International Journal of Computer and Information Sciences, Vol. 1, No. 2, 1972

A CUCH-Machine: The Automatic Treatment of Bound Variables

Corrado Böhm[1] and Mariangiola Dezani[1]

Received December 15, 1971

This paper describes a machine for reducing a λ-formula (explicitly given or implicitly by a system of recursive equations) to principal β-η-normal form, with particular attention to the memory structures needed for the purpose, and with some important features: (1) any kind of collision is permitted; (2) the processing of subformulas which will be thrown away [e.g., $((\lambda xy)x)$ in $((\lambda yz)(\lambda xy)x)$] is avoided; (3) there is no need to introduce any fixed point operator like $\mathcal{O}$, etc. The machine structure entails: (1) some store to memorize as side-effects assignment statements with the r.h.s. of a given shape. (2) a number of stacks, one for every λ in the initial formula, partitioned naturally in classes (chains). These stacks admit as entries only words representing variables and they are peculiar in that the operations admitted on the top are *writing* and *erasing* and the operations admitted on the pseudo-top are *reading*, *read-protecting*, and *resetting readability* (the last two operations are chain operations). This structure is critically motivated. (3) A workstack. (4) A pointerstack. The computation runs through four phases: β-generation, β-run, η-generation, η-run. Every generation- (run-) phase is rather recognition- (transformation-) oriented, but we found it more stimulating to emphasize technical similarities rather than methodological differences. Every phase is described and four examples are extensively developed.

1. INTRODUCTION

From a technical point of view, this paper perhaps fills a gap, in the sense that there has not been published so far to our knowledge a complete algorithmic treatment of the so called α-rule of the λ-calculus (that is, the redenomination rule for variables).[2] From a less technical and more general

[1] Istituto di Scienza della Informazione, Università di Torino, Italy.
[2] The reader will find the fundamental notions about CUCH in the appendix.

point of view, our method can be classified as an attempt to smooth the dualism, too much emphasized today, between the syntax and semantics of a programming language. It is true that the proposed algorithm for both β- and η-reductions has been split into two subsequent phases: *generation* and *run*, the first of which can be thought as a "parsing" or "syntactic recognition," while the second can be considered as a "semantic" phase of interpretation. Neverhteless, we try to point out the common features of both phases, which is given by the creation of side-effects interpreting some assignment statements, that is, the creation of some pair "name:value." Besides, there has been introduced the notion of relative level as subscript for every variable of the initial λ-formula in order to identify unequivocally the subformula where such a variable occurs.

Many authors[1–5,7–11] would probably easily agree that the λ-calculus can be chosen as a simple theoretical model for several other programming languages, some major complications in the latter being mainly of a technical nature. To illustrate the previous statement, there could be produced, among other things, (a) the λ-symbol as a universal type declarator of variables, (b) a formula of type

$$(\lambda x(\lambda y(\lambda zM)))$$

as declaration of the procedure M with formal variables x, y, z, and (c) a formula of type

$$((\lambda xF)G)$$

as the command to apply the procedure F to the actual argument G.

Surprisingly enough, the outcome of this paper displays an inherent complication of the minimal mechanical devices needed for an efficient implementation of the reduction[3] algorithm for the pure λ-calculus.[4]

A little clarification is required for the use of the expressions "efficient" and "mechanical devices" in the previous statement. Ausiello[1] implemented the reduction algorithm in the ROMALISP language. He found out that

[3] Such a reduction, called "normal reduction," is defined in Curry (Ref. 5, p. 140) and it consists in reiterating left-to-right applications of β-η-rules.

[4] Many authors ignore the need for the application of α-rule prior to some β-reductions or, at least, as with Wegner (Ref. 10, pp. 205–208), they state this fact too vaguely to be sure that their implementation is correct. We do not impose any restrictions on the choice of names of variables of the initial λ-formula, allowing any kind of collision (see, for example, Table I). It is not possible to avoid variable collisions by a preliminary pass renaming all bound variables so that they are distinct from each other and from the free variables. Indeed, a λ-formula without variable collisions may generate through a single application of a β-reduction rule a formula where collisions occur. For example,

$$((\lambda x(xx))(\lambda yy)) \xrightarrow{\beta} ((\lambda yy)(\lambda yy))$$

β-rule could be more efficiently executed by assigning values to bound variables instead of executing a material replacement in all occurrences of these variables. The greatest efficiency of this assignment method is shown trivially in the case where a large formula is to be substituted for several occurrences of the same variable and some of these are bound to disappear. In this paper, we try to generalize this assignment method to all parts of the algorithm identifying, as Scott(8) and Strachey(9) did, the execution of assignment statements with the creation of side-effects in some store.

The mechanical device we are presenting is in the spirit of automata theory, i.e., we have tried to identify the whole machinery required with some kind of specialized Turing machine. It comes out that we need several stacks (one for every occurrence of λ in the initial formula) partitioned in classes of read-protection.[5] This means that at a certain moment, it may happen that all entries at a same "height" in one stack class are protected from reading and that at some subsequent moment, this protection will be removed.

In addition to previously mentioned devices, we need two more classical structures: a workstack for characters and subscripted variables and a stack of pointers, both with destructive reading. In order to go deeper into the motivations of the choices made in this paper, let us assume an Algol-like point of view. First, the efficiency claims can be fulfilled by evaluating a variable as late as possible: this means that all variables in every procedure are initialized *by name*. Second, remembering remarks (a) and (b), there is a need for a stack for variable values because a procedure may call itself recursively. Indeed, in this case, different values are assigned to the same formal variable, which must be stored for reading at the right moment. See, for example, Table II.

The stacks of the formal variables associated with the same procedure form a single class in a very natural way. Third, the need for read-protection is illustrated by the successive reduction configurations of the following formula:

$$((\lambda y(yx))(\lambda x(xx)))$$

In this formula, $(\lambda y(yx))$ is a procedure with formal parameter y to be applied to the argument $(\lambda x(xx))$ (which itself is a procedure). The next step is

$$((\lambda x(xx))x)$$

Here, we are faced with the problem of acting chronologically as follows: (1) to assign x as value to x by writing it in the x stack; (2) to read the stack

[5] An alternative solution (suggested to us by H. R. Strong) could be to collect in one stack the stacks belonging to the same class. The necessary information could be recovered allowing every entry to be, instead of a single value, a set of pairs "stack-name:value."

value for x in order to take into account the first occurrence of x; (3) to avoid a loop; (4) to read the stack value for x in order to take into account the second occurrence of x; and (5) again to avoid a loop.

