Journal of Computer Languages, Vol. 1, pp. 287–301. Pergamon Press, 1976. Printed in Northern Ireland

A LAMBDA-CALCULUS MODEL OF PROGRAMMING LANGUAGES—I. SIMPLE CONSTRUCTS†

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—A simple correspondence is presented between the elementary constructs of programming languages and the lambda-calculus. The correspondence is obtained by using intuitive, functional interpretations of programming language constructs, completely avoiding the notions of machine memory and address. In particular, the concepts of program variable and assignment are accounted for in terms of the concepts of mathematical variable and substitution, respectively. Lambda-expression representations are given of assignments, conditional and compound statements, input–output, and blocks. Algol 60 is used as the illustrative language.

Programming Language Semantics Formal Semantics Lambda-Calculus ALGOL 60 Block Structure Assignment vs Substitution Program Correctness

1. INTRODUCTION

OUR AIM in this paper is to explicate the concepts of programming languages in terms of the lambda-calculus. Before we can proceed, we have to be definite about what is to be regarded as the meaning of a program. We take the view that the meaning of a program is a function. A program prescribes the computational steps which produce the value of some function corresponding to a given value of the function argument. It is precisely the function computed by a program that we take to be the meaning of the program. Consequently, in order to model programming languages mathematically, we seek to formulate rules for deriving the mathematical definitions of functions from the computational representations of functions provided in the form of programs.

The problem of obtaining the mathematical definition of a function from the text of a program computing that function is quite non-trivial. To describe computations, programming languages make use of a number of concepts that are not present in the customary mathematical notations for representing functions. Central to the present-day programming languages—and the main source of difference between the diction of mathematics and that of programming languages—is the notion of computer memory. Consider, for example, the concept of program variable. Whereas a mathematical variable denotes a value, a program variable denotes an address in the computer memory. Or, compare the notion of function used in mathematics with the notion of procedure used in programming languages. Again, whereas the evaluation of functions requires straightforward substitution of the argument values in the functional definitions, the execution of procedures requires rather elaborate manipulation of information involving a complex of memory locations.

For a long time now, two types of constituents have been distinguished in programming languages [1]—the *descriptive* elements, such as expressions and functions, and the *imperative* elements, such as assignments, instruction sequencing, and jumps. In addition, high level programming languages also contain *declarative* elements, such as type, array,

† The material in this paper has been derived from the author's Ph.D. Thesis [11]. The author wishes to thank Professor George W. Petznick for his guidance and help in the research for the thesis. The preparation of this paper was partially supported by the NSF Grant GJ 25393.

and block declarations, and name and value specifications for procedure parameters. Although the descriptive and the declarative elements are often lumped together [1, 2], we feel that these two classes should be recognized as quite distinct; while the purpose of the former is mainly to designate values, the purpose of the latter is to remove ambiguities and to impose structure on program and data. Indeed, the declarative features often serve to interconnect the descriptive and the imperative components of a program.

The imperative constituents have their origin in machine languages from which the present-day high-level programming languages have evolved. The descriptive constituents have been introduced for ease and conciseness of notation, as well as for making the programs resemble more closely the functions they compute. The addition of such constituents to programming languages thus represents a step away from the machine and towards mathematics. The declarative constituents have been added for, ostensibly, improving the clarity and transparency of programs. But in actual fact, by introducing sophisticated address-related concepts many declarative features represent a step back to the machine, and their presence often makes the recognition and extraction of the functional meaning of a program more difficult than would be in their absence.

The descriptive features of programming languages lend themselves to mathematical interpretation in quite a natural manner. It is the presence of imperative and certain declarative features that obscures the functional meaning of a program. And the essence of that obscurity is in the dependence of these features upon the concept of computer memory.

Thus, the key to the extraction of functional meaning of programs lies in modelling programming constructs without using the idea of memory address. Such modelling is the task that we undertake to do in this paper. In particular, we seek to explain program variables in terms of mathematical variables, the operation of assignment in terms of substitution, and procedures, programs, and, in general, all program statements in terms of functions.

To express programming constructs, we make use of the notation and terminology of ALGOL 60 [3], with a few, explicitly stated, extensions. As the mathematical theory for modelling the programming constructs, we make use of the (untyped) lambda-calculus [4–6], in which functions are representable in a very natural manner. It should be noted, however, that there may be features of modern programming languages which do not fit easily into this approach, such as pointers, controlled and based variables, associative retrieval, etc.

Since a number of programming language models based on the lambda-calculus have already appeared in the References [1, 2, 7–9], a comparison of our model with others is in order.

1. Our model does not introduce any imperative or otherwise foreign notions to the lambda-calculus. This is in contrast to Landin [1], in which the imperative features of programming languages are accounted for by ad hoc extensions of the lambda-calculus. We find that the calculus, in its purity, suffices as a natural model of programming languages. By not making any additions to the calculus, we have the guarantee that all its properties, in particular, the consistency and the Church–Rosser property [5], are valid in our model. For example, even when a program requires a fixed order of execution, the normal form obtained by evaluating the program representation in any order, whatsoever, represents the program result correctly.

2. In our model, programs are *translated* into lambda-expressions, not *interpreted* by a lambda-calculus interpreter [9]. Thus, programming semantics is completely reduced to

the lambda-calculus semantics, but without commitment to any particular view of the latter. Also, all lambda-expression transformations are applicable to program representations.

3. We model assignments by the substitution operation of the lambda-calculus. Consequently, the notions of memory, address, and fetch and store operations do not enter our model in any explicit manner [2, 9].

4. We represent high-level programming language constructs directly, not in terms of the representations of the machine level operations [7] of the compiled code.

