PROCEDURE LINKAGE OPTIMIZATION
WORKING PAPER

A. Maggiolo-Schettini

Laboratorio di Cibernetica
80072 Arco Felice, ITALY

B. K. Rosen and H. R. Strong

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598, U.S.A.

ABSTRACT: This paper discusses the desirability of procedure linkage optimization and sketches a general theory of interpretive semantics which is motivated by technical problems in specifying and validating program transformations that optimize procedure linkages. One particular transformation is treated in detail.

Recursive ALGOL 60 procedures sometimes pass parameters by name in such a way that the general thunk mechanism is unnecessary and inefficient. We present an optimization which detects this kind of call-by-name and implements it thunklessly. We prove that the transformation preserves semantics and we discuss the effect on running time and memory management.

1. MOTIVATION AND OVERVIEW.

Programs written in procedural languages can devote a substantial portion of their running time to procedure entry and exit. In a language with recursion and block structure the linkages produced by a straightforward compilation may be unnecessarily costly. For example, a variable local to a procedure may be conscientiously stacked during a recursive call but then not be used anywhere after the call. An optimizing compiler should detect and remove many instances of unnecessary stacking and similar inefficiencies.

Procedure linkage optimization is especially important for large programming tasks where modularization is critical. Parnas [9] has shown that the goals of modularization are often best served when the modules do not correspond to the boxes in a flowchart for the process being programmed. He points out that the "information hiding" criterion he proposes has one major drawback: without intensive optimization of procedure linkages, a system modularized according to this criterion is "much less efficient" than one modularized according to major stages in processing [9, page 1057]. To this we add that the procedures in a system modularized according to "information hiding" are likely to be recursive. Recursive procedures cannot be eliminated by simple macroexpansion of calls, so other means of optimization must be explored.

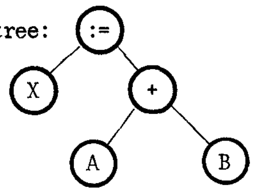
A theory of optimization requires some mathematical representation of programs both before and after application of optimizing transformations. The mathematical framework should be helpful in proving that "optimized" programs really do compute what the programmer intended. None of the available semantic theories are quite right for procedure linkage optimization, so we have assembled yet another interpretive semantics from ideas present in various places in the literature, particularly [2, 8, 13]. This theory of semantics is sketched in Section 3, after a condensed discussion of an unusual representation of programs in Section 2.

Section 4 deals with one particular optimization: thunkless call-by-name. Section 5 sketches further work in progress. Section 6 compares the semantic theory used here with other work.

2. TREE STRUCTURED WEBS.

The nesting of statements and expressions within each other and the roles of each part in the whole are naturally expressed by a tree structure with labelled nodes and an ordering on the children of each node, as in

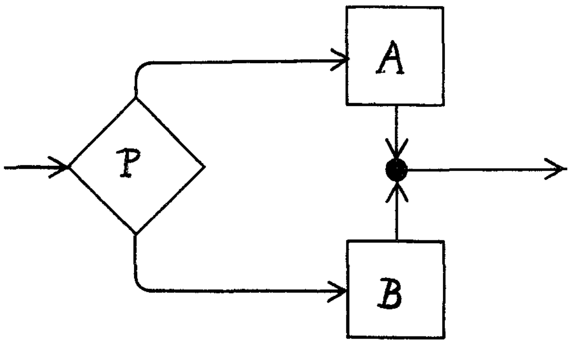
text: X := A + B

tree:


These operator-operand trees are somewhat more convenient for our purposes than VDL trees [13, Section 1].

Trees are not enough for our purposes because control flow is not explicit. We also need a representation like a flowchart: a directed graph with symbols on the nodes and arcs. We call such graphs webs because a similar but less general concept has that name in the literature on graph grammars. The web representing a program should have at least one node for each statement in the program, and the arcs should express possible control flow.

An appropriate web can be obtained by adding arcs between the nodes in an operator-operand tree. For operators representing complex combinations of simpler statements (such as BLOCK for blocks or COND for the conditional construction) we add an extra operand called a _shadow_. Shadows have the niladic operator CONTINUE and act like the dot in



as an explanation of "_if_ P _then_ A _else_ B."

In Figure 1 we simultaneously represent an ALGOL 60 block as a tree and as a web. Most of the symbols used are specific to ALGOL 60 or its close relatives; only DCL, NEXT, and CONTINUE are part of the general theory. The arcs labelled DCL are not control flow arcs; they point from uses of names to their declarations. The spellings y,k,x and other information in lower case are not part of the web, which spells every name ANON and uses explicit DCL arcs rather than repetitions of character strings to connect name uses to name declarations.

