A COMBINATORY LOGIC MODEL

OF

PROGRAMMING LANGUAGES

by

S. Kamal Abdali

A thesis submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN

1974

ACKNOWLEDGMENT

I would like to thank Professor George Petznick for the advice, guidance, and constructive criticism I received from him during the research for this thesis. I am also grateful to Professors Donald Fitzwater, Lawrence Landweber, and Edward Moore for being on my examination committee and offering valuable suggestions. My special appreciation goes to Professor Tad Pinkerton for helping me during some desperate moments.

I also acknowledge my indebtedness to the Courant Institute, where most of this work was done, for a stimulating research environment, and to the AEC Computing Center, for the computing facilities used in the experimental verification of parts of this model.

A COMBINATORY LOGIC MODEL OF PROGRAMMING LANGUAGES

S. Kamal Abdali

Under the supervision of Assistant Professor George W. Petznick

A simple correspondence is presented between a large subset of the ALGOL 60 language and the combinatory logic. With the aid of this correspondence, a program can be translated into a single combinatory object. The combinatory object representing a program is specified, in general, by means of a system of reduction relations among the representations of the program constituents. This object denotes, in terms of the combinatory logic, the function that the program is intended to compute.

The model has been derived by using intuitive, functional interpretations of the constructs of programming languages, completely avoiding the notions of machine command and address. In particular, the concepts of program variable, assignment, and procedure have been accounted for in terms of the concepts of mathematical variable, substitution, and function, respectively.

High-level programming language features are represented in the combinatory logic directly, not in terms of the representations of machine-level operations. Input-output is treated in such a manner that when the representation of a

program is applied to the representations of the input items, the resulting combination reduces to a tuple of the representations of the output items.

The applicability of the model to the problems of proving program equivalence and correctness is illustrated by means of examples.

Approved by George W. Petznick
(signed)

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Preliminary Remarks

Before we can model programming languages, we have to be definite about what is to be regarded as the meaning of a program. In our view, 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 intended to be 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 a 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 substitution used in the functional calculi with the notion of assignment used in programming languages. Again, whereas the former is concerned with values, the latter is concerned with addresses. Similarly, the notion of function used in mathematics is radically different from the notion of procedure used in programming languages; the evaluation of functions requires straightforward substitution of the argument values in the functional definitions, while 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 [17] -- 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, and block declarations, and name and value specifications for procedure parameters. Although the descriptive and the declarative elements are often lumped together [17,36], 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, most 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 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. This is the task that we undertake to do in the present dissertation. 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 [27], with a few, explicitly stated, extensions. As the mathematical theory for modelling the programming constructs, we make use of the combinatory logic [8,9,33,34], originated by Schönfinkel and developed principally by Curry. A remarkable feature of this theory is the absence of variables. Nevertheless, functions can be represented by the objects of this theory in a natural manner. Thus, by using the combinatory interpretation of programming languages, we seek to eliminate variables altogether, program-related or mathematical.

1.2 Background

A number of combinatory logic (or, related, lambda-calculus) models of programming languages have appeared in the literature. The most well known of these are due to Landin. In [16], Landin uses the lambda-calculus to model the semantics of expression evaluation in programming languages. He also prescribes the semantics of the lambda-calculus itself by means of an interpreter, the SECD machine, to evaluate lambda-expressions. But in extending his model to include the imperative features of ALGOL 60, he elects [17] to supplement the lambda-calculus and its evaluator with such concepts as the assigner, the jump operator, and sharing. Due to their pioneering nature and their thoroughness of analysis, Landin's papers have greatly influenced the subsequent work on programming semantics. But they have also been instrumental in creating the rather unfair impression that the pure lambda-calculus is not an appropriate medium for representing the imperative notions of programming languages (see for example [2]).

The direct superimposition of imperative concepts on the lambda-calculus seems to us unsuitable on several grounds. First of all, there is no guarantee of the preservation of consistency in the augmented calculus. Then, by defining the new calculus in terms of the sequentially operating "sharing machine", one throws away the most important property of the lambda-calculus, the Church-Rosser property [9], which implies

that the values of lambda-expressions are independent of their evaluators. Finally, the resulting model of programming languages forces one to identify a program with its execution trace; it fails in capturing the abstract function underlying the computation expressed in the program.

Stratchey [36] also uses the lambda-calculus to model programming language semantics. Unlike Landin, he under-takes to represent the imperative as well as the declarative and descriptive notions of programming languages in terms of the descriptive concepts of the lambda-calculus. To account for the notion of assignment, however, he postulates a number of primitives to represent the generalized concepts of address and the related memory fetch and store operations. Thus, by not going as far as to eliminate program variables in favor of mathematical variables, Stratchey's model is again computational rather than functional. As Burstall points out in describing another lambda-calculus model [5] of programming, it is unnecessary to introduce any assignment-related concepts as primitives. Indeed, Burstall shows how assignments can be naturally modelled by the lambda-calculus operation of substitution. We remark that our approach is closest to Burstall's in spirit, though we adopt a different method of representing assignments.

Orgass and Fitch [28] have developed a theory of programming languages in a system of the combinatory logic. They represent the computer memory as an n-tuple and the

machine state transitions caused by the execution of instructions as functions on n-tuples. Their representation of programming languages is in rather general terms, and it is not clear how it can be used to obtain the representation of specific programs. The main drawback in their model seems to be the lack of treatment of declarative features, so that their discussion of programming languages is applicable, for the most part, to machine-level languages only.

A number of researchers have used the combinatory logic or the lambda-calculus to model only some important programming constructs rather than full-fledged programming languages. In [26], Morris explores the concepts of recursion and types in the lambda-calculus, but his work is also relevant to these concepts as they occur in programs. Ledgard [18] describes a model of type checking in which lambda-expressions are used to abstract out the type relations within a program. In addition, we mention the work of Böhm [4], Petznick [29], Milner [25] and Henderson [11], who obtain the representations of various elementary programming constructs and schematic programs in the combinatory logic or the lambda-calculus.

The semantics of the lambda-calculus itself has attracted considerable attention. We have already mentioned Morris' results on recursion and types [26] and Landin's SECD machine [16] for evaluating lambda-expressions. The formal specification of several lambda-calculus interpreters (originally due to Wegner [37]), and proofs of their mutual

equivalence have been given by McGowan [24]. The Wegner-McGowan and Landin machines employ the so-called "call-by-value" strategy of evaluating the right component of an application before the left one. Consequently, none of these machines is a true normal-form reducer for lambda-expressions. A computer to reduce the objects of the combinatory calculus to their normal forms has been designed and proved correct by Petznick [29]. This computer consists of stack-structured control registers and tree-structured memory, permits shared memory representations of equiform objects, and has provisions for incremental programming and multiple processing. Reynolds [31] defines and interrelates a number of interpreters for the lambda-calculus and for its various extensions with programming language features.

Knuth [15] approaches the semantics of the lambda-calculus from the viewpoint of his general theory of the semantics of context-free languages, in which the meanings of sentences are built up, in the course of sentence generation, from the attributes associated with non-terminal symbols. Finally, the most abstract and incisive work on the semantics of the lambda-calculus has been done by Scott (e.g., [35]) as part of a very general theory of computation. Concerned with an abstract or functional explanation rather than with a computational explanation, Scott represents a lambda-expression as the limit of a certain sequence of constructions on complete lattices. His theory succeeds in providing very

convincing answers to the problems related to recursion and self-application in the lambda-calculus.

Combinatory models constitute but a minute fraction of the total work that has been done so far on the semantics of programming languages. We mention just a few of the other models. In [10], Floyd suggests the method of semantic definition of programming language statements by means of pairs of conditions that hold just before and after the statement execution. Apparently, little use has been made of Floyd's method in the definition of programming languages. But his idea of representing the effect of execution of program statements by assertions has had diverse and far-reaching consequences. (For example, it serves as the basis of most program proving schemes [20]). Of all the methods of defining programming languages, the most elaborate and the most extensively applied is the "Vienna Method" [19,21] -- so called because it was developed at the IBM Vienna Laboratory for the semantic specification of the PL/l language. In this method, programming constructs are represented in terms of the (non-deterministic) state transitions of a machine; it thus constitutes a computational, rather than a functional, approach to semantics.

Among the purely functional semantic models of programming languages based on theories other than the combinatory logic, the most notable are: Burstall's [6] representation of programs by combining the Floyd-type assertions associated with the

programs into the formulas of first-order predicate calculus; and Manna and Vuillemin's [22] Scott-theoretical interpretation of programming constructs as minimal fixed points of "recursive programs".

To conclude, we mention the axiomatic approach to programming semantics, which is different from both functional and computational approaches. It consists in devising the systems of axioms and inference rules which apply directly to programming language constructs, and in which the properties of programs may be derivable as theorems. This approach obviates the circuitousness involved in deriving the same properties when one uses a "programming model" (in which case, one first represents programs as the objects of some mathematical theory, and then works with these representations to derive the properties). Igarashi [13] and deBakker [2] are the first to propose sets of axioms dealing with elementary programming constructs; their systems are, however, rather complex. A very simple and elegant axiom system (in which Floyd's ideas again find a new expression) has recently been presented by Hoare [12].

CHAPTER 2

THE COMBINATORY LOGIC

This chapter summarizes the properties of the combinatory logic [8,9,33,34] which will be utilized later in the modelling of programming languages. The discussion is in terms of a particular system SK consisting of only two primitive objects (S and K), a primitive operation (combination) to form new objects from given ones, and two primitive relations (S- and K-contractions) between objects.

2.1 Morphology and Transformation Rules

The alphabet for SK consists of the symbols "S", "K", "(", and ")". As there is no possibility of confusion, we let "S" and "K" also denote the words consisting solely of S and K, respectively. If a and b denote words over the alphabet, then we denote by "(ab)" the word obtained by concatenating the symbol "(", the word a, the word b, and the symbol ")". Of all the words over the SK alphabet, we distinguish certain words by means of the

(1-1) Definition. The (combinatory) obs are formed according to the following rules:

(1) S and K are obs.

(2) If a and b are obs, then (ab) is an ob.

(3) The only obs are those specified by (1) and (2).

S and K are primitive obs; any other ob, which is necessarily of the form (ab), is a composite ob. The composite ob (ab) is the application of a to b, or the combination of a and b, the obs a and b being its left and right immediate components, respectively. It follows that the immediate components of an ob are uniquely determined and are non-overlapping. A component of an ob is either the ob itself or a component of an immediate component of the ob. A proper component of an ob is a component which is not the ob itself. The length of an ob is the number of symbols S and K in it.

Example. The application of the ob ((KS)((SS)K)) to K is the composite ob (((KS)((SS)K))K) of length 6, some of whose proper components are K, S, (KS), and ((SS)K). The same ob S has three different occurrences as components of the above ob; all these occurrences are to be regarded as different components.

We use the notation a ≡ b to indicate that the obs a and b are equiform, that is, spelt the same over the SK alphabet. We also use this notation for introducing a as a new name for the ob b. We may abbreviate obs by omitting parentheses under the convention that any omitted parentheses are to be re-inserted by association to the left. For example, ((SS)(SK)) maybe abbreviated to (SS)(SK) or SS (SK).

We now state a number of rules for transforming obs.

(1-2) Definition. For all obs a, b, and c:

(1) The ob Sabc S-contracts to the ob ac(bc); in symbols, Sabc $\rightarrow_S$ ac(bc).

(2) The ob Kab K-contracts to the ob a; in symbols, Kab $\rightarrow_K$ a.

If a $\rightarrow_S$ b, then a and b are called, respectively, the S-redex and S-contractum corresponding to each other. K-redex and K-contractum are defined analogously. A redex is either an S-redex or a K-redex.

Let an ob b be obtained from an ob a by replacing a component c of a by an ob d. If it is the case that c $\rightarrow_S$ d or c $\rightarrow_K$ d, then a immediately reduces to b (in symbols, a $\rightarrow_{im}$ b). For example, K(SKSa)Sb $\rightarrow_{im}$ SKSab, and also K(SKSa)Sb $\rightarrow_{im}$ K(Ka(Sa))Sb, depending on the redex selected for contraction.

(1-3) Definition. An ob a is irreducible or in normal form if there is no ob b such that a $\rightarrow_{im}$ b.

Thus, no component of an irreducible ob may be a redex. Some examples of irreducible obs are S, K, KS, SKK, and SS(S(SSK)K).

(1-4) Definition. An ob a reduces to an ob b, denoted a $\rightarrow$ b, if there exist obs a_0, a_1,..., a_n, for some n > 0, such that

(1) a $\equiv$ a_0 ,

(2) b $\equiv$ a_n ,

(3) $a_i \rightarrow_{im} a_{i+1}$, for $0 \leq i < n$.

Example. SKSKSKabc → ac(bc), since

$SKSKSKabc \rightarrow_{im} KK(SK)SKabc \rightarrow_{im} KSKabc \rightarrow_{im} Sabc \rightarrow_{im} ac(bc)$.

If $a \rightarrow_S b$, then we also say that b S-expands to a, and write $b \leftarrow_S a$. In a similar manner, we define K-expansion ($\leftarrow_K$), immediate expansion ($\leftarrow_{im}$), and expansion ($\leftarrow$).

(1-5) Definition. An ob a is interconvertible with an ob b, denoted $a \leftrightarrow b$, if there exist obs $a_0, a_1, \ldots, a_n$, for some $n > 0$, such that

(1) $a \equiv a_0$,

(2) $b \equiv a_n$,

(3) $a_i \rightarrow_{im} a_{i+1}$ or $a_i \leftarrow_{im} a_{i+1}$, for $0 \leq i < n$.

Example. SKKa ↔ SSSSKa, since $SKKa \rightarrow_{im} Ka(Ka) \rightarrow_{im} a$

$\leftarrow_{im} Ka(SSKa) \leftarrow_{im} SK(SSK)a \leftarrow_{im} SS(SS)Ka \leftarrow_{im} SSSSKa$.

Reduction (→) and interconvertibility (↔) are related by the well-known

(1-7) Theorem (Church-Rosser) [9]. If $a \leftrightarrow b$, then there exists an ob c such that $a \rightarrow c$ and $b \rightarrow c$.

(1-8) Corollary. If the obs a and b are irreducible, then $a \leftrightarrow b$ if and only if $a \equiv b$.

(1-9) Definition. A normal form of an ob a is an irreducible ob b, if one exists, such that $a \leftrightarrow b$. An ob is normal

if it has a normal form.

From Theorem 1-7 and Corollary 1-8, it immediately follows that:

(1-10) Theorem. If an ob is normal, then its normal form is unique. Moreover, if b is the normal form of a, then a → b.

That is, it is possible to discover the normal form of a normal ob by a sequence of contractions alone, starting from the given ob. Of course, there are obs that are not normal. An example of such an ob is (aaa), where a ≡ SSK. It is easily seen that all possible reductions of the ob (aaa) either keep producing longer and longer obs or eventually lead to (aaa) again, so that they will continue indefinitely without ever reaching an irreducible form. Moreover, there are obs that are normal but for which not every reduction terminates in a normal form. This is illustrated by the ob KS(aaa), with a defined as above; its normal form S cannot be reached as long as reductions are carried out only inside the component (aaa). Fortunately, there exists a deterministic reduction procedure such that, when applied to normal obs, it always arrives at their normal forms in a finite number of steps. It is as follows:

(1-11) Standard Order Reduction Algorithm (Church-Rosser). [9] To obtain the normal form of a normal ob, start with the ob and successively apply contractions of the leftmost redex, until no further reduction is possible.

A modification of the above scheme is used in Petznick's

combinatory computer [29], a hypothetical machine for reducing obs to normal form. Obs may be so represented in the memory of this machine that all equiform components of an ob may have but one internal representation. The reduction takes place by contracting the leftmost redex; but as a consequence of shared representations, several equiform redexes may be contracted simultaneously.

On account of Definition 1-9 and Theorem 1-10, it seems reasonable to regard the normal form of an ob as the "value" of the ob, and the process of normal form reduction as "evaluation". Consequently, we may regard mutually interconvertible normal obs as "equal" since they have the same value. This view of value and equality is similar to the one taken in other calculi. For example, among the equal arithmetic expressions 2x2+3x4, 4+12, and 16, the last one is distinguished in being "irreducible" by the rules of arithmetic, and thus it is regarded as the value of the three expressions. For obs as well as for arithmetic expressions, it so happens that the value is obtainable by a sequence of reductions only, and it is unique despite the possible nondeterminism involved in the order of reductions. But unlike the case for arithmetical expressions, there is no algorithm to decide whether or not two given obs have the same value (i.e., are interconvertible).

The notion of interconvertibility is generalized in the

following

(1-12) Definition. An ob a is n-interconvertible to an ob b, in symbols $a \leftrightarrow_n b$ if for all obs $c_1,\ldots,c_n$,

$$ac_1\ldots c_n \leftrightarrow bc_1\ldots c_n$$

Clearly, $a \leftrightarrow_n b$ implies $a \leftrightarrow_m b$ for all $m > n$; but the converse does not hold. For example, we have $SKS \leftrightarrow_1 SKK$, since $SKSc \rightarrow c \leftarrow SKKC$ for all c. Yet $SKS \neg\leftrightarrow_0 SKK$ (i.e., $SKS \neg\leftrightarrow SKK$) by Corollary 1-8, as the two obs are irreducible but not equiform. Indeed, it can easily be shown that for a given normal ob a and an integer $n \geq 1$, there exist infinitely many obs b with distinct normal forms such that $a \leftrightarrow_n b$ but, for all $m < n$, $a \neg\leftrightarrow_m b$.

2.2 Functional Abstraction

Our interest in SK derives from the fact that we can represent programming language constructs (e.g., expressions, statements, programs) by obs, and the processes required in the execution of programs (e.g., substitution, expression evaluation, procedure application) by the reduction operation. Such a representation is possible because

1. Programming constructs can be regarded intuitively as functions (in a special sense of the word, explained below), and
2. Functions can be represented by obs.

This section will describe how to represent functions as obs. We will assume that a function F is defined by means of a functional equation of the form

$$F(x_1,...,x_n) = E$$

in which E is an expression that may contain constants, variables, and already defined functions. The variables $x_1,...,x_n$ are called the formal arguments of the function F. To obtain the value of this function for a list of expressions given as actual arguments, the actual arguments are substituted in place of the corresponding formal arguments, and the resulting expression is evaluated.

Confined for the moment to the functions of one argument only, our basic approach to the SK representation of functions may be stated as follows: Let $\mathcal{F}$ be a given set of

one-argument functions to be represented, and let A be the union of the domains and ranges of the given functions. Then we choose obs to represent the members of $\mathcal{F}$ and A so as to satisfy the following condition. For all F in $\mathcal{F}$ and a,b in A, and their respective SK representations $\underline{F}$, $\underline{a}$, $\underline{b}$, if F(a) = b, then $(\underline{F}\ \underline{a}) \rightarrow \underline{b}$.

The above representation applies as such to one-argument functions only. But a basic idea of the combinatory calculus, due to Schönfinkel [34], is to regard even the functions of several arguments as just one-argument functions. This becomes possible if the domains and ranges of one-argument functions are permitted to contain one-argument functions themselves. To see how this idea works, suppose F is a function of two arguments, and let G_x (for a given x) and H be one-argument functions such that

$$G_x(y) = F(x,y) \quad ,$$

$$H(x) = G_x \quad .$$

Then for any arguments a and b, we have

$$F(a,b) = G_a(b) = [H(a)](b)$$

Now F is identified with the one-argument function H, and, consequently, F(a,b) with [H(a)](b). Hence, designating the representative obs by underlining the names of the represented functions or constants, we may choose $\underline{F} = \underline{H}$, thereby

representing F(a,b) by ((H a)b), i.e., (F a b). Note that (F a) represents H(a), i.e. the function G_a . In general, if F is a function of n arguments, then:

(F a_1...a_m) for m < n represents the function G such that $G(x_{m+1},...,x_n) = F(a_1,...,a_m,x_{m+1},...,x_n)$, and (F a_1...a_n) represents $F(a_1,...,a_n)$.

With the above interpretation of functions in mind, every expression involving only functions and constants but not variables, written in the customary functional notation using function application and composition, is representable in SK, provided that its constituent functions and constants are representable. For example, we may represent the expression F(G(a,b),H(J(c),d)) by the ob F(G a b)(H(J c)d), and carry out the evaluation of the former entirely within SK by reducing the latter. But in order to extend our representation to the expressions involving variables also, we require a generalization of obs described next.

We adjoin a denumerable collection of symbols called indeterminates to the alphabet of SK. We do not specify these symbols; but it will always be possible to infer from the context whether or not a symbol is an indeterminate. Of all the words over the augmented SK alphabet, we characterize an ob form to be a word that either consists of a single indeterminate or S or K, or is of the form (e_1 e_2),

where e_1 and e_2 are ob forms (Cf. Definition 1-1). All the definitions, notational conventions, and properties related to obs, as given in the previous section, generalize in an obvious manner for ob forms.

Given an ob form e and an indeterminate x, we say that e contains x, or x occurs in e, in symbols, x oc e, if x is a component of e; x o$\not$c e if it is not the case that x oc e.

