Journal of Computer Languages, Vol. 1, pp. 303–320. Pergamon Press, 1976. Printed in Northern Ireland

A LAMBDA-CALCULUS MODEL OF PROGRAMMING LANGUAGES—II. JUMPS AND PROCEDURES

S. KAMAL ABDALI
Department of Mathematical Sciences, Rensselaer Polytechnic Institute,
Troy, N.Y. 12181, U.S.A.

(*Received* 15 *August* 1974 *and in revised form* 27 *May* 1975)

Abstract—The correspondence between programming languages and the lambda-calculus presented in Part I of the paper is extended here to include iteration statements, jumps, and procedures. Programs containing loops are represented by lambda-expressions whose components are specified recursively by means of systems of simultaneous conversion relations. The representation of call-by-name and side-effects in a program is accomplished without any resort to the concepts of memory and address by making use of a number of variables in addition to those declared by the programs; with the aid of these additional variables, the parameter linkage operations can be simulated by pure substitution. The applicability of the model to the problems of proving program correctness and equivalence is demonstrated by means of examples.

Programming Language Semantics Lambda-Calculus Jumps Procedures Parameters
Call-by-name Side-effects Program Correctness Equivalence of Programs

INTRODUCTION

PART I of this paper dealt with the lambda-calculus representation of assignments, conditional and compound statements, input–output, and blocks. The essential restrictions on the programs representable by the rules presented in that part were the absence of loops and of side-effects. These features will be considered now. (Although forward jumps and side-effect-free procedures could be incorporated rather easily within the representation scheme of Part I, their discussion was postponed until now in the interest of giving a unified treatment of jumps and side-effects.)

The numbering of sections and formulas is here continued from Part I.

4. ITERATION AND JUMP STATEMENTS

4.1. *Recursive definition of lambda-expressions*

In dealing with program loops, we will need the *recursive definitions* of the form

$$f \equiv \ldots f \ldots f \ldots, \tag{i}$$

in which an *LE* is defined in terms of itself. Much towards the precise understanding of such definitions has been contributed, among others, by Morris [1], Rosen [2], and de Bakker [3], who have built upon the pioneering work of Kleene [4] and Scott [5]. Referring the reader to the cited work for a rigorous analysis of recursive definitions, we shall be content here with some informal, intuitive comments.

One possible explanation of (i) is the following: As would be possible in the case of a non-recursive definition, we allow the replacement of a component f in any *LE* by the right-hand side of (i). In other words, we interpret (i) as a *rule of reduction*

$$f \rightarrow \ldots f \ldots f \ldots. \tag{ii}$$

By a sequence of such reductions of f and the other *LC* contractions, it may be possible to reduce an *LE* containing f to a normal form. It has been shown (e.g. Rosen [2]) that the

Church–Rosser property holds with this broader sense of reduction also; consequently, most other important properties of reduction, such as the uniqueness of normal forms and the correctness of the standard-order reduction algorithm, are valid when the reduction rules of form (ii) (with distinct left-hand sides) are admitted.

The interpretation of a recursive definition as a replacement or reduction rule certainly enables us to *use* the definition mechanically. But what, actually, is the object that is so defined? It is obvious that applying the reduction rule (i) to f itself cannot lead to an "ordinary" definition—a non-self-referencing—description of f. However, such a definition is possible if we regard (ii) as a *reduction relation* (not a *rule*) involving an unknown, and regard f as a *solution* of (ii). To reduce an LE containing f, we may, under this latter interpretation of the definition (i), replace f by a solution of (ii), and then apply a sequence of contractions. Now, in general, (ii) may be satisfied by more than one solution, so that we may have the choice of different explicit definitions for the same LE. There is, however, no reason to expect that these different definitions of an LE are compatible to each other or to the interpretation of the recursive definition as a reduction rule—compatible in the sense that all reductions, which start with an LE having a recursively specified LE as a component and which use the different definitions of the recursively specified LE, yield the same normal form (if any). However, those solutions of (ii) for which the resulting definitions of the LE f are compatible with the definitions of the first approach (of taking (ii) as a reduction rule) have been characterized by Morris [1] in terms of the following partial order on LE's: For LE's a and b we say that a *is extended by* b, in symbols, $a \leq b$, if, for all LE's c, it is the case that $ca \leftrightarrow cb$ whenever ca possesses a normal form. (For example, it can be shown that $\Omega \leq b$ for all LE's b, where Ω is as defined in (1).) Now, the particular solutions of (ii) that we are interested in have the property that they are extended by all solutions of (ii). In other words, for an *explicit definition* of the LE f specified by (i), we can take a *minimal solution* of (ii) (with respect to $\leq$).

An explicit definition of the LE specified by (i) is

$$f \equiv \mathbf{Y}h,$$

where $\mathbf{Y} \equiv \lambda x\colon (\lambda y\colon x(yy))(\lambda y\colon x(yy))$, and

$$h \equiv \lambda x\colon \ldots x \ldots x \ldots$$

is the abstraction with respect to the indeterminate x of the LE obtained from the right-hand side of (i) by replacing f in it by x. It is easy to see that for all a, $\mathbf{Y}a \rightarrow a(\mathbf{Y}a)$, and hence that $f \equiv \mathbf{Y}h$ satisfies (ii). Morris [1] has also shown that this solution is minimal.

In general, (ii) has infinitely many, mutually non-convertible, minimal solutions, which are, however, equivalent in the sense that they all have the same intuitive interpretations as functions. The choice of any one of these such as $\mathbf{Y}h$ is rather arbitrary. To leave this choice unspecified, while emphasizing the minimality of the chosen solution, one may employ the *μ-notation* of de Bakker [3]. In this notation, the minimal solution of (ii) is designated by the *μ-expression*

$$\mu x\colon \ldots x \ldots x \ldots,$$

where, the LE to the right of the colon is obtained from the right-hand-side of (i) by replacing f with the indeterminate x.

The above interpretations can also be generalized to include the simultaneous recursive

definition of several *LE*'s in the form

$$f_1 \equiv h_1 f_1 \ldots f_n,$$
$$\ldots \qquad \text{(iii)}$$
$$f_n \equiv h_n f_1 \ldots f_n,$$

where f's do not occur in h's. The generalized interpretations are, briefly:

(a) The definitions (iii) may be regarded as a set of reduction rules (replacing $\equiv$ by $\rightarrow$) without concern as to the explicit values of f's.

(b) The f's defined by (iii) may be regarded to be the minimal solutions of the system of *reduction relations*

$$f_i \rightarrow h_i f_1 \ldots f_n, \qquad 1 \leq i \leq n. \qquad \text{(iv)}$$

An explicit solution of (iv) is given by

$$f_i \equiv \mathrm{Y}_{i,n} h_1 \ldots h_n,$$

where

$$\mathrm{Y}_{i,n} \equiv \lambda z_1 \ldots z_n{:}\mathrm{Y}(\lambda x{:}\langle xz_1, \ldots, xz_n\rangle)(\lambda x_1 \ldots x_n{:}x_i).$$

The interpretations (a) and (b) result in identical normal form reductions of the *LE*'s containing f's.