The combination of tree structure and web structure is crucial to our treatment of macroexpansion [1, Section 3] [8, Section 4.7] independent of syntactic details. In addition to the macrodefinitions indicated by procedure declarations, high level languages have other macrodefinitions, with wide variety of syntax, fixed in their defining interpreters. For example, stepped iteration statements are macrodefined in ALGOL [8, Section 4.6.4.2], and this macrodefinition would be applied when control reaches the node marked ITER in Figure 1.

Macrodefinitions are essentially _rule schemata_ [10, Section 6]: pairs R → S of trees such that some leaves of R and S hold _parameter_ symbols rather than operators. Each parameter X has a _domain_ D_X of trees; occurrences of X are places where an arbitrary tree from D_X might appear in a program. Rule schemata may be applied at nodes in trees, replacing subtrees that match the R by subtrees that match the S. In addition to the considerations in [10, Section 6] we specify a fairly simple convention for carrying along the web arcs.

Here we will not fix on any particular set of tree structured webs as the set of programs, for we do not wish to define any one language here. We only wish to explore the consequences of postulates that are satisfied by fairly natural interpretive definitions of procedural languages. To define any particular language in such a way as to apply the theory, one would have to specify a vocabulary of symbols, a set of programs, domains for parameters in rule schemata, and so on. How this is done is of no concern to us at the moment.

The kind of language considered here is exemplified by ALGOL 60, FORTRAN, ALGOL W [14], BLISS [16], and PL/I minus multitasking and ON conditions. There is one feature of some of these languages that we cannot deal with: declarations which call for nontrivial computations. In array bounds, for example, we require that there be only constants, variables, and arithmetic operators. Conditionals, procedure calls, and other expressions involving a flow of control must be kept out of most declarations. The one exception is, of course, the procedure declaration, which may specify arbitrarily elaborate computations in the procedure body.

The symbols carried by nodes in trees for any given language form its set of _operator symbols_. We assume that all operators have adicities. A K-adic appears only on nodes with K children, for K a nonnegative integer. A _polyadic_ operator appears only on nodes with two or more children. Table I lists the special operators assumed by the theory; any given language uses most of these plus many operators peculiar to itself.

The symbols carried by web arcs for any given language form its set of _cursor_ symbols. Table II lists the special cursors assumed by the theory. Some other operators and cursors used by ALGOL 60 appear in Figure 1. In Figure 1 we used the special cursor START for starting the processing in blocks as well as in GROUPs. This additional use is not part of the general theory.

We assume that operators have been classified in three ways: declarative/imperative, flowsimple/flowcomplex, and macro/nonmacro. The classification for special operators is postulated in Table III; other operators must be classified in such a way that declaratives and macrooperators are flowsimple. Declaratives must also be nonmacro. We also assume that the set of tree structured webs counted as programs has various reasonable properties. For example, a node has at most one DCL outarc.

The only macrooperators in the general theory are the CALL(K) operators, but any given language may other macrooperators whose macrodefinitions are part of the definition of the language. In ALGOL 60, the node carrying ITER in Figure 1 should be macroexpanded according to Figure 2. The dyadic MUNTIL is another macrooperator of ALGOL 60, because the test in an _until_ or _while_ loop may itself be a long _computation_. In our example the test is straightforward, so the MUNTIL statement becomes an UNTIL statement according to the macrodefinition in Figure 3. When the left operand of an MUNTIL

Operator	Adicity	Use
STOP	0	Terminate computations.
CONTINUE	0	Dummy statements.
GROUP	poly	Treat a series of statements as one statement.
PROC(K)	K+1	Declare a procedure with K arguments.
CALL(K)	K+1	Call a procedure with K arguments.
ANON	0	Refer to procedures or variables.
GOTO	0	Transfer control.
FORMAL	0	Formal parameters.