We solved this problem by protecting the top of the x stack from reading at step 3, by releasing this protection at step 4, and by resetting it at step 5.

When we replace a formal variable by its value, we really jump from one procedure M to another. We thus need to forget temporarily not only the value just replaced, but also the values already assigned to the other formal variables of the same procedure M. The operations of *read-protecting* and *resetting readability* must then act on all variable-stacks belonging to the same class.

All this is shown in Table I.

In sections 2 and 3, the algorithm is given for the reduction to the β-normal form of an explicit λ-formula. A theorem due to Curry and Feys (Ref. 5, pp. 132–135) permits us to obtain the β-η-principal normal form from the β-principal normal form executing successively all possible η-reductions, whose algorithm is the object of Section 4. The above-mentioned sections need the introduction of one subscripted auxiliary variable as name for different subformulas. Section 5 allows the introduction of any number of auxiliary variables representing different formulas (possibly combinators) and the treatment of formulas implicitly (recursively) defined. In developing these algorithms, we felt that in this context any dichotomy between syntax and semantics or between compilation and interpretation was more misleading than useful.

We think that the CUCH-machine described here can be considered as a first step toward the identification of the structure of interpreters for actual programming languages provided with procedures called by name and with minimal constraints on the choice of identifiers.

2. β-GENERATION

2.1. β-Generation Statements

According to the particular syntax of λ-calculus, every subformula of the given formula is limited by a pair of parentheses; hence let us say that a subformula is of n-level if the corresponding close-parenthesis is the nth close-parenthesis scanning the whole formula from left to right.

Consequently, the right subformula of the formula of n-level is of $(n - 1)$-level. Let us assign the $(n - 1)$-level to the variables, prefixed or not by λ-symbol, which are inside a formula of n-level. We have thus defined the level of every subformula of the original formula (itself included).

The notion of level permits us to generate the formula by a sequence of assignment statements, where we associate to every variable its level as subscript. To this aim, it is enough to introduce the auxiliary variable $F[i]$ as name of the subformula not atomic of i-level.

Example. Let us consider the λ-formula $((\lambda x(\lambda yx))(\lambda xx))$. We have the following generation:

Assignment statements	Type of assignment statement
$F[1]$: $(\lambda y[0]x[0])$	Abstraction
$F[2]$: $(\lambda x[1]F[1])$	Abstraction
$F[3]$: $(\lambda x[2]x[2])$	Abstraction
$F[4]$: $(F[2]F[3])$	Application

Note. We use the symbol : in order to separate the lhs from the rhs of an assignment statement.

Let us notice that the whole formula is therefore represented by $F[n]$, where n is the last subscript used.

2.2. Stacks and Stack-Chain Creation

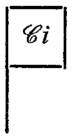
Every creation of an abstraction statement is immediately followed by the creation of a variable-stack identified by the name of the just abstracted variable. The effect of a stack creation will be indicated by the notation

name of variable [level number]:

Let us define as *abstraction chain* a sequence of abstraction assignment statements where every left-hand side is a part of the right-hand side of the next abstraction.

It follows that: (a) Every abstraction statement belongs to exactly one chain; (b) the same happens to the corresponding variable-stack; (c) the chains are partition classes of the set of variable-stacks.

The effect of creating a chain of stacks will be indicated by the notation



$(i = 1, 2, 3, \ldots)$

Example. We consider the same λ-formula of the previous example:

Assignment statements	Stack chains				
$F[1]$: $(\lambda y[0]x[0])$	$\boxed{\mathscr{C}1}$	$y[0]$:			
$F[2]$:$(\lambda x[1]F[1])$		"	$x[1]$:		
$F[3]$:$(\lambda x[2]x[2])$		"	"	$\boxed{\mathscr{C}2}$	$x[2]$:
$F[4]$:$(F[2]F[3])$		"	"		"

2.3. Computing the Generation

The machinery needed for the execution of the generation phase can summarized as follows:

(a) Workstack with following entries:) ("upper case or lower case variable, possibly subscripted."

(b) Transformation rules for the current subformula just completed on the top of the workstack. The application of these rules entails the side-effects corresponding to the creation of assignment statements, of variable-stacks and of variable-stack chains.

(c) A level counter (LC).

2.3.1. *Algorithm of Generation*

Let us denote upper case *or* lower case variables with Φ, Ψ,... and lower case variables with φ, ψ,... .

*P*1. (Initial conditions): LC $\leftarrow$ 1, the workstack is empty.

*P*2. (Copy): We copy the initial λ-formula scanning from the left on the workstack until a subformula is completed.

*P*3. (Distinction between abstraction and application): If the subformula just completed is an application, go to *P*8.

*P*4. (Is this abstraction the first of the chain?): If the right subformula of the subformula just completed is an abstraction, go to *P*7.

*P*5. (Stack and stack chain creation): We write as side-effect

$$\varphi[\mathrm{LC} - 1]:$$

(where φ is the abstracted variable). This variable-stack must be assigned to the newly created stack chain $\mathscr{C}(m + 1)$ (if $\mathscr{C}m$ is the last stack chain created). The first stack must belong to the chain $\mathscr{C}1$.

*P*6. (Creation of an assignment statement and erasure chain-symbol $\mathscr{K}$):

$$F[\mathrm{LC}]\colon (\lambda\varphi[\mathrm{LC}-1]\,\mathscr{K}m\,\Phi[\mathrm{LC}-1])$$

go to *P*9.

*P*7. (Stack creation): We write as side-effect

$$\varphi[\mathrm{LC}-1]\colon$$

(where φ is the abstracted variable). This variable-stack must be assigned to the last stack chain created.

*P*8. (Alternative creation of an assignment statement): We write as side-effect

F[LC]: "subformula just completed in which the level (LC $-$ 1) is assigned to every lower case variable."

*P*9. (Replacement): On the workstack, the subformula just completed is replaced by F[LC].

*P*10. (Is it finished?): If there is no more symbol to be copied, STOP. Otherwise, LC $\leftarrow$ LC $+$ 1, go to *P*2.