(2-1) Definition. Let $e, f_1, \ldots, f_n$ be ob forms and $x_1, \ldots, x_n$ be distinct indeterminates. Then the result of (simultaneous) substitution of f_1 for x_1, f_2 for x_2 ,..., f_n for x_n in e, denoted

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

is defined by induction on n and the structure of e as follows:

(1) $\text{sub } [f_1, x_1; e]$

$$\equiv \begin{cases} e \text{ , if } x_1 \text{ o}\not\text{c } e, \\ f_1 \text{ , if } e \equiv x_1 \\ (\text{sub } [f_1, x_1; g] \text{sub } [f_1, x_1; h]), \\ \qquad \text{otherwise, where } e \equiv (gh). \end{cases}$$

(2) $\text{sub } [f_1, x_1; f_2, x_2; \ldots; f_{n+1}, x_{n+1}; e]$

$\equiv \text{sub } [f_1, z; \text{ sub } [f_2, x_2; \ldots; f_{n+1}, x_{n+1}; \text{sub } [z, x_1; e]]]$

where *z* is an indeterminate which is distinct from each of $x_1, \ldots x_n$ and which does not occur in any of e, f_1, ..., f_n .

As a consequence of the definitions of reduction and substitution, we have

(2-2) Lemma. If $e \rightarrow e'$, then

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

(2-3) Definition. Given an ob form e and indeterminates $x_1,\ldots,x_n$, for some $n \geq 1$, an abstract of e with respect to $x_1,\ldots,x_n$ is an ob form f such that

(1) x_i o$\not\in$ f , $\quad 1 \leq i \leq n$,

(2) $fx_1\ldots x_n \rightarrow e$.

Example. For $e \equiv xyz(y(xy)z)$, we have

$$e \leftarrow S(xy)(y(xy))z \leftarrow SSy(xy)z \leftarrow S(SS)xyz \ .$$

Hence, the ob S(SS) is an abstract of e with respect to x,y,z; the ob forms S(SS)xy, SSy(xy), and S(xy)(y(xy)) are all abstracts of e with respect to z.

Let f be an abstract of an ob form e with respect to the indeterminates $x_1,\ldots,x_n$. The relations (1) and (2) of the above definition are satisfied. Applying Lemma 2-2 to (2), we have for all ob forms $g_1,\ldots,g_n$,

$$\text{sub } [g_1,x_1;\ldots;g_n,x_n;fx_1\ldots x_n] \rightarrow \text{sub } [g_1,x_1;\ldots;g_n,x_n;e].$$

Because of (1), we can simplify the left-hand side in the above relation and restate the relation as follows:
If f is an abstract of e with respect to $x_1,\ldots,x_n$, then for

all ob forms $g_1,...,g_n$, we have

$$f\ g_1...g_n \rightarrow sub\ [g_1,x_1;...;g_n,x_n;e]. \qquad (*)$$

Further properties of abstracts will be described later.

Earlier in this section, we have noted how the expressions that contain already represented constants and functions can be represented by obs. With variables represented by indeterminates (possibly, with the same symbolic denotation), we can extend that scheme to represent by ob forms the expressions that contain variables in addition to constants and functions. Now consider a function F defined by the functional equation

$$F(x_1,...,x_n) = E\ ,$$

in which E is an expression containing constants, variables, and already defined functions. The value of F for given actual arguments $G_1,...,G_n$ is computed by simultaneously replacing x_1 with G_1, ..., x_n with G_n in E, and then evaluating the resulting expression. Let e, $g_1,...,g_n$ be the ob forms representing the expressions E, $G_1,...,G_n$, respectively. Then we would like to represent the function F by an ob form (or, if possible, by an ob[1]) f satisfying

---

1 It will soon become clear that if, in the functional equation $F(x_1,...,x_n) = E$ defining F, the expression E does not involve any variable other than $x_1,...,x_n$, then F is representable by an ob.

the relation

$$fg_1...g_n \rightarrow \text{sub } [g_1,x_1;...;g_n,x_n;e]$$

Furthermore, the above relation must hold for all choices of ob forms $g_1,...,g_n$. But we have just seen that this is precisely the case if f is an abstract of e with respect to $x_1,...,x_n$. Hence, given a functional equation defining a function, any abstract of the ob form representing the right-hand expression in the equation with respect to the formal arguments can be taken as the SK representation of that function.

From the property (*) of abstracts obtained earlier, it follows that all abstracts of the same ob form with respect to the same n indeterminates are mutually n-interconvertible (Definition 1-12). Also, an ob form which is n-interconvertible to an abstract of a given ob form with respect to n given indeterminates is itself one such abstract.

To designate an <u>arbitrarily chosen</u> abstract of the ob form e with respect to the indeterminates $x_1,...,x_n$, we employ the notation $(A^a x_1...x_n:e)$. This notation may be abbreviated by omitting parentheses under the convention that the ob form to the right of the colon sign extends as far to the right as is consistent with its being well formed. For instance, we may abbreviate $(A^a xy:(A^a z:(x(xy)z))$ to $A^a xy:A^a z:x(xy)z$. We state below some important properties of abstracts, including (*) for completeness; some simple arguments pertaining to the

substitution process suffice to establish these properties:

(2-4) Theorem. Let e, e′, $g_1,...,g_n$ be ob forms and $x_1,...,x_n$ distinct indeterminates for some integer $n \geq 1$.

Then:

(1) $(A^a x_1...x_n{:}e)\ g_1...g_n \rightarrow \text{sub}\ [g_1,x_1;...;g_n,x_n;e]$.

(2) If $y_1,...,y_n$ are distinct indeterminates and y_i o$\not{c}$ e for $1 \leq i \leq n$, then
$A^a x_1...x_n{:}e \leftrightarrow_n A^a y_1...y_n{:}\text{sub}\ [y_1,x_1;...;y_n,x_n;e]$.

(3) a) If x o$\not{c}$ e for all $1 \leq i \leq n$, then
$A^a x_1...x_n{:}ex_1...x_n \leftrightarrow_n e$.

b) Generally, for an integer $m \leq n$, if x_i o$\not{c}$ e for $m \leq i \leq n$, then
$A^a x_1...x_n{:}ex_m...x_m \leftrightarrow_n A^a x_1...x_{m-1}{:}e$.

(We consider the right-hand side to be simply e when $m = 1$.)

(4) $A^a x_1...x_{i-1}{:}A^a x_i...x_n{:}e \leftrightarrow_n A^a x_1...x_n{:}e$, $1 \leq i \leq n$.

(5) If $e \leftrightarrow e'$, then $A^a x_1...x_n{:}e \leftrightarrow_n A^a x_1...x_n{:}e'$.

Since a function of n arguments can be represented equally well by any one of a certain class of n-inter-convertible obs, the above theorem suggests several ways of choosing simple functional representations. For example, to represent the function F given by $F(x_1,...,x_n) = E$, one can take the ob $A^a x_1...x_n{:}e'$, where e′ is the normal form

of the ob form e representing the expression E.

The properties given in Theorem 2-4 hold for <u>all</u> abstracts. We may expect that by choosing abstracts in some specific manner, these properties could be strengthened, thus simplifying our work with the abstracts. That this is indeed the case is shown by the following description of (a modified version of) <u>Rosser's abstraction algorithm</u> [33] and the improved properties of the associated abstracts.

(2-5) <u>Definition</u>. Let e be an ob form and $x_1,...,x_n$ be indeterminates. The <u>R-abstract</u> of e with respect to $x_1,...,x_n$, denoted $(A^R x_1...x_n:e)$, is defined by induction on n and the structure of e as follows:

(1) $(A^R x_1:e)$

$$\equiv \begin{cases} Ke \text{ , if } x_1 \text{ o}\not\in e, \\ SKK, \text{ if } e \equiv x_1, \\ f \text{ , if } e \equiv (fx_1) \text{ and } x_1 \text{ o}\not\in f, \\ S(A^R x_1:f)(A^R x_1:g), \text{ otherwise, where } e \equiv (fg) \text{ .} \end{cases}$$

(2) $(A^R x_1 x_2...x_{n+1}:e) \equiv (A^R x_1: (A^R x_2...x_{n+1}:e))$.

Again we may omit parentheses in writing R-abstracts with the understanding that the ob form to the right of the colon extends as far to the right as its well-formedness would permit.

Example.

$A^R z{:}xz(yz) \equiv S(A^R z{:}xz)(A^R z{:}yz) \equiv Sxy,$

$A^R yz{:}xz(yz) \equiv A^R y{:}A^R z{:}xz(yz) \equiv A^R y{:}Sxy \equiv Sx,$

$A^R xyz{:}xz(yz) \equiv A^R x{:}A^R yz{:}xz(yz) \equiv A^R x{:}Sx \equiv S,$

$A^R x{:}xy \equiv S(A^R x{:}x)(A^R x{:}y) \equiv S(SKK)(Ky)$.

We obviously have $x \text{ o}\not\in A^R x{:}e$, and, in general, for all $1 \leq i \leq n$, $x_i \text{ o}\not\in A^R x_1 \ldots x_n{:}e$. Note also that for all indeterminates y, if $y \text{ o}\not\in e$, then $y \text{ o}\not\in A^R x_1 \ldots x_n{:}e$. Thus, if the ob form e contains no indeterminates other than $x_1,\ldots,x_n$, then $A^R x_1 \ldots x_n{:}e$ is just an ob.

(2-6) Lemma. If, for all $1 \leq i \leq n$, $x_i \text{ o}\not\in f$ and $y \neg\equiv x_i$, then

$$\text{sub } [f,y;A^R x_1 \ldots x_n{:}e] \equiv A^R x_1 \ldots x_n{:}\text{sub } [f,y;e] .$$

Proof. Repeated application of Curry's Corollary 4.1 [9, p. 208].

(2-7) Theorem. Let e, e', $g_1,\ldots,g_n$ be ob forms and $x_1,\ldots,x_n$ distinct indeterminates for some integer $n \geq 1$.

Then:

(1) $(A^R x_1 \ldots x_n{:}e)g_1 \ldots g_n \rightarrow \text{sub } [g_1,x_1;\ldots;g_n,x_n;e]$.

(2) If $y_1,\ldots,y_n$ are distinct indeterminates and $y_i \text{ o}\not\in e$ for $1 \leq i \leq n$, then

$$A^R x_1 \ldots x_n{:}e \equiv A^R y_1 \ldots y_n{:}\ \text{sub } [y_1,x_1;\ldots;y_n,x_n;e].$$

(3) a) If $x_i \text{ o}\not\in e$ for all $1 \leq i \leq n$, then

$$A^R x_1 \ldots x_n{:}ex_1 \ldots x_n \equiv e.$$

b) Generally, for an integer $m \leq n$, if $x_i \text{ o}\not\in e$

for $m \leq i \leq n$, then

$A^R x_1 \ldots x_n : e x_m \ldots x_n \equiv A^R x_1 \ldots x_{m-1} : e$.

(The right-hand side is considered to be simply e when $m = 1$.)

(4) $A^R x_1 \ldots x_{i-1} : A^R x_i \ldots x_n : e \equiv A^R x_1 \ldots x_n : e$, $1 \leq i \leq n$.

(5) If $e \leftrightarrow e'$, then

$A^R x_1 \ldots x_n : e \leftrightarrow_n A^R x_1 \ldots x_n : e'$.

Proof. For the case $n = 1$, parts (1) and (2) can be verified by induction on the structure of e, and part (3) is true by the definition of A^R . For $n > 1$, they can be proved by induction. Assuming (1) true for $n \leq k$, we show it for $n = k+1$, as follows. Choose an indeterminate z such that $z \not\circ e$ and, for $1 \leq i \leq k+1$, $z \neg\equiv x_i$ and $z \not\circ g_i$. Then:

$(A^R x_1 x_2 \ldots x_{k+1} : e) g_1 g_2 \ldots g_{k+1}$

$\equiv (A^R x_1 : (A^R x_2 \ldots x_{k+1} : e))\ g_1 g_2 \ldots g_{k+1}$

$\rightarrow \text{sub}\ [g_1, x_1; (A^R x_2 \ldots x_{k+1} : e)]\ g_2 \ldots g_{k+1}$,

by the case $n = 1$,

$\equiv \text{sub}\ [g_1, z; \text{sub}\ [z, x_1; (A^R x_2 \ldots x_{k+1} : e)]]\ g_2 \ldots g_{k+1}$

$\equiv \text{sub}\ [g_1, z; (A^R x_2 \ldots x_{k+1} : \text{sub}\ [z, x_1; e])]\ g_2 \ldots g_{k+1}$,

by Lemma 2-6,

$\equiv$ sub $[g_1,z;(\ A^R x_2 \ldots x_{k+1}$:sub $[z,x_1;e])]$

sub $[g_1,z;g_2] \ldots$ sub $[g_1,z;g_{k+1}]$

by Definition 2-1 (1), since z o$\not\in$ $g_2,\ldots,g_{k+1}$,

$\equiv$ sub $[g_1,z;((\ A^R x_2 \ldots x_{k+1}$: sub $[z,x_1;e])\ g_2,\ldots,g_{k+1})]$

$\rightarrow$ sub $[g_1,z;$ sub $[g_2,x_2;\ldots;g_{k+1},x_{k+1};$ sub $[z,x_1;e]]]$,

by the induction hypothesis and Lemma 2-2,

$\equiv$ sub $[g_1,x_1;g_2,x_2;\ldots;g_{k+1},x_{k+1};e]$,

by Definition 2-1 (2) .

Proofs of other parts are quite simple and are omitted.

It has already been mentioned that x_i o$\not\in$ $A^R x_1 \ldots x_n$: e for all $1 \leq i \leq n$. To verify that R-abstracts are indeed abstracts (Definition 2-3), it suffices to replace g_i by x_i, $1 \leq i \leq n$, in Part (1) of Theorem 2-7. Thus, now we have an algorithm for the SK representation of functions that are defined by functional equations.

A comparison of Theorems 2-4 and 2-7 shows that R-abstracts possess simpler properties than the abstracts in general. One exception is the property in Part (5) of Theorem 2-7, where we do not get any improvement by using R-abstracts. The relation $\leftrightarrow_n$ in the conclusion of that part cannot be strengthened to $\leftrightarrow$, as the following counterexample will show: $Kx(Sx) \leftrightarrow x$, $A^R x{:}Kx(Sx) \equiv SKS$, $A^R x{:}x \equiv SKK$, and $SKS \leftrightarrow_1 SKK$, but not $SKS \leftrightarrow SKK$ (cf. discussion after Definition 1-12).

In spite of their simpler properties as compared to abstracts in general, R-abstracts rapidly increase in length and become tedious to compute as the number of indeterminates for abstraction increases. In Section 2.4, we will describe an alternative, more practical algorithm for obtaining abstracts that have almost the same reduction properties as R-abstracts.

2.3 The Lambda-Calculus

The previous section has introduced the notions of indeterminates, ob forms, and abstracts for the purpose of representing functions in SK. It so turns out that the functions encountered in the representation of programs are defined solely in terms of formal variables; the resulting SK representations are, therefore, just obs, rather than ob forms containing indeterminates. Thus, eventually, the use of indeterminates and the generalization of obs to ob forms are both dispensable; they are employed as a matter of notational convenience only.

An alternative notion is that of lambda-expressions, which closely resemble abstracts in their properties, but in which the use of variables (indeterminates) is not as incidental. Lambda-expressions are the objects of the lambda-calculus LC , which may be regarded either as an augmentation to SK or an independent formal system that is similar to SK in several respects.

We present below a brief description of lambda-expressions for the sake of completeness and comparison with abstracts. For details, consult [8,9,33].

The symbols in the alphabet of LC are "(", ")", "λ" , ":", and a denumerable collection of variables. A lambda-expression (LE) is either a variable, or a word of the form (ef) or (λx:e), where e and f are LE's and x is a variable.

The LE (ef) is the application of e to f, and the LE

(λx:e) is the abstraction of e with respect to x.

To abbreviate LE's, we may omit parentheses under the convention that applications associate to the left and abstractions to the right, with the former taking precedence over the latter. For instance, the LE

(λx: (λy: ((xy) (λz: (((xz) (yz)) u)))))

may be abbreviated to

λx: λy: xy(λz: xz(yz)u) .

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

λxy: xy(λz: xz(yz)u)

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

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 for transformation, called contractions, are these:

(a) $\lambda x{:}e \rightarrow_{\alpha} \lambda y{:}sub\ [y,x;e]$,

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.

(b) $(\lambda x{:}e)f \rightarrow_{\beta} sub\ [f,x;e]$,

provided that no variable with free occurrences in f has bound occurrences in e.

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

Analogously to the development in the case of SK, we define: immediate reduction ($\rightarrow_{im}$) of LE's as the application of an α-, β-, or η-contraction on their parts;[2] reduction ($\rightarrow$) as a sequence of immediate reductions; expansion (or, respectively, α-, β-, η-, immediate expansion) as the converse of reduction (or, respectively, α-, β-, η-contraction, immediate reduction); and interconvertibility ($\leftrightarrow$) as a possibly empty sequence of immediate reductions and expansions. (The use of the same symbols $\rightarrow$, $\rightarrow_{im}$ and $\leftrightarrow$ to represent the different relations of SK and LC should not cause any confusion.) The Church-Rosser theorem holds for LC as well [9, Ch. 4]: If e $\leftrightarrow$ f, then there exists an LE g, such that e $\rightarrow$ g and f $\rightarrow$ g. But note the differences from SK:

---

2 The LE e is an immediate part of the LE's (e f), (f e), and $(\lambda x{:}e)$, where f is an LE and x is a variable. A part of an LE is either the LE itself or a part of an immediate part of the LE.

An LE is irreducible if no β- or η-contraction is applicable to it even after any applications of α-contraction. Given an LE e, if there exists an irreducible LE f such that $e \leftrightarrow f$, then f is a normal form of e. The normal forms of LE's are unique only up to the applications of α-contraction.

The following properties of LE's can be derived easily from the above definitions.

(3-1) Theorem.

(1) If none of $y_1,\ldots,y_n$ has a free occurrence in e and no free occurrence of x_i, $1 \leq i \leq n$, in e becomes a bound occurrence of the corresponding y_i in e by the substitution below, then $\lambda x_1\ldots x_n{:}e \rightarrow \lambda y_1\ldots y_n{:}\mathrm{sub}\,[y_1,x_1;\ldots;y_n,x_n;e]$.

(2) If no variable with a free occurrence in any of $g_1,\ldots,g_n$ has a bound occurrence in e, then

$$(\lambda x_1\ldots x_n{:}e)g_1\ldots g_n \rightarrow \mathrm{sub}\,[g_1,x_1;\ldots;g_n,x_n;e].$$

(3) If none of $x_1,\ldots,x_n$ has a free occurrence in e, then

$$\lambda x_1\ldots x_n{:}ex_1\ldots x_n \rightarrow e.$$

(4a) If $e \rightarrow e'$, then $\lambda x_1\ldots x_n{:}e \rightarrow \lambda x_1\ldots x_n{:}e'$.

(4b) If $e \leftrightarrow e'$, then $\lambda x_1\ldots x_n{:}e \leftrightarrow \lambda x_1\ldots x_n{:}e'$.

The similarity between LC and SK (augmented with ob forms) becomes obvious when we set up a correspondence between variables and indeterminates and then interpret

(1) Ob forms by LE's, by replacing S and K with $\lambda xyz{:}xz(yz)$ and $\lambda xy{:}x$, respectively;

(2) LE's by ob forms, by replacing λ with A^R.

With this interpretation, it is easy to see that whenever two ob forms are mutually related by the SK reduction, their corresponding LE's are related by the LC reduction. The converse, however, is not true. Here is a counterexample. Let

$$f \equiv \lambda x:\ (\lambda xy{:}x)x((\lambda xyz:\ xz(yz))x)$$
$$\rightarrow \lambda x:\ \text{sub}\ [x,x;\ (\lambda xyz{:}xz(yz))x,y;\ x]$$
$$\equiv \lambda x{:}x \equiv g\ ,\ \text{say.}$$

Let f_C and g_C denote the SK interpretations of the LE's f and g, respectively; that is,

$$f_C \equiv A^R x:\ (A^R xy{:}x)x\ ((A^R xyz:\ xz(yz))x)\ ,$$

$$g_C \equiv A^R x{:}x \equiv SKK\ .$$

It can be verified easily that $f_C \equiv SKS$. Thus, in spite of the relation $f \rightarrow g$, it is not the case that $f_C \rightarrow g_C$.
The above example also provides the explanation of this dissimilarity of behavior between LC and SK. During the process of reduction in LC, contractions can be applied to any part of an LE, including the one to be abstracted (i.e., to the right of the colon) in an abstraction. In SK an R-abstract is an abbreviation for an ob form, and there are no provisions for applying contractions on the part to be abstracted before the whole abstract is computed. (Compare Parts (4) of Theorems 2-7 and 3-1.) Thus, while all SK reductions can be carried out (in

terms of interpretations) in LC, additional reductions are possible in LC that have no SK counterparts. By supplementing SK with some special rules, its reductions may be made to correspond exactly with those in LC [9, p. 218]. The modified reductions are termed strong; in contrast, SK reductions are weak. But in strengthening the reductions, one loses the extreme simplicity of the SK reduction process as presently based on S- and K-contractions alone.

Although weaker, the SK reductions completely suffice for the modelling of programming languages, and in the sequel, we describe our model in terms of SK only. But, since all of the reductions that we use also permit LC interpretations, the model may just as well be regarded as being in LC; the only needed modification is to replace the symbol A (the future alternative to A^R) by A, and to consider S and K as abbreviations for the LE's $\lambda xyz:xz(yz)$ and $\lambda xy:x$, respectively. A purely LC formulation of our model is given in [1].