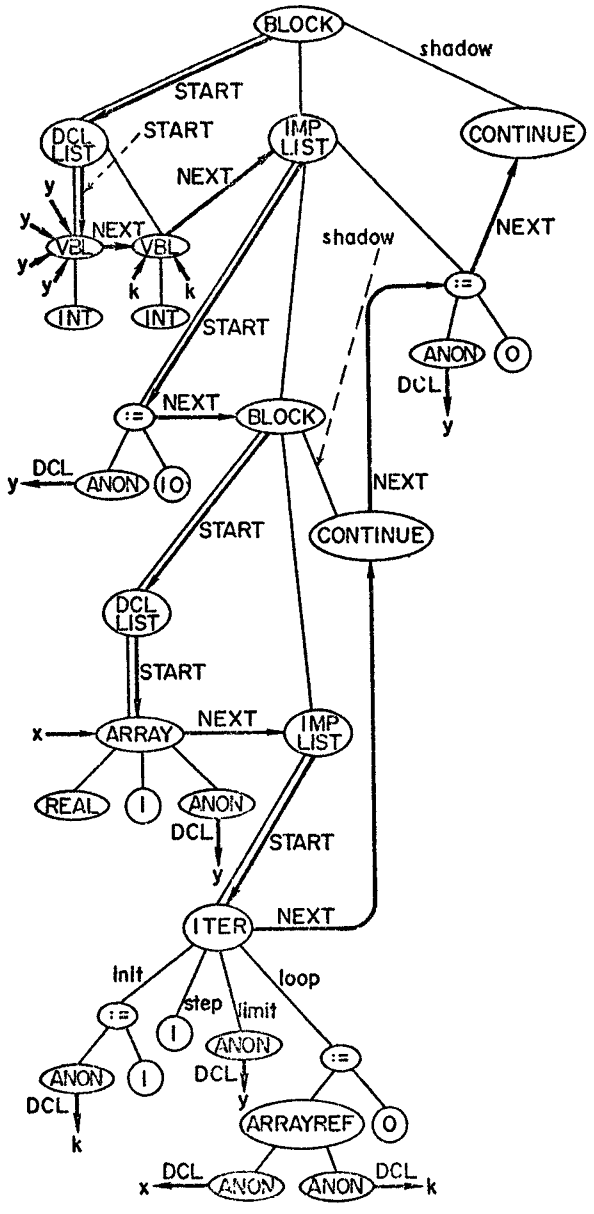
Table I. Special Operator Symbols.

Cursor	Use
START	Pass control from a GROUP to its first constituent.
NEXT	Pass control sequentially.
DCL	Point from uses of names to their declarations.
JUMP	Point from GOTO nodes to their targets.

Table II. Special Cursor Symbols.

Operator	Imperative?	Flowsimple?	Macro?
STOP	YES	YES	NO
CONTINUE	YES	YES	NO
GROUP	YES	NO	NO
PROC(K)	NO	YES	NO
CALL(K)	YES	YES	YES
ANON	YES	YES	NO
FORMAL	YES	YES	NO
GOTO	YES	NO	NO

Table III. Classification of Special Operators.



```
begin integer y, k;
   y := 10;
   begin real array x [1 : y];
      for k := 1 step 1 until y do
      x [k] = 0
   end;
   y := 0
end
```

FIG. 1. Tree structured web for an ALGOL 60 block.

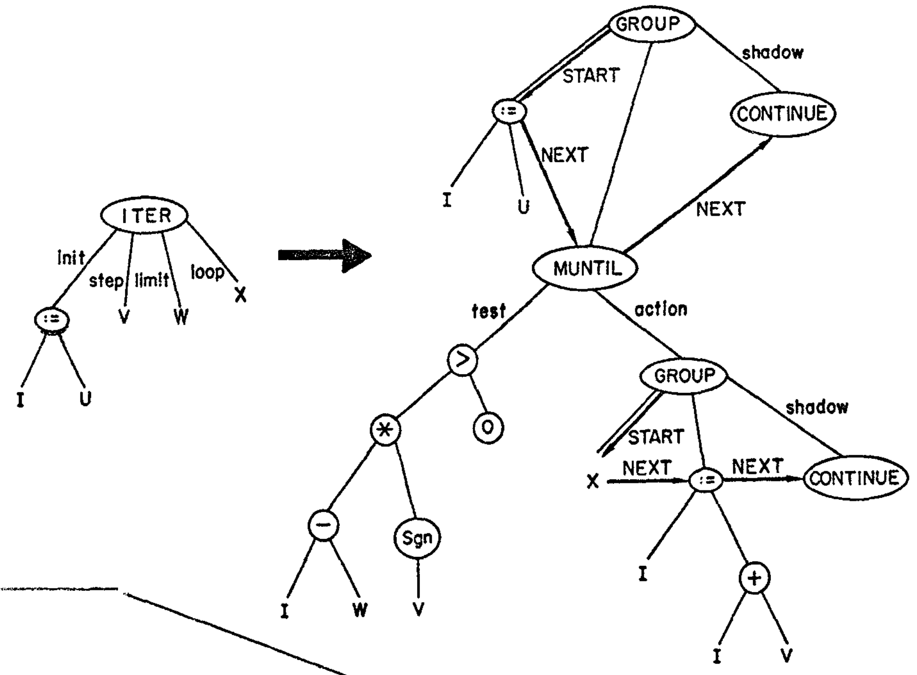


FIG. 2. Macrodefinition of stepped iteration.