Let us follow the application of this algorithm to the λ-formula $((\lambda x(\lambda yx))(\lambda xx))$:

	Side-effects			
Workstack	Assignment statements	Stack chains		
$((\lambda x(\lambda yx)$	$F[1]\colon(\lambda y[0]\mathscr{K}1x[0])$	$\boxed{\mathscr{C}1}$ $y[0]\colon$		
$((\lambda xF[1])$	$F[2]\colon(\lambda x[1]F[1])$	”	$x[1]\colon$	
$(F[2](\lambda xx)$	$F[3]\colon(\lambda x[2]\mathscr{K}2x[2])$	”	”	$\boxed{\mathscr{C}2}$ $x[2]\colon$
$(F[2]F[3])$	$F[4]\colon(F[2]F[3])$	”	”	”
$F[4]$		”	”	”

3. β-RUN

3.1. Initial Conditions

At the end of the generation phase, the following situation is produced: (a) the workstack possesses as unique entry the variable representing the whole λ-formula to be reduced; (b) assignment statements or side-effect giving the generation of the λ-formula; (c) empty stacks of variables; (d) an empty pointerstack, which is a pushdown list.[6]

In the run phase, we will need both terminal and nonterminal variables. The terminal variables will be represented by the letter v suffixed with a natural number in decimal notation or a lower case letter (representing free variables).The nonterminal variables will be represented by a lower case or upper case letter subscripted by a level index.

The workstack is essentially a pushdown list with following possible entries:) ("not terminal variable" "terminal variable."

The variable-stacks created in the generation phase are more specialized than those defined by Ginsburg *et al.*[6] What is peculiar to our stacks is the existence of rules unequivocally determining the reading of one entry not necessarily on the top. Every reading of one entry is immediately followed by a locking, called "read-protection," of all entries belonging to the chain pertinent to that entry. Let us define as a pseudo-top the first unprotected entry from the top. If at some instant some entries of one stack are protected, *writing on the stack* will mean writing on the top and *reading on the stack* will mean reading the pseudo-top of the stack. Symmetrically to the read protection there is a reset operation (of readability) acting again on the pseudo-top of a whole chain.

From the point of view of implementation, the information structure of the read-protected stack is a pushdown list where every entry has a flag bit (0 for readable, 1 for not readable). Later on, we will bar the unreadable entries.

The formula of an i-level represented by $F[i]$ generally entails one pointer i, which points to the right subformula of the formula itself (a possible occurrence of the symbol $\mathscr{K}m$ is considered as belonging to the right subformula).

3.2. Run Evaluation Rules

The following sequence of rules will act like a Markov algorithm. The only difference is the rule 3.2.4, which looks forward and opens more nesting possibilities.

The choice of the next rule to be applied is unequivocally determined by the top entries of the workstack.

3.2.1. $\mathscr{K}m$ Rule

"$\mathscr{K}m$" on the top of the workstack is immediately conveyed on the pointerstack top.

[6] I.e., a list writable, readable/erasable only at the top.

3.2.2. ((λ Rule

"$((\lambda$" on the top of the workstack causes writing on the workstack until and including the close-parenthesis corresponding to the first open, by a (destructive) reading of pointerstack. In this way, we obtain

$$((\lambda\varphi[i]\ \Phi[i])\ \Phi[j])$$

A possible "$\mathscr{K}m$" between $\varphi[i]$ and $\Phi[i]$ is immediately conveyed on the top of the pointerstack.
This formula is now interpreted as an assignment statement. Therefore, we write $\Phi[j]$ on the stack $\varphi[i]$, leaving on the workstack only the formula $\Phi[i]$.

3.2.3. (λφ[i] Rule

The value of

$$\text{"}(\lambda\varphi[i]\text{"}$$

is

$$(\lambda vn$$

where n is the smallest nonnegative integer j such that vj is not in any stack. Moreover, the stack $\varphi[i]$ is loaded with the terminal variable vn.

3.2.4. φ[i] Rule

The value of a nonterminal variable

$$\text{"}\varphi[i]\text{"}$$

is the result of a read operation on the stack $\varphi[j]$ where j is the smallest nonnegative integer not less than i. If this stack does not exist or is empty, the value of $\varphi[i]$ is φ (this condition is verified only in the case of a variable occurring free in the initial λ-formula).

Let us suppose that we read the nth entry from the bottom of the stack $\varphi[j]$ belonging to the chain $\mathscr{C}m$. Then, we must execute a read-protection operation for all n entries of the stack belonging to the chain $\mathscr{C}m$ and we must write on the pointerstack $\mathscr{C}m\mathscr{E}n$.

3.2.5. F[i] Rule

"$F[i]$" on the top of the workstack causes the replacement process with its value to be activated; every entry of that value will be written from left to right on the workstack until one of the rules 3.2.1–3.2.5 is applicable.

If the next applicable rule is one of 3.2.1, 3.2.3–3.2.5, we write on the pointerstack the interruption point consecutively to a left subformula, the ")" symbol consecutively to a right subformula.

3.2.6. β-Completion Rule

If none of the previous rules is applicable, we execute a destructive reading of the pointerstack:

(a) A pointer on the top of the pointerstack causes the writing on the workstack from the corresponding interruption point.

(b) "$\mathscr{C}m\&n$" on the top of the pointerstack causes the execution of a reset operation of readability for all n entries of the stacks belonging to the chain $\mathscr{C}m$.

(c) "$\mathscr{K}m$" on the top of the pointerstack causes the erasure of all tops of the stacks belonging to the chain $\mathscr{C}m$.

(d) ")" on the top of the pointerstack is conveyed on the workstack.

(e) If the pointerstack is empty, the β-reduction of the initial λ-formula is completed.

3.3. Run Machinery

The machinery needed for the execution of the run phase can be summarized as follows: (a) workstack; (b) assignment statements produced during the generation phase; (c) variable-stacks and pertinent chains; (d) pointerstack; (e) run evaluation rules 3.2.1–3.2.6 like a Markov algorithm.

Tables I and II contain the whole reduction to normal form of two λ-formulas.