4.2. *Iteration statements*

The representation of the **for** statement of ALGOL 60 is obtained by expressing this statement in terms of the simple (non-ALGOL 60) **while** loop of the form **while . . . do . . .**. To represent the latter, consider the statement **while** b **do** S appearing in the environment $(v_1, \ldots, v_n)$. Calling this statement by the name T, we may (recursively) describe it, for the purpose of *LC* representation, as

if b **then begin** S; T **end.**

Now the formulas for the representation of compound and conditional statements, (4) and (11) (Part I), respectively, are applicable to the above statement, and its representation $\{T\}$ is, recursively, the *LE*

$$\lambda\phi v_1 \ldots v_n{:}\mathbf{if}\ \{b\}(\{\lambda\phi{:}\{S\}(\{T\}\phi)\}\phi v_1 \ldots v_n)(\phi v_1 \ldots v_n)$$
$$\leftrightarrow \lambda\phi v_1 \ldots v_n{:}\mathbf{if}\ \{b\}(\{S\}(\{T\}\phi))\phi v_1 \ldots v_n.$$

Thus, we adopt the representation

$$\{\textbf{while } b \textbf{ do } S\}_{(v_1,\ldots,v_n)} \qquad (12)$$
$$\equiv \mu x{:}\lambda\phi v_1 \ldots v_n{:}\mathbf{if}\ \{b\}(\{S\}(x\phi))\phi v_1 \ldots v_n.$$

Alternative definitions of the same *LE*, call it A, are

$$A \equiv \lambda\phi v_1 \ldots v_n{:}\mathbf{if}\ \{b\}(\{S\}(A\phi))\phi v_1 \ldots v_n,$$
$$A \equiv \mathrm{Y}(\lambda x\phi v_1 \ldots v_n{:}\mathbf{if}\ \{b\}(\{S\}(x\phi))\phi v_1 \ldots v_n).$$

Example. At this point, we illustrate the *LC* representations introduced so far by means of a complete program. Also, as an application of the model, we derive the correctness of the program in terms of its representation.

Statements	*Representations*
begin integer x, y;	
read x;	$a \equiv \lambda\phi xyoi{:}\phi iyo$
$y := 0$;	$b \equiv \lambda\phi xy{:}\phi x\underline{0}$
begin integer z;	
$z := 0$;	$c \equiv \lambda\phi zxy{:}\phi\underline{0}xy$
while $z < x$ **do**	
begin	
$y := 1+y+2\times z$;	$d \equiv \lambda\phi zxy{:}\phi zx(+(+\underline{1}\,y)(\times\underline{2}z))$
$z := z+1$	$e \equiv \lambda\phi zxy{:}\phi(+z\underline{1})xy$
end	$f \equiv \lambda\phi{:}d(e\phi)$
end of while	$g \equiv \lambda\phi zxy{:}$**if** $(<zx)\ (f(g\phi))\phi zxy$
end;	$h \equiv \lambda\phi{:}c(g(\lambda z{:}\phi))\Omega$
write y	$j \equiv \lambda\phi xyo{:}\phi xy(o;y)$
end	$k \equiv \lambda\phi{:}a(b(h(j(\lambda xy{:}\phi))))\Omega\Omega$
{program}	$\equiv P \equiv k\mathbf{II}$.

We wish to prove that on reading a non-negative integer n, this program will print out the integer n^2. According to our input-output conventions, we need to show that

$$P\underline{n} \to \langle\underline{n^2}\rangle, \quad \text{for all integers } n \geq 0. \tag{i}$$

This is done in four steps, as follows:

(a) We show that, for all *LE*'s ϕ and all integers n and i,

$$g\phi\underline{i}\ \underline{n}\ \underline{i^2} \to \phi\underline{i}\ \underline{n}\ \underline{i^2}, \quad \text{if } i \geq n, \tag{ii}$$
$$g\phi\underline{i}\ \underline{n}\ \underline{i^2} \to g\phi\underline{i+1}\ \underline{n}\ \underline{(i+1)^2}, \quad \text{if } i < n. \tag{iii}$$

By the definition of g, we obtain

$$g\phi\underline{i}\ \underline{n}\ \underline{i^2} \to \mathbf{if}\ (<\underline{i}\ \underline{n})\ (f(g\phi))\phi\underline{i}\ \underline{n}\ \underline{i^2}.$$

If $i \geq n$, then $(<\underline{i}\ \underline{n}) \to$ **false**, so that (ii) is immediate. Otherwise, $(<\underline{i}\ \underline{n}) \to$ **true**, and the above *LE*

$$\begin{aligned}&\to f(g\phi)\underline{i}\ \underline{n}\ \underline{i^2} \to d(e(g\phi))\underline{i}\ \underline{n}\ \underline{i^2} \to e(g\phi)\underline{i}\ \underline{n}(+(+\underline{1}\ \underline{i^2})(\times\underline{2}\ \underline{i}))\\ &\to e(g\phi)\underline{i}\ \underline{n}\ \underline{1+i^2+2\times i} \to e(g\phi)\underline{i}\ \underline{n}\ \underline{(i+1)^2}\\ &\to g\phi\ \underline{i+1}\ \underline{n}\ \underline{(i+1)^2}.\end{aligned}$$

(b) Next, for all integers n and i such that $0 < i \leq n$, we have

$$g\phi\underline{0}\ \underline{n}\ \underline{0} \to g\phi\underline{i}\ \underline{n}\ \underline{i^2}. \tag{iv}$$

This is proved by induction on i. From (iii) one easily verifies (iv) both for $i = 1$, and for $i = j+1 \leq n$ when the case for $i = j < n$ is assumed.

(c) Next, we claim that for all integers $n \geq 0$, it is the case that

$$h\phi\underline{n}\ \underline{0} \to \phi\underline{n}\ \underline{n^2}. \tag{v}$$

For, we have

$$\begin{aligned}h\phi\underline{n}\ \underline{0} &\equiv (\lambda\phi{:}c(g(\lambda z{:}\phi))\Omega)\phi\underline{n}\ \underline{0}\\ &\to c(g(\lambda z{:}\phi))\Omega\underline{n}\ \underline{0}\\ &\to g(\lambda z{:}\phi)\underline{0}\ \underline{n}\ \underline{0}.\end{aligned}$$

Now if $n = 0$, then from (ii) it follows that

$$g(\lambda z{:}\phi)\underline{0}\ \underline{\mathrm{n}}\ \underline{0} \to (\lambda z{:}\phi)\underline{\mathrm{n}}\ \underline{\mathrm{n}}\ \underline{\mathrm{n}^2} \to \phi\underline{\mathrm{n}}\ \underline{\mathrm{n}^2}.$$

On the other hand, if $n > 0$, then for the case $i = n$ (iv) yields

$$\begin{aligned} g(\lambda z{:}\phi)\underline{0}\ \underline{\mathrm{n}}\ \underline{0} &\to g(\lambda z{:}\phi)\underline{\mathrm{n}}\ \underline{\mathrm{n}}\ \underline{\mathrm{n}^2} \\ &\to (\lambda z{:}\phi)\underline{\mathrm{n}}\ \underline{\mathrm{n}}\ \underline{\mathrm{n}^2} \qquad \text{by (ii)} \\ &\to \phi\underline{\mathrm{n}}\ \underline{\mathrm{n}^2}. \end{aligned}$$