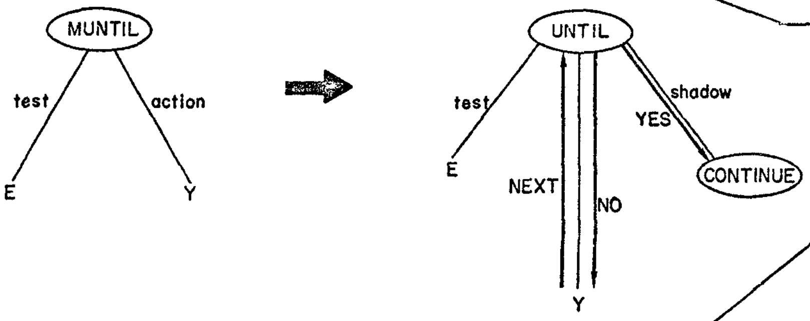


FIG. 3. Partial macrodefinition of MUNTIL. The domain of E consists of trees without implicit control flow.

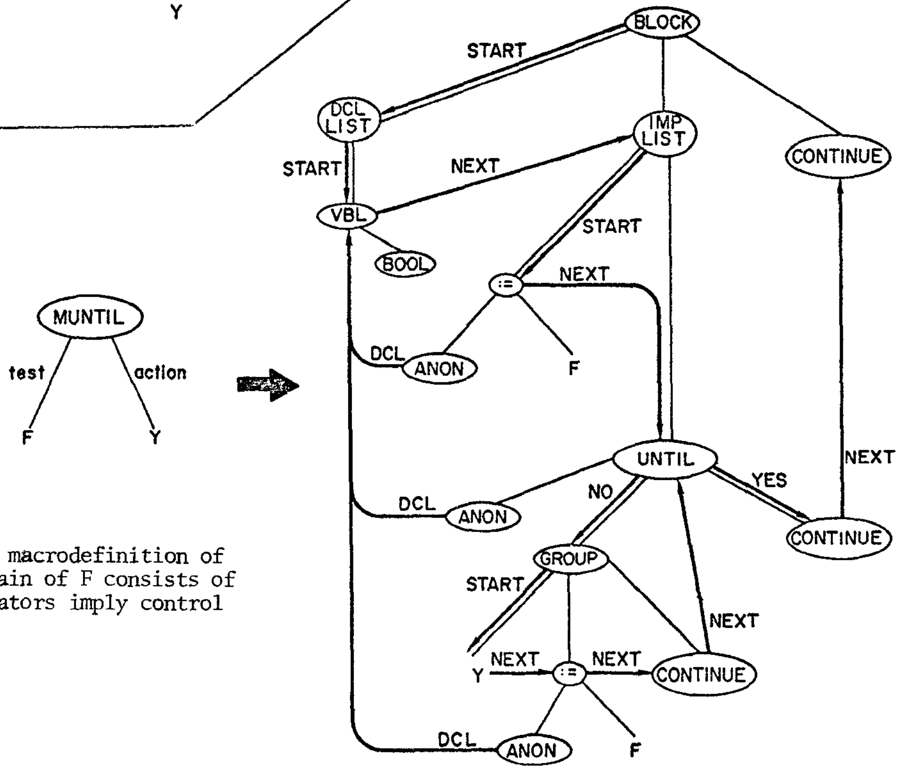


FIG. 4. Partial macrodefinition of MUNTIL. The domain of F consists of trees whose operators imply control flow.

node involves control flow, then Figure 4 is used.

The definition of ALGOL 60 includes macrodefinitions such as Figures 2-4. These definitions use niladic macroparameter symbols as placeholders for which actual trees may be substituted. Each macroparameter Z has a domain D_Z of trees that may replace it. For example, D_I for Figure 2 contains just ANON but D_U contains all arithmetic expressions. The theory postulates certain closure properties for the domains and establishes some technical lemmas to support the results stated in Section 3. Space does not permit details here.

We close this section with some terminology to be used later. As in [10, Section 4], nodes in trees are strings of nonnegative integers. The set of all such strings is NN*. The children of a node n in a program P are all nodes n·(k) that appear in P, for k a nonnegative integer. If p is a child of n then PARp = n. An arc is a triple a = (m,c,p) where m,p are in NN* and c is a cursor. The tail of a is Ta = m; the head of a is Ha = p.