We also remark that LC supports extensionality -- the property that if fa $\leftrightarrow$ ga for all a, then f $\leftrightarrow$ g -- while SK does not. As a result, two representations of the same intuitive function of n arguments are mutually interconvertible in LC but, in general, only n-interconvertible in SK. This, however, is no hindrance at all: It is the mechanical process

of functional evaluation that we keep simple by adhering to weak reductions. But in making formal arguments about functions, such as proving equivalence of functions and in choosing function representations, we may freely employ n-interconvertibility, which is as easy to use in these cases as interconvertibility.

2.4 Additional Obs

In this section, we define a number of obs and ob families that will be employed in the representation of programs. We begin by introducing the most important of these:

(4-1) Definition.

$$I \equiv SKK , \qquad B \equiv S(KS)K ,$$

$$C \equiv S(S(KS)(S(KK)S))(KK) ,$$

$$W \equiv SS(SK) , \quad Z \equiv KI ,$$

$$T \equiv CI , \qquad D \equiv WI ,$$

$$\beta \equiv B(BS)B, \quad Y \equiv B(SWW)B , \quad \Omega \equiv YK .$$

Clearly, I, B, C, and W are normal obs. Though it may not be so obvious from their definitions, the obs Z, T, D, β, and Y can be easily seen to be normal also. However, it will follow from the property of Y given below that Ω does not possess a normal form.

We shall use the term "rule" to designate frequently used reduction properties. We list a set of rules (including S- and K-contraction for completeness) that can be easily derived from the above definitions.

(4-2) Rule. For all obs a, b, c, d:

(1) $Sabc \rightarrow ac(bc)$,

(2) $Kab \rightarrow a$,

(3) $Ia \rightarrow a$,

(4) Babc → a(bc) ,

(5) Cabc → acb ,

(6) Wab → abb ,

(7) Zab → b ,

(8) Tab → ba ,

(9) Da → aa ,

(10) βabcd → a(bd)(cd),

(11) Ya → a(Ya) ,

(12) Ωa → Ω .

We may wish to incorporate some rules directly in an ob reducing mechanism in order to avoid intermediate reduction steps. This places the selected rules at par with contractions. Equivalently, we may wish to extend the calculus by admitting the obs with such rules as new primitive obs (in addition to S and K), and the rules themselves as contractions. In general, we have the choice of many rules for the same ob. As an example, for the ob B we have:

(a) B → S(KS)K

(b) Ba → S(Ka)

(c) Bab → S(Ka)b

(d) Babc → a(bc)

(e) Babcd → a(bc)d

For the sake of determinateness in reduction, however, we should allow only one rule for each primitive ob. (Note that if B is regarded as a primitive ob and (d) as the

"B-contraction," then, by Definition 1-4, the ob Bab has to be considered irreducible whenever a and b are irreducible.) Although SK is formulated in terms of S and K alone, we shall state unique rules for the obs that are important enough to be candidates for use as primitives in an extended calculus. Now, whenever any extensions to SK are stipulated, the following question naturally arises: Do the Church-Rosser Theorem and related properties remain valid in the extended calculus? Fortunately, this theorem holds in very general situations involving replacement rules (Rosen [32]). It follows from Rosen's work that the theorem is supported, for example, by the calculus containing S, K, and the obs of Definition 3-1 as primitives and Rules 3-2 as contractions. Furthermore, the standard-order reduction algorithm continues to be valid in the extended calculus.

We shall next describe the representation of natural numbers and number-theoretic functions. Following Church [8, Chap. 3], we represent a natural number n by an ob $\underline{n}$ with this desired property:

(4-3) $\underline{n}xy \rightarrow \underbrace{x(x(\ \ldots\ (x\ y)\ \ldots))}_{\text{n occurrences of x}}$.

The ob $\underline{n}$ is defined inductively in terms of the obs $\underline{\text{suc}}$ and $\underline{0}$, representing the successor function and zero, respectively, as follows:

(4-4) Definition.

$$\underline{suc} \equiv SB \quad ,$$

$$\underline{0} \equiv Z \quad ,$$

$$\underline{n+1} \equiv \underline{suc}\ \underline{n} \ .$$

That is, we have

$$\underline{0} \equiv Z \ , \quad \underline{1} \equiv SBZ \ , \quad \underline{2} \equiv SB(SBZ) \ , \quad \ldots \quad .$$

Using the above definitions, we immediately obtain:

(4-5) Rule.

(1) $\underline{0}xy \rightarrow y$,

(2) $\underline{n+1}\ xy \rightarrow x(\underline{n}xy)$,

from which the property (4-3) follows by induction.

At this point, we introduce some notation due to Curry [9]: We write $a \circ b$ for Bab and $a_{(n)}$ for $\underline{n}Ba$. Consequently, we have:

(4-6) Rule.

(1) $(a \circ b)c \rightarrow a(bc)$,

(2) $a_{(n)}bc_1 \ldots c_n \rightarrow a(bc_1 \ldots c_n)$.

In particular:

(3) $K_{(n)}ab_1 \ldots b_n c \rightarrow ab_1 \ldots b_n$,

(4) $B_{(n)}ab_1 \ldots b_n cd \rightarrow ab_1 \ldots b_n\ (cd)$.

In expressions involving $\circ$ we shall regard $\circ$ as being of lower precedence than application and of higher precedence

than another ∘ at the right. Thus a∘bc will stand for a∘(bc), not (a∘b)c, and a∘b∘c for (a∘b)∘c, not a∘(b∘c). Note, however, that a∘b∘c $\leftrightarrow_1$ a∘(b∘c).

Anticipating the forthcoming discussion of tuples, we next introduce the representation of ordered pairs and triples.

(4-7) Definition.

$$\langle a,b\rangle \equiv C(Ta)b \ ,$$

$$\langle a,b,c\rangle \equiv C\langle a,b\rangle c \ .$$

The definition yields the following reduction properties.

(4-8) Rule.

(1) $\langle a,b\rangle c \rightarrow cab$,

(2) $\langle a,b\rangle K \rightarrow a$ and $\langle a,b\rangle Z \rightarrow b$,

(3) $\langle a,b,c\rangle d \rightarrow dabc$,

(4) $\langle a,b,c\rangle(K\circ K) \rightarrow a$, $\langle a,b,c\rangle(KK) \rightarrow b$,

$\langle a,b,c\rangle(KZ) \rightarrow c$.

We can now describe the representation of the predecessor function on natural numbers.

(4-9) Definition. $\underline{pred} \equiv \langle S(BCT)\underline{suc}\circ T\underline{0}, \langle \underline{0},\underline{0}\rangle, K\rangle$.

(4-10) Rule.

$$\underline{pred}\ n \rightarrow \begin{cases} \underline{0}, & \text{if } n = 0, \\ \underline{n-1}, & \text{if } n > 0. \end{cases}$$

To prove this rule, let h ≡ S(BCT) $\underline{suc}\circ T\underline{0}$. Then it can be easily shown that

$$h\ \langle \underline{m}, \underline{n}\rangle \rightarrow \langle \underline{n}, \underline{n+1}\rangle \ .$$

Hence,

$$\underline{\text{pred}}\ \underline{n} \equiv \langle h, \langle \underline{0}, \underline{0}\rangle, K\rangle\ \underline{n}$$

$$\rightarrow \underline{n}\ h\langle \underline{0}, \underline{0}\rangle K$$

$$\rightarrow \underbrace{h(h(h(\ \ldots\ (h\ \langle \underline{0}, \underline{0}\rangle)\ \ldots\ )))K}_{\uparrow\ n \text{ occurrences of } h}$$

$$\rightarrow \begin{cases} \langle \underline{0}, \underline{0}\rangle K \rightarrow \underline{0}, & \text{if } n = 0\ , \\ \langle \underline{n-1}, \underline{n}\rangle K \rightarrow \underline{n-1}, & \text{if } n > 0\ . \end{cases}$$

It will be found convenient to define the ob families by which the properties of K, I, and B are generalized in the following manner.

(4-11) Rule.

(1) $K_n a b_1 \ldots b_n \rightarrow a, \quad n \geq 1\ .$

(2) $I^m_n a_1 \ldots a_n \rightarrow a_m\ , \quad n \geq m \geq 1\ ,$

(3) $B^m_n a b_1 \ldots b_m c_1 \ldots c_n \rightarrow a(b_1 c_1 \ldots c_n) \ldots (b_m c_1 \ldots c_n),$

$m,n \geq 1\ .$

Thus, we will have

$$K \leftrightarrow_2 K_1\ , \quad I \leftrightarrow_1 I^1_1\ , \quad B \leftrightarrow_3 B^1_1\ , \quad S \leftrightarrow_3 B^1_2 I,$$

$$a_{(n)} \leftrightarrow_{n+1} B^1_n a\ .$$

The above rules can be realized by making the following

(4-12) Definition.

(1) $K_n \equiv \underline{n}\ K\ , \quad n \geq 1\ ,$

(2) $I^m_n \equiv \underline{m-1}\ K\ (\underline{n-m}\ K), \quad n \geq m \geq 1\ ,$

(3) $B^m_n \equiv \underline{n}(\underline{m-1}(C(\underline{3}\ B)\beta \circ TI)B)\ , \quad m,n \geq 1\ .$

It is easy to verify parts (1) and (2) of Rule 4-l1 from the above definitions. We shall prove part (3).

Let $h \equiv C(\underline{3}\ B)\beta \circ TI$. We first show by induction on m that

(*) $\qquad \underline{m\text{-}1}\ hBab_1 \ldots b_m c_1 \rightarrow a(b_1c_1)\ldots(b_mc_1),\ (m \geq 1).$

This result is immediate for m = 1. Assume it is true for m = k; then

$\underline{k}\ h\ Bab_1 \ldots b_{m+1}c_1$

$\rightarrow h(\underline{k\text{-}1}\ h\ B)\ ab_1 \ldots b_{m+1}c_1$

$\rightarrow (C(\underline{3}\ B)\ \beta \circ TI)\ (\underline{k\text{-}1}\ h\ B)\ ab_1 \ldots b_{m+1}c_1$

$\rightarrow C(\underline{3}\ B)\ \beta\ (TI\ (\underline{k\text{-}1}\ h\ B))\ ab_1 \ldots b_{m+1}c_1$

$\rightarrow \underline{3}\ B(TI(\underline{k\text{-}1}\ h\ B))\ \beta\ ab_1 \ldots b_{m+1}c_1$

$\rightarrow \underline{3}\ B(\underline{k\text{-}1}\ h\ BI)\ \beta\ ab_1 \ldots b_{m+1}c_1$

$\rightarrow \underline{k\text{-}1}\ h\ B\ I(\beta a\ b_1b_2)\ b_3 \ldots b_{m+1}c_1$, by 4-6 (2) ,

$\rightarrow I(\beta ab_1b_2c_1)(b_3c_1)\ldots(b_{m+1}c_1)$, by induction hypothesis,

$\rightarrow a(b_1c_1)(b_2c_1)\ \ldots\ (b_{m+1}c_1)$.

So (*) has been established. To prove 4-1l (3) now, we use induction on n. For $m \geq 1$, we have

$B_1^m\ ab_1 \ldots b_m c_1 \equiv \underline{1}(\underline{m\text{-}1}\ hB)\ ab_1 \ldots b_m c_1$

$\rightarrow \underline{m\text{-}1}\ hB(\underline{0}(\underline{m\text{-}1}hB))a)\ b_1 \ldots b_m c_1$

$\rightarrow \underline{m\text{-}1}\ hB\ ab_1 \ldots b_m c_1$

$\rightarrow a(b_1c_1)(b_2c_1)\ \ldots\ (b_mc_1)$, by (*).

Hence assuming 4-11 (3) true for $m \geq 1$ and $n = k$, we obtain

$$B^{m}_{k+1}ab_1\ldots b_m c_1\ldots c_{k+1} \equiv \underline{k+1}(\underline{m-1}\ h\ B)\ ab_1\ldots b_m c_1\ldots c_{k+1}$$

$$\rightarrow \underline{m-1}\ hB\ (\underline{k}(\underline{m-1}\ hB)a)\ b_1\ldots b_m c_1\ldots c_{k+1}$$

$$\rightarrow \underline{k}(\underline{m-1}\ hB\ a(b_1c_1)\ldots(b_mc_1)c_2\ldots c_{k+1}\ ,\ \text{by (*)},$$

$$\equiv\ B^{m}_{k}a(b_1c_1)\ldots(b_mc_1)c_2\ldots c_{k+1}$$

$$\rightarrow a(b_1c_1c_2\ldots c_{k+1})\ldots\ (b_mc_1c_2\ldots c_{k+1})\ ,$$

by induction hypothesis.

As shown by the next definition and the succeeding rues, it is possible to generate the above ob families from single obs. We can thus avoid postulating infinitely many primitives in an extended calculus.

(4-13) Definition.

(1) $\mathcal{K} \equiv TK$,

(2) $\mathcal{I} \equiv \beta B(C\underline{pred}K)(CCK\circ T\underline{pred})$,

(3) $\mathcal{B} \equiv T\circ C(C\underline{pred}(C(\underline{3}B)\ \beta\circ TI))B$.

(4-14) Rule.

(1) $\mathcal{K}\ \underline{n} \rightarrow K_n$, $\quad n \geq 1$,

(2) $\mathcal{I}\ \underline{m}\ \underline{n} \rightarrow I^{m}_{n}$, $\quad n \geq m \geq 1$,

(3) $\mathcal{B}\ \underline{m}\ \underline{n} \rightarrow B^{m}_{n}$, $\quad m,n \geq 1$.

Proof. Omitted.

Before listing further obs, we state an algorithm to obtain abstracts in terms of the obs K_n , I_n^m, and B_n^m. Unlike the algorithm of Definition 2-5 in which abstraction is carried out for one indeterminate at a time, the present algorithm performs the abstraction with respect to all specified indeterminates in a single step, and amounts to simple replacements of indeterminates with special obs.

(4-15) Definition.

(1) An initial component of an ob form is either the ob form itself or the left immediate component of an initial component of the form.

(2) A primal component of an ob form is either its shortest initial component, or a right immediate component of one of its initial components.

Example. The ob form e ≡ SK(x(KK)yz)(S(wz)(SSy))(xyz) has five initial components, namely, S, SK, SK(x(KK)yz), SK(x(KK)yz)(S(wz)(SSy)), and e itself; the primal components of e are S, K, x(KK)yz, S(wz)(SSy), and xyz.

(4-16) Definition. Let e be an ob form and $x_1,...,x_n$ ($n \geq 1$), be indeterminates. Then the *-abstract of e with respect to $x_1,...,x_n$, denoted $A^*x_1...x_n:e$, is the first of the following ob forms, selected in the given order, according as the condition is satisfied:

(1) $K_n e$, if x_i o$\not$ e for all $1 \leq i \leq n$,

(2) I , if $e \equiv x_1 \ldots x_n$,

(3) I_n^i , if $e \equiv x_i$ for some $1 \leq i \leq n$,

(4) f , if $e \equiv f x_1 \ldots x_n$, and x_i o$\not$ f for all $1 \leq i \leq n$,

(5) $B_n^m I \; I_n^m (A^* x_1 \ldots x_n : f_2) \ldots (A^* x_1 \ldots x_n : f_m)$,

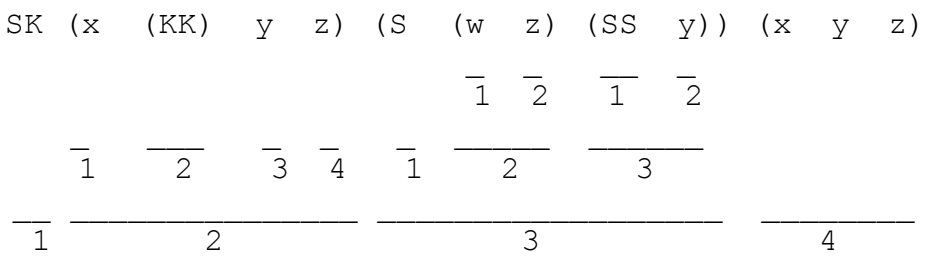
if $e \equiv f_1 f_2 \ldots f_m$, $f_1, f_2, \ldots, f_m$ are primal components of e, and $f_1 \equiv x_i$ for some $1 \leq i \leq n$,

(6) $B_n^{m-1} f_1 (A^* x_1 \ldots x_n : f_2) \ldots (A^* x_1 \ldots x_n : f_m)$,

if $e \equiv f_1 f_2 \ldots f_m$, f_1 is the longest initial component of e such that x_i o$\not$ f_1 for all $1 \leq i \leq n$, and $f_2, \ldots, f_m$ are primal components of e.

Note that for cases (5) and (6), we decompose the ob form e into an initial component and a number of primal components, with the initial component chosen to be either a single indeterminate among $x_1, \ldots, x_n$, if possible, or else the longest possible ob form not containing any of $x_1, \ldots, x_n$.

Example. To find $A^* xyz : e$, where e is as defined in the previous example. Diagrammed below is the decomposition of e and its components according to the conditions prescribed in (5) and (6) above.



Hence, A*xyz:e

$\equiv B_3^3\ (SK)(B_3^4\, II_3^1\ (K_3(KK))I_3^2\ I_3^3)(B^2{}_3S(B_3^1 w\ I_3^3)(B_3^1\ (SS)I_3^2))I$.

By using Rules 4-10, it can be easily verified that, except for parts (3b) and (4), Theorem 2-7 remains valid when A^R is replaced by A*. It is possible to restore (3b) by a slight modification in the above definition of abstracts. Let $A^{**}x_1...x_n$:e be specified similarly to $A^*x_1...x_n$:e, except for modifying the clause (4) in Definition 4-16 to

(4) $A^{**}x_1...x_{m-1}$:f, if for some $1 \leq m < n$,

$e \equiv fx_m...x_n$, and for $m \leq i \leq n$, x_i o$\not\in$ f .

(We consider A^{**} $x_1...x_{m-1}$:f to be simply f when m = 1.) Let e be the same ob form as before. Then, we have

$A^{**}xyz:e \equiv B_3^3\ (SK)(B_1^2 II_1^1\ (K_1(KK)))(B_3^2S(K_2w)(B_3^1\ (SS)I_3^2))I$.

It can be verified that with A** used instead of A^R , all parts except (4) of Theorem 2-7 remain valid.

In the sequel, we will drop the superscript on A altogether, thereby leaving the abstraction algorithm unspecified.

A function on natural numbers is partial recursive if it can be defined using the following functions and function-forming schemes [14]:

1. Successor Function $S(x) = x+1$.
2. Constant Functions $C_q^n(x_1,\ldots,x_n) = q$, q a natural number.
3. Identity Functions $U_i^n(x_1,\ldots,x_n) = x_i$, $1 \leq i \leq n$.
4. Composition Scheme: Given functions $g, h_1, \ldots, h_m$, to obtain f such that
   $f(x_1,\ldots,x_n) = g(h_1(x_1,\ldots,x_n),\ldots,h_m(x_1,\ldots,x_n))$.
5. Primitive Recursion Scheme. Given functions g and h, to obtain f such that

   $$f(x_1,\ldots,x_m,0) = g(x_1,\ldots,x_m) ,$$

   $$f(x_1,\ldots,x_m,y+1) = h(x_1,\ldots,x_m,y,f(x_1,\ldots,x_m,y)).$$
6. Minimalization Scheme. Given a function g, to obtain f such that

   $$f(x_1,\ldots,x_m) = (\mu y)[g(x_1,\ldots,x_m,y) = 0]$$

   = the least integer y, if one exists, such that

   $$g(x_1,\ldots,x_m,y) = 0 .$$

Now the obs suc, $K_n q$ and I_n^i clearly represent the first three functions of the above list. Further, if the obs

$\underline{g}$ and $\underline{h_i}$ are the representations of the functions g and h_i, respectively, then $\underline{f} \equiv B_n^m \underline{g}\underline{h_1}\ldots\underline{h_m}$ represents the f of Scheme 4. Thus, to complete the combinatory representation of partial recursive functions, it only remains to provide the obs $\underline{primrec}_m$ and $\underline{mnm\ell z}_m$ such that the definitions $f \equiv \underline{primrec}_m\ \underline{h}\ \underline{g}$ and $f \equiv \underline{mnm\ell z}_m\ \underline{g}$ may correspond to Schemes 5 and 6. For this purpose, we give the following definitions and rules, adapted from Petznick [29] with minor modifications.

(4-17) Definition.

(1) $\underline{primrec}_0 \equiv Axy{:}\langle Az{:}\langle \underline{suc}(zK), x(zK)(xZ)\rangle, \langle \underline{0}, y\rangle, Z\rangle$,

(2) $\underline{primrec}_m \equiv B_4^2\ \underline{primmrec}_0$, $m \geq 1$,

(3) $\underline{mnm\ell z}_0 \equiv D(Axyz{:}zy(K(xx(\underline{suc}\ y)z))y)\underline{0}$,

(4) $\underline{mnm\ell z}_m \equiv \underline{mnm\ell z}_{0(m)}$ $\quad (\equiv \underline{m}\ B\ \underline{mnm\ell z}_0)$, $m \geq 1$.

(4-18) Rules.

(1) $\underline{primrec}_m\ abc_1\ldots c_m n$

$$\rightarrow \begin{cases} bc_1\ldots c_m \text{ , if } n = 0, \\ ac_1\ldots c_m \underline{n-1}(\underline{primrec}_m abc_1\ldots c_m \underline{n-1}), \text{ if } n > 0. \end{cases}$$

(2) $\underline{mnm\ell z}_m ab_1\ldots b_m \rightarrow \underline{n}$,

provided that $ab_1\ldots b_m \rightarrow \underline{0}$, and, for all $0 \leq p < n$, there exists some $q > 0$ such that $ab_1\ldots b_m\underline{p} \rightarrow \underline{q}$.