(d) Finally, to prove (i), we simply use the definitions of the *LE*'s a through k, obtaining, for all integers $n \geq 0$,

$$\begin{aligned} P\underline{\mathrm{n}} \equiv k\mathbf{I}\underline{\mathrm{n}} &\to a(b(h(j(\lambda xy{:}\mathbf{I}))))\boldsymbol{\Omega}\boldsymbol{\Omega}\mathbf{I}\underline{\mathrm{n}} \\ &\to b(h(j(\lambda xy{:}\mathbf{I})))\underline{\mathrm{n}}\boldsymbol{\Omega}\mathbf{I} \\ &\to h(j(\lambda xy{:}\mathbf{I}))\underline{\mathrm{n}}\ \underline{0}\mathbf{I} \\ &\to j(\lambda xy{:}\mathbf{I})\underline{\mathrm{n}}\ \underline{\mathrm{n}^2}\mathbf{I} \qquad \text{by (v)} \\ &\to (\lambda xy{:}\mathbf{I})\underline{\mathrm{n}}\ \underline{\mathrm{n}^2}(\mathbf{I};\underline{\mathrm{n}^2}) \\ &\to (\mathbf{I};\underline{\mathrm{n}^2}) \\ &\to \langle\underline{\mathrm{n}^2}\rangle. \end{aligned}$$

4.3. *Jump statements*

We regard the execution of the statement $S \equiv$ **goto** L in a program as the substitution of the part of the program following L for the one following S. This viewpoint provides us with the representation of both labels and jump statements.

A label is identified with the part of the program following it. To be accurate, the representation of a label L occurring in a program P is taken to be the representation of the program P' obtained from P by deleting all the statements, but retaining the declarations, that appear above L. This representation can be obtained in a simpler manner by using the following inductive scheme: Let the label L occur in a block b whose declared variables are $v_1, \ldots, v_n$.

(a) If L is followed by statements $S_1, \ldots, S_m$, and a label M, in that order, all within b, then $\{L\} \equiv \{S_1\}(\{S_2\}(\ldots(\{S_m\}\{M\})\ldots))$.

(b) If $S_1, S_2, \ldots, S_m$ are the statements following L to the end of b, then

$$\{L\} \equiv \{S_1\}(\{S_2\}(\ldots(\{S_m\}(\lambda v_1 \ldots v_n{:}N)\ldots)),$$

where $N \equiv \mathbf{I}$, if b is the outermost block, else N is the representation of the program part following b, i.e. of the (possibly imaginary) label immediately after the end of b.

According to the rules of ALGOL, the label to which a jump can be made must be in a block which is the same as, or outer to, the block containing the jump statement. It follows that (the list of variables constituting) the environment of a jump statement must contain the environment of the referred label as a final segment. Suppose $(v_1, \ldots, v_n)$ is the environment of the statement $S \equiv$ **goto** L, and $(v_m, \ldots, v_n)$, where $1 \leq m \leq n$, is the environment of L, and let ϕ represent as usual the program remainder of S. The execution of S causes the program to compute the function $\{L\}(v_m, \ldots, v_n)$ instead of $\phi(v_1, \ldots, v_n)$. Hence, the representation of S can be taken to be the *LE*

$$\lambda\phi v_1 \ldots v_n{:}\{L\}v_m \ldots v_n,$$

which simplifies (by η-contraction) to

$$\lambda\phi v_1 \ldots v_{m-1}{:}\{L\}.$$

Thus, we define

$$\{\text{goto } L, \text{ environment } (L) = (v_m, \ldots, v_n),\ 1 \leq m \leq n\}_{(v_1, \ldots, v_n)} \equiv \lambda\phi\colon (\lambda v_1 \ldots v_{m-1}\colon\{L\}). \tag{13}$$

It is sometimes convenient, specially in conection with conditional statements, to write the right-hand-side in the alternative forms:

$$\lambda\phi v_1 \ldots v_n\colon\{L\}v_m \ldots v_n,$$
$$\lambda\phi v_1 \ldots v_n\colon(\lambda v_1 \ldots v_{m-1}\colon\{L\})v_1 \ldots v_n.$$

Example. The representation of **goto** statements and labels is illustrated by means of a complete program. The program below has been derived from the program given in the previous example simply by expressing the **while** loop in terms of **goto**'s. As another application of the model, we prove the (input–output) equivalence of the two programs.

As before, the representations of individual statements are shown on the same line as the statement, or on the last line for a multiple-line statement, and are designated identifying names. The *LE*'s common to the representation of both programs have the same names.

The label *M* serves to illustrate the case (a) of label representations discussed above; it is otherwise superfluous.

```
begin integer x,y;
  read x;                          a ≡ λφxyoi:φiyo
  y := 0;                          b ≡ λφxy:φx0
  begin integer z;
     z := 0;                       c ≡ λφzxy:φ0xy
L:   if z=y then goto N
          else goto M;             d' ≡ λφzxy:if (=zy) (λz:N)Mzxy
M:   y := y+2×z+1;                e' ≡ λφzxy:φzx(+(+y(×2z))1)
     z := z+1;                     f' ≡ λφzxy:φ(+z1)xy
     goto L                        g' ≡ λφ:L
  end;                             h' ≡ λφ:c(d'(e'(f'(g'(λz:φ)))))Ω
N:   write y                       j ≡ λφxyo:φxy(o;y)
end                                k' ≡ λφ:a(b(h'(j(λxy:φ))))ΩΩ
                        {program} ≡ P' ≡ k'II
                                   L ≡ d'M
                                   M ≡ e'(f'(g'(λz:N)))
                                   N ≡ j(λxy:I).
```

We wish to prove that the above program and the program of the previous example produce the same output when executed with the same non-negative integer as the input data. That is, in terms of their representations, we wish to show that for all integers $n \geq 0$,

$$P\underline{n} \leftrightarrow P'\underline{n}. \tag{i}$$

Of course, this can be shown by using the previously obtained result $P\underline{n} \rightarrow \langle\underline{n^2}\rangle$ in conjunction with a direct proof of the fact that $P'\underline{n} \rightarrow \langle\underline{n^2}\rangle$. But we will prove the equivalence of the programs by verifying, in effect, that their differing parts do the same work when the programs are executed. These differing parts are represented by the *LE*'s h and h'. If we can show that for all integers $n \geq 0$,

$$hN\underline{n}\ \underline{0} \leftrightarrow h'N\underline{n}\ \underline{0}, \tag{ii}$$

(where $N \equiv j(\lambda xy{:}\mathrm{I})$, defined in the present example), then (i) is demonstrated as follows. From the previous example, part (d), we know that for all $n \geq 0$,

$$P\underline{n} \to h(j(\lambda xy{:}\mathrm{I}))\underline{n}\ \underline{0}\mathrm{I} \equiv hN\underline{n}\ \underline{0}\mathrm{I}.$$

But using the definitions of the present example, we also have

$$\begin{aligned} P'\underline{n} &\equiv k'\mathrm{II}\underline{n} \\ &\to a(b(h'(x(\lambda xy{:}\mathrm{I}))))\Omega\Omega\mathrm{I}\underline{n} \\ &\to b(h'(j(\lambda xy{:}\mathrm{I})))\underline{n}\Omega\mathrm{I} \\ &\to h'(j(\lambda xy{:}\mathrm{I}))\underline{n}\ \underline{0}\mathrm{I} \equiv h'N\underline{n}\ \underline{0}\mathrm{I}. \end{aligned}$$

Hence, it follows from (ii) that $P\underline{n} \leftrightarrow P'\underline{n}$.