5. Our model potentially spans the full ALGOL 60 language. It is also applicable to a number of other advanced programming features, such as collateral statements, the use of labels and procedures as assignable values, coroutines, etc.

6. As a matter of opinion, it seems that our representations are much simpler and clearer than the ones given in other models.

We describe the model informally, and only for a representative set of programming language constructs. But we provide enough motivating details and illustrations to, hopefully, convey the method and suggest its extension to other programming features. (More detailed treatment can be found in [10, 11].) This part of the paper describes the representation of block structured programs composed of assignment, conditional, and input–output statements. Part II will extend the model to jump statements, loops, and procedures.

2. THE LAMBDA-CALCULUS

To fix our terminology and notation for the *lambda-calculus* (*LC*), we include the following definitions, referring the reader to [4–6] for more information.

We assume given a collection of entities called *indeterminates*. A *lambda-expression* (*LE*) is recursively defined to be either an indeterminate, or the *application* (*ef*) of an *LE e* to an *LE f*, or the *abstraction* (λx:*e*) of an *LE e* with respect to an indeterminate *x*. In writing *LE*'s, parentheses may be omitted under the convention that applications associate to the left and abstractions to the right, with the former taking precedence over the latter. For instance,

$$(\lambda x\colon (\lambda y\colon (\lambda z\colon (((xz)(yz))u))))$$

may be abbreviated to

$$\lambda x{:}\lambda y{:}\lambda z{:}xz(yz)u.$$

As an additional convention, the above may be further abbreviated to

$$\lambda xyz{:}xz(yz)u.$$

Identity of *LE*'s is indicated by the symbol '$\equiv$', which is also used as a definition symbol.

In the *LE* (λx:*e*), the first occurrence of *x* is a *binding* occurrence, and *e* is the *range* of that occurrence. An occurrence of an indeterminate in an *LE* is *bound* if it is either binding or in the range of any binding occurrence of the same indeterminate; otherwise, the occurrence is *free*. If $e, f_1, \ldots, f_n$ are *LE*'s and $x_1, \ldots, x_n$ indeterminates, then

$$\text{sub}\,[f_1, x_1; \ldots; f_n, x_n; e]$$

denotes the *result of simultaneously substituting* f_i *for all free occurrences of* x_i $(1 \leq i \leq n)$ *in e*.

The basic *LC* rules of transformation, called *contractions*, are these:

(α) $\lambda x{:}e \to_\alpha \lambda y{:}\text{sub}\,[y, x; e]$,
provided that y has no free occurrences in e, and no free occurrence of x in e becomes a bound occurrence of y by the substitution.

(β) $(\lambda x{:}e)f \to_\beta \text{sub}\,[f, x; e]$,
provided that no indeterminate with free occurrences in f has bound occurrences in e.

(η) $\lambda x{:}ex \to_\eta e$,
provided that x has no free occurrences in e.

The converse of β- (or η-) contraction is called β- (or η-) *expansion*. Contractions and expansions may be applied to the whole or a part (which must itself be an *LE*) of an *LE*. *Reduction* ($\to$) consists of a (possibly empty) sequence of contractions. As an example of reduction, it can be shown that

$$(\lambda x_1 \ldots x_n{:}e)f_1 \ldots f_n \to \text{sub}\,[f_1, x_1; \ldots; f_n, x_n; e],$$

provided that no indeterminate with free occurrences in f's has bound occurrences in e. *Conversion* ($\leftrightarrow$) consists of a (possibly empty) sequence of contractions and expansions. If $e \leftrightarrow f$, then there exists an *LE* g such that $e \to g$ and $f \to g$ (Church–Rosser Theorem [5]). An *LE* is *irreducible* if no β- or η-contraction is applicable to it even after any α-contractions. Given an *LE* e, if there exists an irreducible *LE* f such that $e \leftrightarrow f$, then f is a *normal form* of e. If e possesses a normal form f, then f is unique up to the applications of α-contractions, and, moreover, $e \to f$; f can be obtained from e by using, among other possibilities, the *standard-order reduction algorithm*, in which one always chooses the leftmost applicable β- or η-contraction at each step in reduction.

A natural interpretation of *LE*'s is as functions whose domains and ranges themselves contain functions. With the indeterminates regarded as arbitrary functions, the *LE* (fe) may be regarded as the functional expression $f(e)$, and $(\lambda x{:}e)$ as the function f defined by $f(x) = e$ in the customary functional notation. With this interpretation, the *LE* $(e_1e_2e_3 \ldots e_n)$ may be viewed variously as $e_1(e_2, \ldots, e_n)$, or $(e_1(e_2))(e_3, \ldots, e_n)$, etc.

Following are some definitions and abbreviations that will be used later:

$$\begin{aligned} \langle a_1, \ldots, a_n\rangle &\equiv \lambda x{:}xa_1 \ldots a_n, \qquad n \geq 0 \\ \mathbf{I} &\equiv \lambda x{:}x \\ \boldsymbol{\Omega} &\equiv (\lambda xy{:}xx)(\lambda xy{:}xx) \\ \mathbf{Y} &\equiv \lambda x{:}(\lambda y{:}x(yy))(\lambda y{:}x(yy)) \\ a;b &\equiv \lambda x{:}axb. \end{aligned} \tag{1}$$

3. BASIC PROGRAMMING FEATURES

3.1. *An overview*

There are several different ways in which a program may be regarded as a function, depending upon what we consider to be the arguments of the program and what we regard as the finally computed results. These different functional interpretations of programs may result in different choices of the representations of individual programming constructs. We will take the view that it is the external input–output behavior that most appropriately characterizes a program, and choose our representations with the goal of making this

behavior as explicit as possible. Consider, e.g. the program:

begin integer a, b, c;
read a; **read** c; $b := a+c$; **write** b; $b := b-2\times c$; **write** b
end.