Table I. Program: ((λy(yx))(λx(xx)))

β-Generation Workstack	Assignment statements	Stack chains	
$((\lambda y(yx)$	$F[1]{:}(y[0]x[0])$		
$((\lambda yF[1])$	$F[2]{:}(\lambda y[1]\mathscr{K}1F[1])$	$\boxed{\mathscr{C}1}$ $y[1]{:}$	
$(F[2](\lambda x(xx)$	$F[3]{:}(x[2]x[2])$	,,	
$(F[2](\lambda xF[3])$	$F[4]{:}(\lambda x[3]\mathscr{K}2F[3])$	,,	$\boxed{\mathscr{C}2}$ $x[3]{:}$
$(F[2]F[4])$	$F[5]{:}(F[2]F[4])$	,,	,,
$F[5]$		,,	,,

Table continued

Table I (continued)

β-RUN

Rule number Workstack	Pointerstack	Stack chains	
$F[5]$	—	$\boxed{\mathscr{C}1}$ $y[1]$:	$\boxed{\mathscr{C}2}$ $x[3]$:
.5			
$(F[2]$	5	,,	,,
.5			
$((\lambda y[1]F[1])F[4])$	$\mathscr{K}1$	$y[1]{:}F[4]$	,,
.2			
$F[1]$	$\mathscr{K}1$	,,	,,
.5			
$(y[0]$	$1\ \mathscr{K}1$	,,	,,
.4			
$(F[4]$	$\mathscr{C}1\ \mathscr{E}11\ \mathscr{K}1$	$y[1]{:}\overline{F[4]}$	,,
.5			
$((\lambda x[3]F[3])x[0])$	$\mathscr{K}2\ \mathscr{K}1$	$y[1]{:}F[4]$	$x[3]{:}x[0]$
.2			
$F[3]$	$\mathscr{K}2\ \mathscr{K}1$	,,	,,
.5			
$(x[2]$	$3\ \mathscr{K}2\ \mathscr{K}1$	,,	,,
.4			
$(x[0]$	$\mathscr{C}2\ \mathscr{E}1\ 3\ \mathscr{K}2\ \mathscr{K}1$	,,	,,
.4			
$(x$	$\mathscr{C}2\ \mathscr{E}1\ 3\ \mathscr{K}2\ \mathscr{K}1$	,,	
.6			
$(xx[2]$	$)\ \mathscr{K}2\ \mathscr{K}1$	,,	$x[3]{:}x[0]$
.4			
$(xx[0]$	$\mathscr{C}2\ \mathscr{E}1\ )\ \mathscr{K}2\ \mathscr{K}1$	,,	$x[3]{:}\overline{x[0]}$
.4			
$(xx$	$\mathscr{C}2\ \mathscr{E}1\ )\ \mathscr{K}2\ \mathscr{K}1$	,,	,,
.6			
(xx)	$\mathscr{K}2\ \mathscr{K}1$	,,	$x[3]{:}x[0]$
.6			
(xx)	$\mathscr{K}1$	,,	$x[3]$:
.6			
(xx)	—	$y[1]$:	,,

Table II. Program: $((\lambda x(\lambda y(x(xy))))(\lambda x(\lambda y(x(xy)))))$

β-Generation

Workstack	Assignment statements	Stack chains					
$((\lambda x(\lambda y(x(xy)$	$F[1]:(x[0]y[0])$						
$((\lambda x(\lambda y(xF[1])$	$F[2]:(x[1]F[1])$						
$((\lambda x(\lambda yF[2])$	$F[3]:(\lambda y[2]\mathscr{K}1F[2])$	$\boxed{\mathscr{C}1}$	$y[2]$:				
$((\lambda xF[3])$	$F[4]:(\lambda x[3]F[3])$		,,	$x[3]$:			
$(F[4](\lambda x(\lambda y(x(xy)$	$F[5]:(x[4]y[4])$		,,	,,			
$(F[4]((\lambda x(\lambda y(xF[5])$	$F[6]:(x[5]F[5])$		,,	,,			
$(F[4](\lambda x(\lambda yF[6])$	$F[7]:(\lambda y[6]\mathscr{K}2F[6])$		,,	,,	$\boxed{\mathscr{C}2}$	$y[6]$:	
$(F[4]((\lambda xF[7])$	$F[8]:(\lambda x[7]F[7])$		,,	,,		,,	$x[7]$:
$(F[4]F[8])$	$F[9]:(F[4]F[8])$		,,	,,		,,	,,
$F[9]$			,,	,,		,,	,,

β-Run

Rule number / Workstack	Pointerstack	Stack chains					
$F[9]$	—	$\boxed{\mathscr{C}1}$	$y[2]$:	$x[3]$:	$\boxed{\mathscr{C}2}$	$y[6]$:	$x[7]$:
.5							
$(F[4]$	9		,,	,,		,,	,,
.5							
$((\lambda x[3]F[3])F[8])$	—		,,	,,		,,	,,
.2							
$F[3]$	—		,,	$x[3]:F[8]$		,,	,,
.5							
$(\lambda y[2]$	3		,,	,,		,,	,,
.3							
$(\lambda v0$	3		$y[2]:v0$	,,		,,	,,

.6.1					
$(\lambda v0 F[2]$	$\mathscr{K}1\,)$	,,	,,	,,	,,
.5					
$(\lambda v0(x[1]$	$2\ \mathscr{K}1\,)$	,,	,,	,,	,,
.4					
$(\lambda v0(F[8]$	$\mathscr{C}1\ \mathscr{E}1\ 2\ \mathscr{K}1\,)$	$y[2]{:}\overline{v0}$	$x[3]{:}\overline{F[8]}$	,,	,,
.5					
$(\lambda v0((\lambda x[7]F[7])F[1])$	$\mathscr{K}1\,)$	$y[2]{:}v0$	$x[3]{:}F[8]$	,,	,,
.2					
$(\lambda v0 F[7]$	$\mathscr{K}1$	,,	,,	,,	$x[7]{:}F[1]$
.5					
$(\lambda v0(\lambda y[6]$	$7\ \mathscr{K}\ 1\,)$	,,	,,	$y[6]{:}v1$	,,
.3					
$(\lambda v0(\lambda v1$	$7\ \mathscr{K}1\,)$	,,	,,	,,	,,
.6.1					
$(\lambda v0(\lambda v1 F[6]$	$\mathscr{K}2\,)\ \mathscr{K}1\,)$	,,	,,	,,	,,
.5					
$(\lambda v0(\lambda v1(x[5]$	$6\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	,,	,,	,,	,,
.4					
$(\lambda v0(\lambda v1(F[1]$	$\mathscr{C}2\ \mathscr{E}1\ 6\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	,,	,,	$y[6]{:}\overline{v1}$	$x7{:}\overline{F[1]}$
.5					
$(\lambda v0(\lambda v1((x[0]$	$1\ \mathscr{C}2\ \mathscr{E}1\ 6\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	,,	,,	,,	,,
.4					
$(\lambda v0(\lambda v1((F[8]$	$\mathscr{C}1\ \mathscr{E}1\ 1\ \mathscr{C}2\ \mathscr{E}1\ 6\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	$y[2]{:}\overline{v0}$	$x[3]{:}\overline{F[8]}$	,,	,,
.5					
$(\lambda v0(\lambda v1(((\lambda x[7]F[7])y[0])$	$\mathscr{C}2\ \mathscr{E}1\ 6\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	$y[2]{:}v0$	$x[3]{:}F[8]$	,,	,,
.2					
$(\lambda v0(\lambda v1(F[7]$	$\mathscr{C}2\ \mathscr{E}1\ 6\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	,,	,,	,,	$x[7]{:}\overline{F[1]}\ y[0]$
.5					
$(\lambda v0(\lambda v1((\lambda y[6]F[6])F[5])$	$\mathscr{K}2\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	,,	,,	$y[6]{:}v1F[5]$	$x[7]{:}F[1]\ y[0]$
.2					
$(\lambda v0(\lambda v1 F[6]$	$\mathscr{K}2\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	,,	,,	,,	,,
.5					
$(\lambda v0(\lambda v1(x[5]$	$6\ \mathscr{K}2\ \mathscr{K}2\,)\ \mathscr{K}1\,)$	,,	,,	,,	,,