A node m in a program P may be a macronode because it carries a macrooperator or because some proper descendant n of m calls for some computations before the operation indicated by Pm can take place. For example, := is not a macrooperator in ALGOL 60, but

arrayidentifer [functioncall] := expression

must still be macroexpanded. Here the call leads to a proper descendant n of the node with :=, and the operator Pn is macro. If Pn were FORMAL or if n had control outarcs in P (i.e. outarcs with cursors other than DCL), then n would also presuppose computations. When a node n in a program P has no presuppositions, it is said to be ready. Thus, a macronode is one that carries a macrooperator or has unready proper descendants. If n is a macronode in P and P' is the result of performing the appropriate macroexpansion at n in P, then we write $P \Rightarrow_n P'$.

A procedure declaration is simply a node p in a program P such that Pp is PROC(K). The subtree of P at p·(0) is the procedure template. (Think of this as synonymous with "procedure body" [8, Sec. 5.4.3] for the moment.) Nodes descended from p·(0) which carry FORMAL and have DCL outarcs to children p·(k) with $1 \leq k \leq K$ represent uses of formal parameters in the procedure's computations.

3. ABSTRACT INTERPRETIVE SEMANTICS.

A state ξ of the abstract interpreter for a language is an assignment of values to all the "variables" in a possibly infinite collection. The general theory reserves a few variables for special purposes and puts a few restrictions on the transition relation $\vdash$ among states.

The variables include a set CELLS of "cells" which may hold scalars, arrays, stacks, or what-have-you: whatever is appropriate for a specific language. A few variables not in CELLS have special functions: IN and OUT hold the input and output files; PROG holds a tree structured web; ENV holds a partial function mapping declarations in PROG to cells (this is the "environment"); CONTROL holds a node in PROG.

The notion of control flow is as much like ordinary flowchart control flow as possible. In an initial state ξ, ξCONTROL is the root of ξPROG. In a final state η, ηCONTROL carries STOP. What happens in between is determined by inspecting the locus of control in the current state of the program, "nearby" nodes, and the values of expressions built up from cell contents and primitive operations.

The source programs of a language are those allowed as values of PROG in initial states. For BLISS and other languages which have no distinction between imperative statements and value generating expressions, all programs are source programs. For ALGOL 60, the source programs (which are the "programs" of [8]) form a proper subset of the programs. Straightforward macroexpansion of the procedure call in A := B + FIDDLE (C) leads to an assignment statement of the form A := B + (begin...end). This is thoroughly intelligible and desirable [16, Secs. 1.2.1, 1.2.2], although it violates the rules of ALGOL 60. The simplest way to include both ALGOL 60 and BLISS in our theory (without making either appear unduly pathological) is to allow the set of source programs to be properly contained in the set of programs.

A state ξ of the interpreter is initial iff

ξPROG is a source program;
ξCONTROL is the root of ξPROG;
ξENV has empty domain;
ξOUT is defaulted;
each ξx for x in CELLS is defaulted. (3.1.1)

Here, the default values for OUT and for cells are part of the definition of any specific language. Empty files, zero numbers, and null strings would be natural choices. For each pair (P, in) such that P is a source program and in is a possible value for IN, there is a unique initial state ξ such that ξPROG is P and ξIN is in.

Like the set of programs, the transition relation $\vdash$ is language dependent. We are concerned with properties of $\vdash$ in procedural languages, not with a metalanguage in which to specify $\vdash$. Before imposing restrictions on $\vdash$, we state the obvious definition for the input/output relation computed by any source program P. If in and out are values for IN and OUT, then

$$[P]\ \underline{in} = \underline{out} \text{ iff } (\exists \eta, \text{ a final state})(\xi \vdash^* \eta \ \&\ \eta\text{OUT} = \underline{out}), \tag{3.1.2}$$

where ξ is the initial state with ξPROG = P and ξIN = in. To say that η is final is to say that ηCONTROL carries STOP in ηPROG.

The partial function [P] is the meaning of P under consideration when we attempt to show that "optimized" programs have the same meaning as their sources. We will find it convenient to treat infinite, jammed, and successful computations in a uniform way. We require that the transition relation $\vdash$ be a total function on states: for each state ξ there is exactly one state η such that $\xi \vdash \eta$. If $\eta = \xi$ then ξ is said to be a dead state. Final states are dead, of course. A computation is any infinite sequence

$$\xi_0 \vdash \xi_1 \vdash \xi_2 \vdash \ldots$$

of states, with ξ_0 initial. If some ξ_i is dead the computation is said to terminate. If some ξ_i is final, it terminates successfully. Even successful computations are infinite sequences; they just stabilize eventually with $\xi_i = \xi_{i+1} = \xi_{i+2} = \cdots$.

If X is a set of variables and ξ,η are states then

$$\xi \equiv \eta \text{ mod } X \text{ iff } \xi y = \eta y, \text{ all } y \text{ not in } X. \quad (3.2.1)$$

In order to minimize the use of curly brackets, we also set

$$\xi \equiv \eta \text{ mod } x \text{ iff } \xi \equiv \eta \text{ mod } \{x\}, \quad (3.2.2)$$

for each variable x.