(We use **read** and **write** as standard statements for performing single-item input–output operations.) As far as the external input–output is concerned, the above program behaves like a two-argument function which produces as value two quantities, the sum and difference of its arguments. Thus, this program may be intuitively interpreted as the function f given by

$$f(x, y) = \langle x + y, x - y \rangle.$$

In LC notation, this function can be expressed by

$$\lambda xy{:}\langle x + y, x - y \rangle. \tag{i}$$

(Although the expressions $x + y$ and $x - y$ are not exactly *LE*'s, they can be easily translated to be such.) Accordingly, we would like to set up our model in such a manner that the representation of the above program may turn out to be the *LE* (i) (or an *LE* which can be reduced to (i)). So constructed, the model would, in essence, enable us to abstract out of a program code the function from the input space to the output space that the program computes.

More generally, let P be a program, $i_1, \ldots, i_p$ its inputs, and $o_1, \ldots, o_q$ its outputs. Then we would like that the representations provided by our model satisfy the reduction relation

$$\{P\}\{i_1\} \ldots \{i_p\} \rightarrow \langle \{o_1\}, \ldots, \{o_q\} \rangle, \tag{2}$$

in which the representations are denoted (anticipating a forthcoming notation) by enclosing within braces the symbols for the corresponding represented entities.

The subsequent sections will show how the representations of various programming constructs may be chosen to fulfil the above requirement.

3.2. *Variables*

For each program that we wish to represent in our model, we shall need a fixed correspondence between the variables declared in the program and the indeterminates. In block-structured programming languages, it is permissible to employ the same identifier to denote different program variables as long as those variables have different scopes. In choosing the correspondence between program variables and indeterminates, it will not be necessary to distinguish between any two identically denoted variables that are declared in disjoint blocks. But it will be necessary to distinguish all the variables that are declared in a set of nesting blocks. (To distinguish two such variables denoted by the same identifier, it suffices, for example, to superscript the identifier by the respective block level numbers—zero for the outermost (program) block, and $n + 1$ for the blocks immediately enclosed by a block at level n.) The above mentioned correspondence will be set up by assigning distinct indeterminates to distinct variables (in the above sense) in some order.† Since we have not

† We assign single indeterminates to both simple variables and array variables. However, arrays will not be discussed in this paper; the reader is referred to [10, 11] for their treatment.

specified the notation for indeterminates, we shall, for convenience in expressing our representations, denote the indeterminates by the same symbols by which the corresponding program variables are denoted.

The representation of a programming construct in a given program depends upon, among other things, the variable declarations in whose scope the construct appears. To account for this, we need the following notion: The *environment* of a construct in a program is a list of all the program variables that have been declared in the blocks enclosing the point at which the construct occurs. In this list, the variables are to be arranged in their order of declaration within individual blocks, with the blocks taken in the innermost to the outermost order.†

From what has been stated earlier about distinguishing program variables, it follows that the variables constituting an environment are all distinct.

Example. Consider the following schematic program, in which it is assumed that the omitted statements indicated by ellipses do not contain declaration.

```
A: begin integer x, y;
     B: . . . ;
     begin integer y, z;
       C: . . . ;
     end;
     begin integer x, w, z;
       D: . . . ;
     end
   end.
```

From our view-point, this program makes use of six distinct variables, namely, x^0, y^0, y^1, z^1, x^1 and w^1, where the superscripts indicate block level numbers. For simplicity, let us omit the superscripts from x^0, y^0, z^1, and w^1. Then the environments of the statements labelled A (i.e. the whole program), B, C and D are, respectively, the null list ( ), (x, y), (y^1, z, x, y) and (x^1, w, z, x, y).

In general, the representation of a construct depends upon the construct's own environment as well as the environments of its constituents. We denote the representation of a construct X appearing in the environment E by $\{X\}_E$; we drop the subscript if X is a constant or if the representation of X is the same in all environments (in which X can legally occur).

A formula specifying the representation of a construct in terms of its environment and the representations and environments of its constituents will be referred to as a *representation rule*. In such a formula, the environments of the constituents will generally be omitted if they are the same as the environment of the construct under representation.

3.3. *Constants, operations, relations, expressions*

The *LC* representation of the natural number arithmetic is well-known [4]. The representation can be easily extended to signed integers and rational numbers also [11]. Since this paper is not concerned with the purely numerical aspects of programming, we just take for granted, without going into the details of their actual construction, the *LE*'s representing the constants, operations, and relations of various types employed in programming

† When the program contains procedures, the environments may also include formal variables and a number of other variables which are not explicitly declared in the program; these additional variables are introduced in Part II of the paper.

languages. We use the following notation for such *LE*'s: **true** and **false** denote the *LE*'s representing the corresponding boolean values; **if** denotes the representation of the connective **if . . . then . . . else**; the *LE*'s representing numbers are denoted by underlining the numerals or the expressions standing for them; the symbols of operations and relations are also used to denote the corresponding *LE*'s. We assume that the representations are defined so as to satisfy the following reduction relations:

$$+\ \underline{m}\ \underline{n} \to \underline{m+n}, \qquad (m, n \text{ being arbitrary numbers.}\dagger)$$
$$-\ \underline{m}\ \underline{n} \to \underline{m-n},$$
$$<\ \underline{m}\ \underline{n} \to \begin{cases}\textbf{true}, & \text{if } m < n,\\ \textbf{false}, & \text{otherwise,}\end{cases}$$

etc. Also, for all *LE*'s b, c, d, we require

$$\textbf{if}\ bcd \to \begin{cases}c, & \text{if } b \to \textbf{true}\\ d, & \text{if } b \to \textbf{false.}\end{cases}$$

It follows from the above that the *LC* representation of an *expression*‡ may be obtained by simply writing the expression in the pre-fix notation, with all operator-operands combinations parenthesized, and then replacing all operators and operands by their individual representations. For example, $E \equiv (x, y, z)$ being the environment, we have

$$\{x + \textbf{if}\ y \neq \underline{0}\ \textbf{then}\ z - y \div x\ \textbf{else}\ \underline{15} \times x\}_E$$
$$\equiv +x(\textbf{if}\ (\neq y\underline{0})(-z(\div yx))(\times \underline{15}x)),$$

where we denote the indeterminates representing program variables by the variable symbols themselves.