We shall make use of certain special forms of recursion in specifying some ob sequences. It is possible to give individual, explicit definitions for the obs of these sequences. We indicate in the table below how all members of such ob sequences can be generated from a single ob, thus avoiding the need to postulate infinitely many primitives in an extended calculus containing such obs.

(4-19) Table.

	Specification of the ob sequence f_i	Definition of the ob f such that f $\underline{i} \rightarrow f_i$
(1)	$f_0 \equiv g$, $f_{n+1} \equiv h\ f_n$.	$f \equiv \langle h,g \rangle$.
(2)	$f_1 \equiv g$, $f_{n+1} \equiv h\ f_n$.	$f \equiv \langle h,g \rangle \circ e\ \underline{pred}$.
(3)	$f_0 \equiv g$, $f_{n+1} \equiv h\ \underline{n}$.	$f \equiv C(SI(K\circ(h\circ\underline{pred})))g$.
(4)	$f_0 \equiv g$, $f_{n+1} \equiv h\ \underline{n}\ f_n$.	$f \equiv \underline{primrec}_0\ h\ g$.
(5)	$f_0 \equiv g$, $f_{n+1}a_1 \ldots a_{n+1} \equiv h(f_n a_1 \ldots a_n)a_{n+1}$.	$f \equiv \underline{primrec}_0 \langle B,h \rangle g$.
(6)	$f_0 \equiv g$, $f_{n+1}a_1 \ldots a_{n+1} \equiv ha_1(f_n a_2 \ldots a_{n+1})$.	$f \equiv \underline{primrec}_0(C\circ(CBh\circ TB))g$.

For each entry in the Table 4-19, the relation $f\underline{n} \to f_n$ can be verified from the definition of f. For example, we prove the case of entry (3). Let $p \equiv K\circ(h\circ\underline{pred})$, so that $f \equiv C(SIp)g$. Then:

$$f\underline{0} \equiv C(SIp)g\underline{0} \to SIp\underline{0}g \to I\underline{0}(p\underline{0})g \to \underline{0}(p\underline{0})g \to g \equiv f\underline{0}.$$

$$f\underline{n+1} \equiv C(SIp)g\underline{n+1} \to SIp\underline{n+1}g \to I\underline{n+1}(p\underline{n+1})g$$
$$\to \underline{n+1}(p\underline{n+1})g \to p\underline{n+1}(\underline{n}(p\underline{n+1})g)$$
$$\equiv (K\circ(h\circ\underline{pred}))\underline{n+1}(\underline{n}(p\underline{n+1})g)$$
$$\to K(h(\underline{pred}\ \underline{n+1}))(\underline{n}(p\underline{n+1}g)$$
$$\to h\underline{n} \equiv f\underline{n+1}\ .$$

To facilitate the representation of a succession of function applications, we introduce the ob family $\underline{nest}_n$:

(4-20) Definition.

$$\underline{nest}_0 \equiv I\ ,$$

$$\underline{nest}_{n+1}a_1\ldots a_{n+1} \equiv Ba_1(\underline{nest}_n\ a_2\ldots a_{n+1})\ ,$$
$$[a_1,\ldots,a_n] \equiv \underline{nest}_n\ a_1\ldots a_n\ .$$

(4-21) Rule. $[a_1,a_2,\ldots,a_n]b \to a_1(a_2(\ldots(a_n\ b)\ldots))$.

Note the following properties of nests:

$$[a,\ldots,a] \leftrightarrow_1 \underline{n}\ a\ ,$$

(with n occurrences)

$$[a_1,\ldots,a_m,b_1,\ldots,b_n] \leftrightarrow_1 [a_1,\ldots,a_m]\circ[b_1,\ldots,b_n]\ .$$

The purpose of the next set of obs is to represent functional operators for permuting and duplicating the arguments of functions. These obs thus generalize the effect of C and W.

(4-22) Definition.

(1) $\underline{\text{rot}\ell}_1 \equiv I$,

$\underline{\text{rot}\ell}_{n+1} \equiv (BC\circ B)\ \underline{\text{rot}\ell}_n$, $\quad n \geq 1.$

(2) $\underline{\text{rotr}}_1 \equiv I$,
$\underline{\text{rotr}}_{n+1} \equiv C(B\circ B)C\ \underline{\text{rotr}}_n$, $\quad n \geq 1.$

(3) $\underline{\text{swap}}^m_n \equiv [\underline{\text{rotr}}_m , \underline{\text{rotr}}_n , \underline{\text{rot}\ell}_{m+1} , \underline{\text{rot}\ell}_n]$, $1 \leq m \leq n$.

(4) $\underline{\text{perm}}^n{}_{i_1,\ldots,i_m} \equiv B^m_n\ I\ I^{i_1}{}_n \ldots I^{i_m}{}_n$,

$1 \leq i_j \leq n$ for all $1 \leq j \leq m$.

(5) $\text{dup}_0 \equiv I$,
$\underline{\text{dup}}_{n+1} \equiv \underline{\text{dup}}_n \circ (W_{(n+1)}\underline{\text{rotr}}_{n+1})_{(n+1)}$, $n \geq 1$.

(4-23) Rule.

(1) $\underline{\text{rot}\ell}_n\ a\ b_1b_2\ldots b_n \rightarrow a\ b_2\ldots b_n\ b_1$, $\quad n \geq 1$.

(2) $\underline{\text{rotr}}_n\ a\ b_1\ldots b_{n-1}b_n \rightarrow a\ b_nb_1\ldots b_{n-1}$, $\quad n \geq 1$.

(3) $\underline{\text{swap}}^m_n\ a\ b_1\ldots b_n \rightarrow a\ b_1\ldots b_{m-1}b_nb_{m+1}\ldots b_{n-1}b_m$,

$1 \leq m \leq n$.

(4) $\underline{\text{perm}}^n{}_{i_1,\ldots,i_m} a_1\ldots a_n \rightarrow a_{i_1}\ldots a_{i_m}$,

$1 \leq i_j < n$ for all $1 \leq j \leq m$.

(5) $\underline{\text{dup}}_n\ a\ b_1\ldots b_n \rightarrow a\ b_1\ldots\ b_nb_1\ldots\ b_n$.

Proof. (1), (2), and (5) by induction on n,
(3) by (1) and (2), and (4) by Rule 4-10 (3).

Our representation of ordered tuples is adopted from Church [8]. The essential idea is to regard the tuple $\langle a_1,\ldots,a_n\rangle$ as the abstract Ax: $xa_1\ldots a_n$, that is, to define tuples so as to obtain the rule

$$\langle a_1,\ldots,a_n\rangle b \rightarrow ba_1\ldots a_n \ .$$

(Two special cases of ordered tuples, namely, pairs and triples, and their associated rules were introduced earlier (4-7 and 4-8). In addition to the tuple-forming operators, we will require a number of obs to represent various manipulations on tuples, such as inserting, retrieving, or changing elements. Here are the necessary definitions and rules.

(4-24) Definition.

(1) $\underline{tup}_0 \equiv I$,

$\underline{tup}_{n+1}\ a_1\ldots a_{n+1} \equiv C(\underline{tup}_n\ a_1\ldots a_n)a_{n+1}$,

$\langle a_1,\ldots,a_n\rangle \equiv \underline{tup}_n\ a_1\ldots a_n$, $\quad n \geq 0$.

(2) $\underline{insert} \equiv C$.

(3) $\underline{elem}_n^m \equiv I_n^m$, $\quad 1 \leq m \leq n$.

(4) $\underline{replace}_n^m \equiv [\underline{rot\ell}_{m+1},\underline{rotr}_m,K]\underline{tup}_n$, $\quad 1 \leq m \leq n$.

(5) $\underline{fput}_n^m \equiv B_n^2 I(\underline{rotr}_{n+1}\ \underline{replace}_n^m)$, $\quad 1 \leq m \leq n$.

(6) $\underline{funtup}_n^m = B_n^m\ \underline{tup}_n$, $\quad 1 \leq m \leq n$.

(4-25) Rule.

(1) $\langle a_1,\ldots,a_n\rangle b \rightarrow b a_1\ldots a_n$.

(2) insert $\langle a_1,\ldots,a_n\rangle b \rightarrow \langle a_1,\ldots,a_n,b\rangle$.

(3) $\underline{\text{elem}}_n^m\ a_1\ldots a_n \rightarrow a_m$, $\quad 1 \le m \le n$.

(4) $\underline{\text{replace}}_n^m\ b\ a_1\ldots a_n \rightarrow \langle a_1,\ldots,a_{m-1},b,a_{m+1},\ldots,a_n\rangle$,

$1 \le m \le n$.

$\langle a_1,\ldots,a_n\rangle\ (\underline{\text{replace}}_n^m\ b)$

$\rightarrow \langle a_1,\ldots,a_{m-1},b,a_{m+1},\ldots,a_n\rangle$, $1 \le m \le n$.

(5) $\underline{\text{fput}}_n^m\ f a_1\ldots a_n \rightarrow \langle a_1,\ldots,a_{m-1},f a_1\ldots a_n,a_{m+1},\ldots,a_n\rangle$,

$1 \le m \le n$.

$\langle a_1,\ldots,a_n\rangle\ (\underline{\text{fput}}_n^m\ f)$

$\rightarrow \langle a_1,\ldots,a_{m-1},f a_1\ldots a_n,a_{m+1},\ldots,a_n\rangle$,

$1 \le m \le n$.

(6) $\underline{\text{funtup}}_n^m\ f_1\ldots f_m\ a_1\ldots a_n \rightarrow \langle f_1 a_1\ldots a_n,\ldots,f_m a_1\ldots a_n\rangle$.

Proof. Omitted

We claim that there exist obs nest, rotℓ, rotr, tup, elem, and replace possessing the following reduction properties:

$\underline{\text{nest}}\ \underline{\text{n}} \rightarrow \underline{\text{nest}}_{\text{n}}$, $n \geq 0$,

$\underline{\text{rot}\ell}\ \underline{\text{n}} \rightarrow \underline{\text{rot}\ell}_{\text{n}}$, $n \geq 1$,

$\underline{\text{rotr}}\ \underline{\text{n}} \rightarrow \underline{\text{rotr}}_{\text{n}}$, $n \geq 1$,

$\underline{\text{tup}}\ \ \underline{\text{n}} \rightarrow \underline{\text{tup}}_{\text{n}}$, $n \geq 0$,

$\underline{\text{elem}}\ \underline{\text{m}}\ \underline{\text{n}} \rightarrow \underline{\text{elem}}_{\text{n}}^{\text{m}}$, $1 \leq m \leq n$,

$\underline{\text{replace}}\ \underline{\text{m}}\ \underline{\text{n}} \rightarrow \underline{\text{replace}}_{\text{n}}^{\text{m}}$, $1 \leq m \leq n$.

The definitions of nest, rotℓ, rotr, and tup follow directly from Table 4-19 and Definitions 4-20,22,24; in addition, we can take

$\underline{\text{elem}} \equiv I$,

$\underline{\text{replace}} \equiv Axy: [\underline{\text{rot}\ell}\ (\underline{\text{suc}}\ x), \underline{\text{rotr}}\ x, K]\ (\underline{\text{tup}}\ y)$.

It should be noted that, indeed, each ob sequence that we have introduced so far, or will introduce in the sequel, can be generated from a finite number of obs.

CHAPTER 3

BASIC PROGRAMMING FEATURES

In this chapter, we undertake the representation of the more elementary features of high-level programming languages, postponing the discussion of jumps and procedures to later chapters. We use the ALGOL 60 [27] terminology and notation, whenever possible, to express programming language features.

3.1 An Overview

To construct our model, we start with the representations of the atomic constituents of program, such as constants and variables. We then develop the rules for obtaining the representations of larger and larger programming constructs by combining the representations of their syntactic units in certain ways, eventually deriving the rules for representing whole programs. Throughout this development, we are guided by intuitive interpretations of programming constructs as functions.

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, for example, the program:

```
begin integer a,b,c;

  read a; read c; b := a+c; write b; b:=b-2xc;
  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

$$f(x,y) = \langle x-y, x+y \rangle \ . \qquad \text{(i)}$$

The function f is representable in SK by an abstract

$$f \equiv Axy{:}\langle x-y, x+y \rangle \qquad \text{(ii)}$$

having the reduction property

$$\underline{f}xy \rightarrow \langle x-y, x+y \rangle \ . \qquad \text{(iii)}$$

(Although the expressions x-y and x+y are not exactly ob forms, they can be easily translated to be such -- as we will soon see.) Accordingly, we would like to set up the model in such a manner that the representation of the above program may turn out to be an ob f, satisfying (iii). 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. For, f represents precisely this input-output function.

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

(1-1) $\{P\}\{i_1\}...\{i_p\} \rightarrow \langle\{o_1\}...\{o_q\}\rangle$,

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 fulfill 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 of (the augmented) SK. Now 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+l for the blocks immediately enclosed by a block at level n.) The correspondence between program variables and indeterminates will be set up by assigning distinct indeterminates to distinct variables (in the above sense) in some order.[1] Since we

---

1 We shall assign single indeterminates to both simple variables and array variables, however, the treatment of arrays is deferred to a later section, and until that discussion all variables are meant to be simple.

have not specified the alphabet 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 notion of environment defined as follows: 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.[2]

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 declarations.

---

2 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 will be introduced in Chapter 5.

```
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 SK representation of a construct depends upon the construct's own environment as well as the environments of its constituents. We denote the SK representation of a construct X appearing in the environment E by $\{X\}_E$. We drop the subscript from the above notation 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

The general criterion for choosing the SK representations of the values of the various types employed in programming languages and associated operations and relations may be stated thus: Let * denote a unary operation and ** a binary operation or relation. Then for all operands a and b of the proper types for which * and ** are defined, it should be the case that

{*}{a} → {value of (*a)} ,

{**}{a}{b} → {value of (a**b) .

In this way, it becomes possible to interpret the computation of values as simply the SK reduction process. Of course, the above criterion may be met by several different representations, in which case the choice is dictated by the simplicity of the resulting obs.

We begin with the Boolean values. The values true and false are represented by the obs K and Z, (Definition 2.4-1), respectively. The motivation behind this choice is that it leads to a very simple representation of conditional expressions; namely,

{if b then c else d) ≡ {b}{c}{d} (i)

Quite short representations of logical operators can then be provided by using McCarthy's well-known technique of expressing these operators in terms of conditionals [23].

In particular, we obtain:

(2-1) Definitions and Rules.

(1)	true ≡ K ,	true a b → a .
(2)	false ≡ Z ,	false a b → b .
(3)	¬ ≡ <false,true> ,	¬ a → a false true .
(4)	∧ ≡ CC false ,	∧ a b → a b false .
(5)	∨ ≡ T true ,	∨ a b → a true b .
(6)	imp ≡ CC true ,	imp a b → a b true .
(7)	eqv ≡ CS ¬ ,	eqv a b → a b (¬ b) .

It is an easy matter to verify that the above presented obs correctly represent the corresponding logical operators. For example, it is seen that

imp true true → true true true → true

imp true false → true false true → false

imp false true → false true true → true

imp false false → false false true → true

So that a conditional expression may be recognizable as such even when represented in SK, we make the following trivial definition and restate the representation rule (1).

(2-2) Definition. if ≡ I .

(2-3) Representation Rule.

{if b then c else d } ≡ if {b}{c}{d} .

Let us now turn to the representation of arithmetic in SK. We will only be concerned with integers and rational numbers. (The type called "real" in the programming parlance will be termed "rational" here.) The representation of natural numbers has already been described (Definition 2.4-4). Following the procedure common in analysis, one can consider integers and rational numbers to be equivalence classes of natural numbers and integers, respectively. For convenience in carrying out reductions with their representations, it seems preferable, however, to represent these numbers in terms of uniquely selected members of the equivalence classes they stand for. Thus, an integer p is traditionally defined to be the set of all pairs <m,n> of natural numbers in and n, such that, intuitively, p is a solution of the equation m = n+p; but we prefer to define the SK representation of p to be the representation of that particular pair in the set which has the smallest m and n (at least one of these being necessarily zero). Again, a rational number r is the set of all pairs <p,q> of integers p and q such that r intuitively satisfies $p = q \times r$; but we prefer to represent r by the representation of that pair for which q is positive, p and q are relatively prime, and, furthermore, if p = 0, then q = 1.

Numbers of different types denoted by the same symbols in a programming language are in reality different entities possessing different ob representations. We denote them by underlined numerals, unsubscripted in the case of natural numbers, and with subscripts Z and Q added for integers and rational numbers, respectively. Thus, the obs $\underline{3}$, $\underline{3}_Z$, and $\underline{3}_Q$ and represent the natural number 3, the integer 3 and the rational number 3. As discussed above, we define them as follows:

$\underline{3} \equiv \underline{suc}\ (\underline{suc}\ (\underline{suc}\ \underline{0}))$, where $\underline{suc} \equiv SB$, $\underline{0} \equiv Z$,

$\underline{3}_Z \equiv \langle \underline{3}, \underline{0} \rangle$,

$\underline{3}_Q \equiv \langle \underline{3}_Z, \underline{1}_Z \rangle \equiv \langle\langle \underline{3}, \underline{0} \rangle, \langle \underline{1}, \underline{0} \rangle\rangle$.

Additional examples:

$\underline{0}_Z \equiv \langle \underline{0}, \underline{0} \rangle$, $\underline{0}_Q \equiv \langle \underline{0}_Z, \underline{1}_Z \rangle \equiv \langle\langle \underline{0}, \underline{0} \rangle, \langle \underline{1}, \underline{0} \rangle\rangle$,

$\underline{-3}_Z \equiv \langle \underline{0}, \underline{3} \rangle$, $\underline{-3}_Q \equiv \langle \underline{-3}_Z, \underline{1}_Z \rangle \equiv \langle\langle \underline{0}, \underline{3} \rangle, \langle \underline{1}, \underline{0} \rangle\rangle$,

$\underline{-2.6}_Q \equiv \langle \underline{-13}_Z, \underline{5}_Z \rangle \equiv \langle \underline{0}_Z, \underline{13}_Z \rangle \equiv \langle\langle \underline{0}, \underline{13} \rangle, \langle \underline{5}, \underline{0} \rangle\rangle$.

Having settled upon the representation of numbers themselves, let us turn to the representation of operations and relations defined on numbers. We denote the representations by the corresponding algebraic symbols, again unsubscripted in the case of natural numbers and subscripted with the letters Z and Q in the case of integers and rational numbers, respectively. Thus, $+$, $+_Z$, and $+_Q$ represent natural number, integer, and rational addition, respectively, and are to be so

defined that, for example,

$+\ \underline{3}\ \underline{2} \rightarrow \underline{5}$.

$+_Z\ \underline{3}_Z\ \underline{-2}_Z \rightarrow \underline{1}_Z$, i.e., $+_Z\ \langle\underline{3},\underline{0}\rangle\langle\underline{0},\underline{2}\rangle \rightarrow \langle\underline{1},\underline{0}\rangle$.

$+_Q\ \underline{5/6}_Q\ \underline{-4/9}_Q \rightarrow \underline{7/18}_Q$.

Obs representing the successor and predecessor functions on natural numbers were defined earlier (Definition 2.4-4,2.4-9). Using the method of representing recursively defined functions given in Section 2.4, the SK representations of other operations and relations on natural numbers can be obtained from their well-known recursive definitions [14]. For strong combinatory calculi, this is done in Church [8]; for the weaker SK reductions, some representations are given in Petznick [29). Finally, to represent integer and rational operations, we can make use of the definitions of these operations in terms of, respectively, the natural number and integer operations on the components of the pairs representing their operands. Consider integer operations for example. First we will require an ob $\underline{\text{norm}\ell}_Z$ to represent the "normalization" operation of converting an arbitrary pair of natural numbers in the equivalence class denoting an integer to the unique pair chosen for the CC representation of that integer; e.g.,

$$\underline{\text{norm}\ell}_Z\ \langle\underline{16},\underline{5}\rangle \rightarrow \langle\underline{11},\underline{0}\rangle\ ,$$

Having represented the proper subtraction and minimum selection

operations on natural numbers by the obs $\dot{-}$ and min, respectively, we would need the rule

$$\underline{\text{norm}\ell_Z}\ \langle \underline{m},\underline{n}\rangle \rightarrow \langle \dot{-}\ \underline{m}\ (\underline{\text{min}}\ \underline{m}\ \underline{n}),\ \dot{-}\ \underline{n}\ (\underline{\text{min}}\ \underline{m}\ \underline{n})\rangle\ .$$

In view of Rule 2.4-8(2), an adequate definition for this purpose is:

$$\underline{\text{norm}\ell_Z} \equiv \text{Ax:}\langle\ \dot{-}\ (xK)(\underline{\text{min}}(xK)(xZ)),\ \dot{-}\ (xZ)(\underline{\text{min}}(xK)(xZ))\rangle\ .$$

Now we can represent integer addition and subtraction (in terms of natural number addition on the components of the integer operands) by the obs defined to satisfy the reduction relations

$$+_Z\ \langle \underline{a},\underline{b}\rangle\ \langle \underline{c},\underline{d}\rangle \rightarrow \underline{\text{norm}\ell_Z}\ \langle +\ \underline{a}\ \underline{c}\ ,\ +\ \underline{b}\ \underline{d}\rangle\ ,$$

$$\dot{-}_Z\ \langle \underline{a},\underline{b}\rangle\ \langle \underline{c},\underline{d}\rangle \rightarrow \underline{\text{norm}\ell_Z}\ \langle +\ \underline{a}\ \underline{d}\ ,\ +\ \underline{b}\ \underline{c}\rangle\ .$$

As additional examples, let us consider the familiar programming operations of type-conversion from integers to rational numbers, and the "mixed addition" of rational and integer operands giving a rational result. The obs float and $+_{QZQ}$ representing these operations have to satisfy, for example:

$$\underline{\text{float}}\ \underline{2_Z} \rightarrow \underline{2_Q}\ ,$$

$$+_{QZQ}\ \underline{-3.14_Q}\ \underline{2_Z} \rightarrow \underline{-1.14_Q}\ .$$

The following definitions clearly suffice for these obs:

$\underline{\text{float}} \equiv Ax{:}\langle x, \underline{1}_Z\rangle$,

$+_{QZQ} \equiv Axy{:}+_Q\ x(\underline{\text{float}}\ y)$.