It remains to verify (ii). From (v) in the previous example, we have for all integers $n \geq 0$,

$$hN\underline{n}\ \underline{0} \to N\underline{n}\ \underline{n^2}.$$

So (ii) would follow if we can also prove

$$h'N\underline{n}\ \underline{0} \to N\underline{n}\ \underline{n^2}. \qquad \text{(iii)}$$

To outline the proof of (iii), we simply state the sequence of reduction relations leading to it.

(a) $L\underline{i}\ \underline{n}\ \underline{i^2} \to \begin{cases} N\underline{n}\ \underline{n^2}, & \text{if } i = n, \\ L\underline{i+1}\ \underline{n}\ \underline{(i+1)^2}, & \text{if } i \neq n. \end{cases}$

(b) $L\underline{0}\ \underline{n}\ \underline{0} \to L\underline{i}\ \underline{n}\ \underline{i^2}$, for $0 < i \leq n$.

(c) $L\underline{0}\ \underline{n}\ \underline{0} \to N\underline{n}\ \underline{n^2}$, for $n \geq 0$.

(d) $h'\phi\underline{n}\ \underline{0} \to N\underline{n}\ \underline{n^2}$, for $n \geq 0$.

5. PROCEDURES

5.1. *F-procedures*

We use the term *F-procedure* to denote a type procedure without any side effects. Specifically, an *F*-procedure is a procedure in which

(a) the procedure name is typed,
(b) all parameters are called by value,
(c) no global variables are modified,
(d) no jumps are made outside the procedure body,
(e) no procedures are used other than *F*-procedures.

Because of the above restrictions, the representation of *F*-procedures is much simpler than that of general procedures. Since many procedures encountered in programs are truly *F*-procedures, it seems useful to deal with them as a special case.

For the moment, let us consider only the *F*-procedures which do not involve global variables at all. For these, the environment of the declaration is immaterial. Let f be an *F*-procedure and $p_1, \ldots, p_n$ be its parameters. We wish to represent f in such a manner that for all expressions $e_1, \ldots, e_n$

$$\{f\}\{e_1\} \ldots \{e_n\} \to \{f(e_1, \ldots, e_n)\}. \qquad \text{(i)}$$

Such a representation is accomplished as follows: We use a variable π to denote the *F*-procedure value; that is, all assignments to f are represented as if made to π. Further,

we represent the statement S constituting the body of f by taking its environment to be $(\pi, p_1, \ldots, p_n)$. Now, starting with an arbitrary value of π, and the values e_i of p_i, the execution of S has the effect of assigning the value $f(e_1, \ldots, e_n)$ to π, and certain values to p_i which are irrelevant to the result; say, we have

$$\{S\}\phi\pi\{e_1\}\ldots\{e_n\} \rightarrow \phi\{f(e_1, \ldots, e_n)\}p_1 \ldots p_n. \tag{ii}$$

To obtain (i) from (ii), we may initialize π with Ω, and choose the LE $\lambda\phi p_1 \ldots p_n{:}\pi$ for ϕ and $\{S\}\phi\pi$ for $\{f\}$. Thus we adopt the following representation rule:

$$\begin{aligned}&\{\textbf{F-procedure } f(p_1, \ldots, p_n) \text{ with body } S\}\\ &\equiv \{S\}_{(\pi, p_1, \ldots, p_n)}(\lambda\, \pi p_1 \ldots p_n{:}\pi)\Omega.\end{aligned} \tag{14}$$

It should be pointed out that a label appearing in the body of an F-procedure is to be represented as the part of the F-procedure (not the program) that follows the label.

Example.

integer procedure mod (x,y);
value x,y; **integer** x,y;
begin integer q;
$q := x \div y$;
mod $:= x - y \times q$
end q;

$a \equiv \lambda\phi q\pi xy{:}\phi(\div xy)\pi xy$
$b \equiv \lambda\phi q\pi xy{:}\phi q(-x(\times yq))xy$
$c \equiv \lambda\phi{:}a(b(\lambda q{:}\phi))\Omega$
mod $\equiv c(\lambda\pi xy{:}\pi)\Omega$.

Example. Representation of the factorial function.

integer procedure fact (n); **value** n; **integer** n;
fact $:=$ **if** $n = 0$ **then** 1 **else** $n \times$ fact $(n-1)$;

As the body of this F-procedure consists of a single assignment statement, we have

$$\{\text{body}\} \equiv \lambda\phi\pi n{:}\phi((=n\underline{0})\underline{1}(\times n(\textbf{fact}(-n\underline{1}))))n.$$

Hence, the representation of the F-procedure is given by the recursively defined LE

$$\textbf{fact} \equiv \{\text{body}\}(\lambda\pi n{:}\pi)\Omega \rightarrow \lambda n{:}(=n\underline{0})\underline{1}(\times n(\textbf{fact}(-n\underline{1}))).$$

An explicit definition of the above LE is

$$\textbf{fact} \equiv \mathbf{Y}(\lambda zn{:}(=n\underline{0})\underline{1}(\times n(z(-n\underline{1})))).$$

Finally, it is easy to remove the restriction about global variables imposed earlier on functions: In case the global variable values are used (but not, of course, modified) in an F-procedure, we append the global variables to the actual arguments as if they also were parameters in addition to the explicitly declared parameters of the F-procedure. This is illustrated below.

Example.

begin integer x,y;
integer procedure $f(n)$;
value n; **integer** n;
$f := n+x$;
...
begin integer z;
$x := f(y)+z$;
...

$a \equiv \lambda\phi\pi nxy{:}\phi(+nx)nxy$
$f \equiv a(\lambda\pi nxy{:}\pi)\Omega$
$\lambda\phi zxy{:}\phi z(+(fyxy)z)y$

5.2. *Call-by-name, side-effects*

In the previous section, we have described the *LC* representation of procedures subject to rather stringent conditions. We will now show how the representations can be extended to more general procedures, allowing call-by-name, the modification of global variables, and side effects. However, we limit ourselves here to considering the formal parameters of the type integer and label only. The extension of the model to include real and boolean parameters is, of course, trivial.

In ALGOL 60, a procedure call is intended to have the effect of an appropriately modified copy of the procedure body [6]. The modification in the case of call-by-name consists in replacing each instance of a called-by-name formal parameter by the corresponding actual parameter. (It is understood that any name conflicts between the variables appearing in the actual parameter expressions and the local variables of the procedure are to be first removed by renaming the latter variables.) Instead of performing such symbolic substitution, however, which would require keeping procedures in text form at the execution time, most ALGOL compilers accomplish the same effect by treating formal parameter references in procedures as calls on special "parameter procedures" generated from actual parameters [7]. As a result, if an operation refers to a formal parameter during the execution of a procedure, then the procedure execution is suspended to evaluate the corresponding actual parameter *in the environment of the procedure calling statement*, and then the procedure execution is resumed using the thus-acquired value in the operation. Of course, depending upon the type and use of a parameter, the actual parameter evaluation may yield a value (e.g. an arithmetic or boolean quantity when the formal parameter is an operand in an expression) or a name (e.g. the address of a variable when the formal parameter appears to the left of an assignment statement.) Our *LC* interpretation is based on a similar idea. But we are able to avoid the notion of address, and work exclusively with values, by making use of a number of different "parameter procedures" for different operations performed with the same parameter; namely, the evaluation of actual parameter expressions, making assignments to the variables provided as actual parameters, and jump to an actual label.