3.4. *Assignments*

Before discussing any particular type of statement, we will first indicate the general idea behind our representations. Consider a given statement S of a program. Let $(v_1, \ldots, v_n)$ be the environment of S, and denote by F the section of the program following S and extending all the way to the program end. (F will be referred to as the *program remainder*§ of S.) The two parts of the program, one consisting of F alone, and the other composed of S and F together, may be interpreted as two functions ϕ and ϕ', respectively, of the arguments $v_1, \ldots, v_n$. The effect of interposing S in the program is to transform ϕ into ϕ'. As the representation of S, therefore, we take precisely the function (to be accurate, the functional operator) σ given by

$$(\sigma(\phi))(v_1, \ldots, v_n) = \phi'(v_1, \ldots, v_n), \tag{i}$$

which accomplishes the above transformation.

Now (i) may be written in *LC* as

$$\sigma\phi v_1 \ldots v_n \to \phi' v_1 \ldots v_n, \tag{ii}$$

† Of course, distinct *LE*'s are needed to represent the addition operations on natural numbers, integers and rational numbers. For simplicity, we denote all these by the same symbol $+$, leaving the choice of the applicable *LE* dependent on the context.

‡ At present we limit ourselves to the expressions which do not contain function designators. This restriction will be lifted when procedures are discussed (Part II).

§ What is called the program remainder here and in [10] is essentially equivalent to the "continuation" proposed by Lockwood Morris and Wadsworth (see Strachey and Wadsworth [12]); the two concepts were, however, arrived at quite independently.

and hence in the more convenient functional form

$$\sigma(\phi, v_1, \ldots, v_n) = \phi'(v_1, \ldots, v_n). \tag{iii}$$

If we can express the right-hand-side of (iii) in terms of $\phi, v_1, \ldots, v_n$, and possibly some constants, then we may take (iii) as the definition of σ as a function of formal arguments $\phi, v_1, \ldots, v_n$. (Note that while the domains of the arguments $v_1, \ldots, v_n$ are the values of the corresponding program variables, the domain of ϕ consists of the program remainders considered as function. Furthermore, in view of (ii), if a statement S is modelled by the *LE* σ, then the execution of S is simulated by the reduction of the *LE*

$$\sigma\hat{\phi}\hat{v}_1 \ldots \hat{v}_n,$$

in which the symbols $\hat{v}_1, \ldots, \hat{v}_n$ denote the *LC* representations of the *values* of the corresponding variables immediately prior to the execution of S, and $\hat{\phi}$ denotes the representation of the program remainder of S.)

A key step in representing a programming language statement is thus to define a suitable equation of the form (iii) for it. The choice of ϕ' is, of course, based on our intuitive understanding of the effect of the statement.

Let us now look at the assignment statement $v_i := e$ in the environment $(v_1, \ldots, v_n)$. The ϕ' in this case is obtained from ϕ by setting the argument v_i to e. Thus, in effect, this assignment statement behaves like the function σ such that

$$\begin{aligned}(\sigma(\phi))(v_1, \ldots, v_n) &= \phi'(v_1, \ldots, v_n)\\ &= \phi(v_1, \ldots, v_{i-1}, e, v_{i+1}, \ldots, v_n).\end{aligned}$$

Accordingly, we adopt the following representation rule:

$$\{v_i := e\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n{:}\phi v_1 \ldots v_{i-1}\{e\}v_{i+1} \ldots v_n. \tag{3}$$

(We recall from Section 3.2 the convention that in a representation rule, the environments of all representations are considered to be the same as that of the construct under representation, unless specified otherwise. Also, we assume in the above representation that any type conversion needed for the assignment has been incorporated within e itself.)

The multiple assignments of ALGOL 60 and the collateral (parallel) assignments of ALGOL 68 [13] present no special problem. Omitting the formal rules for their representation, we will simply illustrate their treatment in the following example.

Example. Presented side by side below are some assignment statements and their *LC* representations. Note that the environment of the first two statements is (x, y), and of the next three is (z, y^1, x, y^0); the superscripts in the latter environment are block level numbers, and are used to distinguish the two variables designated by the same identifier.

Program	*Statement representations*
begin integer x, y;	
x:=2;	$\lambda\phi xy{:}\phi\underline{2}y$ (which $\rightarrow \lambda\phi x{:}\phi\underline{2}$)
y:=x+19;	$\lambda\phi xy{:}\phi x(+x\underline{19})$
begin integer z, y;	
y:=x−5;	$\lambda\phi zy^1xy^0{:}\phi z(-x\underline{5})xy^0$
x:=y:=$z\uparrow(x+y)$;	$\lambda\phi zy^1xy^0{:}\phi z(\uparrow z(+xy^1))(\uparrow z(+xy^1))y^0$
(y:=z, z:=y);	$\lambda\phi zy^1xy^0{:}\phi y^1zxy^0$
...	
end	
end.	

