beta_U(\langle\Sigma_1\rangle)\rangle, \ldots, \langle\beta_U(\langle\Sigma_k\rangle)\rangle$.

11. THEOREMS ON MULTIPLE BETA REDUCTION

THEOREM 11.1. If $\langle\Omega\rangle$, $\langle\Sigma_1\rangle$, $\langle\Sigma_2\rangle$, ... are U-correct, then

$$\langle\beta_U(\langle S(\ldots, \langle\Sigma_2\rangle, \langle\Sigma_1\rangle; \langle\Omega\rangle)\rangle = \\ = \langle S(\ldots, \langle\beta_U(\langle\Sigma_2\rangle)\rangle, \langle\beta_U(\langle\Sigma_1\rangle)\rangle; \langle\beta_U(\langle\Omega\rangle)\rangle)\rangle.$$

PROOF: For easier reading we shall drop the signs $\rangle$ and $\langle$ throughout this proof.

The proof has to be read twice. The first time we deal with the proof of

$$\beta_U S(\ldots, \Sigma_2, \Sigma_1; \Omega) = S(\ldots, \beta_U\Sigma_2, \beta_U\Sigma_1; \beta_U\Omega) \tag{11.1}$$

in the case that the Σ_i are integers. (This case is intuitively clear, but it takes little extra trouble to derive it formally.) In the second reading the result of the first reading can be used.

We apply induction with respect to the structure of Ω, using the definition of substitution as given in section 6. (Note that in the first reading the

induction hypothesis is used only for cases belonging to the first reading.) The cases (i), (ii), (iv) of section 6 are very simple, and so is case (iii) if the constant γ is not in U. We concentrate on the two remaining cases, viz. $\Omega=\lambda\Gamma$ and $\Omega=\gamma(\lambda\Delta, \Gamma)$ with $\gamma\in U$.

If $\Omega=\lambda\Gamma$ we apply (v) of section 6 twice:

$$\beta_U S(\ldots, \Sigma_2, \Sigma_1; \lambda\Gamma)=\beta_U\lambda S(\ldots, \Lambda_2, \Lambda_1, 1; \Gamma), \tag{11.2}$$

$$S(\ldots, \beta_U\Sigma_2, \beta_U\Sigma_1; \lambda\beta_U\Gamma)=\lambda S(\ldots, \Lambda_2^*, \Lambda_1^*, 1; \beta_U\Gamma), \tag{11.3}$$

where Λ_i is given by (6.2), and $\Lambda_i^*=S(\ldots, 4, 3, 2; \beta_U\Sigma_i)$.

By section 10 (iii) and by the induction hypothesis, the right-hand side of (11.2) equals

$$\lambda\beta_U S(\ldots, \Lambda_2, \Lambda_1, 1; \Gamma)=\lambda S(\ldots, \beta_U\Lambda_2, \beta_U\Lambda_1, 1; \beta_U\Gamma). \tag{11.4}$$

In the first reading of the proof the Σ_i and Λ_i are integers, whence $\beta_U\Sigma_i=\Sigma_i$, and therefore $\Lambda_i^*=\Lambda_i=\beta_U\Lambda_i$. So the right-hand sides of (11.3) and (11.4) are equal, hence the left-hand sides of (11.2) and (11.3) are equal. In the second reading of the proof we may use the theorem for the case that the Σ_i are integers; hence

$$\beta_U\Lambda_i=\beta_U S(\ldots, 4, 3, 2; \Sigma_i)=S(\ldots, 4, 3, 2; \beta_U\Sigma_i)=\Lambda_i^*,$$

and the right-hand sides of (11.2) and (11.3) are equal.