5.3. *Integer parameters*

In the absence of procedures we were able to express each statement in a program as a function which had for its arguments the variable ϕ, denoting the program remainder (that is, part of the program following the statement), and the variables constituting the environment of the statement. Clearly the representation of a statement S in a procedure body would involve two sets of program remainders and environments—namely, one set for S itself and one for the statement, say T, that calls the procedure. The program remainder of T corresponds to the familiar "return" address or label for the procedure call. Now, any formal parameter instances in S give rise to actual parameter evaluations in the environment of T, but after the evaluation the control must eventually transfer back to S. Hence the representation of parameter evaluation also involves the two sets of environments and program remainders; but this time the program remainder of S serves as the return address. We will use the variable ρ to indicate the program remainder at the return point and ϕ, as usual, for the program remainder at the current point.

We have so far represented, and will continue to represent, each program variable by a single indeterminate. The representation of an assignment statement may be conceived as "binding" the indeterminate representing the variable appearing at the left-hand side to the

representation of the right-hand expression.† In general, the indeterminates representing program variables are "bound" at any time to the current values of the corresponding program variables. With each called-by-value formal parameter we similarly need to associate a single indeterminate, bound to the current "value" of the parameter at any time. However, we need to carry more information with a called-by-name formal parameter. Depending on the type and use of a parameter, we shall associate a number of indeterminates with it. For each called-by-name formal parameter of type integer, we require three indeterminates best thought of as being bound, respectively, to the "value" associated with it and to the "parameter procedures" for evaluating it and making assignments to it. If p is an integer parameter, then these three indeterminates will be usually denoted by p, p_ε, and p_α. (The parameter of type label will be discussed later.) The environment of a statement in a procedure body will contain the variables corresponding to all of the above mentioned indeterminates; specifically, it will consist of the following in the given order:

(a) variables local to the procedure,
(b) ρ, the "return" variable,
(c) variables representing the formal parameters,
(d) variables global to the procedure.

Next, let us turn to the procedure call. Associated with each called-by-name actual parameter p of type integer, and *individual to each procedure call*, is an *LE* that represents the "parameter procedure" for its evaluation. In case p is a program variable (rather than an expression), there is also another *LE* which represents the "parameter procedure" to effect the assignments to p called for in the procedure. These *LE*'s, referred to as "actual evaluation" and "actual assignment" operators, are denoted $\varepsilon_p{}^a$ and $\varepsilon_p{}^a$, respectively, with further distinguishing marks added when more than one procedure call is involved.

Last, let us consider the procedure declaration. Associated with each called-by-name formal parameter of type integer, and unique to each environment within the procedure body, are two *LE*'s which represent the calls on the "actual evaluation" and "actual assignment" parameter procedures mentioned above. For convenience, these *LE*'s are referred to as "formal evaluation" and "formal assignment" operators, and are denoted by $\varepsilon_p{}^f$ and $\alpha_p{}^f$, where p is the formal parameter, with further distinguishing marks added if more than one environment is involved. If, in a statement in a procedure body, a formal parameter appears as an operand of an expression, the statement will be represented as if preceded by a formal evaluation; likewise, if a formal parameter occurs at the left-hand side of an assignment statement, that statement will be represented as if immediately followed by a formal assignment.

The above ideas will now be illustrated by means of a very simple example in which the declaration and the call of a procedure have the same environment.

```
begin integer y;
  procedure P(x); integer x; x := x+2;
  y := 1;
  P(y)
end.
```

† The present descriptive use of "binding" and "bound" has no connection with the terms defined at the beginning of Section 2, Part I.

The body of the above procedure consists of a single statement, and that statement needs to be both preceded by a formal evaluation and followed by a formal assignment. Thus, it is represented by the compound

$$\lambda\phi{:}\varepsilon_x^{\,f}(a(\alpha_x^{\,f}\phi)) \equiv b,$$

say, where a is the representation of $x{:=}x{+}2$ as an ordinary assignment statement. Since there are no local variables in the procedure, the environment of this latter statement consists of the following:

ρ	the "return" variable,
x_ε	the "parameter evaluation" variable,
x_α	the "parameter assignment" variable,
x	the "parameter" variable, and
y	the global variable.

Thus we can write

$$a \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\phi\rho x_\varepsilon x_\alpha(+x\underline{2})y.$$

Now, as the variable x_ε is bound to the actual evaluation operator, and the formal evaluation consists of just an application of this LE, we define $\varepsilon_x^{\,f}$ to be

$$\lambda\phi\rho x_\varepsilon x_\alpha xy{:}x_\varepsilon\rho\phi x_\varepsilon x_\alpha xy,$$

or more simply,

$$\lambda\phi\rho x_\varepsilon x_\alpha x{:}x_\varepsilon\rho\phi x_\varepsilon x_\alpha x.$$

Note the interchange of ϕ and ρ above; this signifies that the program remainder at the return point of procedure call becomes the current program remainder during parameter evaluation, and vice versa. In a similar manner, we define

$$\alpha_x^{\,f} \equiv \lambda\phi\rho x_\varepsilon x_\alpha x{:}x_\alpha\rho\phi x_\varepsilon x_\alpha x.$$

(In general, the global variables of the procedure need not appear in the formal evaluation and assignment operators.)

The whole procedure may be represented by

$$P \equiv b(\lambda\rho x_\varepsilon x_\alpha x{:}\rho),$$

which displays the effect that once the procedure execution is over, (after the application of b), only the return variable is retained, and the other variables, namely, the ones connected with parameters, are deleted from the environment.

Next, let us look at the procedure call. There is only one call-by-name actual parameter of type integer in this statement. So we need to define two LE's $\varepsilon_x^{\,a}$ and $\alpha_x^{\,a}$, the actual evaluation and assignment operators. These serve essentially as the fictitious assignment statements $x{:=}y$ and $y{:=}x$ (in the environment of the procedure call), respectively, and thus can be defined by

$$\varepsilon_x^{\,a} \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\rho\phi x_\varepsilon x_\alpha yy,$$
$$\alpha_x^{\,a} \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\rho\phi x_\varepsilon x_\alpha xx.$$

Again the interchange of ϕ and ρ is needed to represent the fact that after evaluating the actual parameter in the environment of the procedure calling statement, the control passes back to the procedure body.†

† It should not be difficult to see that coroutines can be represented by using the same idea, as follows: the "remainder" of each coroutine may be represented by a different variable. The coroutine calls are then representable by the LE's which simply permute these variables to bring the remainder of the called coroutine in front. We will soon see how we can also account for the private variables of a coroutine by "covering" them when the control passes out of it and "uncovering" them on return.

The purpose of the procedure calling statement itself is three-fold:

(a) to extend the environment from (y) to $(x_\varepsilon, x_\alpha, x, y)$
(b) to initialize the added variables; that is, substitute $\varepsilon_x{}^a$ for x_ε, $\alpha_x{}^a$ for x_α, and, by convention, Ω for x.