When ξCONTROL is a macronode in ξPROG, the theory sketched in Section 2 specifies what to do. When it is a nonmacronode carrying a special operator, we have $\xi \vdash \eta$ for the obvious choice of η. (Like CONTINUE, each PROC(K) merely passes control.) For other nonmacronodes, we have yet to rule out absurdities like

if (this program now contains 17 array declarations)
    then (---) else (* * *),

which cannot be executed without scanning the entire program. We must formalize the dependence of each nonmacronode on only a small set of nearby nodes. The next two postulates will do this. First we must say just what "nearby" means here.

The vicinity of a node n in a program P consists of n together with all proper descendants p of n in P such that

(PARp is in VICn); (3.3.1)

(no control arc has tail PARp and head p). (3.3.2).

POSTULATE 3.4. Trichotomy Principle. Let ξ be a state. Let P be ξPROG and n be ξCONTROL. Then ξ is dead or "macrolive" or "nonmacrolive". If ξ is macrolive then $\xi \vdash \eta$ with

(no ancestor of n is declarative in P); (1)

(no proper descendant of n in P is in the domain of ξENV); (2)

$\xi \equiv \eta$ mod PROG & $P \Rightarrow_n \eta$PROG. (3)

If ξ is nonmacrolive then

(no proper ancestor of n is declarative in P); (4)

(every node $p \neq n$ in VIC n is ready in the sense of Sec. 2); (5)

(Pn is a nonmacrooperator and is not FORMAL); (6)

P = ηPROG; (7)

(some control arc a has Ta = n & Ha = ηCONTROL). (8)

POSTULATE 3.5. Program Locality Principle. Let ξ, ξ', η be states and let n be ξCONTROL, P be ξPROG, and P' be ξ'PROG. Suppose ξ is nonmacrolive and $\xi \vdash \eta$. Suppose $\xi \equiv \xi'$ mod PROG, that VIC n is the same subset of NN* in P and in P', and that all p in VIC n have Pp = P'p. Suppose that all nodes in VICn have the same outarcs in P and in P'. Then ξ' is nonmacrolive and some state η' has $\xi' \vdash \eta'$ and $\eta' \equiv \eta$ mod PROG.

LEMMA 3.6. Let P be a program and let P' be the result of macroexpanding a node m in P. Then [P] = [P'].

LEMMA 3.7. Let P be a program and let in be in the domain of [P]. Then there is a program P' such that

$P \Rightarrow^* P'$ & [P]in = [P']in, (1)

and no macroexpansions are performed in the computation of P' on in.

The copy rule semantics of procedure calls have been included in our theory, but we have not yet specified the semantics of procedure declarations, or any other declarations. Since we are concerned with meaning, not implementation, a very simple postulate is appropriate.

POSTULATE 3.8. Declaration Principle. Let ξ be a state such that ξCONTROL has a NEXT outarc a and carries a declarative operator. Then $\xi \vdash \eta$, where ηCONTROL is Ha and

$\xi \equiv \eta$ mod {CONTROL, ENV}. (1)

If the operator is PROC(K) for some K then ξENV = ηENV. Otherwise there is a cell x such that

(ηENV)(ξCONTROL) = x; (2)

(ηENV)n = (ξENV)n, all n $\neq$ ξCONTROL. (3)

For ALGOL 60 and other languages

without a provision for "sharing" of storage by names, we could add that x is not already a value of ξENV.

We close this section with some remarks on the scope of our interpretive semantics. As was remarked in Section 2, we cannot allow declarations to involve nontrivial computations. Array bounds may be large arithmetic expressions, but they must be flowsimple and nonmacro. This restriction to a subset of languages like ALGOL 60 is not much inconvenience to the programmer. The effect of

```
begin real array stuff [1: fiddle (gamma)]; ...end
```

can be obtained by

```
begin integer top;
      top := fiddle (gamma);
      begin real array stuff [1 : top];...end
end
```

instead. But to prove this is so would require a temporary extension of the theory. The combinatorics of Section 2 are unpleasant enough already. We prefer to honor the spirit of phrases like "the units of operation...are called statements" [8, Section 4] or "every executable form...(that is, every form except the declarations)" [16, Section 1.2.1] by restricting declarations enough to make these phrases meaningful. Declarations are in fact executed, and at run-time too [8, Section 5].

Except for the restrictions on declarations, we have resisted the temptation to make abuse-prone programming language features indescribable. We allow for unrestricted GOTOs. We allow global references in procedure templates. These things made the theory more complex then it would otherwise have been, but the size and shape of the added burden are what would be expected on intuitive grounds.