The second case we have to deal with, is $\Omega=\gamma(\lambda\Delta, \Gamma)$ with $\gamma\in U$. We have to show

$$\beta_U S(\ldots, \Sigma_2, \Sigma_1; \gamma(\lambda\Delta, \Gamma))=S(\ldots, \beta_U\Sigma_2, \beta_U\Sigma_1; \beta_U\gamma(\lambda\Delta, \Gamma)). \tag{11.5}$$

The right-hand side equals, by section 10 (iv)

$$S(\ldots, \beta_U\Sigma_2, \beta_U\Sigma_1; S(\ldots, 3, 2, 1, \beta_U\Gamma; \beta_U\Delta)).$$

By the formulas on composite substitution (section 7) this is

$$S(\ldots, \beta_U\Sigma_2, \beta_U\Sigma_1, S(\ldots, \beta_U\Sigma_2, \beta_U\Sigma_1; \beta_U\Gamma); \beta_U\Delta). \tag{11.6}$$

The left-hand side of (11.5) equals, according to 6 (iii),

$$\beta_U\gamma(S(\ldots, \Sigma_2, \Sigma_1; \lambda\Delta), S(\ldots, \Sigma_2, \Sigma_1; \Gamma)). \tag{11.7}$$

By 6 (v) we have

$$S(\ldots, \Sigma_2, \Sigma_1; \lambda\Delta)=\lambda\Phi$$

where

$$\Phi=S(\ldots, \Lambda_2, \Lambda_1, 1; \Delta), \quad \Lambda_i=S(\ldots, 3, 2; \Sigma_i).$$

Applying 10 (iv) we can write for (11.7)

$$S(\ldots, 2, 1, \beta_U S(\ldots, \Sigma_2, \Sigma_1; \Gamma); \beta_U\Phi). \tag{11.8}$$

By the induction hypothesis we have

$$\beta_U\Phi=S(\ldots, \beta_U\Lambda_2, \beta_U\Lambda_1, 1; \beta_U\Delta),$$

and so we can apply the formula for composite substitution (section 7) to (11.8); it becomes

(11.9) $$S(\ldots, \Pi_2, \Pi_1; \beta_U \Delta)$$

where

$$\Pi_1 = S(\ldots, 2, 1, \beta_U S(\ldots, \Sigma_2, \Sigma_1; \Gamma); 1) = \beta_U S(\ldots, \Sigma_2, \Sigma_1; \Gamma)$$

$$\Pi_{i+1} = S(\ldots, 2, 1, \beta_U S(\ldots, \Sigma_2, \Sigma_1; \Gamma); \beta_U \Lambda_i) \quad (i = 1, 2, \ldots).$$

We have to show that (11.9) equals (11.6). By the induction hypothesis we have $\Pi_1 = S(\ldots, \beta_U \Sigma_2, \beta_U \Sigma_1; \beta_U \Gamma)$, so it remains to show that $\Pi_{i+1} = \beta_U \Sigma_i$ $(i = 1, 2, \ldots)$.

In the first reading of the proof the Σ_i are positive integers. Therefore the Λ_i are integers > 1; it follows that $\beta_U \Lambda_i = \Lambda_i > 1$, whence $\Pi_{i+1} = \Lambda_i - 1 = \Sigma_i = \beta_U \Sigma_i$.

In the second reading of the proof we may use the result of the first reading:

$$\beta_U \Lambda_i = \beta_U S(\ldots, 3, 2; \Sigma_i) = S(\ldots, 3, 2; \beta_U \Sigma_i),$$

and the formula for Π_{i+1} now results in (cf. section 7)

$$\Pi_{i+1} = S(\ldots, 3, 2, 1; \beta_U \Sigma_i) = \beta_U \Sigma_i.$$

THEOREM 11.2. Let U and V be subsets of the set of constants, and let $\langle \Sigma \rangle$ be both U-correct and V-correct. Then $\langle \beta_U(\langle \Sigma \rangle) \rangle$ is V-correct, $\langle \beta_V(\langle \Sigma \rangle) \rangle$ is U-correct and $\langle \beta_U(\langle \beta_V(\langle \Sigma \rangle) \rangle) \rangle = \langle \beta_V(\langle \beta_U(\langle \Sigma \rangle) \rangle) \rangle$.

PROOF: Again we omit the $\langle$'s and $\rangle$'s. The V-correctness of $\beta_U \Sigma$ is easily proved by recursion: use the definition of β_U of section 10. In 10 (iv) we have to use Theorem 10.1.