(c) to apply P before applying the program remainder; that is, substitute $P\phi$ for ϕ

Consequently, the statement $P(y)$ above may be represented by the *LE*

$$(\lambda x_\varepsilon x_\alpha x{:}(\lambda\phi y{:}P\phi x_\varepsilon x_\alpha xy))\varepsilon_x{}^a\alpha_x{}^a\Omega,$$

or, more simply, by

$$\lambda\phi y{:}P\phi\varepsilon_x{}^a\alpha_x{}^a\Omega y.$$

Putting together the representations obtained piece-meal above, and adding the ones for the assignment and the block, we can now complete the representation of the program:

Example.

begin integer y;
  procedure $P(x)$; **integer** x;

$$\varepsilon_x{}^f \equiv \lambda\phi\rho x_\varepsilon x_\alpha x{:}x_\varepsilon\rho\phi x_\varepsilon x_\alpha x$$
$$\alpha_x{}^f \equiv \lambda\phi\rho x_\varepsilon x_\alpha x{:}x_\alpha\rho\phi x_\varepsilon x_\alpha x$$
$$a \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\phi\rho x_\varepsilon x_\alpha(+x\underline{2})y$$
$$b \equiv \lambda\phi{:}\varepsilon_x{}^f(a(\alpha_x{}^f\phi))$$
$$P \equiv b(\lambda\rho x_\varepsilon x_\alpha x{:}\rho)$$
$$c \equiv \lambda\phi y{:}\phi\underline{1}$$
$$d \equiv \lambda\phi y{:}P\phi\varepsilon_x{}^a\alpha_x{}^a\Omega y$$
$$\varepsilon_x{}^a \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\rho\phi x_\varepsilon x_\alpha yy$$
$$\alpha_x{}^a \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\rho\phi x_\varepsilon x_\alpha xx$$
$$e \equiv \lambda\phi{:}c(d(\lambda y{:}\phi))\Omega$$
$$\{\text{program}\} \equiv e\mathbf{\Pi}.$$

    $x := x+2;$
    $y := 1;$
    $P(y)$
end

Next, let us consider the representation of type procedures in which a value is associated with the procedure identifier. In this case we will use an additional variable π to denote the procedure value in representing the statements of the procedure body. The representations are otherwise similar to that for the untyped procedures discussed above. A statement in which the function designator of a procedure is used as an operand of an expression will be represented as if it were compounded of two statements—the first a procedure call to obtain the value of the procedure, and the second using that value in the expression.

The representation of a type procedure is shown in the following example, which also illustrates the treatment of call-by-value in our present scheme of procedure representation. (Some explanation follows the program.)

Example.

begin integer u, v;
  integer procedure $P(x, y)$;
    integer x, y; **value** y;

$$\varepsilon_x{}^f \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xy{:}x_\varepsilon\rho\phi\pi x_\varepsilon x_\alpha xy$$
$$\alpha_x{}^f \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xy{:}x_\alpha\rho\phi\pi x_\varepsilon x_\alpha xy$$

```
begin
  P := x−y;

  x := y

  end of compound
end of P;
u:=v:=3;
u:=P(v, u+1) + u;

end
```

$$
\begin{aligned}
a &\equiv \lambda\phi\rho\pi x_e x_\alpha xyuv{:}\phi\rho(-xy)x_e x_\alpha xyuv\\
b &\equiv \lambda\phi{:}\varepsilon_x^{\,f}(a\phi)\\
c &\equiv \lambda\phi\rho\pi x_e x_\alpha xyuv{:}\phi\rho\pi x_e x_\alpha yyuv\\
d &\equiv \lambda\phi{:}c(\alpha_x^{\,f}\phi)\\
e &\equiv \lambda\phi{:}b(d\phi)\\
P &\equiv e(\lambda\rho\pi x_e x_\alpha xy{:}\rho\pi)\\
f &\equiv \lambda\phi uv{:}\phi\underline{3}\,\underline{3}\\
g &\equiv \lambda\phi uv{:}P\phi\Omega\varepsilon_x^{\,a}\alpha_x^{\,a}\Omega(+u\underline{1})uv\\
\varepsilon_x^{\,a} &\equiv \lambda\phi\rho\pi x_e x_\alpha xyuv{:}\rho\phi\pi x_e x_\alpha vyuv\\
\alpha_x^{\,a} &\equiv \lambda\phi\rho\pi x_e x_\alpha xyuv{:}\rho\phi\pi x_e x_\alpha xyux\\
h &\equiv \lambda\phi\pi uv{:}\phi(+\pi u)v\\
k &\equiv \lambda\phi{:}g(h\phi)\\
m &\equiv \lambda\phi{:}f(k(\lambda uv{:}\phi))\Omega\Omega\\
\{\text{program}\} &\equiv m\Pi.
\end{aligned}
$$

The environment of the statements in the procedure above consists of eight variables: the return variable ρ, the procedure value variable π, the three variables x_e, x_α, and x for the called-by-name parameter x, the single called-by-value parameter variable y, and finally the two global variables u and v. Of these, the four parameter variables are effectively discarded at the end of the procedure body execution by the component $(\lambda\rho\pi x_e x_\alpha xy{:}\rho\pi)$ of P above. The procedure call is represented as the compound of two statements f and g: f computes π, the procedure value, and g makes use of this in the assignment statement.

In both previous examples, the environments of the procedure declaration and the procedure call are the same. In the general case, these environments may be different; this is so, for example, when a procedure call takes place in a block enclosed by the block that declares the procedure. When this happens, there arises the problem of "covering" the local variables of the calling point whose scopes do not include the procedure declaration. Of course, the covering must be such that the variables may be "uncovered" on return to the calling point. Notice the contrast with jumps in which the variables that do not have valid declarations at the jump label are simply discarded permanently. Covering is also needed in specifying the formal evaluation and assignment operators for use with statements inside a block in a procedure body, since in this case, again, the variables local to the procedure body are invisible at the calling point.

The following example shows a way of covering the non-overlapping parts of the environment, in order to overcome the environment conflict problem. (See the explanation below.)

Example.

```
begin integer x;
  procedure P(y); integer y;
    begin integer z;

      z := y+3;

    end of block
  end of P;
```

$$
\begin{aligned}
\varepsilon_y^{\,f} &\equiv \lambda\phi z\rho y_e y_\alpha y{:}\rho y_e\langle\phi,z\rangle y_e y_\alpha y\\
\alpha_y^{\,f} &\equiv \lambda\phi z\rho y_e y_\alpha y{:}\rho y_\alpha\langle\phi,z\rangle y_e y_\alpha y\\
a &\equiv \lambda\phi z\rho y_e y_\alpha yx{:}\phi(+y\underline{3})\rho y_e y_\alpha yx\\
b &\equiv \lambda\phi{:}\varepsilon_y^{\,f}(a\phi)\\
c &\equiv \lambda\phi{:}b(\lambda z{:}\phi)\Omega\\
P &\equiv c(\lambda\rho y_e y_\alpha y{:}\rho\mathrm{I})
\end{aligned}
$$

```
begin integer u;
  P(u+x);
  ...
end
end.
```

$$d \equiv \lambda\phi ux{:}P\langle\phi,u\rangle\varepsilon_y{}^a\Omega\Omega x$$
$$\varepsilon_y{}^a \equiv \lambda\phi u\rho y_\varepsilon y_\alpha yx{:}\rho\mathbf{I}\langle\phi,u\rangle y_\varepsilon y_\alpha(+ux)x$$

In representing the procedure call in the above example, $\langle\phi, u\rangle$ is passed as the return point argument instead of ϕ, thus covering u. Since $\langle\phi, u\rangle A \to A\phi u$, for all LE's A, the application of $\langle\phi, u\rangle$ to any LE has the effect of uncovering u and restoring the environment; e.g. in $\varepsilon_y{}^f$, the application is made to y_ε, and in P, to $\mathbf{I}$. Note that in the representation of the procedure call, namely, d, we have used Ω for what would otherwise have been $\alpha_y{}^a$; this is so, because no assignment can be made to the particular actual parameter in this case.