Not all operations, of course, are so easy to deal with. Furthermore, the operations such as the exponentiation to fractional powers, which lead to irrational numbers, can be represented in our scheme only by defining them in terms of functions that approximate the results to some desired precision, using the techniques for representing functions that will be described later. However, the actual details of representing various arithmetic operations and relations are not crucial to our model, for this model is not intended to be used in studying the purely numerical aspects of programs. The purpose of the discussion in this section is simply to indicate that it is possible to represent arithmetic in SK, and to sketch a way of carrying out this representation.

3.4 Expressions

We limit ourselves at present to the expressions that do not contain function designators. (This restriction will be lifted when procedures are discussed.) According to the requirements imposed by programming languages, any variable occurring in such an expression must also occur in the environment of the expression. Let e be an expression and E its environment. Then, in view of the SK representations chosen for operations and constants, the representation $\{e\}_E$ will be defined by induction on the (parse) structure of e to be the following ob form:

1) If e consists of a constant represented by c, then $\{e\}_E \equiv c$.
2) If e consists of a program variable corresponding to the indeterminate x, then $\{e\}_E \equiv x$.
3) If e consists of a k-ary operation (or relation) represented by o, with the subexpressions $e_1,\ldots,e_k$ as operands, then $\{e\}_E \equiv (o\ \{e_1\}_E\ ,\ldots,\ \{e_k\}_E)$.

It follows that the SK representation of an expression may be obtained simply by writing the expression in the prefix notation, with all operator-operands combinations parenthesized, and then replacing all operators and operands by their representations. It further follows that the

representation of an expression is the same in all environments in which it can legally appear. Thus, in accordance with a convention stated in Section 3.2, we may omit the mention of environment in denoting the representation of expressions.

Consider, for example, the following expression in which x, y, z denote program variables:

$$x + \underline{\text{if}}\ y \neq 0\ \underline{\text{then}}\ z+y\ \underline{\text{else}}\ 15. \qquad \text{(i)}$$

In the prefix form, this expression may be written as:

$$(+\ x\ (\underline{\text{if}}\ (\neq\ y\ 0)\ (+\ z\ y)\ 15)). \qquad \text{(ii)}$$

In accordance with the conventions stated in Section 3.2, let us use the symbols x, y, z for indeterminates as well as the program variables. Now if the program variables have been declared to be all of type integer, and the value of the above expression is to be of type integer also, then the expression may be represented by the ob form

$$+_Z\ x\ (\underline{\text{if}}\ (\neq_Z\ y\ \underline{0}_Z)\ (+_Z\ z\ y)\ \underline{15}_Z). \qquad \text{(iii)}$$

On the other hand, if the program variable x and the result are of the type rational, and y and z of type integer, then the expression may be represented by

$$+_Q\ x\ (\underline{\text{float}}\ (\underline{\text{if}}\ (\neq_Z\ y\ \underline{0}_Z)\ (+_Z\ z\ y)\ \underline{15}_Z)). \qquad \text{(iv)}$$

where the ob <u>float</u> represents the type-conversion operation

mentioned in the previous section. But for closer resemblance between the given expression and its representation, we find it preferable to write, instead of (iv),

$$+_{QZQ}\ x\ (\underline{if}\ (\neq_Z\ y\ \underline{0}_Z)\ (+_Z\ z\ y)\ \underline{15}_Z)\ , \qquad \text{(v)}$$

with the ob $+_{QZQ}$ representing the "mixed" addition of a rational number to an integer, producing a rational number as value.

For conciseness of notation in stating the representation of expressions, we shall henceforth assume that all type-conversions are absorbed within mixed operations as above. Furthermore, we shall omit all type indicating subscripts, leaving the types to be inferred from the context. For example, we shall abbreviate (v) to

$$+\ x\ (\underline{if}\ (\neq\ y\ \underline{0})\ (+z\ y)\ \underline{15})\ . \qquad \text{(vi)}$$

No serious ambiguity will arise out of the above convention, because, except for using natural numbers in a few, explicitly indicated, instances, we shall use the numbers and operations of type integer only.

We shall often need the abstract of the representation of an expression with respect to the indeterminates representing the variables in the environment of the expression. The procedure of Definition 2.4-16 is particularly easy to apply in this case. Specifically, for the special case of

expression representations, we may state the following

(4-1) Abstraction Algorithm. Let e be the ob form representing an expression which appears in the environment $(v_1,...,v_n)$.

Then to obtain $Av_1... v_n$:e, rewrite e, replacing

(1) each ob o representing a k-ary operation by $(B_n^k$ o)

(2) each indeterminate v_i, $1 \leq i \leq n$, by I_n^i, and

(3) each ob c representing a constant operand by $(K_n c)$.

Example. Let (x,y,z,w) be the environment of the expression

if y = 218 ∧ x+3 < -z then z+(7+y) else
if x × y ≥ z then entier (x/z) else -12.

The above expression may be represented by the ob form

e ≡ if (∧ (= y 218) (< (+ x 3)(minus z))) (+ z (+ 7 y))
(if (≥ (x × y) z) (entier(/ x z)) -12) ,

where minus and entier have obvious significance. Using the above procedure, we obtain

$Axyzw{:}e \equiv B_4^3$ if$(B_4^2 \wedge (B_4^2 = I_4^2 (K_4$ 218$))(B_4^2 <(B_4^2 + I_4^1 (K_4$3$))$
$(B_4^1$ minus $I_4^3)))$ $(B_4^2 + I_4^3 (B_4^2 + (K_4$ 7$)I_4^2))$
$(B_4^3$ if $(B_4^2 \geq (B_4^2 \times I_4^1 I_4^2) I_4^3)$ $(B_4^1$ entier $(B_4^2/I_4^1 I_4^3))$
$(K_4$ -12$))$.

It is only in the representation of expressions that we are compelled to employ indeterminates. In representing larger constructs, such as statements, which contain expressions as syntactic units, the expression representations will always be used as parts of abstracts of the above form, eliminating all indeterminates.

3.5 Assignments

Before discussing any particular type of statement, we will first indicate the general idea behind our SK representations. Consider a given statement S of a program. Let $(v_1,...,v_n)$ 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 sometimes 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 $v_1,...,v_n$. With this interpretation in mind, the effect of the statement S 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

$$(\sigma(\phi))(v_1,...,v_n) = \phi'(v_1,...,v_n) \qquad \text{(i)}$$

which accomplishes the above transformation.

Using the Schönfinkel interpretation of functions (Section 2.2), the above relation may also be written as

$$\sigma(\phi,v_1,...,v_n) = \phi'(v_1,...,v_n) \ . \qquad \text{(ii)}$$

Now suppose we can somehow express the right-hand-side of (ii) in terms of the function ϕ, the variables $v_1,...,v_n$,

and possibly some constants. Then we may take (ii) to be the functional equation defining σ, with $v_1,...,v_n$, and even ϕ, as formal arguments. (Note that while the domains of the arguments $v_1,...,v_n$ are the values of the corresponding program variables, the domain of ϕ consists of the program remainders considered as functions.) Now by interpreting (ii) in combinatory terms, and by abstracting the ob form representing its right-hand-side with respect to the indeterminates $\phi,v_1,...,v_n$, we may obtain a definition of σ as an ob.

We remark that if the statement S is modelled as above by the ob σ, then the execution of S is modelled by the reduction of

$$\sigma\ \underline{\phi}\ \underline{v_1}\ \ldots\ \underline{v_n}\ ,$$

in which the symbols $\underline{v_1},...,\ \underline{v_n}$ denote the representations of the values of the corresponding variables immediately prior to the execution of S, and $\underline{\phi}$ denotes the representation of the program remainder of S.

A key step in representing a programming statement is to define a suitable equation of the form (i) or (ii) for it. (The choice of ϕ' is, of course, based on our intuitive understanding of the effect of the statement.) For the sake of motivation, we will include the details of this step in describing the first few of our representations.

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

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

In SK notation, this amounts to

$$\sigma\phi v_1...v_n \rightarrow \phi v_1...v_{i-1}\ \{e\}_{(v_1,...,v_n)} v_{i+1}...v_n) .$$

Accordingly, we adopt the following SK representation of assignment statements:

(5-1) Representation Rule.

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

(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.)

It has been mentioned earlier that only the variables

contained in the environment of an expression can occur in the expression. Thus, no indeterminates other than $v_1,...,v_n$ can occur in the ob form {e} in the above representation rule. It follows from the abstractions specified in that rule that the SK representation of an assignment statement is an ob, not an ob form containing occurrences of indeterminates.

Note that the multiple assignments of ALGOL 60 [27] and the collateral (parallel) assignments of ALGOL 68 138] present no special problem. Omitting the formal rules for their representation, we simply illustrate their treatment in the following example.

Example. Presented side by side below are some assignment statements and their SK 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;	$A\phi xy{:}\phi 2y$ (or, $A\phi x{:}\phi 2$)
y:=x+l9;	$A\phi xy{:}\ \phi x(+xl9)$
begin integer z,y;	
y:=x-5;	$A\phi zy^1xy^0{:}\phi z(-x5)xy^0$
x:=y:=z+(x+y) ;	$A\phi zy^1xy^0{:}\phi z(+z(+xy^1))(+z(+x\ y^1))y^0$

```
        (y:=z, z:=y);          Aϕzy¹xy⁰:ϕy¹zxy⁰
        ...
    end
end
```

In order to express the representation of assignments explicitly as obs rather than abstracts, we introduce some new obs. Although their names are suggestive of the programming actions they represent, we emphasize that they are just ordinary obs, with no imperative notions attached to them.

(5-2) Definition.

$$\underline{eval}_m \equiv C(B_n^2 I \circ \underline{rotr}_{n+1}) \ , \qquad n \geq 1.$$

$$\underline{store}_i \equiv \underline{swap}_{i+2}^2 K \ , \qquad i \geq 1.$$

$$\underline{assign}_n^i \equiv CB\underline{store}_i \circ \underline{eval}_n \ , \qquad n \geq i \geq 1.$$

The following reduction properties are easy to derive.

(5-3) Rule.

$\underline{eval}_n \ f\phi x_1 \ldots x_n \rightarrow \phi(fx_1 \ldots x_n)x_1 \ldots x_n$.

$\underline{store}_i \ \phi e x_1 \ldots x_i \rightarrow \phi x_1 \ldots x_{i-1} e$.

$\underline{assign}_n^i \ f\phi \ x_1 \ldots x_n \rightarrow \phi x_1 \ldots x_{i-1}(fx_1 \ldots x_n)x_{i+1} \ldots x_n$.

Hence, we have the following alternative to (5-1).

(5-4) Representation Rule.

$$\{v_i := e\}_{(v_1,\ldots,v_n)} \equiv \underline{\text{assign}}_n^i \ (Av_1\ldots v_n:\{e\}) \ .$$

Example. Consider the statement y := x+19 in the environment (x,y). In this case, we have

$$\{e\} \equiv \{x+19\} \equiv +x\underline{19} \ ,$$

$$Axy:\{e\} \equiv B_2^2+I_2^1 \ (K_2\underline{19}) \ , \quad \text{by (4-1)},$$

$$\{y := x+19\}_{(x,y)} \equiv \underline{\text{assign}}_2^2 \ (B_2^2 + I_2^1 \ (K_2\underline{19})) \ .$$

To conclude this section, we point out the fact that we have eliminated the concepts of memory and address from our model, and have reduced the concept of assignment to that of substitution or function evaluation.

3.6 Compound Statements

Consider the compound statement S begin S1;S2 end appearing in the environment $(v_1,...,v_n)$. 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 $v_1,...,v_n$, and we are interested in the representation σ of S with the functional transformation property

$$(\sigma(\phi))(v_1,...,v_n) = \phi'(v_1,...,v_n) \ .$$

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

$$(\sigma_1(\phi^*))(v_1,...,v_n) = \phi'(v_1,...,v_n) \ ,$$
$$(\sigma_2(\phi))(v_1,...,v_n) = \phi^*(v_1,...,v_n) \ .$$

The above functional conditions can be expressed in SK as the following reduction relations:

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

These relations will certainly hold if we choose

$\phi^* \equiv \sigma_2\phi$, $\phi' \equiv \sigma_1(\sigma_2\phi)$, and hence, $\sigma \equiv A\phi{:}\sigma_1(\sigma_2\phi)$.

The generalization to the case of an n-component compound is now obvious. So we are led to the following SK representation of compound statements:

(6-1) Representation Rule.

$$\{\underline{\text{begin}}\ S_1;S_2;\ldots;S_n\ \underline{\text{end}}\} \equiv A\phi{:}\{S_1\}(\{S_2\}(\ldots(\{S_n\}\phi)\ldots))$$

$$\text{or,} \quad \equiv [\{S_1),\{S_2\},\ldots,\{S_n\}]$$

(where the notation [...] for nests is as introduced in Definition 2.4-20).

Notice the convenient fact that in the above nest the individual statement representations appear from left to right in the same order in which the statements occur in the compound (cf. Stratchey [36]).

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;	$a \equiv A\phi xy{:}\phi\underline{2}y$

```
          y := x+3;                b ≡ Aϕxy:ϕx(+x3)
          x := y+x                 c ≡ Aϕxy:ϕ(+yx)y
     end                           d ≡ Aϕ:a(b(cϕ))
(ii) begin
          y := 5;                  e ≡ Aϕxy:ϕx5
          x := y+2                 f ≡ Aϕxy:ϕ(+y2)y
     end                           g ≡ Aϕ:e(fϕ)
```

The compound statements (i) and (ii) are intuitively equivalent. How can their equivalence be demonstrated in our model? The obs representing the statements (i) and (ii), namely, d and g, must have the same interpretation as a function of the variables ϕ, x, and y. Alternatively, d and g must perform the same reduction when applied to the same obs $\underline{\phi}$, $\underline{x}$, and $\underline{y}$; that is,

$$d\ \underline{\phi}\ \underline{x}\ \underline{y} \leftrightarrow g\ \underline{\phi}\ \underline{x}\ \underline{y}\ . \qquad (*)$$

To verify (*), we reduce both sides to the same ob.

$g\ \underline{\phi}\ \underline{x}\ \underline{y} \equiv A\phi{:}e(f\phi)\underline{\phi}\ \underline{x}\ \underline{y}$

$\rightarrow e(f\underline{\phi})\ \underline{x}\ \underline{y}$, since $\phi\ o\not\in e$ and $\phi\ o\not\in f$,

$\equiv$ (Aϕxy:ϕx$\underline{5}$)(f$\underline{\phi}$) $\underline{x}$ $\underline{y}$

$\rightarrow$ f $\underline{\phi}$ $\underline{x}$ $\underline{5}$

$\equiv$ (Aϕxy:ϕ(+y$\underline{2}$)y) $\underline{\phi}$ $\underline{x}$ $\underline{5}$

$\rightarrow$ $\underline{\phi}$ (+ $\underline{5}$ $\underline{2}$)$\underline{5}$

$\rightarrow$ $\underline{\phi}$ $\underline{7}$ $\underline{5}$.

Similarly, it is easy to see that

d $\underline{\phi}$ $\underline{x}$ $\underline{y}$ $\rightarrow$ $\underline{\phi}$ $\underline{7}$ $\underline{5}$.

In general, to show that two statements appearing in the same environment of n variables are equivalent, we need to prove that the SK representations f_1 and f_2 of those statements are mutually (n+1)-interconvertible (Definition 2.1-l1). This, however, is an unnecessarily strict condition. It is often the case that in all the intended executions of a program (that is, with the input data satisfying the program specifications), the values of program variables range over certain restricted domains only. In such cases, the equivalence of two statements in the environment of n variables may be established by proving that, for $\underline{x_1},...,\underline{x_n}$ representing not all arbitrary obs but only the possible values corresponding to the variables $x_1,...,x_n$ of the environment, and for all obs $\underline{\phi}$, it is the case that

f_1 $\underline{\phi}$ $\underline{x_1}...\underline{x_n}$ $\leftrightarrow$ f_2 $\underline{\phi}$ $\underline{x_1}...\underline{x_n}$.

3.7 Blocks

Next, let us consider a block S whose head declares the variables $u_1,...,u_m$ and initializes these to the values[3] $c_1,...,c_m$, and whose body consists of the statements $S_1,...,S_p$, 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 $u_1,...,u_m$ (initialized at $c_1,...,c_m$).
2) Execution of the compound begin $S_1;...;S_p$ end.
3) Deletion of the, variables $u_1,...,u_m$ from the environment.

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

$$(\alpha(\phi))(v_1,...,v_n) = \phi(c_1,...,c_m,v_1,...,v_n)$$

$$(\beta(\phi))(u_1,...,u_m,v_1,...,v_n) = (\sigma_1(\sigma_2(...(\sigma_p(\phi))...)))(u_1,...,u_m,v_1,...,v_n)$$

$$(\gamma(\phi))(u_1,...,u_m,v_1,...,v_n) = \phi(v_1,...,v_n)$$

$$(\sigma(\phi))(v_1,...,v_n) = (\alpha(\beta(\gamma(\phi))))(v_1,...,v_n)$$

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

$$\sigma \equiv A\phi v_1...v_n:\ \sigma_1(\sigma_2(...(\sigma_p(Au_1...u_m{:}\phi))...))c_1...c_m v_1...v_n\ .$$

---

3 We assume that the expressions $c_1,...,c_m$ do not contain the variables $u_1,...,u_m$; they may, however, contain the variables in the environment of S.

Consequently, we choose the following representation of blocks.

(7-1) Representation Rule.

$$\{\underline{\text{begin}}\ \langle\text{type}\rangle\ u_1 := c_1;\ldots;\langle\text{type}\rangle\ u_m := c_m;\ S_1;\ldots;S_p\ \underline{\text{end}}\}_{(v_1,\ldots,v_n)}$$

$$\equiv A\phi v_1\ldots v_n{:}\{S_1\}_F(\{S_2\}_F(\ldots(\{S_p\}_F(Au_1\ldots u_m{:}\phi))\ldots))\ \{c_1\}_E\ldots(c_m\}_E v_1\ldots v_n\ ,$$

where $E \equiv (v_1,\ldots,v_n)$ and $F \equiv (u_1,\ldots,u_m,v_1,\ldots,v_n)$. (We assume that the expressions c_i include any needed type-conversions.)

Using the notation of nests and tuples (Definitions 2.4-20, 2.4-24), an explicit combinatory description of the above abstract is

$$[\langle\{c_1\}_E,\ldots,(c_m\}_E\rangle\ ,\ [\{S_1\}_F,\ldots,\{S_p\}_F\ ,K_m]]$$

In the case that the variables are left uninitialized in the block-head -- as is normal in ALGOL 60 -- any arbitrary ob can be used for $\{c_i\}$ in the above representation. One might wish to use for this purpose an ob which would play the role of the everywhere undefined function. This function is modelled, for example, by the ob Ω (Definition 2.4-1) having the property

Ω 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 (7-1), 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.[4]

Being the representation of statements, the components $\{S_i\}_F$, $1 \leq i \leq p$, of the right-hand-side of (7-1) do not contain any indeterminates. But being the representations of expressions in the environment $(v_1,...,v_n)$, $\{c_i\}$, $1 \leq i \leq m$, may possibly contain $v_1,...,v_n$. If the variables declared in the block head are not initialized, then, by recourse to a suitable abstraction algorithm (Theorem 2.2-4(3)), the indeterminates $v_1,...,v_n$ can be dropped from the right-hand-side of (7-1). We thus obtain the following simplified representation:

(7-2) Representation Rule.

$$\{\underline{\text{begin}}\ \langle\text{type}\rangle u_1;...;\langle\text{type}\rangle u_m;S_1;...;S_p\ \underline{\text{end}}\}_{(v_1,...,v_n)}$$

$$\equiv A\phi v_1...v_n:\{S_1\}_F(\{S_2\}_F(...(\{S_p\}_F(Au_1...u_m:\phi))...))\underbrace{\Omega\Omega...\Omega}_{m \text{ times}}$$

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

---

4 This situation may be avoided by using the ob

$$\Omega' \equiv D(B(S(BSC))(BC(C(KD)))K$$

instead of Ω. It is easy to verify both that Ω' is normal and that, for all a, $\Omega' a \rightarrow \Omega'$.

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 (7-2), except that the constants are used instead of the corresponding Ω.

Example. The environment of the following block is assumed to be (w). The individual statements and their representations are given side by side below. The representations have been given identifying names for reference purposes.