Table continued

Table II (continued)

β-Run

Rule number	Workstack	Pointerstack	Stack chains $\boxed{\mathscr{C}1}$		Stack chains $\boxed{\mathscr{C}2}$	
.4	$(\lambda v0(\lambda v1(y[0]$	$\mathscr{C}2\ \mathscr{E}2\ 6\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	$y[2]{:}v0$	$x[3]{:}F[8]$	$y[6]{:}v1\ \overline{F[5]}$	$x[7]{:}F[1]\ \overline{y[0]}$
.4.5	$(\lambda v0(\lambda v1(v0F[5]$	$)\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}v1\ F[5]$	$x[7]{:}F[1]y[0]$
.5	$(\lambda v0(\lambda v1(v0(x[4]$	$5\ )\ \mathscr{K}\ 2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	,,	,,
.4	$(\lambda v0(\lambda v1(v0(y[0]$	$\mathscr{C}2\ \mathscr{E}2\ 5\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ 1\ )$	,,	,,	$y[6]{:}v1\ \overline{F[5]}$	$x[7]{:}F[1]\ \overline{y[0]}$
.4	$(\lambda v0(\lambda v1(v0(v0$	$\mathscr{C}1\ \mathscr{E}1\ \mathscr{C}2\ \mathscr{E}2\ 5\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ 1\ )$	$y[2]{:}\overline{v0}$	$x[3]{:}\overline{F[8]}$	,,	,,
.6	$(\lambda v0(\lambda v1(v0(v0y[4]$	$)\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	$y[2]{:}v0$	$x[3]{:}F[8]$	$y[6]{:}v1\ F[5]$	$x[7]{:}F[1]\ y[0]$
.4	$(\lambda v0(\lambda v1(v0(v0F[5]$	$\mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}v1\ \overline{F[5]}$	$x[7]{:}F[1]\ \overline{y[0]}$
.5	$(\lambda v0(\lambda v1(v0(v0(x[4]$	$5\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	,,	,,
.4	$(\lambda v0(\lambda v1(v0(v0(F[1]$	$\mathscr{C}2\ \mathscr{E}1\ 5\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}\overline{v1}\ \overline{F[5]}$	$x[7]{:}\overline{F[1]}\ \overline{y[0]}$
.5	$(\lambda v0(\lambda v1(v0(v0((x[0]$	$\mathscr{C}2\ \mathscr{E}1\ 5\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	,,	,,
.4	$(\lambda v0(\lambda v1(v0(v0((F[8]$	$\mathscr{C}1\ \mathscr{E}1\ 1\ \mathscr{C}2\ \mathscr{E}1\ 5\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	$y[2]{:}\overline{v0}$	$x[3]{:}\overline{F[8]}$	,,	,,
.5	$(\lambda v0(\lambda v1(v0(v0(((\lambda x[7]F[7])y[0])$	$\mathscr{C}2\ \mathscr{E}1\ 5\ \mathscr{C}2\ \mathscr{E}2\ ))\ \mathscr{K}2\ \mathscr{K}2\ \ )\ \mathscr{K}1\ )$	$y[2]{:}v0$	$x[3]{:}F[8]$	,,	,,
.2	$(\lambda v0(\lambda v1(v0(v0(F[7]$	,,	,,	,,	,,	$x[7]{:}\overline{F[1]}\ \overline{y[0]}\ y[0]$