The evaluation and assignment operators, both formal and actual, have been defined above slightly differently than in the two previous examples in which covering was not required. These two examples are worked out once again so as to make the treatment uniform, whether or not covering is needed in a particular case.

Example.

```
begin integer y;
  procedure P(x); integer x;

  x := x+2;

  y := 1;
  P(y)

end
```

$$\varepsilon_x{}^f \equiv \lambda\phi\rho x_\varepsilon x_\alpha x{:}\rho x_\varepsilon\langle\phi\rangle x_\varepsilon x_\alpha x$$
$$\varepsilon_x{}^f \equiv \lambda\phi\rho x_\varepsilon x_\alpha x{:}\rho x_\alpha\langle\phi\rangle x_\varepsilon x_\alpha x$$
$$a \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\phi\rho x_\varepsilon x_\alpha(+x\underline{2})y$$
$$b \equiv \lambda\phi{:}\varepsilon_x{}^f(a(\alpha_x{}^f\phi))$$
$$P \equiv b(\lambda\rho x_\varepsilon x_\alpha x{:}\rho\mathbf{I})$$
$$c \equiv \lambda\phi y{:}\phi\underline{1}$$
$$d \equiv \lambda\phi y{:}P\langle\phi\rangle\varepsilon_x{}^a\alpha_x{}^a\Omega y$$
$$\varepsilon_x{}^a \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\rho\mathbf{I}\langle\phi\rangle x_\varepsilon x_\alpha yy$$
$$\alpha_x{}^a \equiv \lambda\phi\rho x_\varepsilon x_\alpha xy{:}\rho\mathbf{I}\langle\phi\rangle x_\varepsilon x_\alpha xx$$
$$e \equiv \lambda\phi{:}c(d(\lambda y{:}\phi))\Omega$$
$$\{\text{program}\} \equiv e\mathbf{II}$$

Example.

```
begin integer u, v;
  integer procedure P(x,y);
  integer x,y; value y:

  begin
    P := x−y;

    x := y;

    end of compound
  end of P;
  u := v := 3;
  u := P(v,u+1)+u;
```

$$\varepsilon_x{}^f \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xy{:}\rho x_\varepsilon\langle\phi\rangle\pi x_\varepsilon x_\alpha xy$$
$$\alpha_x{}^f \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xy{:}\rho x_\alpha\langle\phi\rangle\pi x_\varepsilon x_\alpha xy$$
$$a \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xyuv{:}\phi\rho\ (-xy)x_\varepsilon x_\alpha xyuv$$
$$b \equiv \lambda\phi{:}\varepsilon_x{}^f(a\phi)$$
$$c \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xyuv{:}\phi\rho\pi x_\varepsilon x_\alpha yyuv$$
$$d \equiv \lambda\phi{:}c(\alpha_x{}^f\phi)$$
$$e \equiv \lambda\phi{:}b(d\phi)$$
$$P \equiv e(\lambda\rho\pi x_\varepsilon x_\alpha xy{:}\rho\mathbf{I}\pi)$$
$$f \equiv \lambda\phi uv{:}\phi\underline{3}\,\underline{3}$$
$$g \equiv \lambda\phi uv{:}P\langle\phi\rangle\Omega\varepsilon_x{}^a\alpha_x{}^a\Omega(+u\underline{1})uv$$
$$\varepsilon_x{}^a \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xyuv{:}\rho\mathbf{I}\langle\phi\rangle\pi x_\varepsilon x_\alpha vyuv$$
$$\alpha_x{}^a \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xyuv{:}\rho\mathbf{I}\langle\phi\rangle\pi x_\varepsilon x_\alpha xyux$$

$$h \equiv \lambda\phi\pi uv{:}\phi(+\pi u)v$$
$$k \equiv \lambda\phi{:}g(h\phi)$$
$$l \equiv \lambda\phi{:}f(k(\lambda uv{:}\phi))\Omega\Omega$$
$$\{\text{program}\} \equiv l\mathrm{I}.$$

end

A procedure body may contain a procedure call, possibly a recursive one, in which the formal parameters are used in actual parameter expressions. And the parameters of the nested call may themselves by called be name. The representation in such a case requires the covering of all the variables associated with the procedure body, including the local variables, the return variable, and the parameter variables. This is illustrated below.

Example.

```
begin integer x;
  procedure P(y,n); integer y,n; value n;
    begin
      ...
    end P;
  procedure Q(z); integer z;
    begin integer w;
      P(z,x);
      ...
    end Q;
  ...
end.
```

If the representation of the body of the procedure P is a, then the representation of P itself is

$$P \equiv a(\lambda\rho y_\varepsilon y_\alpha yn{:}\rho\mathrm{I}).$$

The representation of the statement $P(z, x)$ is

$$\lambda\phi{:}\varepsilon_z^{\,1}(b\phi),$$

where $\varepsilon_z^{\,1}$ is the formal evaluation operator for z in Q, and b represents the call on P, as follows:

$$b \equiv \lambda\phi w\rho z_\varepsilon z_\alpha zx{:}P\langle\phi,w,\rho,z_\varepsilon,z_\alpha,z\rangle\varepsilon_y{}^a\alpha_y{}^a\Omega xx$$
$$\varepsilon_y{}^a \equiv \lambda\phi w\rho z_\varepsilon z_\alpha z\rho_1 y_\varepsilon y_\alpha yx{:}\rho_1\mathrm{I}\langle\phi,w,\rho,z_\varepsilon,z_\alpha,z\rangle y_\varepsilon y_\alpha zx$$
$$\alpha^a{}_y \equiv \lambda\phi w\rho z_\varepsilon z_\alpha z\rho_1 y_\varepsilon y_\alpha yx{:}\rho_1\mathrm{I}\langle\phi,w,\rho,z_\varepsilon,z_\alpha,y\rangle y_\varepsilon y_\alpha yx.$$

The next example illustrates the representation of a procedure calling statement in which an actual parameter itself consists of a call on a procedure.

Example.

```
begin integer x;
  procedure P(r,s); integer r,s; begin ... end P;
  procedure Q(t); integer t; begin ... end Q;
  begin integer y;
    ... ; P(x,Q(y)); ...
  end
end.
```

Because the second actual parameter, $Q(y)$, in the above procedure calling statement $P(x, Q(y))$ does not require an assignment operator,† the latter statement is represented by the *LE*

$$\lambda\phi yx{:}P\langle\phi,y\rangle\varepsilon_\tau{}^a\alpha_\tau{}^a\Omega\varepsilon_s{}^a\Omega\Omega x.$$

The first actual parameter, x, poses no problem, other than the covering of the variable y not visible to the procedure declaration of P; hence, we define

$$\varepsilon_\tau{}^a \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r s_\varepsilon s_\alpha s x{:}\rho\mathrm{I}\langle\phi,y\rangle r_\varepsilon r_\alpha x s_\varepsilon s_\alpha s x,$$
$$\alpha_\tau{}^a \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r s_\varepsilon s_\alpha s x{:}\rho\mathrm{I}\langle\phi,y\rangle r_\varepsilon r_\alpha r s_\varepsilon s_\alpha s r.$$