By the same recursion we shall prove $\beta_U \beta_V \Sigma = \beta_V \beta_U \Sigma$. The only case where the induction step is non-trivial is the case $\Sigma = \gamma(\lambda \Omega, \Gamma)$ with $\gamma \in U \cup V$. If $\gamma \in U$ we have by 10 (iv)

$$\beta_V \beta_U \Sigma = \beta_V S(\ldots, 3, 2, 1, \beta_U \Gamma; \beta_U \Omega).$$

By Theorem 11.1 this equals

(11.10) $$S(\ldots, 3, 2, 1, \beta_V \beta_U \Gamma; \beta_V \beta_U \Omega).$$

If $\gamma \notin U, \gamma \in V$ we find by 10 (ii), 10 (iii)

$$\beta_V \beta_U \Sigma = \beta_V \gamma(\lambda \beta_U \Omega, \beta_U \Gamma),$$

and by 10 (iv) this equals (11.10). So $\gamma \in U \cup V$ implies that $\beta_V \beta_U \Sigma$ equals (11.10). On behalf of the induction hypothesis (11.10) is symmetric, whence $\beta_V \beta_U \Sigma = \beta_U \beta_V \Sigma$.

12. THE CHURCH-ROSSER THEOREM FOR BETA REDUCTION

We consider an NF expression Σ with a single constant A that can be used for β-reduction. We label all A's in Σ so that they become all different.

Next we take a subset U of the labelled Λ's, we apply β_U and then remove the labels. This gives an NF expression Σ'. We say that Σ' is a multiple reduction of Σ', and we write $\Sigma \geqslant_m \Sigma'$. If U has only one element, and if that element has just one occurrence in Σ, the reduction is called single, and we write $\Sigma \geqslant_s \Sigma'$.

If Σ_1 and Σ_2 satisfy either $\Sigma_1 \geqslant_s \Sigma_2$ or $\Sigma_2 \geqslant_s \Sigma_1$, we write $\Sigma_1 \sim \Sigma_2$. The Church-Rosser theorem for beta reduction says: If $\Sigma_1 \sim \Sigma_2 \sim \ldots \sim \Sigma_n$ then there are $\Lambda_1, \ldots, \Lambda_k$ and $\Pi_1, \ldots, \Pi_h$ with

$$\Sigma_1 \geqslant_s \Lambda_1 \geqslant_s \ldots \geqslant_s \Lambda_k,\ \Sigma_n \geqslant_s \Pi_1 \geqslant_s \ldots \geqslant_s \Pi_h,\ \Lambda_k = \Pi_h.$$

This can now be proved as follows. From theorem 11.2 we easily obtain: if $\Sigma_1 \geqslant_m \Sigma_2$, $\Sigma_1 \geqslant_m \Sigma_3$ then there is a Σ_4 with $\Sigma_2 \geqslant_m \Sigma_4$, $\Sigma_3 \geqslant_m \Sigma_4$. Moreover it can be shown: If $\Sigma \geqslant_m \Lambda$ then there is a sequence

$$\Sigma \geqslant_s \Sigma_1 \geqslant_s \Sigma_2 \geqslant_s \ldots \geqslant_s \Sigma_m = \Lambda.$$

(Actually, if every element of U occurs at most once in the U-correct expression Σ, then we can arrange the elements of U as $u_1, \ldots, u_m$ in such a way that

$$\beta_{\{u_m\}} \ldots \beta_{\{u_1\}} \Sigma = \beta_U \Sigma).$$

The Church-Rosser theorem now follows by a trivial reduction argument.

The above proof can be easily adapted to lambda calculus with expressions as types (see section 13).

13. Notation in AUTOMATH

The mathematical language AUTOMATH (see [2]) has lambda calculus with types, and these types are again expressions. That is, instead of λ_x we have things that can be visualized as $\lambda_{x \in \langle \Phi \rangle} \langle \Omega \rangle$, where Φ and Ω denote name-carrying expressions. We may think of x to be a variable of the type $\langle \Phi \rangle$. It is clear that we do not want x to have any binding influence on $\langle \Phi \rangle$. In order to achieve this, we create a new lingual constant T (just like we added A to our set of constants in section 3), and we write

$$T(\langle \Phi \rangle, \lambda_x \langle \Omega \rangle) \tag{13.1}$$

instead of $\lambda_{x \in \langle \Phi \rangle} \langle \Omega \rangle$. Now (13.1) can be transformed into a namefree expression just like any other name-carrying expression.