.5	$(\lambda v0(\lambda v1(v0(v0((\lambda y[6]F[6])y[4])$	$\mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	$y[2]{:}v0$	$x[3]{:}F[8]$	$y[6]{:}v1\ \overline{F[5]}\ y[4]$	$x[7]{:}F[1]\ \overline{y[0]}\ y[0]$
.2	$(\lambda v0(\lambda v1(v0(v0F[6]$	,,	,,	,,	,,	,,
.5	$(\lambda v0(\lambda v1(v0(v0(x[5]$	$6\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2)\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	,,	,,
.4	$(\lambda v0(\lambda v1(v0(v0(y[0]$	$\mathscr{C}2\ \mathscr{E}3\ 6\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}v1\ \overline{F[5]}\ \overline{y[4]}$	$x[7]{:}F[1]\ \overline{y[0]}\ \overline{y[0]}$
.4	$(\lambda v0(\lambda v1(v0(v0(v0$	$\mathscr{C}1\ \mathscr{E}1\ \mathscr{C}2\ \mathscr{E}3\ 6\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	$y[2]{:}\overline{v0}$	$x[3]{:}\overline{F[8]}$	,,	,,
.6	$(\lambda v0(\lambda v1(v0(v0(v0F[5]$	$)\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	$y[2]{:}v0$	$x[3]{:}F[8]$	$y[6]{:}v1\ \overline{F[5]}\ y[4]$	$x[7]{:}F[1]\ \overline{y[0]}\ y[0]$
.5	$(\lambda v0(\lambda v1(v0(v0(v0(x[4]$	$5\ )\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	,,	,,
.4	$(\lambda v0(\lambda v1(v0(v0(v0(y[0]$	$\mathscr{C}2\ \mathscr{E}3\ 5\ )\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}v1\ \overline{F[5]}\ \overline{y[4]}$	$x[7]{:}F[1]\ \overline{y[0]}\ \overline{y[0]}$
.4	$(\lambda v0(\lambda v1(v0(v0(v0(v0$	$\mathscr{C}1\ \mathscr{E}1\ \mathscr{C}2\ \mathscr{E}3\ 5\ )\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	$y[2]{:}\overline{v0}$	$x[3]{:}\overline{F[8]}$	,,	,,
.6	$(\lambda v0(\lambda v1(v0(v0(v0(v0y[4]$	$)\ )\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	$y[2]{:}v0$	$x[3]{:}F[8]$	$y[6]{:}v1\ \overline{F[5]}\ y[4]$	$x[7]{:}F[1]\ \overline{y[0]}\ y[0]$
.4	$(\lambda v0(\lambda v1(v0(v0(v0(v0y[4]$	$\mathscr{C}2\ \mathscr{E}3\ )\ )\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}v1\ \overline{F[5]}\ \overline{y[4]}$	$x[7]{:}F[1]\ \overline{y[0]}\ \overline{y[0]}$
.4	$(\lambda v0(\lambda v1(v0(v0(v0(v0v1$	$\mathscr{C}2\ \mathscr{E}1\ \mathscr{C}2\ \mathscr{E}3\ )\ )\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}\overline{v1}\ \overline{F[5]}\ \overline{y[4]}$	$x[7]{:}\overline{F[1]}\ \overline{y[0]}\ \overline{y[0]}$
.6	$(\lambda v0(\lambda v1(v0(v0(v0(v0v1)$	$)\ \mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}v1\ \overline{F[5]}\ y[4]$	$x[7]{:}F[1]\ \overline{y[0]}\ y[0]$
.6	$(\lambda v0(\lambda v1(v0(v0(v0(v0v1))$	$\mathscr{K}2\ \mathscr{C}2\ \mathscr{E}2\ )\ )\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	,,	,,
.6	$(\lambda v0(\lambda 1(v0(v0(v0(v0v1)))$	$)\ \mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	$y[6]{:}v1\ F[5]$	$x[7]{:}F[1]\ y[0]$
.6	$(\lambda v0(\lambda v1(v0(v0(v0(v0v1))))$	$\mathscr{K}2\ \mathscr{K}2\ )\ \mathscr{K}1\ )$	,,	,,	,,	,,
.6	$(\lambda v0(\lambda v1(v0(v0(v0(v0v1)))))$	$\mathscr{K}1\ )$	,,	,,	$y[6]{:}$	$x[7]{:}$
.6	$(\lambda v0(\lambda v1(v0(v0(v0(v0v1))))))$		$y[2]{:}$	$x[3]{:}$	$y[6]{:}$	$x[7]{:}$

4. η-REDUCTION

We will η-reduce an already β-reduced λ-formula and therefore in this formula every bound variable is represented by a v-symbol suffixed with a nonnegative integer in decimal notation. The formula is subject to a

Table III. Program: (λv0(λv1((v0(λv2(λv3(v2v3))))v1)))

η-Generation

Workstack	Assignment statements	Counter	"Black list"	
$(\lambda v0$		0		
$(\lambda v0(\lambda v1$		1		
$(\lambda v0(\lambda v1((v0$		,,	$v0$	
$(\lambda v0(\lambda v1((v0(\lambda v2$		2	,,	
$(\lambda v0(\lambda v1((v0(\lambda v2(\lambda v3$		3	,,	
$(\lambda v0(\lambda v1((v0(\lambda v2(\lambda v3(v2v3)$	$F[1]$: $(v2v3)$	,,	,,	$v2$
$(\lambda v0(\lambda v1((v0(\lambda v2(\lambda v3F[1])$	$F[2]$: $(\lambda v3F[1])$	,,	,,	,,
$(\lambda v0(\lambda v1((v0(\lambda v2F[2])$	$F[3]$: $(\lambda v2F[2])$	,,	,,	,,
$(\lambda v0(\lambda v1((v0F[3])$	$F[4]$: $(v0F[3])$	,,	,,	,,
$(\lambda v0(\lambda v1(F[4]v1)$	$F[5]$: $(F[4]v1)$	,,	,,	,,
$(\lambda v0(\lambda v1F[5])$	$F[6]$: $(\lambda v1F[5])$	,,	,,	,,
$(\lambda v0F[6])$	$F[7]$: $(\lambda v0F[6])$	,,	,,	,,
$F[7]$		3	,,	,,

η-Run

Workstack	Pointerstack				Side-effects		"Black list"	
$F[7]$							$v0$	$v2$
$(\lambda v0F[6]$	)						,,	,,
$(\lambda v0(\lambda v1F[5]$	)	)					,,	,,
$(\lambda v0(\lambda v1(F[4]v1))$	)						,,	,,
$(\lambda v0F[4]$	)				$v2{:}v1$	$v3{:}v2$	$v0$	$v1$
$(\lambda v0(v0F[3]$	)	)			,,	,,	,,	,,
$(\lambda v0(v0(\lambda v2$	3	)	)		,,	,,	,,	,,
$(\lambda v0(v0(\lambda v1F[2]$	)	)	)		,,	,,	,,	,,
$(\lambda v0(v0(\lambda v1(\lambda v3$	2	)	)	)	,,	,,	,,	,,
$(\lambda v0(v0(\lambda v1(\lambda v2F[1]$	)	)	)	)	,,	,,	,,	,,
$(\lambda v0(v0(\lambda v1(\lambda v2(v1v2))$	)	)	)		,,	,,	,,	,,
$(\lambda v0(v0(\lambda v1v1$	)	)	)		,,	,,	,,	,,
$(\lambda v0(v0(\lambda v1v1)$	)	)			,,	,,	,,	,,
$(\lambda v0(v0(\lambda v1v1))$	)				,,	,,	,,	,,
$(\lambda v0(v0(\lambda v1v1)))$								

generation phase like β-generation, where, however, no level of calculus is assigned to the variables, which are now all terminal. No stacks are created. The bound variable, which occurs as the lhs of an application or more than once as the rhs, is marked in a "black list," because the formula is certainly not reducible with regard to this variable. Moreover, the greatest number n that occurs as suffix for a v in the λ-formula is stored.