In the absence of procedures, only the variables contained in the environment of an expression can legally occur in the expression. Thus, no indeterminates other than $v_1, \ldots, v_n$ can occur in the $LE\{e\}$ in (3). It follows from the abstractions specified in that representation rule that our LC representation of an assignment statement does not contain any free indeterminates. This property holds for all *elementary* statements (e.g. assignment, input–output) from which the *composite statements* (e.g. compound, conditional, block) are constructed. From the way the latter type of statements are represented, it will follow that the property of not containing free indeterminates is enjoyed by the LC representations of *all* types of statements.

3.5. *Compound statements*

Consider the compound statement $S \equiv$ **begin** S_1; S_2 **end** appearing in the environment $(v_1, \ldots, v_n)$. Let F be the segment of the program that follows S. We can interpret the program segments F and $(S; F)$ to be two functions ϕ and ϕ', respectively, of the arguments $v_1, \ldots, v_n$, and we are interested in the representation σ of S with the functional transformation property

$$\sigma\phi v_1 \ldots v_n \rightarrow \phi' v_1 \ldots v_n.$$

Now the execution of $(S; F)$ has precisely the same effect as $(S_1; S_2; F)$. Denoting by ϕ^* the functional interpretation of the program segment $(S_2; F)$, and by σ_1 and σ_2 the representations of the statements S_1 and S_2, respectively, we have

$$\sigma_1\phi^* v_1 \ldots v_n \rightarrow \phi' v_1 \ldots v_n,$$
$$\sigma_2\phi v_1 \ldots v_n \rightarrow \phi^* v_1 \ldots v_n.$$

These relations will certainly hold if we choose $\phi^* \equiv \sigma_2\phi$, $\phi' \equiv \sigma_1(\sigma_2\phi)$, and hence, $\sigma \equiv \lambda\phi{:}\sigma_1(\sigma_2\phi)$.

The generalization to the case of an n-component compound is now obvious, and we are led to the following LC representation of compound statements:

$$\{\textbf{begin } S_1; S_2; \ldots; S_n \textbf{ end}\} \equiv \lambda\phi{:}\{S_1\}(\{S_2\}(\ldots(\{S_n\}\phi)\ldots)). \tag{4}$$

Notice the convenient fact that in the right-handside of (4) the individual statement representations appear from left to right in the same order in which the statements occur in the compound (cf. Stratchey [2]).

Example. The representation of compound statements is illustrated below. Individual statement representations are shown on the same line as the statements (on the last line for multiple-line statements), and are given names for reference purposes. The environment is assumed to be (x, y).

Statements	*Representations*
(i) **begin**	
$x := 2$;	$a \equiv \lambda\phi xy{:}\phi\underline{2}y$
$y := x+3$;	$b \equiv \lambda\phi xy{:}\phi x(+x\underline{3})$
$x := y+x$	$c \equiv \lambda\phi xy{:}\phi(+yx)y$
end	$d \equiv \lambda\phi{:}a(b(\sigma(c\phi))$

(ii) **begin**
$y := \underline{5}$; $\quad e \equiv \lambda\phi xy{:}\phi x\underline{5}$
$x := y+2$ $\quad f \equiv \lambda\phi xy{:}\phi(+y\underline{2})y$
end $\quad g \equiv \lambda\phi{:}e(f\phi)$.

The compound statements (i and ii) are intuitively equivalent. Their equivalence is derived in our model by showing that the *LE*'s representing them, namely, d and g, are mutually convertible. Specifically, it is easily seen that $d \rightarrow \lambda\phi xy{:}\phi\underline{7}\,\underline{5} \leftarrow g$. Note that the normal form $\lambda\phi xy{:}\phi\underline{7}\,\underline{5}$ of these representations corresponds to the collateral assignment statement $(x{:=}7, y{:=}5)$, which may be thought of as the *simplest* version of the above code.

3.6. *Blocks*

Next, let us consider a block S whose head declares the variables $u_1, \ldots, u_m$ and initializes these to the values† $c_1, \ldots, c_m$, and whose body consists of the statements $S_1, \ldots, S_p$, in that order. The execution of S can be broken down into three operations performed in succession:

(1) Extension of the existing environment by the variables $u_1, \ldots, u_m$ (initialized at $c_1, \ldots, c_m$).
(2) Execution of the compound **begin** $S_1; \ldots; S_p$ **end**.
(3) Deletion of the variables $u_1, \ldots, u_m$ from the environment.

Let these three operations be denoted by the functions α, β, and γ. Let $(v_1, \ldots, v_n)$ be the environment of S. Then with the obvious significance of other symbols, we have

$$(\alpha(\phi))(v_1, \ldots, v_n) = \phi(c_1, \ldots, c_m, v_1, \ldots, v_n)$$
$$(\beta(\phi))(u_1, \ldots, u_m, v_1, \ldots, v_n) = (\sigma_1(\sigma_2(\ldots(\sigma_p(\phi))\ldots)))(u_1, \ldots, u_m, v_1, \ldots, v_n)$$
$$(\gamma(\phi))(u_1, \ldots, u_m, v_1, \ldots, v_n) = \phi(v_1, \ldots, v_n)$$
$$(\sigma(\phi))(v_1, \ldots, v_n) = (\alpha(\beta(\gamma(\phi))))(v_1, \ldots, v_n)$$

By expressing the above in *LC* notation, and making use of proper abstractions and simplifications, we obtain

$$\sigma \equiv \lambda\phi v_1 \ldots v_n{:}\sigma_1(\sigma_2(\ldots(\sigma_p(\lambda u_1 \ldots u_m{:}\phi))\ldots))c_1 \ldots c_m v_1 \ldots v_n.$$

Consequently, we choose the following representation of blocks.

$$\{\textbf{begin } \langle\text{type}\rangle u_1 := c_1; \ldots; \langle\text{type}\rangle u_m := c_m; S_1; S_2; \ldots; S_p \textbf{end}\}_{(v_1,\ldots,v_n)}$$
$$\equiv \lambda\phi v_1 \ldots v_n{:}\{S_1\}_F(\{S_2\}_F(\ldots(\{S_p\}_F(\lambda u_1 \ldots u_m{:}\phi))\ldots))\{c_1\}_E \ldots \{c_m\}_E v_1 \ldots v_n, \tag{5}$$

where

$$E \equiv (v_1, \ldots, v_n) \quad \text{and} \quad F \equiv (u_1, \ldots, u_m, v_1, \ldots, v_n).$$

(We assume that the expressions c_i include any needed type-conversions.)