For motivational purposes we considered "procedure template" synonymous with "procedure body". In applications, templates will consists of bodies (expressed as tree structured webs) augmented by prologs and epilogs appropriate to the parameter passing mechanisms used. For example, suppose an ALGOL 60 procedure with body B has a real formal VALPARAM which is passed by value. Then the template corresponds to

```
begin real VALPARAM;
      VALPARAM := FORMAL;
      B
end,
```

and the actual for VALPARAM in a call on the procedure would be substituted by name for FORMAL.

This reduction of call-by-value to call-by-name is part of the definition of ALGOL 60[8, Section 4.7.3]. Wirth and Hoare [14, pp. 422-423] state the same reduction more formally, along with a similar reduction of call-by-result to call-by-name. The only other parameter passing mechanism we are familiar with is call-by-reference. This can be reduced to call-by-name via a reduction to call-by-value: to pass E by reference is to pass the address of E by value [15, page 91]. In short, our theory applies to languages with various parameter passing mechanisms. The necessary reductions to call-by-name have already been used "in nature" to explain these mechanisms.

4. THUNKLESS CALL BY NAME.

Recursive ALGOL 60 procedures sometimes use call by name in a very simple way. Once nonrecursive calls have made available some addresses, the recursive calls merely pass the addresses down to deeper levels of recursion. Using a formal criterion which detects this situation as a special case, an optimizer can eliminate some of the parameters passed by name to a recursive procedure p.

For us, a procedure in a program is a procedure declaration node: a node p such that p carries the operator PROC(K) for some K. The nodes p•(1),...,p•(K) are the formals of p. A call on p within the template (that is, descended from p•(0)) is said to be directly recursive. Under certain conditions on the directly recursive calls and certain other conditions on the other calls on p, the thunkless call by name transformation will be applicable. Throughout this section, we suppose that P is a program, p carries PROC(K) in P, and Y is a subset of the set {p•(1),...,p•(K)} of formals.

Let a be any DCL or JUMP arc in P. Then we say that a is under any node n such that n is an ancestor of Ta. If n is also an ancestor of Ha, then a is bound under n; otherwise a is free under n. The bound/free distinction is much the same as local/global or internal/external. We choose words from logic instead so as to avoid possible minor clashes with the usage in specific programming languages.

Now suppose that n is an actual parameter node in a procedure call within the template of p: the parent m of n has p•(0) ANC m and m•(i) = n for some i ≠ 0 and Pm = CALL(J) for some J. Any DCL or JUMP arc a which is under n is said to be thunkless iff it is either bound under n or free under p. Thus the name resolved by a may be either strictly local to the actual parameter n or global with respect the procedure p.

CONDITION 4.1. Let n be an actual parameter node in a directly recursive call on p, and let n correspond to a member of Y: n has the form m•(k) for some k with p•(k) in Y. Then one of the following holds:

Pn = FORMAL & DCLn is in Y; (1)

(every arc under n is thunkless). (2)

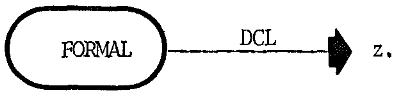
Informally, this condition says that the procedure cannot pass just anything to itself as an actual for a member of Y. It can pass a member of Y; it can pass something relatively global; it can pass an expression which declares the names it uses. The last possibility does not arise in ALGOL 60. It does arise in BLISS and other languages which consider all statements to be expressions.

The second condition for thunkless call-by-name is simply that recursion cannot be smuggled in from the outside by passing a call on p by name to a call on p:

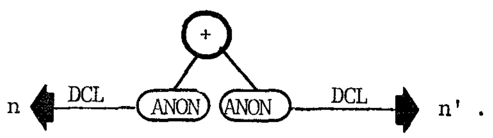
CONDITION 4.2. Let n be an actual parameter in a call on p which is not directly recursive, and let n correspond to a member of Y. Then no call on p is descended from n.

If both conditions hold, we say that Y is a <u>thunkless</u> set of formals. In order to exploit this situation we must define a monoid of transformations that act on actual/formal correspondences.