The run phase has essentially only one rule:

$$(\lambda vi(\Phi[j]\, vi))$$

on the top of the workstack, iff vi does not occur in the "black list," it is replaced by

$$\Phi[j]$$

In addition, we write as side-effects

$$\begin{aligned}&v(i+1){:}vi\\&v(i+2){:}v(i+1)\\&\dots\\&vn{:}v(n-1)\end{aligned}$$

Also, in the η-run, the pointerstack allows us to follow the nesting of the subformulas. Table III shows an example.

5. PREFIXING TO A λ-FORMULA A SEQUENCE OF ASSIGNMENT STATEMENTS

We present an extension of the method so far described which allows us to write any number of assignment statements to the left of the given λ-formula to be reduced. The aim is twofold: (1) to handle efficiently more than one occurrence of the same λ-subformula (i.e., generating it once only) and/or (2) to define a λ-formula recursively.

As an example, we would like to process expressions of the kind

$$(D := \mathscr{D}) \quad (G := \mathscr{G}[D]) \quad (H := \mathscr{H}) \quad (HG)$$

where $\mathscr{D}$, $\mathscr{G}$, $\mathscr{H}$ are given λ-formulas and D occurs in $\mathscr{G}$.

5.1. Generation

In addition to the generation rules, let us add for every assignment statement the following prescriptions:

(a) The auxiliary variable of the rhs must be the corresponding lhs symbol

Table IV. Program: (K:=(λx(λyx)))(I:=(λxx))(KI)

β-Generation Workstack	Assignment statements	Stack chains				
(K:= (λx(λyx)	K[1]:(λKy[0]$K\mathscr{K}1$Kx[0])	$\boxed{K\mathscr{C}1}$	Ky[0]:			
(K:= (λxK[1])	K[2]:(λKx[1]K[1])		,,	Kx[1]:		
(K:= K[2])	K:K[2]		,,	,,		
(I:= (λxx)	I[1]:(λIx[0]$I\mathscr{K}1$Ix[0])		,,	,,	$\boxed{I\mathscr{C}1}$	Ix[0]:
(I:= I[1])	I:I[1]		,,	,,		,,
(KI)	F[1]:(KI)		,,	,,		,,
F[1]						

β-Run Rule number	Workstack	Pointerstack	Stack chains				
	F[1]		$\boxed{K\mathscr{C}1}$	Ky[0]:	Kx[1]:	$\boxed{I\mathscr{C}1}$	Ix[0]:
.5							
	(K	F1		,,	,,		,,
.5							
	(K[2]	,,		,,	,,		,,
.5							
	((λKx[1]K[1])I)			,,	,,		,,
.2							
	K[1]			,,	Kx[1]:I		,,
.5							
	(λKy[0]	K1		,,	,,		,,
.3							
	(λv0	,,		Ky[0]:v0	,,		,,
.6.1							
	(λv0Kx[0]	) $K\mathscr{K}1$		,,	,,		,,
.4							
	(λv0I	$K\mathscr{C}1$ $\mathscr{E}1$) $K\mathscr{K}1$		Ky[0]:$\overline{v0}$	Kx[1]:$\overline{I}$		,,
.5							
	(λv0(λIx[0]	I1 $K\mathscr{C}$ 1$\mathscr{E}$ 1) $K\mathscr{K}1$		,,	,,		,,
.3							
	(λv0(λv1	,,		,,	,,		Ix[0]:v1
.6							
	(λv0(λv1Ix0	$I\mathscr{K}1$) $K\mathscr{C}1$ $\mathscr{E}1$) $K\mathscr{K}1$		,,	,,		,,
.4							
	(λv0(λv1v1	$I\mathscr{C}1$ $\mathscr{E}1$ $I\mathscr{K}1$)$K\mathscr{C}1\mathscr{E}1$)$K\mathscr{K}1$		,,	,,		Ix[0]:$\overline{v1}$
.6							
	(λv0(λv1v1)	$K\mathscr{C}1$ $\mathscr{E}1$) $K\mathscr{K}1$		,,	,,		Ix[0]:
.6							
	(λv0(λv1v1))			Ky[0]:	Kx[1]:		Ix[0]:

(b) Every chain symbol and every lower case variable belonging to the rhs must be prefixed with the corresponding lhs symbol.

(c) To take into account a := assignment statement, it is sufficient to generate[7] rhs $\equiv$ lhs $[n]$ and to construct

$$\text{lhs} : \text{rhs}$$

Collisions between auxiliary variables are not permitted, therefore no λ-formula can be assigned to the symbol F.

5.2. Run

All rules behave as before with the exception of rule 3.2.4, which is also entered by any occurrence of an upper case letter subscripted or not. Table IV gives an example.

APPENDIX

As a language, the CUCH has an alphabet which is the union of the following three alphabets: (1) an alphabet consisting of infinite *variables*; (2) an alphabet consisting of a finite number of *constants*; (3) the alphabet $\{(,\lambda,)\}$.

CUCH words are called *formulas*. The construction of this language, which permit building up new formulas from atomic formulas, is as follows:

(1) Each variable and each constant is an atomic formula.

(2) If $\mathscr{A}$ and $\mathscr{B}$ are formulas, then $(\mathscr{A}\mathscr{B})$ is a formula obtained (by definition) by means of the *application* of $\mathscr{A}$ to $\mathscr{B}$.

(3) If we denote by $\mathscr{F}[x]$ any formula F, and if we are interested in the free occurrences of x in $\mathscr{F}[x]$ (notice that x may not occur in F), then (by definition) $(\lambda x\mathscr{F}[x]) \equiv (\lambda xF)$ is a formula obtained from F by *abstraction* (relative to x). However each occurrence of x in (λxF) is said to be *bound* (from λ; notice that *free* $\equiv$ not bound).

The morphology of CUCH is, according to the Backus notation,

$$\langle\text{variable}\rangle = \cdots \mid t \mid u \mid v \mid w \mid x \mid y \mid z \cdots$$

$$\langle\text{constant}\rangle = \underset{\wedge}{B} \mid \underset{\wedge}{C} \mid \underset{\wedge}{I} \mid \underset{\wedge}{K} \mid \underset{\wedge}{O} \mid \underset{\wedge}{C_*} \mid \underset{\wedge}{W} \cdots$$

$$\langle\text{formula}\rangle = \langle\text{variable}\rangle \mid \langle\text{constant}\rangle \mid (\lambda\, \langle\text{variable}\rangle\langle\text{formula}\rangle) \mid (\langle\text{formula}\rangle\langle\text{formula}\rangle)$$

[7] Here, n is the final level.