Statements	Representations
begin integer x:=5,y;	
y := x-7;	a ≡ Aϕxyw:ϕx(-x7)w
begin integer z;	
z := 3+y;	b ≡ Aϕzxyw:ϕ(+3y)xyw
x := z × x	c ≡ Aϕzxyw:ϕz(×zx)yw
end	d ≡ Aϕ:b(c(Az:ϕ))Ω
end	e ≡ Aϕ:a(d(Axy:ϕ))5Ω

Explicit combinatory definitions of the above obs, in accordance with our previous representation rules, are as follows:

$$a \equiv \underline{assign}_3^2\ (B_3^2 - I_3^1\ (K_3 7))\ ,$$

$$b \equiv \underline{assign}_4^1\ (B_4^2 + (K_4 3)\, I_4^3)\ ,$$

$$c \equiv \underline{assign}_4^2\ (B_4^2 \times I_4^1\ I_4^2)\ ,$$

$d \equiv [\langle \Omega \rangle, [b, c, K_1]]$,

$e \equiv [\langle \underline{5}, \Omega \rangle, [a, d, K_2]]$.

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 $a_1,...,a_n$ will be represented by the tuple

$$< \{a_1\},...,\{a_n\}> .$$

(The empty file is represented by the null tuple $< > \equiv I$.) For given ob forms u and v, we will abbreviate the ob form insert u v by u,v ; also we will denote u,v,w by u,v,w , and so on. By Rule 2.4-25 (2), we have

$$<x_1,...,x_n>,y \equiv <x_1,...,x_n,y> \quad ,$$

$$I,x_1,...,x_n \equiv <x_1,...,x_n> \quad ,$$

so that "," 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 $v_1,...,v_n$ of a program, and let σ be the ob representing S. In our discussion so far, σ has been defined as an abstract of the form

$$A\phi v_1...v_n:... \qquad (*)$$

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

$$\sigma\ \phi\ v_1...v_n \quad ,$$

in which the underlined 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 ob

(8-1) $\sigma\ \phi\ \underline{v_1} \ldots \underline{v_n}\ \underline{w}\ \underline{u_1}\ \underline{u_2} \ldots \underline{u_m}$,

with w denoting the output file and u_i the i^{th} 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

$$A\ \phi\ v_1 \ldots v_n\ o\ i_1 \ldots i_m : \ldots ,$$

where $o, i_1, \ldots, i_m$ 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 (*) by choosing abstracts properly. Furthermore, in the case of input-output statements, the following choice of SK representations is obvious:

(8-2) Representation Rule.

$$\underline{\{\text{read}}\ v_j\}_{(v_1,\ldots,v_n)} \equiv A\phi v_1 \ldots v_n o\colon \phi v_1 \ldots v_{j-1} i v_{j+1} \ldots v_n o ,$$

$$\underline{\{\text{write}}\ e\}_{(v_1,\ldots,v_n)} \equiv A\phi v_1 \ldots v_n o\colon \phi v_1 \ldots v_n\ \underline{o, \{e\}} ,$$

where e is some expression to be output.

In order to provide explicit ob representation of input-output statements, we introduce the following obs

(8-3) Definition.

(1) $\underline{\text{read}}_n^j \equiv \underline{\text{swap}}_{n+3}^{j+1} K_{(n+1)}$, $1 \leq j \leq n$,

(2) $\underline{\text{write}}_n \equiv [B_{n+1}^2 SK_{(n)}, K, (CC)_{(n)}]$.

(8-4) Rule.

(1) $\underline{\text{read}}_n^j \ \phi x_1 \ldots x_n \circ i \rightarrow \phi x_1 \ldots x_{j-1} \ i \ x_{j+1} \ldots x_n \circ$,

(2) $\underline{\text{write}}_n \ f\phi x_1 \ldots x_n \circ \rightarrow \phi x_1 \ldots x_n \ \underline{\circ, f x_1 \ldots x_n}$.

Proof of (2).

$\underline{\text{write}}_n \ f\phi x_1 \ldots x_n \circ$

$\equiv [B_{n+1}^2 SK_{(n)}, K, (CC)_{(n)}] f\phi x_1 \ldots x_n \circ$

$\rightarrow B_{n+1}^2 SK_{(n)} (K((CC)_{(n)} f)) \phi x_1 \ldots x_n \circ$, by Rule 2.4-21,

$\rightarrow S(K_{(n)} \phi x_1 \ldots x_n)(K((CC)_{(n)} f) \phi x_1 \ldots x_n) \circ$

$\rightarrow K_{(n)} \phi x_1 \ldots x_n \circ (K((CC)_{(n)} f) \phi x_1 \ldots x_n \circ)$

$\rightarrow \phi x_1 \ldots x_n ((CC)_{(n)} f x_1 \ldots x_n \circ)$, by Rule 2.4-6,

$\rightarrow \phi x_1 \ldots x_n (C \circ (f x_1 \ldots x_n) \circ)$

$\rightarrow \phi x_1 \ldots x_n (C \circ (f x_1 \ldots x_n)) \equiv \phi x_1 \ldots x_n \ \underline{\circ, f x_1 \ldots x_n}$.

In view of the above rules, we propose the following alternative to (8-2):

(8-5) Representation Rule.

$\underline{\{\text{read}\ v_j\}}_{(v_1, \ldots, v_n)} \equiv \underline{\text{read}}_n^j$,

$\underline{\{\text{write}\ e\}}_{(v_1, \ldots, v_n)} \equiv \underline{\text{write}}_n \ (A v_1 \ldots v_n : \{e\})$.

3.9 Programs

Let the input file initially presented to a given program consist of items $i_1,...,i_p$, and let $o_1,...,o_q$ 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

$$\{\text{program }\}\{i_1\}...\{i_p\} \rightarrow \langle \{o_1\},...,\{o_q\} \rangle . \qquad \text{(i)}$$

Now the execution of a particular statement of the program is modelled by the reduction of an ob given by (8-1) in the previous section. 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 (8-1), we obtain the following conditions:

$\sigma \equiv$ {program block }

n = 0 as the environment is null ,

$\underline{w} \equiv$ I , as the output file may be considered empty at the start of the program ,

m = p , and $u_j = i_j$, $1 \leq j \leq p$.

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

$$\{\text{program block }\}\ I\ I\ \{i_1\}...\{i_p\} . \qquad \text{(ii)}$$

Next, consider (8-1) again -- but this time for the case when the entire program has been executed. Now we have:

$\sigma \equiv$ I, the null program segment,

$\underline{n} = 0$, as the environment is null,

$\underline{w} = < \{o_1\},\ldots,\{o_q\} >$, representing the final output file,

$m = 0$, assuming the program exhausts the input file,

$\phi \equiv$ I.

Thus, (8-1) in this case becomes the ob

$$I\ I < \{o_1\},\ldots,\{q\} > ,$$

which reduces to

$$< \{o_1\},\ldots,\{q\} > . \qquad \text{(iii)}$$

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

$$\{\text{program block}\}I\ I\ \{i_1\}\ldots\{i_p\} \rightarrow < \{o_1\},\ldots,\{o_q\} >. \qquad \text{(iv)}$$

Comparing (i) and (iv), we obtain:

(9-1) Representation Rule.

{program} $\equiv$ (program block) I I .

(9-2) Remarks.

(1) From (iv) and (9-1) it follows that

$$< \{i_1\},\ldots,\{i_p\} > \{\text{program}\} \rightarrow < \{o_1\},\ldots,\{o_q\} > ,$$

that is,

<input file> [program} → <output file> .

(2) In the ob representing a program, the component ({program block} I) will be found to be of interest by itself; we will refer to it as the routine of the program.

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 ≡ Aϕabcoi:ϕibco
read c;	g ≡ Aϕabcoi:ϕabio
b := a+c;	h ≡ Aϕabc: ϕa(+ac)c
write b;	j ≡ Aϕabco:ϕabco,b
b := b-2xc;	k ≡ Aϕabc:ϕa(-b(×2c))c
write b	j
end	m ≡ Aϕ:f(g(h(j(k(j(Aabc:ϕ)))))ΩΩΩ

Since the ob in represents the program block, the representation of the whole program is p ≡ mII. Now it can be verified that, for all integers a and b,

p a b → <a+b,a-b> .

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 SK reduction.

p 5 3 ≡ m I I 5 3 → f(g(h(j(k(j(Aabc:I))))))Ω Ω Ω I 5 3

→ g(h(j(k(j(Aabc:I))))) 5 Ω Ω I 3

→ h(j(k(j(Aabc:I)))) 5 Ω 3 I

→ j(k(j(Aabc:I))) 5 (+ 5 3) 3 I

→ j(k(j(Aabc:I))) 5 8 3 I

→ k(j(Aabc:I)) 5 8 3 I,8 ≡ k(j(Aabc:I)) 5 8 3 <8>

→ j(Aabc:I) 5 (-8 (× 2 3)) 3 <8> → j(Aabc:I) 5 2 3 <8>

→ (Aabc:I) 5 2 3 <8>, 2 → (Aabc:I) 5 2 3 <8,2>

→ I <8,2> → <8,2> .

3.10 Conditional Statements

Recall that the SK representation of a Boolean expression b has the property

{b} p q → p , if b has the value true,

→ q , if b has the value false.

In view of the above property, we choose the representation of a two-branch conditional statement as follows:

(10-1) Representation Rule.

{if b then S_1 else S_2 }$(v_1,...,v_n)$

≡ A$\phi v_1...v_n$: {b}{S_1}{S_2}$\phi v_1...v_n$.

For the purpose of representation, a one-branch conditional statement (an if statement, in ALGOL 60 terminology) may be viewed as a two-branch conditional with a dummy or "do-nothing" statement for the second branch. When appearing in the environment $(v_1,...,v_n)$, the "do-nothing" statement can obviously be represented by

$$A\phi v_1...v_n:\ \phi v_1...v_n \ ,$$

that is, I. Substituting the "do-nothing" statement for S_2 in (10-1), we obtain the

(10-2) Representation Rule.

$$\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\}_{(v_1,...,v_n)} \equiv A\phi v_1...v_n:\{b\}\{S_1\}I\phi v_1...v_n \ .$$

In order to describe the above representations explicitly as obs, we first introduce a new ob sequence and its associated reduction rule.

(10-3) Definition. $\underline{\text{cond}}_n \equiv \beta\circ(B_n^3 I)$.

(10-4) Rule.

$$\underline{\text{cond}}_n\ bs_1s_2\phi x_1...x_n \rightarrow bx_1...x_n(s_1\phi x_1...x_n)(s_2\phi x_1...x_n) \ .$$

It is easy to see that if b is a Boolean expression, then for all ob forms p, q, and r, the following interconvertibility relation holds:

$$\{b\}pqr \leftrightarrow \{b\}(pr)(qr) \ .$$

Applying this relation to (10-1), we may

obtain the following alternatives to (10-1) and (10-2).

(10-5) Representation Rule.

(1) $\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \}_{(v_1,\ldots,v_n)}$

$\equiv A\phi v_1 \ldots v_n$: $\{b\}(\{S_1\}\phi v_1 \ldots v_n)(\{S_2\}\phi v_1 \ldots v_n)$

$\equiv \underline{\text{cond}}_n\ (Av_1 \ldots v_n$: $\{b\}\{S_1\}\{S_2\}$.

(2) $\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\}_{(v_1,\ldots,v_n)}$

$\equiv A\phi v_1 \ldots v_n$: $\{b\}(\{S_1\}\phi v_1 \ldots v_n)(\phi v_1 \ldots v_n)$

$\equiv \underline{\text{cond}}_n\ (Av_1 \ldots v_n$: $\{b\}\{S_1\}I$.

3.11 Arrays

Arrays can be interpreted as tuples and combinations of tuples. An array of a single dimension is represented by a tuple of the representations of the individual array elements, taken in the order of the lowest to the highest subscript. An array of dimension n+1 is represented by a tuple whose elements are the representations of the n-dimensional subarrays (or slices, in the ALGOL 68 terminology [38]) obtained by fixing the first subscript in turn from the lowest to the highest possible value. For example, the array A [1:2, 1:3] is represented by

$< < \{A_{11}\},\{A_{12}\},\{A_{13}\}> , < \{A_{21}\},\{A_{22}\},\{A_{23}\}> >$,

where $(A_{ij}\}$ is the representation of the array element A_{ij}. As in the case of simple variables, an array identifier can

itself be used for the indeterminate assigned to the array variable.

With the above interpretation of arrays, we next describe the representation of subscripted variables in expressions, assignments to subscripted variables, and array declarations. In this description, we assume for simplicity that all arrays have the lowest subscript bound of 1. To obtain the correct representation in the case of an array one of whose subscript bounds, ℓ, is different from 1, one simply needs to first increment the corresponding bound and subscript expressions throughout the program by $1-\ell$.

1) Subscripted variable as an operand in an expression

The representation in this case is just the corresponding element of the tuple representing the array. Thus, given the declaration <type> array v [1:n], we have, on the basis of Rule 2.4-25 (3),

$$\{v\ [i]\} \equiv v\ \underline{elem}_n^i \quad .$$

This representation is inadequate, since, in general, n and i are given as expressions rather than constants, and their values may not be known at the time of SK translation of the program. However, we have seen (cf. end of Section 2.4) that there exists an ob elem such that for all obs a and b if a → i and b → n, where i, n represent natural numbers i, n such that

$1 \leq i \leq n$,

$$\underline{\text{elem}}\ a\ b \rightarrow \underline{\text{elem}}^{i}_{n} \ .$$

Hence, given the array declaration v [1:e], we specify

$$\{v\ [f]\} \equiv v(\underline{\text{elem}}\ [f\}\{e\}) \ .$$

More generally, for the array v $[1:e_1,...,1:e_m]$, we have

$$\{v\ [f_1,...,fm]\} \equiv v(\underline{\text{elem}}\ \{f_1\}\{e_1\})...(\underline{\text{elem}}\ \{f_m\}\{e_m\}).$$

2) Assignments to subscripted variables

In this case, the representation consists in replacing the designated element of the tuple representing the array with the representation of the new value. Let us first consider the arrays of a single dimension only. We have already seen (Section 2.4, end) that there exists an ob replace such that, for all natural number representations i, m such that $1 \leq i \leq m$ and for all ob forms $a_1,...,a_m,b$,

$$\langle a_1,...,a_m\rangle(\underline{\text{replace}}\ \underline{i}\ \underline{m}\ b) \rightarrow \langle a_1,...,a_{i-1},b,a_{i+1},...,a_m\rangle \ .$$

Hence, given the declaration <type> array $v_j[1: e]$, we have

$$\{v_j[f] := g\}_{(v_1,...,v_n)}$$

$$\equiv A\phi v_1...v_n: \phi v_1...v_{j-1}(v_j(\underline{\text{replace}}\ \{f\}\{e\}\{g\}))v_{j+1}...v_n \ .$$

The representation in the case of a higher dimensional array involves the replacement of all slices of the array

that are affected by the assignment. It is easy to see that, when the array v_j is declared to be of bounds $[l{:}e_1,\ldots,l{:}e_m]$, the following is a suitable representation:

$$\{v_j\ (f_1,\ldots,f_m] := g\ \}$$

$$\equiv A\phi v_1\ldots v_n{:}\ \phi v_1\ldots v_{j-1} h v_{j+1}\ldots v_n\ ,$$

where

$$h \equiv v_j(\underline{\text{replace}}\ \{f_1\}\{e_1\}(v_j((\underline{\text{elem}}\ \{f_1\}\{e_1\})\ (\underline{\text{replace}}\ \{f_2\}\{e_2\}(v_j((\underline{\text{elem}}\ \{f_1\}\{e_1\})(\underline{\text{elem}}\ \{f_2\}\{e_2\})\ (\underline{\text{replace}}\ \{f_3\}\{e_3\}(\ \ldots\ v_j((\underline{\text{elem}}\ \{f_1\}\{e_1\})\ldots(\underline{\text{elem}}\ \{f_{m-1}\}\{e_{m-1}\})\ \underline{\text{replace}}\ \{f_m\}\{e_m\}\{g\}))\ldots\ ))\ ))\ ))\ .$$

3) Array declaration

We treat array declarations in the same way as the declaration of simple variables with respect to environments and the representation of initialized variable values. However, there is the following exception: if an array is not initialized at the time of declaration, the block representation is obtained by assuming all the array elements to be Ω. Thus, for an array with the bound pairs [1:2,1:3], the initial value is represented by $\langle\langle\Omega,\Omega,\Omega\rangle,\langle\Omega,\Omega,\Omega\rangle\rangle$.

Since, in general, array bounds may be specified by expressions, we need to create tuples of arbitrary dimensions and sizes in which all elements are Ω. This will be possible by means of the ob tupinit having the property

$$\underline{\text{tupinit}}\ \underline{1}\ \underline{m} \rightarrow \underbrace{\langle \Omega, \Omega, \ldots, \Omega \rangle}_{\uparrow\ m\ \text{elements}} ,$$

$$\underline{\text{tupinit}}\ n{+}1\ m_1 \ldots m_{n+1} \rightarrow \underbrace{\langle a, a, \ldots, a \rangle}_{\uparrow\ m_1\ \text{elements}} , \qquad (*)$$

where $a \equiv \underline{\text{tupinit}}\ n\ m_2 \ldots m_{n+1}$.

In order to define tupinit, we need the following definition, making use of some obs of Section 2.4:

(11-1) Definition. $\underline{\text{maketup}} \equiv \beta(TW)\underline{\text{pred}}\ \underline{\text{tup}}$.

(11-2) Rule. $\underline{\text{maketup}}\ \underline{n}\ x \rightarrow \underbrace{\langle x, .., x \rangle}_{\uparrow\ n\ \text{times}}$, for integer $n > 0$.

(11-3) Definition.

$$\underline{\text{tupinit}}_0 \equiv \Omega ,$$

$$\underline{\text{tupinit}}_{n+1}\ a_1 a_2 \ldots a_{n+1} \equiv \underline{\text{maketup}}\ a_1\ (\underline{\text{tupinit}}_n\ a_2 \ldots a_{n+1}) .$$

Now from Table 2.4-19 we can define an ob such that for all natural numbers $n \geq 0$,

$$\underline{\text{tupinit}}\ \underline{n} \rightarrow \underline{\text{tupinit}}_n .$$

It is quite straightforward to check that with this choice of tupinit, the relations (*) are indeed satisfied.

Example. The array representations discussed above are illustrated on the following block which is assumed to occur in the environment (n).

Statements	Representations
begin integer array p [1:n],	
q [1:2,1:3]; integer r;	
r := q [n,n+1];	a ≡ Aϕpqrn:ϕpq(q(e1em n 2) (elem(+ n 1)3))n
p [r]:= r+3	b ≡ Aϕpqrn:ϕ(p(rep1ace rn(+ r 3)))qrn
end	σ ≡ Aϕn:a(b(Apqr:ϕ)) (tupinit 1 n)
	<<Ω,Ω,Ω>,<Ω,Ω,Ω>>Ωn

CHAPTER 4

ITERATION AND JUMP STATEMENTS

4.1 Recursive Specification of Obs

In dealing with program loops, we shall need obs having the property that they are equiform to one or more components of certain obs to which they reduce. That is, these obs are to possess given reduction properties of the form

$$F \rightarrow \ldots F \ldots F \ldots \quad . \qquad \text{(i)}$$

We refer to such obs as recursively specified obs, and describe two approaches to define these.

One approach to define recursively specified obs is to admit them as primitive obs, taking the respective properties required of them as the reduction rules associated with them. Since the reductions rules so added are recursive, in the sense that they reduce an ob in terms of itself, it is not at all obvious that the Church-Rosser property would hold in the extended calculus. But it follows from the work of Rosen [32] that the Church-Rosser property is indeed preserved by such extensions, and, consequently, most other properties of reduction, such as the uniqueness of normal forms and the correctness of the standard-order reduction algorithm, also continue to be valid.

Another approach to defining the obs specified by the properties of the form (i) is to look for solutions of (i), treating such formulas as reduction relations involving an unknown. Now, in general, (i) may be satisfied by more than one solution, so that we may have the choice of different explicit definitions for the same ob. There is, however, no reason to expect that these different definitions of an ob are compatible to each other or to the definition of the ob as a new primitive -- compatible in the sense that all reduction sequences, which start with an ob having a recursively specified ob as a component and which use the different definitions of the recursively specified ob, yield the same normal form (if any). In fact, incompatibilities do occur, as the following trivial example indicates: Let F be an ob specified by

$$F \rightarrow F,$$

and let it be required to reduce the ob $G \equiv SF$. By the definition of reduction (Definition 2.1-4), the given property of F is satisfied by every ob. In particular, with $F \equiv S$ and $F \equiv K$ chosen as two possible definitions of F, the same ob G may be reduced to SS in one case and to SK in another! Moreover, both these results are in conflict with the one obtained by taking F as a primitive ob and $F \rightarrow F$ as the associated reduction rule. For, then, G does not even have a

normal form!

Thus, not all solutions of (i) are acceptable for a definition of the ob F. Following Morris [26], to characterize those solutions of (i) for which the resulting definitions of the ob F are compatible with the definitions of the first approach (of taking (i) as a reduction rule), we may proceed thus. Let us introduce a partial order on obs as follows: For obs a and b we say that a is extended by b, in symbols, $a \leq b$, if, for all obs 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 obs b, where Ω is as given in Definition 2.4-1. Now, the particular solutions of (i) that we are interested in have the property that they are extended by all solutions of (i). In other words, for an explicit definition of the ob F specified by (i), we can take a minimal solution of (i) (with respect to $\leq$).