In the case that the variables are left uninitialized in the block-head—as is normal in ALGOL 60—any arbitrary *LE* can be used for $\{c_i\}$ in the above representation. One might wish to use for this purpose an *LE* which would play the role of the everywhere

† We assume that the expressions $c_1, \ldots, c_m$ do not contain the variables $v_1, \ldots, v_m$; they may, however, contain the variables in the environment of S.

undefined function. This function is modelled, for example, by the $LE\Omega$ (see (1) in Section 2) which has the property $\Omega a \rightarrow \Omega$ for all a. It should be noted, however, that Ω does not possess a normal form. As a result, if Ω is used in place of the missing c_i's in (5), then the presence of any variables that remain undefined throughout the program execution would cause the program representation to behave as if the program contained an infinite loop.

Being the representation of statements, the components $\{S_i\}_F$, $1 \leq i \leq p$, of the right-hand-side of (5) do not contain any free indeterminates. But being the representations of expressions in the environment $(v_1, \ldots, v_n)$, $\{c_i\}$, $1 \leq i \leq m$, may possibly contain $v_1, \ldots, v_n$. If the variables declared in the block head are not initialized, then the indeterminates $v_1, \ldots, v_n$ have no free occurrences in the expression at the right-hand-side of (5), and hence can be dropped by using η-contraction. We thus obtain the following simplified representation rule:

$$\{\textbf{begin } \langle \text{type} \rangle u_1; \ldots; \langle \text{type} \rangle u_m; S_1; S_2; \ldots; S_p \textbf{ end}\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi{:}\{S_1\}_F(\{S_2\}_F(\ldots(\{S_p\}_F(\lambda u_1 \ldots u_m{:}\phi))\ldots))\underbrace{\Omega\Omega\ldots\Omega}_{m \text{ times}} \quad (6)$$

where

$$F \equiv (u_1, \ldots, u_m, v_1, \ldots, v_n).$$

Note that if only constants are used to initialize the declared variables, then again the variables v_i can be dropped, and the representation is similar to (6), except that the constants are used instead of the corresponding Ω.

Example. The environment of the following block is assumed to be (w).

Statements	*Representations*
begin integer $x := 5, y;$	
$y := x - 7;$	$a \equiv \lambda\phi xyw{:}\phi x(\underline{-}x\underline{7})w$
begin integer $z;$	
$z := 3+y;$	$b \equiv \lambda\phi zxyw{:}\phi(\underline{+}\underline{3}y)xyw$
$x := z \times x$	$c \equiv \lambda\phi zxyw{:}\phi z(\times zx)yw$
end	$d \equiv \lambda\phi{:}b(c(\lambda z{:}\phi))\Omega$
end	$e \equiv \lambda\phi{:}a(d(\lambda xy{:}\phi))\underline{5}\Omega.$

3.7. *Conditional statements*

In view of the reduction property of **if** given in Section 3.1, we choose the representation of a two-branch conditional statement as follows:

$$\{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n{:}\ \textbf{if}\{b\}\{S_1\}\{S_2\}\phi v_1 \ldots v_n. \quad (7)$$

For the purpose of representation, a one-branch conditional statement may be viewed as two-branch conditional with a dummy or "do-nothing" statement for the second branch. When appearing in the environment $(v_1, \ldots, v_n)$, the "do-nothing" statement can obviously be represented by

$$\lambda\phi v_1 \ldots v_n{:}\phi v_1 \ldots v_n$$

which reduces to I (defined in (1)). Substituting the "do-nothing" statement for S_2 in (7), we obtain:

$$\{\textbf{if } b \textbf{ then } S_1\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n{:}\textbf{if } \{b\}\{S_1\}\mathrm{I}\phi v_1 \ldots v_n. \quad (8)$$

3.8. *Input–output*

We shall assume for simplicity that the program input and output operations are each restricted to a single file. A file of items $a_1, \ldots, a_n$ will be represented by the *tuple*

$$\langle\{a_1\}, \ldots, \{a_n\}\rangle$$

as defined in (1) (Section 2). The empty file is represented by the null tuple $\langle\ \rangle \equiv \mathbf{I}$. From the definition of tuples and the ";" operation (also defined in (1)), it follows that

$$\langle x_1, \ldots, x_n\rangle; y \to \langle x_1, \ldots, x_n, y\rangle.$$

Denoting $(a; b); c$ by $a; b; c$, and so on, we also have

$$\mathbf{I}; x_1; \ldots; x_n \to \langle x_1, \ldots, x_n\rangle.$$

Thus, ";" may be regarded as the operation of writing on a file, and the file resulting from writing an item a on a given file b may be represented by $(\{b\}; \{a\})$.