The actual notation in AUTOMATH is different. Instead of (13.1) AUTOMATH uses $[x, \langle \Phi \rangle] \langle \Omega \rangle$, and for the application $A(\langle \Phi \rangle, \langle \Gamma \rangle)$ AUTOMATH uses $\{\langle \Gamma \rangle\} \langle \Phi \rangle$.

14. Algorithms

An algorithm for turning an NF expression into a name-carrying one, can be described on the basis of the recursive definition of substitution in section 6. Let $\langle \Omega \rangle$ be an NF expression. Take a free variable list $\ldots, x_3, x_2, x_1$ consisting of distinct letters which do not belong to our

alphabet of constants. Now add these x_i to that alphabet, and evaluate

$$\langle S(\ldots, x_3, x_2, x_1; \langle \Omega \rangle) \rangle$$

This is a namefree expression; if we proclaim the x_i's to be variables again, it becomes an intermediate expression where the free variables have names but the bound variables are nameless. If we want to have names for the bound variables too, we have to modify S slightly. We take an infinite store of letters $y_1, y_2, \ldots$ (different from the x_i's and different from the constants), and we take a modified form of (6.1). Any time we get to apply (6.1) we take a fresh y (i.e. one that has not been used before) and we replace the right-hand side of (6.1) by

$$\lambda_y \langle S(\ldots, \langle A_3 \rangle, \langle A_2 \rangle, \langle A_1 \rangle, y; \langle \Gamma \rangle) \rangle.$$

It is not very hard either to give algorithms that transform name-carrying expressions into namefree ones. This can be done if a free variable list is given (and then it has to be checked, during the execution of the algorithm, whether this list is adequate), but we can also write an algorithm that produces a free variable list itself. For the case of the first-mentioned possibility we give a brief description of the crucial steps. Let $\ldots, x_3, x_2, x_1$ be a free variable list, and let $\langle \Omega \rangle$ be the name-carrying expression we want to transfer into the namefree expression $\langle \Omega^* \rangle$. If $\langle \Omega \rangle$ equals one of the x's, then $\langle \Omega^* \rangle$ is an integer, viz. the index of that x. If $\langle \Omega \rangle$ is a variable, but not one of the x's, the answer is "free variable list was wrong". If $\langle \Omega \rangle = \lambda_y \langle \Gamma \rangle$ then we transform $\langle \Gamma \rangle$ into the nameless expression $\langle \Gamma^* \rangle$ by means of the free variable list $\ldots, x_3, x_2, x_1, y$, and we have $\langle \Omega^* \rangle = \lambda \langle \Gamma^* \rangle$. The other cases (the cases (i) $\langle \Omega \rangle = \langle \Omega_1 \rangle, \langle \Omega_2 \rangle$, (ii) $\langle \Omega \rangle =$ a constant, (iii) $\langle \Omega \rangle = \langle \beta \rangle (\langle \Omega_1 \rangle)$) are very easy.

Technological University,
Department of Mathematics,
Eindhoven, Netherlands.

REFERENCES

1. Barendregt, H. P., Some extensional models for combinatory logics and λ-calculi. Doctoral Thesis, Utrecht 1971.
2. Bruijn, N. G. de, The mathematical language AUTOMATH, its usage, and some of its extensions, Symposium on Automatic Demonstration (Versailles December 1968), Lecture Notes in Mathematics, Vol. 125, Springer-Verlag, 29–61 (1970).
3. Church, A., The Calculi of Lambda Conversion, Annals of Math. Studies, vol. 6, Princeton University Press, 1941.
4. Curry, H. B. and R. Feys, Combinatory Logic. North-Holland Publishing Company, Amsterdam 1958.