For example, consider the relation $F \rightarrow F$ again. Since this relation is satisfied by all obs, Ω is a minimal solution for it. Thus, $F \equiv \Omega$ may be taken as a definition of the ob specified by $F \rightarrow F$. Under this definition of F, the ob $G \equiv SF$ does not possess a normal form. This agrees with the result obtained by taking F as a primitive ob, with $F \rightarrow F$ as its associated reduction rule.

To obtain an explicit definition of the ob F given by (i), we may proceed as follows: Let x be an indeterminate,

and let

$$H \equiv Ax: \ldots x \ldots x \ldots$$

be an abstract with respect to x of the ob form obtained from the right-hand-side of (i) by replacing the components equiform to F by x. Now

$$YH \rightarrow H(YH) , \qquad \text{by Rule 2.4-1(18),}$$

$$\rightarrow \ldots(YH)\ldots(YH)\ldots , \qquad \text{by Theorem 2.2-4.}$$

Hence, YH is seen to be a solution of (i). It has been shown by Morris [26] that this solution is also minimal. Consequently,

$$F \equiv YH \equiv Y(Ax:\ldots x \ldots x \ldots) \qquad \text{(ii)}$$

is an explicit definition of the ob specified by (i).

In general, (i) has infinitely many, mutually noninterconvertible, 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 for the explicit solution of (i), such as YH in (ii), is rather arbitrary. To leave this choice unspecified, while emphasizing the minimality of the chosen solution, one may employ the μ-notation of deBakker [3]. In this notation, the minimal solution of (i) is designated by the μ-expression

$$\mu x:\ldots x \ldots x \ldots ,$$

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

Since the formula (i) has the appearance of a relation, which may not necessarily suggest that it is intended to define anything, we shall use the notation

$$F \equiv \ldots F \ldots F \ldots$$

to indicate that the ob F is being defined as specified by (i).

The above treatment of recursively specified obs can also be generalized to include the simultaneous recursive specification of several obs, such as

$$\begin{aligned} F_1 &\rightarrow H_1 F_1 \ldots F_n \ , \\ &\ldots \\ F_n &\rightarrow H_n F_1 \ldots F_n \ , \end{aligned} \qquad \text{(iii)}$$

where $F_1,\ldots,F_n$ do not occur as components in $H_1,\ldots,H_n$. The definitions of $F_1,\ldots,F_n$ may be obtained as follows:

1) The F's specified by (iii) are considered primitive obs whose associated reduction rules are just the formulas (iii).
2) The F's specified by (iii) may be explicitly defined as the minimal solutions of the system of

reduction relations (iii).

An explicit solution may be obtained as follows: Consider

$$\begin{aligned} & <H_1F_1...F_n,...,H_1F_1...F_n> \\ \leftarrow\ & \underline{\text{funtup}}^n_n\ H_1...H_n\ F_1...F_n, \qquad \text{by Rule 2.4-25,} \\ \leftarrow\ & <F_1,...,F_n>(\underline{\text{funtup}}^n_n\ H_1...H_n) \\ \leftarrow\ & <\underline{\text{funtup}}^n_n\ H_1...H_n><F_1,...,F_n> \ . \end{aligned}$$

Hence, we may take, for $1 \leq i \leq n$,

$$F_i \equiv Y\ <\underline{\text{funtup}}^n_n\ H_1...H_n>\ \underline{\text{elem}}^i_n \ .$$

As before, we shall employ the notation

$$\begin{aligned} F_1 &\equiv H_1F_1...F_n \ , \\ &... \\ F_n &\equiv H_nF_1...F_n \ , \end{aligned}$$

to indicate the definition of the obs $F_1,...,F_n$ by means of the formula (iii).

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 ... repeat... . To represent the latter, consider the statement while b repeat S appearing in the environment $(v_1,...,v_n)$. Calling this statement by the name T, we may (recursively!) describe it, for the purpose of SK representation, as

if b then begin S;T end .

Now the formulas for the representation of compound and conditional statements, (3.6-1) and (3.10-5(1)) respectively, are applicable to the above statement, so that its representation {T} is, recursively, the ob

$$A\phi v_1...v_n\colon \{b\}((A\phi\colon \{S\}(\{T\}\phi))(\phi v_1...v_n)\ (\phi v_1...v_n)$$

$$\leftrightarrow_{n+1} A\phi v_1...v_n\colon \{b\}(\{S\}(\{T\}\phi))\phi v_1...v_n\ .$$

Thus, we adopt the

(2-1) Representation Rule.

$$\{\text{while b repeat S}\}_{(v_1,...,v_n)}$$

$$\equiv \mu x\colon A\phi v_1...v_n\colon \{b\}(\{S\}(x\phi))\phi v_1...v_n\ .$$

Alternative definitions of the same ob, call it X, are

$$X \equiv A\phi v_1...v_n\colon \{b\}(\{S\}(X\phi))\phi v_1...v_n\ ,$$
$$X \equiv Y(Ax\phi v_1...v_n\colon \{b\}(\{S\}(x\phi))\phi v_1...v_n)\ ,$$
$$X \equiv \text{cond}_n\ \{b\}\ [\{S\},X]\ I\ .$$

Example. At this point, we illustrate the SK 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. Given below are the individual statement representations, shown on the same line as the statements (or on the last line for multiple line statements), and have been designated names for reference purposes.

```
begin integer x,y;
  read x;                          a ≡ Aϕxyoi:ϕiyo
  y := 0;                          b ≡ Aϕxy:ϕx0

  begin integer z;
    z := 0;                        c ≡ Aϕzxy:ϕ0xy
  while z < x repeat
      begin
        y := 1+y+2×z;              d ≡ Aϕzxy:ϕzx(+(+1y)(×2z))
        z := z+1                   e ≡ Aϕzxy:ϕ(+z1)xy
      end                          f ≡ Aϕ:d(eϕ)
  end while                        g ≡ Aϕzxy:(<zx)(f(gϕ))ϕzxy
                                   h ≡ Aϕ:c(g(Az:ϕ))Ω
  write y                          j ≡ Aϕxyo: ϕxy o,y
end                                k ≡ Aϕ: a(b(h(j(Axy:ϕ))))ΩΩ
                                   {program ≡ P ≡ kII
```

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

$$P\ \underline{n} \rightarrow \langle \underline{n^2} \rangle\ , \quad \text{for all integers } n \geq 0\ . \qquad \text{(i)}$$

This is done in four steps, as follows:

(a) We show that, for all obs ϕ, and all integers n and i,

$$g\ \phi\ \underline{i}\ \underline{n}\ \underline{i^2} \rightarrow \phi\ \underline{i}\ \underline{n}\ \underline{i^2}\ , \qquad \text{if } i \geq n, \qquad \text{(ii)}$$

$$g\ \phi\ \underline{i}\ \underline{n}\ \underline{i^2} \rightarrow g\ \phi\ \underline{i+1}\ \underline{n}\ \underline{(i+1)^2}\ , \text{ if } i < n\ . \qquad \text{(iii)}$$

By the definition of g, we obtain

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

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

$$\rightarrow f(g\phi)\underline{i}\ \underline{n}\ \underline{i^2} \rightarrow d(e(g\phi))\underline{i}\ \underline{n}\ \underline{i^2} \rightarrow e(g\phi)\underline{i}\ \underline{n}(+(+\ \underline{1}\ \underline{i^2})(\times\ \underline{2}\ \underline{i}))$$

$$\rightarrow g\phi\ \underline{i+1}\ \underline{n}\ \underline{(i+1)^2}\ .$$

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

$$g\ \phi\ \underline{0}\ \underline{n}\ \underline{0} \rightarrow g\ \phi\ \underline{i}\ \underline{n}\ \underline{i^2}\ . \qquad \text{(iv)}$$

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

$$h\ \phi\ \underline{n}\ \underline{0} \rightarrow \phi\ \underline{n}\ \underline{n^2} \quad . \qquad \text{(v)}$$

For, we have

$$\begin{aligned} h\ \phi\ \underline{n}\ \underline{0} &\equiv (A\phi{:}c(g(Az{:}\phi))\Omega)\phi\ \underline{n}\ \underline{0} \\ &\rightarrow c(g(Az{:}\phi))\Omega\ \underline{n}\ \underline{0} \\ &\rightarrow g(Az{:}\phi)\ \underline{0}\ \underline{n}\ \underline{0} \ . \end{aligned}$$

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

$$g(Az{:}\phi)\ \underline{0}\ \underline{n}\ \underline{0} \rightarrow (Az{:}\phi)\ \underline{n}\ \underline{n}\ \underline{n^2} \rightarrow \phi\ \underline{n}\ \underline{n^2} \ .$$

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

$$\begin{aligned} g(Az{:}\phi)\ \underline{0}\ \underline{n}\ \underline{0} &\rightarrow g(Az{:}\phi)\ \underline{n}\ \underline{n}\ \underline{n^2} \\ &\rightarrow (Az{:}\phi)\ \underline{n}\ \underline{n}\ \underline{n^2} \qquad \text{by (ii)} \\ &\rightarrow \phi\ \underline{n}\ \underline{n^2} \ . \end{aligned}$$

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

$$\begin{aligned} P\ \underline{n} \equiv k\ I\ I\ \underline{n} &\rightarrow a(b(h(j(Axy{:}I))))\ \Omega\ \Omega\ I\ \underline{n} \\ &\rightarrow b(h(j(Axy{:}I)))\ \underline{n}\ \Omega\ I \\ &\rightarrow h(j(Axy{:}I))\ \underline{n}\ \underline{0}\ I \\ &\rightarrow j(Axy{:}I)\ \underline{n}\ \underline{n^2}\ I \qquad \text{by (v)} \\ &\rightarrow (Axy{:}I)\ \underline{n}\ \underline{n^2}\ \underline{I,n^2} \\ &\rightarrow \underline{I,n^2} \\ &\rightarrow \langle \underline{n^2} \rangle \end{aligned}$$

Returning to the discussion of iteration statements, we can express the general for statement of ALGOL 60 in terms of the simple while loop treated above. For example, we can reformulate the statement

for v_i := e_1 step e_2 until e_3 do S

as

begin v_i := e_1 while $(v_i - e_3) \times sign(e_2) \leq 0$ repeat
begin S; v_i := $e_1 + e_2$ end end .

The latter form can then be represented as an ob by employing the representations of compound and while statements. Omitting the details of derivation, we list below the SK representations for the three cases of for list elements, namely, arithmetic expression, (ALGOL 60) while element, and step-until element:

$\{$for v_i := e do S$\}_{(v_1,\ldots,v_n)}$

$\equiv A\phi v_1 \ldots v_n$: $\{S\}\phi v_1 \ldots v_{i-1}\{e\}v_{i+1}\ldots v_n$.

$\{$for v_i := e while b do S$\}_{(v_1,\ldots,v_n)}$

$\equiv \mu x{:}A\phi v_1 \ldots v_n{:}(Av_i{:}\{b\})\{e\}(\{S\}(x\phi))\phi v_1 \ldots v_{i-1}\{e\}v_{i+1}\ldots v_n$

$\equiv Y(Ax\phi v_1 \ldots v_n{:}(Av_i{:}\{b\})\{e\}(\{S\}(x\phi))\phi v_1 \ldots v_{i-1}\{e\}v_{i+1}\ldots v_n)$.

$\{$for v_i := e_1 step e_2 until e_3 do S$\}_{(v_1,\ldots,v_n)}$

$\equiv Ax\phi v_1 \ldots v_n$: $(Y(Ax\phi v_1 \ldots v_n{:}\{(v_i - e_3)\times sign(e_2) \leq 0\}$

$(\{S\}(Av_1 \ldots v_n{:}x\phi v_1 \ldots v_{i-1}\{v_i + e_2\}v_{i+1}\ldots v_n))$

$\phi v_1 \ldots v_n))\phi v_1 \ldots v_{i-1}\{e_1\}v_{i+1}\ldots v_n$.

4.3 Jump Statements

We regard the execution of the statement S ≡ 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 routine (Remark 3.9-2(2)) 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 $v_1,...,v_n$.

(1) If L is followed by statements $S_1,...,S_m$, and a label M, in that order, all within b, then

$$\{L\} \equiv \{S_1\}(\{S_2\}(...(\{S_m\}\{M\})...))$$

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

$$\{L\} \equiv \{S_1\}(\{S_2\}(...(\{S_m\}(Av_1...v_n:N))...)) \ ,$$

where N ≡ I, if b is the outermost block, else N is the representation of the program part following b, that is, 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 $(v_1,...,v_n)$ is the environment of the statement S ≡ goto L, and $(v_m,...,v_n)$, 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 $\{L\}(v_m,...,v_n)$ instead of $\phi(v_1,...,v_n)$. Hence, the representation of S can be taken to be the ob

$$A\phi v_1...v_n: \{L\}v_m...v_n \ ,$$

or the (n+1)-interconvertible ob

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

Thus, we choose:

(3-1) Representation Rule.

$\{$goto L, where the environment of L is $(v_m,...,v_n)$,

$$1\leq m\leq n\}_{(v_1,...,v_n)} \equiv A\phi:(Av_1...v_{m-1}:\{L\}) \equiv K_m\{L\} \ .$$

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

$$A\phi v_1...v_n:\{L\}v_m...v_n \ ,$$

$$A\phi v_1...v_n:(Av_1...v_{m-1}:\{L\})v_1...v_n \ .$$

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 obs common to the representation of both programs have the same names.

The label M serves to illustrate the case (1) of label representations discussed above; it is otherwise superfluous.

```
begin integer x,y;
   read x;                    a ≡ Aϕxyoi:ϕiyo
   y := 0;                    b ≡ Aϕxy:ϕx0
   begin integer z;
     z := 0;                  c ≡ Aϕzxy:ϕ0xy
L:   if z=y then goto N
            else goto M;      d'≡ Aϕzxy:(=zy)(Az:N)Mzxy
M:   y := y+2×z+1;            e'≡ Aϕzxy:ϕzx(+(+y(×2z))1)
     z := z+1;                f'≡ Aϕzxy:ϕ(+zl)xy
     goto L                   g'≡ Aϕ:L
   end;                       h'≡ Aϕ:c(d'(e'(f'(g'(Az:ϕ)))))Ω
```

N:write y

end

{program} ≡ P ≡ k′II

j ≡ Aϕxyo:ϕxy o,y

k′≡ Aϕ:a(b(h′(j(Axy:ϕ))))ΩΩ

L ≡ d′M

M ≡ e′(f′(g′(Az:N)))

N ≡ j(Axy: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 ≥ 0,

$$P\ \underline{n} \leftrightarrow P'\underline{n}\ . \qquad \text{(i)}$$

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 obs h and h′. If we can show that for all integers n ≥ 0,

$$h\ N\ \underline{n}\ \underline{0} \leftrightarrow h'N\ \underline{n}\ \underline{0} \qquad \text{(ii)}$$

(where N ≡ j(Axy:I), defined in the present example), then (i) is demonstrated as follows. From the previous example, part (d), we know that for all n ≥ 0,

$$P\underline{n} \rightarrow h(j(Axy{:}I))\underline{n}\ \underline{0}\ I \equiv hN\underline{n}\ \underline{0}\ I\ .$$

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

$$P'\underline{n} \equiv k'II\underline{n}$$

$$\rightarrow a(b(h'(j(Axy{:}I))))\Omega\Omega I\underline{n}$$

$$\rightarrow b(h'(j(Axy{:}I)))\underline{n}\Omega I$$

$$\rightarrow h'(j(Axy{:}I))\underline{n}\ \underline{0}I \equiv h'N\underline{n}\ \underline{0}I\ .$$

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 $n \geq 0$,

$$h\ N\ \underline{n}\ \underline{0} \rightarrow N\ \underline{n}\ \underline{n^2}\ .$$

So (ii) would follow if we can also prove

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

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

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

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

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

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

The treatment of designational expressions and switches is omitted, except for an example which should suffice to indicate

how these may be represented as obs. In the schematic program below, b and c denote Boolean, and e and f, arithmetic expressions. It is assumed that the omitted statements indicated by ellipses do not contain any declarations.

```
begin integer x;
   M: ...
   begin integer y;
       ...
      begin integer z;

         switch P := N, if b then P [e] else L, M;
         ...
         N: ...
         begin integer w;
              ...
            goto if c then N else P [f];
         end w;

      end z;
      L: ...

   end y
end
```

The representations of the switch and goto statements in the above program are, respectively,

{P } ≡ <{N},{b}(Azyx:{P}(elem {e}3)zyx)(Az:{L}),Azy:{M}>,

and

Aϕwzyx:{c}(Awzy:{M})(Aw:{P}(elem {f}3)).

(Cf. Section 3.11.) Note that in the above two formulas 3, {e}, and {f} are to be natural number representations.

CHAPTER 5

PROCEDURES

5.1 F-procedures

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

(1) the procedure name is typed,

(2) all parameters are called by value,

(3) no global variables are modified,

(4) no jumps are made outside the procedure body,

(5) 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 $p_1,...,p_n$ be its parameters. We wish to represent f in such a manner that for all expressions $e_1,...,e_n$

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

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 $(\pi,p_1,...,p_n)$. 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 $f(e_1,...,e_n)$ to π, and certain values to p_i which are irrelevant to the result; say, we have

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

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

(1-1) Representation Rule.

{F-procedure f$(p_1,...,p_n)$ with body S}

$$\equiv \{S\}_{(\pi,p_1,...,p_n)}(A\pi p_1...p_n{:}\pi)\Omega$$

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. The following is an F-procedure; hence (1-1) is applicable.

```
integer procedure mod(x,y);
  value x,y; integer x,y;
  begin integer q;
    q := x+y;
    mod := x-y×q
  end q;
```

$a \equiv A\phi q\pi xy{:}\phi(+xy)\pi xy$

$b \equiv A\phi q\pi xy{:}\phi q(-x(\times yq)xy$

$c \equiv A\phi{:}a(b(Aq{:}\phi))\Omega$

$\underline{mod} \equiv c(A\pi xy{:}\pi)\Omega$

Example. Representation of the factorial function.

```
integer procedure fact(n); value n; integer n;
   fact := if n = 0 then 1 else n × fact (n-1);
```

As the body of this F-procedure consists of a single assignment statement, we have, by (3.5-1),

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

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

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

A non-recursive definition of the above ob is

$\underline{fact} \equiv Y(Azn{:}(=n\underline{0})\underline{1}(\times n(z(-n\underline{1}))))$.