For the second actual parameter, $Q(y)$, things are slightly more complex. (Note, however, that only an evaluation operator is needed in this case; the assignment operator is undefined.) First, we have to provide for a call on Q—which requires covering all the variables associated with the call on P—with the following actual evaluation and assignment operators:

$$\varepsilon_t{}^a \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r s_\varepsilon s_\alpha s\rho_1\pi t_\varepsilon t_\alpha t x{:}\rho_1\mathrm{I}\langle\phi,y,\rho,r_\varepsilon,r_\alpha,r,s_\varepsilon,s_\alpha,s\rangle\pi t_\varepsilon t_\alpha y x$$
$$\alpha_t{}^a \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r s_\varepsilon s_\alpha s\rho_1\pi t_\varepsilon t_\alpha t x{:}\rho_1\mathrm{I}\langle\phi,t,\rho,r_\varepsilon,r_\alpha,r,s_\varepsilon,s_\alpha,s\rangle\pi t_\varepsilon t_\alpha t x.$$

Now, $\varepsilon_s{}^a$ is defined in terms of a call on Q, followed by an assignment of the resulting value to s, as follows:

$$a \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r s_\varepsilon s_\alpha s x{:}Q\langle\phi,y,\rho,r_\varepsilon,r_\alpha,r,s_\varepsilon,s_\alpha,s\rangle\Omega\varepsilon_t{}^a\alpha_t{}^a\Omega x,$$
$$b \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r s_\varepsilon s_\alpha s\pi x{:}\rho\mathrm{I}\langle\phi,y\rangle r_\varepsilon r_\alpha r s_\varepsilon s_\alpha\pi x,$$
$$\varepsilon_s{}^a \equiv \lambda\phi{:}a(b\phi).$$

5.4. *Label parameters*

The representation of label parameters is actually much simpler than of the integer parameters. The reason is that two different operations, evaluation and assignments, are possible with the latter type; in addition, the value of the parameter at any time has to be carried also along within the representation. In the case of a label parameter, the only possible actual operation is a jump to it. Thus, with each formal label parameter, p, we need to associate only one variable, denoted by p_γ, which is to be bound to the operator for effecting the actual **goto** operation. (The variable p_γ is, of course, an element of the environment of the procedure body.) Next, associated with each actual label parameter, and individual to each procedure call, is an *LE* that represents the parameter procedure to effect the jump to the actual label. For a parameter p, this "actual **goto**" operator is denoted by $\gamma_p{}^a$, with further distinguishing marks added when more than one procedure call is involved. Last, associated with each formal label parameter, and unique to each environment within the procedure body, is an *LE*, the "formal **goto**" operator, that represents a call on the actual parameter procedure, that is, an application of the actual **goto** operator; the formal **goto** operator for the parameter p is denoted $\gamma_p{}^f$, again with further distinguishing marks added if more than one environment is involved.

In conjunction with our detailed treatment of jumps (Section 4.3) and procedures with integer parameters (Section 5.3), the example below should suffice to explain how to represent label parameters.

† As explained earlier, an assignment operator is required for those actual parameters which consist of a single program variable.

Example.

```
begin integer q;
procedure R(v); label v;
  goto v;

begin integer r;
  procedure P(x,z); integer x; label z;

    R(z);

  begin integer s,t;
      P(t,L)

    end;
  L: . . .
  end
end.
```

$$a \equiv \gamma_v^{\,f} \equiv \lambda\phi\rho v_\gamma{:}\rho v_\gamma\langle\phi\rangle v_\gamma$$
$$R \equiv a(\lambda\rho v_\gamma{:}\rho\mathbf{I})$$

$$\varepsilon_x^{\,f} \equiv \lambda\phi\rho x_\varepsilon x_\alpha x z_\gamma{:}\rho x_\varepsilon\langle\phi\rangle x_\varepsilon x_\alpha x z_\gamma$$
$$\alpha_x^{\,f} \equiv \lambda\phi\rho x_\varepsilon x_\alpha x z_\gamma{:}\rho x_\alpha\langle\phi\rangle x_\varepsilon x_\alpha x z_\gamma$$
$$\gamma_z^{\,f} \equiv \lambda\phi\rho x_\varepsilon x_\alpha x z_\gamma{:}\rho z_\gamma\langle\phi\rangle x_\varepsilon x_\alpha x z_\gamma$$
$$b \equiv \lambda\phi\rho x_\varepsilon x_\alpha x z_\gamma rq{:}$$
$$R\langle\phi,\rho,x_\varepsilon,x_\alpha,x,z_\gamma,r\rangle\gamma_v^{\,a}q$$
$$\gamma_v^{\,a} \equiv \lambda\phi\rho x_\varepsilon x_\alpha x z_\gamma r\rho_1 v_\gamma q{:}\gamma_z^{\,f}\phi\rho x_\varepsilon x_\alpha x z_\gamma rq$$

$$c \equiv \lambda\phi strq{:}P\langle\phi,s,t\rangle\varepsilon_x^{\,a}\alpha_x^{\,a}\Omega\gamma_z^{\,a}rq$$
$$\varepsilon_z^{\,a} \equiv \lambda\phi st\rho x_\varepsilon x_\alpha x z_\gamma rq{:}$$
$$\rho\mathbf{I}\langle\phi,s,t\rangle x_\varepsilon x_\alpha t z_\gamma rq$$
$$\alpha_x^{\,a} \equiv \lambda\phi st\rho x_\varepsilon x_\alpha x z_\gamma rq{:}$$
$$\rho\mathbf{I}\langle\phi,s,x\rangle x_\varepsilon x_\alpha x z_\gamma rq$$
$$\gamma_z^{\,a} \equiv \lambda\phi st\rho x_\varepsilon x_\alpha x z_\gamma rq{:}Lrq$$

6. CONCLUSION

By interpreting programming language constructs intuitively as functions rather than machine commands, we have succeeded in modelling programming languages in the pure lambda-calculus. An immediate application of our model is in a functional (as opposed to computational) semantic definition of high-level programming languages, as the lambda-calculus interpretations of the individual programming constructs can themselves be taken as *the* semantic specification of the constructs. Of more interest, however, is the potential of the present model in studying the properties of programs—such as, convergence, correctness, and equivalence—and in performing useful program transformations—such as program simplification (source code level) and optimization (compiled code level). Since we describe a program as a lambda-expression, the above described applications essentially reduce to transformations within the lambda-calculus. The possibilities of some of these applications have been indicated by examples in the body of the paper. In the case of loop-free programs, these applications most often involve straightforward lambda-calculus reduction. For the programs containing loops, our proofs of correctness and equivalence are rather ad hoc; the development of systematic methods to deal with these applications warrants further research.

REFERENCES

1. J. H. Morris, Lambda-calculus models of programming languages, Ph.D. Dissertation, Project MAC, MIT, MAC-TR-57 (1968).
2. B. K. Rosen, Tree-manipulating systems and Church–Rosser theorems, *J. ACM* **20**, 160 (1973).

3. J. W. deBakker, *Recursive Procedures*, Mathematical Center, Amsterdam (1972).
4. S. C. Kleene, *Introduction to Metamathematics*, van Nostrand, Princeton, NJ (1950).
5. D. Scott, Continuous lattices, in *Proc.* 1971 *Dalhousie Conf.*, *Springer-Verlag Lecture Notes in Maths* #274, Springer, Berlin (1972).
6. P. Naur (Ed.) Revised Report on the algorithmic language ALGOL 60, *Comm. ACM* **6**, 1 (1963).
7. B. Randell and L. J. Russel, *ALGOL* 60 *Implementation*, Academic Press, NY (1964).