Now let S be a statement appearing in the environment $(v_1, \ldots, v_n)$ of a program, and let σ be the *LE* representing S. In our discussion so far, σ has been defined as an *LE* of the form

$$\lambda\phi v_1 \ldots v_n : \ldots \tag{i}$$

with the indeterminate ϕ standing for the program remainder of S. Accordingly, the execution of S has been modelled by the reduction of the *LE*

$$\sigma\hat{\phi}\hat{v}_1 \ldots \hat{v}_n,$$

in which the hatted symbols denote the representations of the values of the corresponding variables immediately prior to the execution of S. In order to take input–output into account, we will generalize the representations so as to model the above execution by the reduction of the *LE*

$$\sigma\hat{\phi}\hat{v}_1 \ldots \hat{v}_n\hat{w}\hat{u}_1\hat{u}_2 \ldots \hat{u}_m, \tag{9}$$

with w denoting the output file and u_i the ith of the m items remaining on the input file at the moment of execution. (As soon as an item is read, it is supposed to disappear from the input file.) This arrangement requires that the representations of statements be generally of the form

$$\lambda\phi v_1 \ldots v_n \mathrm{o} i_1 \ldots i_m : \ldots,$$

where $\mathrm{o}, i_1, \ldots, i_m$ are the extra indeterminates corresponding to the output file and input items. It must be evident, however, that the representations of those statements which do not involve input–output can be simplified back to the form (i) by using η-contraction. Furthermore, in the case of input–output statements, the following choice of *LC* representations is obvious:

$$\{\text{read } v_j\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n \mathrm{o} i : \phi v_1 \ldots v_{j-1} i v_{j+1} \ldots v_n \mathrm{o}, \tag{10}$$

$$\{\text{write } e\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n \mathrm{o} : \phi v_1 \ldots v_n(\mathrm{o}; \{e\}),$$

where e is some expression to be output.

3.9. *Programs*

Let the input file initially presented to a given program consist of items $i_1, \ldots, i_p$, and let $\mathrm{o}_1, \ldots, \mathrm{o}_p$ constitute the items of the final output file produced by the program. As

remarked in Section 3.1, we wish to choose a program representation so as to obtain the relation

$$\{\text{program}\}\{i_1\}\ldots\{i_p\} \rightarrow \langle\{o_1\},\ldots,\{o_q\}\rangle. \tag{i}$$

Now the execution of a particular statement of the program is modelled by the reduction of an *LE* given by (9). Suppose that as an instance of such a statement we take the entire outermost block of the program. Recalling the significance of symbols used in connection with (9), we obtain the following conditions:

$\sigma \equiv$ {program block},
$n = 0$, as the environment is null,
$\hat{w} \equiv \mathbf{I}$, as the output file may be considered empty at the start of the program,

$$m = p, \quad \text{and} \quad u_j = i_j, \quad 1 \leq j \leq p.$$

Furthermore, in place of $\hat{\phi}$, the "null" program remainder, we may arbitrarily choose to employ the *LE* I. On substituting these values, the execution of the program is seen to amount to the reduction of the *LE*

$$\{\text{program block}\}\mathbf{II}\{i_1\}\ldots\{i_p\}. \tag{ii}$$

Next, consider (9) again—but this time for the case when the entire program has been executed. Now we have:

$\sigma \equiv \mathbf{I}$, the null program segment,
$n = 0$, as the environment is null,
$\hat{w} = \langle\{o_1\},\ldots,\{o_q\}\rangle$, representing the final output file,
$m = 0$, assuming the program exhausts the input file,
$\hat{\phi} \equiv \mathbf{I}$.

Thus, (9) in this case becomes the *LE*

$$\mathbf{II}\langle\{o_1\},\ldots,\{o_q\}\rangle,$$

which reduces to

$$\langle\{o_1\},\ldots,\{o_q\}\rangle. \tag{iii}$$

If our representations work properly, then the *LE* (ii) should reduce to the *LE* (iii); that is,

$$\{\text{program block}\}\mathbf{II}\{i_1\}\ldots\{i_p\} \rightarrow \langle\{o_1\},\ldots,\{o_q\}\rangle. \tag{iv}$$

Comparing (i) and (iv), we obtain the representation rule:

$$\{\text{program}\} \equiv \{\text{program block}\}\mathbf{II}. \tag{11}$$

Example. Following is the representation of the simple program mentioned at the beginning of Section 3.1.

Statements	*Representations*
begin integer a, b, c;	
read a;	$f \equiv \lambda\phi abcoi{:}\phi ibco$
read c;	$g \equiv \lambda\phi abcoi{:}\phi abio$
$b := a+c$;	$h \equiv \lambda\phi abc{:}\phi a(+ac)c$
write b;	$j \equiv \lambda\phi abco{:}\phi abc(o; b)$
$b := b-2\times c$;	$k \equiv \lambda\phi abc{:}\phi a(-b(\times \underset{j}{2}c))c$
write b	
end	$m \equiv \lambda\phi{:}f(g(h(j(k(j(\lambda abc{:}\phi))))))\Omega\Omega\Omega.$

Since the $LE\,m$ represents the program block, the representation of the whole program is $P \equiv m\mathbf{II}$. Now it can be verified that

$$m\mathbf{II} \rightarrow \lambda xy{:}\langle \underline{x-y},\ \underline{x+y}\rangle.$$

Thus the program representation P indeed abstracts out the input–output behavior of the program (cf. Section 3.1). The execution trace of the above program, when run with the integers 5 and 3 as data items, is reflected in the following LC reduction.