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;                          a ≡ Aϕπnxy:ϕ(+nx)nxy
...                                   f ≡ a(Aπnxy:π)Ω
begin integer z;
   x := f(y)+z;                       Aϕzxy:ϕz(+(fyxy)z)y
...
```

5.2 Call-by-name, Side-effects

In the previous section, we have described the SK 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 trivial.

In ALGOL 60, a procedure call is intended to have the effect of an appropriately modified copy of the procedure body [27]. 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 [30]. 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 SK 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, the 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.[1] 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:

---

1 The present descriptive use of "binding" and "bound" has no connection with the terms defined at the beginning of Section 2.3.

(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 ob that represents the "parameter procedure" for its evaluation. In case p is a program variable (rather than an expression), there is also another ob which represents the "parameter procedure" to effect the assignments to p called for in the procedure. These obs, referred to as "actual evaluation" and "actual assignment" operators, are denoted ε_p^a and α_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 obs which represent the calls on the "actual evaluation" and "actual assignment" parameter procedures mentioned above. For convenience, these obs are referred to as "formal evaluation" and "formal assignment" operators, and are usually denoted ε_p^f and α_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 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

$$A\phi\colon \varepsilon_x^f\ (a(\alpha_x^f\phi)) \equiv b\ ,$$

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

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

Now, as the variable x_ε is bound to the actual evaluation operator, and the formal evaluation consists of just an application of this ob, we define ε^f_x to be

$$A\phi\rho x_\varepsilon x_\alpha xy\colon\ x_\varepsilon\rho\phi x_\varepsilon x_\alpha xy \ ,$$

or more simply,

$$A\phi\rho x_\varepsilon x_\alpha x\colon\ x_\varepsilon\rho\phi x_\varepsilon x_\alpha x \ .$$

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

$$\alpha^f_x \equiv A\phi\rho x_\varepsilon x_\alpha x\colon\ x_\alpha\rho\phi x_\varepsilon x_\alpha x \ .$$

(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

$$P \equiv b(A\rho x_\varepsilon x_\alpha x\colon\rho)$$

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 obs ε^a_x and α^a_x, 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

$$\varepsilon^a_x \equiv A\phi\rho x_\varepsilon x_\alpha xy\colon\ \rho\phi x_\varepsilon x_\alpha yy \quad ,$$

$$\alpha^a_x \equiv A\phi\rho x_\varepsilon x_\alpha xy\colon\ \rho\phi x_\varepsilon x_\alpha xx \quad .$$

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.[2]

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 ε^a_x for x_ε, α^a_x for x_α, and, by convention, Ω for x.

---

2 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 obs 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.

(c) to apply P before the rest of the program; that is, substitute $P\phi$ for ϕ.

Consequently, the statement P(y) above may be represented by the ob

$$(A x_\varepsilon x_\alpha x\colon\ (A\phi y\colon\ P\phi x_\varepsilon x_\alpha x y))\varepsilon^a_x \alpha^a_x \Omega \quad ,$$

or, more simply, by

$$A\phi y\colon\ P\ \phi\ \varepsilon^a_x\ \alpha^a_x\ \Omega\ y \quad .$$

Putting together the representations obtained piecemeal 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;
                                  ε^f_x  ≡ Aφρx_εx_αx: x_ερφx_εx_αx
                                  α^f_x  ≡ Aφρx_εx_αx: x_αρφx_εx_αx
      x := x+2;                   a      ≡ Aφρx_εx_αxy: φρx_εx_α(+x2)y
                                  b      ≡ Aφ: ε^f_x (a(α^f_xφ))
                                  P      ≡ b(Aρx_εx_αx:ρ)
   y := 1;                        c      ≡ Aφy: φ1
   P(y)                           d      ≡ Aφy: Pφε^a_xα^a_xΩy
                                  ε^a_x  ≡ Aφρx_εx_αxy: ρφx_εx_αyy
                                  α^a_x  ≡ Aφρx_εx_αxy: ρφx_εx_αxx
 end                              e      ≡ Aφ:c(d(Ay:φ))Ω
                                  {prog }  ≡ eII
```

Next, let us consider the SK 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 those 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;
```

$$\varepsilon_x^f \equiv A\phi\rho\pi x_\varepsilon x_\alpha xy : x_\varepsilon \rho\phi\pi x_\varepsilon x_\alpha xy$$

$$\alpha_x^f \equiv A\phi\rho\pi x_\varepsilon x_\alpha x : x_\alpha \rho\phi\pi x_\varepsilon x_\alpha xy$$

```
     begin
        P := x-y;
```

$$a \equiv A\phi\rho\pi x_\varepsilon x_\alpha xyuv : \phi\rho(-xy)x_\varepsilon x_\alpha xyuv$$

$$b \equiv A\phi : \varepsilon_x^f(a\phi)$$

```
        x := y
```

$$c \equiv A\phi\rho\pi x_\varepsilon x_\alpha xyuv : \phi\rho\pi x_\varepsilon x_\alpha yyuv$$

$$d \equiv A\phi : c(\alpha_x^f\phi)$$

```
    end compound                e    ≡ Aϕ:b(dϕ)
  end P;                        P    ≡ e(Aρπx_εx_αxy:ρπ)
  u:=v:=3;                      f    ≡ Aϕuv:ϕ3 3
  u:=P(v,u+1) + u;              g    ≡ Aϕuv:Pϕ Ω ε^a_x α^a_x Ω(+ul)uv
                                ε^a_x ≡ Aϕρπx_εx_αxyuv:ρϕπx_εx_αvyuv
                                α^a_x ≡ Aϕρπx_εx_αxyuv:ρϕπx_εx_αxyux
                                h    ≡ Aϕπuv:ϕ(+πu)v
                                k    ≡ Aϕ:g(hϕ)
end                             m    ≡ Aϕ:f(k(Auv:ϕ)) Ω Ω
                          {prog } ≡ mII
```

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_ε, 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 $(A\rho\pi x_\varepsilon x_\alpha xy:\rho\pi)$ 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 environment 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 nonoverlapping parts of the environment, in order to overcome the environment conflict problem. (See explanations below.)

Example.

```
begin integer x;
   procedure P(y);integer y;
      begin integer z;
                                  ε^f_y ≡ Aφzρy_εy_αy:ρy_ε(Aψ:ψφz)y_εy_αy
                                  α^f_y ≡ Aφzρy_εy_αy:ρy_α(Aψ:ψφz)y_εy_αy
         z := y+3;                a ≡ Aφzρy_εy_αyx:φ(+y3)ρy_εy_αyx
                                  b ≡ Aφ:ε^f_y(aφ)
  end block                       c ≡ Aφ:b(Az:φ)Ω
```

```
end P;                          P ≡ c(Aρy_εy_αy:ρI)

begin integer u;
   P(u+x);                      d ≡ Aϕux:P(Aψ:ψϕz)ε^a_yΩΩx
   ...                  ε^a_y ≡ Aϕuρy_εy_αy:ρI(Aψ:ψϕz)y_εy_α(+ux)x
end
```

end

In representing the procedure call in the above example, $(A\psi:\psi\phi z)$ is passed as the return point argument instead of ϕ, thus covering u. The application of $(A\psi:\psi\phi z)$ to any ob has the effect of uncovering u and restoring the environment; e.g., in ε^{f}_{y} the application is made to y_ε, and in P, to I. Note that in the representation of the procedure call, namely, d, we have used Ω for what would otherwise have been α^{a}_{y}; 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
```

$$\varepsilon^{f}_{x} \equiv A\phi\rho x_{\varepsilon}x_{\alpha}x:\rho x_{\varepsilon}(A\psi:\psi\phi)\,x_{\varepsilon}x_{\alpha}x$$

$$\alpha^{f}_{x} \equiv A\phi\rho x_{\varepsilon}x_{\alpha}x:\rho x_{\alpha}(A\psi:\psi\phi)\,x_{\varepsilon}x_{\alpha}x$$

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

$$b \equiv A\phi:\varepsilon^{f}_{x}(a(\alpha^{f}_{x}\phi))$$

$$P \equiv b(A\rho x_{\varepsilon}x_{\alpha}x:\rho I)$$

$$c \equiv A\phi y:\phi\underline{1}$$

$$d \equiv A\phi y:P(A\psi:\psi\phi)\varepsilon^{a}_{x}\alpha^{a}_{x}\Omega y$$

$$\varepsilon^{a}_{x} \equiv A\phi\rho x_{\varepsilon}x_{\alpha}xy:\rho I(A\psi:\psi\phi)\,x_{\varepsilon}x_{\alpha}yy$$

$$\alpha^{a}_{x} \equiv A\phi\rho x_{\varepsilon}x_{\alpha}xy:\rho I(A\psi:\psi\phi)\,x_{\varepsilon}x_{\alpha}xx$$

$$e \equiv A\phi:c(d(Ay:\phi))\Omega$$

$$\{prog\} \equiv eII$$

Example.

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

$$\varepsilon^{f}_{x} \equiv A\phi\rho\pi x_{\varepsilon}x_{\alpha}xy:\rho x_{\varepsilon}(A\psi:\psi\phi)\,\pi x_{\varepsilon}x_{\alpha}xy$$

$$\alpha^{f}_{x} \equiv A\phi\rho\pi x_{\varepsilon}x_{\alpha}x:\rho x_{\alpha}(A\psi:\psi\phi)\,\pi x_{\varepsilon}x_{\alpha}xy$$

$$a \equiv A\phi\rho\pi x_{\varepsilon}x_{\alpha}xyuv:\phi\rho(-xy)\,x_{\varepsilon}x_{\alpha}xyuv$$

$$b \equiv A\phi:\varepsilon^{f}_{x}(a\phi)$$

$$c \equiv A\phi\rho\pi x_{\varepsilon}x_{\alpha}xyuv:\phi\rho\pi x_{\varepsilon}x_{\alpha}yyuv$$

$$d \equiv A\phi:c(\alpha^{f}_{x}\phi)$$

```
    end compound                    e ≡ Aϕ:b(dϕ)
  end P;                            P ≡ e(AρπxεxαxY:ρIπ)
  u := v := 3;                      f ≡ Aϕuv:ϕ3 3
  u := P(v,u+1)+u;                  g ≡ Aϕuv:P(Aψ:ψϕ)Ωε(a)x α(a)x Ω(+ul)uv
                                  ε(a)x ≡ Aϕρπxεxαxyuv:ρI(Aψ:ψϕ)πxεxαvyuv
                                  α(a)x ≡ Aϕρπxεxαxyuv:ρI(Aψ:ψϕ)πxεxαxyux
                                    h ≡ Aϕπuv:ϕ(+πu)v
                                    k ≡ Aϕ:g(hϕ)
end                                 ℓ ≡ Aϕ:f(k(Auv:ϕ))ΩΩ
                                {prog } ≡ ℓII
```

For subscripted variables occurring as actual parameters, the actual evaluation and assignment operators are again chosen so as to represent the fictitious assignments between the formal and actual parameter variables. But now this involves the obs elem and replace introduced in the discussion of arrays (Section 3.11). We will simply illustrate the representation by means of an example. (The statements denoted by ellipses are assumed not to contain any declarations.)

Example.

```
begin integer n;
   ...
   begin integer array x [l:n];
     procedure P(u,v); integer u,v;
```

```
            begin
             ...
            end P;
        begin integer y;
             ...
            P(y,x [y])
        end
    end
end
```

For the above program, the representation of the procedure calling statement P(y,x [y]) is the ob

$$A\phi yxn:P(A\psi:\psi\phi y)\varepsilon^{a}_{u}\alpha^{a}_{u}\Omega\varepsilon^{a}_{v}\alpha^{a}_{v}\Omega xn,$$

where

$\varepsilon^{a}_{u} \equiv A\phi y\rho u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}vxn:\rho I(A\psi:\psi\phi y)u_{\varepsilon}u_{\alpha}yv_{\varepsilon}v_{\alpha}vxn,$

$\alpha^{a}_{u} \equiv A\phi y\rho u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}vxn:\rho I(A\psi:\psi\phi u)u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}vxn,$

$\varepsilon^{a}_{v} \equiv A\phi y\rho u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}vxn:\rho I(A\psi:\psi\phi y)u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}(x(\underline{elem}\ yn)xn,$

$\alpha^{a}_{v} \equiv A\phi y\rho u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}vxn:\rho I(A\psi:\psi\phi y)u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}v(x(\underline{replace}\ ynv)n.$

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 be called by 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

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

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

$$A\phi{:}\varepsilon^f_z(b\phi) \ ,$$

where ε^f_z is the formal evaluation operator for z in Q, and b represents the call on P, as follows:

$$b \equiv A\phi w\rho z_\varepsilon z_\alpha z \ x{:}\ P(A\psi{:}\psi\phi w\rho z_\varepsilon z_\alpha z)\varepsilon^a_y\alpha^a_y\Omega x \ x \ ,$$

$$\varepsilon^a_y \equiv A\phi w\rho z_\varepsilon z_\alpha z \ \rho_1 y_\varepsilon y_\alpha y \ x{:}\ \rho_1 I(A\psi{:}\psi\phi w\rho z_\varepsilon z_\alpha z) y_\varepsilon y_\alpha z \ x \ ,$$

$$\alpha^a_y \equiv A\phi w\rho z_\varepsilon z_\alpha z \ \rho_1 y_\varepsilon y_\alpha y \ x{:}\ \rho_1 I(A\psi{:}\psi\phi w\rho z_\varepsilon z_\alpha z) y_\varepsilon y_\alpha y \ x \ .$$

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,[3] the latter statement is represented by the ob

$$A\phi yx{:}P(A\psi{:}\psi\phi y)\varepsilon\,{}^{a}_{r}\alpha\,{}^{a}_{r}\alpha\Omega\varepsilon\,{}^{a}_{s}\Omega\Omega x \ .$$

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

$$\varepsilon^{a}_{r} \equiv A\phi y\rho r_{\varepsilon}r_{\alpha}rs_{\varepsilon}s_{\alpha}sx{:}\rho I(A\psi{:}\psi\phi y)\,r_{\varepsilon}r_{\alpha}xs_{\varepsilon}s_{\alpha}sx \ ,$$

$$\alpha^{a}_{r} \equiv A\phi y\rho r_{\varepsilon}r_{\alpha}rs_{\varepsilon}s_{\alpha}sx{:}\rho I(A\psi{:}\psi\phi y)\,r_{\varepsilon}r_{\alpha}rs_{\varepsilon}s_{\alpha}sr \ .$$

---

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

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:

$\varepsilon^{a}_{t} \equiv A\phi y\rho\ r_{\varepsilon}r_{\alpha}r\ s_{\varepsilon}s_{\alpha}s\ \rho_1\pi t_{\varepsilon}t_{\alpha}tx{:}\rho_1 I(A\psi{:}\psi\phi y\rho r_{\varepsilon}r_{\alpha}rs_{\varepsilon}s_{\alpha}s)\pi t_{\varepsilon}t_{\alpha}yx$

$\alpha^{a}_{t} \equiv A\phi y\rho\ r_{\varepsilon}r_{\alpha}r\ s_{\varepsilon}s_{\alpha}s\ \rho_1\pi t_{\varepsilon}t_{\alpha}tx{:}\rho_1 I(A\psi{:}\psi\phi t\rho r_{\varepsilon}r_{\alpha}rs_{\varepsilon}s_{\alpha}s)\pi t_{\varepsilon}t_{\alpha}tx$

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

$a \equiv A\phi y\rho\ r_{\varepsilon}r_{\alpha}r\ s_{\varepsilon}s_{\alpha}s\ x{:}Q(A\psi{:}\psi\phi y\rho r_{\varepsilon}r_{\alpha}rs_{\varepsilon}s_{\alpha}s)\Omega\varepsilon^{a}_{t}\alpha^{a}_{t}\Omega x$,

$b \equiv A\phi y\rho\ r_{\varepsilon}r_{\alpha}r\ s_{\varepsilon}s_{\alpha}s\ \pi x{:}\rho I(A\psi{:}\psi\phi y)r_{\varepsilon}r_{\alpha}rs_{\varepsilon}s_{\alpha}\pi x$,

$\varepsilon^{a}_{s} \equiv A\phi{:}a(b\phi)$.

5.4 Label parameters

The representation of label parameters is actually much simpler than of the integer variety. The reason is that two different operations, evaluation and assignment, 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 ob that represents the parameter procedure to effect the jump to the actual label. For a parameter p, this "actual goto" operator is denoted by γ_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 ob, 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 γ_p^f , again with further distinguishing marks added if more than one environment is involved.

For anyone who has followed the previous 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.

Example

```
 begin integer q;
     procedure R(v); label v;
        goto v;        a ≡ γ^f_v ≡ Aϕρv_γ: ρv_γ(Aψ:ψϕ)v_γ
                       R ≡ a(Aρv_γ:ρI)
     begin integer r;
        procedure P(x,z); integer x; label z;
                       ε^f_x ≡ Aϕρx_εx_αxz_γ:ρx_ε(Aψ:ψϕ)x_εx_αxz_γ
                       α^f_x ≡ Aϕρx_εx_αxz_γ:ρx_α(Aψ:ψϕ)x_εx_αxz_γ
                       γ^f_z ≡ Aϕρx_εx_αxz_γ:ρz_γ(Aψ:ψϕ)x_εx_αxz_γ
           R(z);       b ≡ Aϕρx_εx_αxz_γrq:R(Aψ:ψϕρx_εx_αxz_γr)γ^a_vq
                       γ^a_v ≡ Aϕρx_εx_αxz_γrρ_1v_γq:γ^f_zϕρx_εx_αxz_γrq
        begin integer s,t;
           P(t,L)      c ≡ Aϕstrq:P(Aψ:ψϕρst)ε^a_xα^a_xΩγ^a_zrq
                       ε^a_x ≡ Aϕstρx_εx_αxz_γrq:
                                   ρI(Aψ:ψϕst)x_εx_αtz_γrq
                       α^a_x ≡ Aϕstρx_εx_αxz_γrq:ρI(Aψ:ψϕsx)x_εx_αxz_γrq
                       γ^a_z ≡ Aϕstρx_εx_αxz_γrq: L r q
        end;
     L: ...
     end
end
```

CHAPTER 6

CONCLUSION

In this dissertation, we have presented a combinatory logic (or, equivalently, lambda-calculus) model of programming languages. Since a number of programming language models based on the same calculi have already appeared in the literature [5,11,16,17,18,28,31,36], 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 [17], 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, 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 (Reynolds [31]). 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 (Stratchey [36], Reynolds [31]).

4. We represent high-level programming language constructs directly, not in terms of the representations of the machine level operations (Orgass-Fitch [28]) 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 have described the model informally, and only for a representative set of programming language constructs. But we have provided enough motivating details and illustrations to, hopefully, convey the method and suggest its extension to other programming features. Our explanations, we believe, are quite adequate for the detailed construction of an effective procedure, say, in the form of a compiler, to

translate ALGOL 60 programs into lambda-expressions.

An immediate application of our model is in a functional (as opposed to computational) semantic definition of high-level programming languages, as the combinatory 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 or a combinatory object, the above-described applications essentially reduce to transformations within the respective calculi. The possibilities of some of these applications have been indicated by examples. 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. Abdali, S. K., A simple lambda-calculus model of programming languages, AEC R & D Report cOO-3077-28, Courant Inst. Math. Sci., New York Univ. (July 1973).
2. deBakker, J. W., Formal Definition of Programming Languages, Mathematisch Centrum, Amsterdam (1967).
3. _____, Recursive Procedures, Mathematical Center, Amsterdam (1972).
4. Böhm, C., The CUCH as a formal and description language, in Formal Language Description Languages ed. Steel, T. B.)., North-Holland, Amsterdam (1966), 179—197.
5. Burstall, R. M., Semantics of assignment, in Machine Intelligence 2 (ed. Dale, E., and Michie, D.), Oliver and Boyd, Edinburgh (1967), 3-20.
6. _____, Formal description of program structure and semantics in first-order logic, in Machine Intelligence 5 (ed. Meltzer, B., and Michie, D.), Edinburgh Univ. Press, Edinburgh (1970), 79-98.
7. Cadiou, J. M. and Manna, Z., Recursive definition of partial functions and their computations, SIGPLAN Notices 7, 1 (Jan. 1972), 58—65.
8. Church, A., The Calculi of Lambda-Conversion, Princeton Univ. Press, Princeton, N. J. (1941).

9. Curry, H. B., and Feys, R., Combinatory Logic, Vol. I, North-Holland, Amsterdam (1956).

10. Floyd, R. W., Assigning meanings to programs, Proc. Symp. in Appl. Math., Amer. Math. Soc. 19 (1967), 19—32.

11. Henderson, P., Derived semantics for programming languages, Comm. ACM 15, 11 (Nov. 1972), 967—973.

12. Hoare, C. A. R., An axiomatic basis for computer programming, Comm. ACM 12, 10 (Oct. 1969), 576—580,583.

13. Igarashi, S., On the equivalence of programs represented by ALGOL-like statements, Report of Computer Center, Univ. of Tokyo 1, 1 (1968), 103-118.

14. Kleene, S. C., Introduction to Metamathematics, van Nostrand, Princeton, N. J. (1950).

15. Knuth, D. E., Examples of formal semantics, in Symp. on Semantics of Algorithmic Languages (ed. Engeler, E.), Springer-Verlag (1971), 212—235.

16. Landin, P. J., The mechanical evaluation of expressions, Computer J. 6, 4 (Jan. 1964), 308—322.

17. _____, A correspondence between ALGOL 60 and Church's lambda-notation, Comm. ACM 8, 2-3 (Feb., Mar. 1965), 89—101, 158—165.

18. Ledgard, H. F., A model of type-checking, Comm. ACM 15, 11 (Nov. 1972) 956—966.

19. Lee, J. A. N., Computer Semantics, van Nostrand Reinhold, New York (1972).

20. London, R. L., The current state of proving programs correct, Proc. ACM Annual Conf. (Aug. 1972), 39-46.

21. Lucas, P., Lauer, P., and Stigleitner, H., Method and notation for the formal definition of programming languages, Report TR 25.087, IBM Laboratory, Vienna (1968).

22. Manna, Z., and Vuillemin, J., Fixpoint approach to the theory of computation, Comm. ACM 15, 7 (July 1972), 528—536.

23. McCarthy, J., A basis for a mathematical theory of programming, in Computer Programming and Formal Systems (ed. Braffort, P., and Hirshberg, D.), North-Holland, Amsterdam, (1963), 33—70.

24. McGowan, C., Correctness results for lambda-calculus interpreters, Ph.D. thesis, Comp. Sci. Dept., Cornell Univ. (1971).

25. Milner, R., Implementation and application of Scott's logic for computable functions, SIGPLAN Notices 7, 1 (Jan. 1972), 1—6.

26. Morris, J. H., Lambda-calculus models of programming languages, Ph.D. thesis, Project MAC, MIT, MAC-TR-57 (Dec. 1968).

27. Naur, P. et al., Revised report on the algorithmic language ALGOL 60, Comm. ACM 6, 1 (Jan. 1963), 1-17.

28. Orgass, R. J., and Fitch, F. B, A theory of programming languages, Studium Generale 22 (1969), 113—136.

29. Petznick, G. W., Combinatory programming, Ph.D. thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison (1970).

30, Randell, B., and Russel, L. J., ALGOL 60 Implementation, Academic Press, New York (1964).

31. Reynolds, J. C., Definitional interpreters for higher-order programming languages, Proc. ACM Annual Conf. (Aug. 1972), 717—740.

32. Rosen, B. K., Tree-manipulating systems and Church-Rosser theorems, J. ACM 20, 1 (Jan. 1973), 160—187.

33. Rosser, 3. B., Deux Esquisses de Logigue, Gauthiers-Villars, Paris (1955).

34. Schönfinkel, M., Über die Bausteine der Mathematischen Logik, Mathematische Annallen 92 (1924), 305-316.

35. Scott, D., Lattice theory, data types, and semantics, in Formal Semantics of Programming Languages (ed. Rustin, R.), Prentice-Hall, New Jersey (1972), 65—106.

36. Stratchey, C., Towards a formal semantics, in Formal Language Description Languages (ed. Steel, T.B.), North-Holland, Amsterdam (1966), 198-220.

37. Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, New York (1968).

38. van Wijngaarden, et al., Report on the algorithmic language ALGOL 68, Numerische Math. 14 (1969), 79—218.