The CUCH constants are of three types. Some of them, like $\underset{\wedge}{S}$, $\underset{\wedge}{B}$, $\underset{\wedge}{C}$, $\underset{\wedge}{I}$, $\underset{\wedge}{K}$, $\underset{\wedge}{O}$, $\underset{\wedge}{W}$, $\underset{\wedge}{C}_*$, etc., are names of closed formulas.[8] Let us give, for example the definition of $\underset{\wedge}{I}$, $\underset{\wedge}{K}$, and $\underset{\wedge}{O}$:

$$\underset{\wedge}{I} = (\lambda xx), \qquad \underset{\wedge}{K} = (\lambda x(\lambda yx)), \qquad \underset{\wedge}{O} = (\lambda x(\lambda yy))$$

Some of them, like θ, are means of defining *classes* of formulas: In such cases, the corresponding reduction rule holds for each element of the class. The rule for θ is

$$\theta a = (a(\theta a)) \quad \text{where} \quad a \quad \text{is any formula}$$

The operator δ (universal discriminator for normal forms) is not representable by any closed formula.

The relation between pairs of formulas is as follows:

(1) *Immediate reduction.* $\mathscr{A}$ is said to be immediately reducible to $\mathscr{B}$ by means of the α-, β-, and η-rules, respectively, and we write $\mathscr{A} \rightarrow \mathscr{B}$ if at least one of the following definitions holds:

(α) $F \xrightarrow{\alpha} G$ iff $F \equiv (\lambda x\mathscr{F}[x])$ and $G \equiv (\lambda y\mathscr{F}[y])$, where F is a subformula of $\mathscr{A}$, $\mathscr{B}$ is the result of the substitution of G for one of the occurrences of F in $\mathscr{A}$, and y does not occur free in $\mathscr{F}[x]$ but it is free for x in $\mathscr{F}[x]$.[9]

(β) $F \xrightarrow{\beta} G$ iff $F \equiv ((\lambda x\mathscr{F}[x])M)$, $G \equiv \mathscr{F}[M]$, where F and G are as in (α), and M is a formula free for x in $\mathscr{F}[x]$.

(η) $F \xrightarrow{\eta} G$ iff $F \equiv (\lambda x(Mx))$ and $G \equiv M$, where F and G are as in (α), M is a formula, and x does not occur free in M.

(2) *Reduction.* $\mathscr{A}$ is said to be reducible to $\mathscr{B}$, and we write $\mathscr{A} \Rightarrow \mathscr{B}$, if there is a finite sequence of formulas, $\mathscr{A} \equiv F1, F_2, \ldots, F_n \equiv \mathscr{B}$ such that for each $1 \leqslant i \leqslant n-1$, we have $Fi \rightarrow F(i+1)$.

(3) *Conversion.* $\mathscr{A}$ is said to be convertible to $\mathscr{B}$, and we write $\mathscr{A} = \mathscr{B}$, iff, under the conditions of the preceding paragraph, we have $Fi \rightarrow F(i+1)$ or $F(i+1) \rightarrow Fi$.

According to the Church–Rosser theorem (Ref. 5, pp. 108–150), if $\mathscr{A} = \mathscr{B}$, then there is $\mathscr{C}$ such that $A \Rightarrow \mathscr{C}$ and $\mathscr{B} \Rightarrow \mathscr{C}$. An important consequence of this theorem is that if $\mathscr{A} = \mathscr{B}$ and in $\mathscr{B}$ neither the β-rule nor

[8] "Closed" means "without free variables."

[9] y does not occur free in $\mathscr{F}[x]$ both if it does not occur and if it occurs bound. A formula M is said to be free for x in $\mathscr{F}[x]$ if, denoting by $\mathscr{F}[M]$ the result of the substitution of M for every free occurrences of x in $\mathscr{F}[x]$, every free occurrence of a variable z in M is still free in $\mathscr{F}[M]$. If x does not occur free in $\mathscr{F}[x]$, then we have $\mathscr{F}[M] = \mathscr{F}[x]$.

the η-rule is applicable, then $\mathscr{B}$ is unique up to some applications of the α-rule. In this case, is said to be in *normal form.* A rather obvious refinement of this theorem is: Any two formulas reducible to normal forms and mutually convertible can be reduced to the same *principal normal form,* where all bound variables belong to a given alphabet and occur in a fixed order. Another important consequence of the Church–Rosser theorem is that if $\mathscr{A}$ does admit a normal form, then an algorithm does exist for obtaining it.

ACKNOWLEDGMENT

We are indebted to the referee for indicating topics needing greater clarity.

REFERENCES

1. G. Ausiello, "Automatic reduction of CUCH expressions by means of the value method," *Atti del I Congresso Nazionale dell'AICA, Napoli 26–29 settembre 1968* (Rome, 1971), pp. 174–184.
2. C. Böhm and W. Gross, "Introduction to the CUCH," in *Automata Theory,* ed. by E. R. Caianiello (Academic Press, New York, 1966), pp. 35–65.
3. C. Böhm, "The CUCH as a formal and description language," in *Formal Languages Description Languages for Computer Programming,"* ed. by T. B. Steel, Jr. (North-Holland, Amsterdam, 1966), pp. 179–197.
4. A. Church, "The calculi of lambda-conversion," Ann. Math. Stud. No. 6, Princeton University Press, 1941.
5. J. B. Curry and R. Feys, *Combinatory Logic,* Vol. 1 (North-Holland, Amsterdam, 1958).
6. S. Ginsburg, Sheila A. Greibach, and Michael A. Harrison, "One-way stack automata," *ACM,* **14**(2): 389–418 (1967).
7. P. Landin, "A correspondence between ALGOL 60 and Church's lambda notation," *Comm. ACM* (February/March 1965).
8. D. Scott, "Outline of a mathematical theory of computation," Oxford University Computing Laboratory Programming Research Group (1970).
9. C. Strachey, "Towards a formal semantics," in *Formal Languages Description Languages, for Computer Programming,* ed. by T. B. Steel, Jr. (North-Holland, Amsterdam, 1966), pp. 198–216.
10. P. Wegner, *Programming Languages, Information Structures and Machine Organisation* (McGraw-Hill, New York, 1968).
11. C. McGowan, "The correctness of a modified SECD machine," 2nd Annual ACM Symposium on the Theory of Computation, 1970, North Hampton, pp. 149–157.

Printed in Belgium