$$\begin{aligned}
P\,\underline{5}\,\underline{3} \equiv m\mathbf{II}\underline{5}\,\underline{3} &\rightarrow f(g(h(j(k(j(\lambda abc{:}\mathbf{I})))))\Omega\Omega\Omega\mathbf{I}\underline{5}\,\underline{3}\\
&\rightarrow g(h(j(k(j(\lambda abc{:}\mathbf{I})))))\underline{5}\Omega\Omega\mathbf{I}\underline{3}\\
&\rightarrow h(j(k(j(\lambda abc{:}\mathbf{I}))))\underline{5}\Omega\underline{3}\mathbf{I}\\
&\rightarrow j(k(j(\lambda abc{:}\mathbf{I})))\underline{5}(+\underline{5}\,\underline{3})\underline{3}\mathbf{I}\\
&\rightarrow j(k(j(\lambda abc{:}\mathbf{I})))\underline{5}\,\underline{8}\,\underline{3}\mathbf{I}\\
&\rightarrow k(j(\lambda abc{:}\mathbf{I}))\underline{5}\,\underline{8}\,\underline{3}(\mathbf{I};\underline{8}) \rightarrow k(j(\lambda abc{:}\mathbf{I}))\underline{5}\,\underline{8}\,\underline{3}\langle\underline{8}\rangle\\
&\rightarrow j(\lambda abc{:}\mathbf{I})\underline{5}(-\underline{8}(\times\underline{2}\,\underline{3}))\underline{3}\langle\underline{8}\rangle \rightarrow j(\lambda abc{:}\mathbf{I})\underline{5}\,\underline{2}\,\underline{3}\langle\underline{8}\rangle\\
&\rightarrow (\lambda abc{:}\mathbf{I})\underline{5}\,\underline{2}\,\underline{3}(\langle\underline{8}\rangle;\underline{2}) \rightarrow (\lambda abc{:}\mathbf{I})\underline{5}\,\underline{2}\,\underline{3}\langle\underline{8},\underline{2}\rangle\\
&\rightarrow \mathbf{I}\langle\underline{8},\underline{2}\rangle \rightarrow \langle\underline{8},\underline{2}\rangle.
\end{aligned}$$

SUMMARY

A mapping from the elementary constructs of programming languages to the untyped lambda-calculus is developed in order to represent programs by lambda-expressions. Program variables are represented by the lambda-calculus indeterminates such that a distinct indeterminate corresponds to each variable declared in a set of nesting blocks. Well-known lambda-expression representations are adapted for numerals, logical values, operations, and relations. Expression representations are also obtained from the representations of constituent operands and operations in a familiar manner. The environment of a point is defined to be the list of all the variables declared in the blocks enclosing the point. The representation of a programming construct depends upon both the construct and the environment in which it occurs. The program remainder of a statement is the part of the program between the statement and the program end. A statement is regarded as a function of the variables in its environment as well of its program remainder; the representation of a statement is obtained by expressing the corresponding function in the lambda-calculus notation. An assignment statement is represented by the application of its program remainder to the program variables in its environment, with the assignment-affected variable replaced by the expression assigned to it. Since the program remainder itself serves to propagate the effects of the computations performed by a statement to the rest of the program, statement sequencing and compounding are represented by nested application of individual statement representations. Representation rules are given for blocks with or without declared variable initialization, conditional statements, input–output statements restricted to single files, and whole programs. The representations of programs is so chosen that the result of applying the lambda-expression representing a program to the representations of the input items reduces to a lambda-expression which is a tuple of the output item representations. The representation of a program expresses in the lambda-calculus notation the function from the input space to the output space that the program computes.

About the Author—S. Kamal Abdali received the B.E. degree in electrical engineering from the American University of Beirut, the M.Sc. degree in computer science from the University of Montreal, and the Ph.D. degree in computer science from the University of Wisconsin, Madison, in 1963, 1968 and 1974, respectively. From 1963 to 1967, he worked in the field of

switching and communication, holding positions with Government Engineering College, Karachi, Pakistan, Telephone Industries of Pakistan, Haripur, Pakistan, and International Telephone and Telegraph, Montreal, Canada. From 1971 to 1973, he was with the Computer Science Department of New York University. Since 1973, he has been with the Mathematical Sciences Department of Rensselaer Polytechnic Institute, Troy, N.Y. His current research interests include the theory of computing, program verification, and the combinatory logic.

Dr. Abdali is a member of the Association for Computing Machinery and the Association for Symbolic Logic.

REFERENCES

1. P. J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation, *Comm. ACM* **8**, 89 and 158 (1965).
2. C. Strachey, Towards a formal semantics, in *Formal Language Description Languages* (T. B. Steel Ed.), North-Holland, Amsterdam (1966).
3. P. Naur (Ed.), Revised report on the algorithmic language ALGOL 60, *Comm. ACM* **6**, 1 (1963).
4. A. Church, *The Calculi of Lambda-Conversion*, Princeton University Press, NJ (1941).
5. H. B. Curry and R. Feys, *Combinatory Logic*, Vol. 1. North-Holland, Amsterdam (1958).
6. J. R. Hindley, B. Lercher and J. P. Seldin, *Introduction to Combinatory Logic*, Cambridge University Press, London (1972).
7. R. J. Orgass and F. B. Fitch, A theory of programming languages, *Studium Generale* **22**, 113 (1969).
8. P. Henderson, Derived semantics for programming languages, *Comm. ACM* **15**, 967 (1972).
9. J. C. Reynolds, Definitional interpreters for higher-order programming languages, *Proc. ACM Annual Conf.* Boston, pp. 717–740 (1972).
10. S. K. Abdali, A simple lambda-calculus model of programming languages, AEC R & D Report C00-3077-28, Courant Inst. Math. Sci., New York University (1973).
11. S. K. Abdali, A combinatory logic model of programming languages, Ph.D. Dissertation, Comp. Sci. Dept., University of Wisconsin (1974).
12. C. Stratchey and C. P. Wadsworth, Continuations—a mathematical semantics for handling full jumps, Technical Monograph PRG-11, Oxford University, London (1973).
13. A. van Wijngaarden (Ed.), Report on the algorithmic language ALGOL 68, *Numerische Math.* **14**, 79 (1969).