Each call m on p determines an actual/formal correspondence C_m for Y. For each y in Y, say $y = p\cdot(k)$, $C_m\, y$ is the tree $P/[m\cdot(k)]$ together with all web arcs inherited from P, including any arcs from descendants of $m\cdot(k)$ to nondescendants of $m\cdot(k)$. For example, m might be a directly recursive call and the actual for y might be a reference to a member z of Y. Then C_m would be:



On the other hand, $C_m y$ might be a sum of quantities declared at nodes n and n':



For each directly recursive call m, C_m determines a transformation $[C_m]$ which acts on such actual/formal correspondences. For any correspondence A and any y in Y, let k be chosen with $y = p\cdot(k)$. Let n be $m\cdot(k)$. Then

$$([C_m]A)y = \underline{\text{if}}\ (Pn = \text{FORMAL}\ \&\ \text{DCL}n = z \text{ in } Y)\ \underline{\text{then}}\ Az\ \underline{\text{else}}\ C_m y. \qquad (4.3.1)$$

Now consider the set of transformations

$$G = \{[C_m]\ |\ m \text{ a directly recursive call on } p\}. \quad (4.3.2)$$

This set generates a monoid

$$M = \{g_0 \circ g_1 \circ \cdots \circ g_{J-1}\ |\ J \geqslant 0\ \&\ g_j \text{ in } G, \text{ all } j<J\}, \quad (4.3.3)$$

where $\circ$ is the usual composition, $g \circ h$ = (g after h).

In the example we began with, each m has P n = FORMAL and DCL n = y for all choices of $y = p\cdot(k)$ in Y and $n = m\cdot(k)$. Thus $[C_m]$ is the identity mapping and $|M|$ is one. In general $|M|$ may be much larger than $|G|$, but it will always be <u>finite</u>.

What is the significance of M? Consider an arbitrary call m' on p which is not directly recursive. Let m be a directly recursive call. Macroexpansion at m' leads to a new call on p at the node $m'' = m'\cdot(m/[p\cdot(0)])$. By Condition 4.1, the new actual/formal correspondence for m'' is derived from the old one for m' in a very simple way:

$$C_{m''} = [C_m]\, C_{m'}. \qquad (4.3.4)$$

Further macroexpansions lead to further calls. For any such call, say q, derived from the call m', we have

$$C_q = f\, C_{m'} \qquad (4.3.5)$$

for some member f of M.

Now we can exploit thunklessness. For each choice of m' as a call on p which is not directly recursive and each choice of f in M, we form a new procedure p'[m',f], with only K - |Y| formals. The template for p'[m',f] differs from that of p in two ways. First, any descendant of p·(0) with a DCL outarc to some y in Y is replaced in p'[m',f] by a node whose descendants form $(fC_{m'})y$. Second, each directly recursive call m on p is replaced by a call on $p'[m',[C_m]f]$ in p'[m',f]. Formals not in Y are treated just as in p.

Let there be H calls on p in P which are not directly recursive, and let J be |M|. Thus there are HJ procedure declarations p'[m',f]. The new program P' is like P except at p and at calls on p which are not directly recursive. In P', p carries GROUP and has HJ + 1 children. These are the p'[m',f] declarations, listed in an arbitrary order, plus the shadow. Each call m' in P becomes a call on p'[m',I] in P', where I is the unit element in M.

THEOREM 4.4. Let p be a procedure declaration node in a program P and let Y be a subset of the set of formals of p. Suppose Y is thunkless. Then the new program P' has the same meaning as P.

PROOF. Let <u>in</u> be a value for IN; we must show that [P] <u>in</u> = [P']<u>in</u>. We may assume at least one of these is defined. Suppose [P]<u>in</u> = <u>out</u>, the argument in the other case being similar. We show [P']<u>in</u> = <u>out</u>.

By Lemma 3.7, there is a sequence $(P_0,\ldots,P_J)$ of programs such that P_0 = P, P_j macroexpands in one step to P_{j+1} for all j < J, and P_J computes from <u>in</u> to <u>out</u> with no macroexpansion.

Let P'_0 be P'. For each j < J, P'_j is like P_j except for differences in procedure declarations and calls: each call on p in P_j appears as a call on some p'[m',f] in P'_j. By macroexpanding in P'_j to mimic the expansion that takes P_j to P_{j+1}, we can form P'_{j+1} so as to be like P_{j+1} except for the p vs. p'[...] differences. By the Program Locality Principle (Post. 3.5) and the procedural case of the Declaration Principle (Post. 3.8), the fact that P_J computes from <u>in</u> to <u>out</u> without macroexpansion implies that $[P'_J]$ <u>in</u> = <u>out</u> also. By J applications of Lemma 3.6, we conclude that [P'] <u>in</u> = <u>out</u>. ∎

In order to give a self-contained statement of Theorem 4.4, we make the weakening assumption that Y is the set of formals of p and offer the following.

INFORMAL SUMMARY OF THEOREM 4.4:
Let p be a procedure in a program P (with all of its parameters passed by name). If, in any directly recursive call, p passes to itself only

(a) its formals,

(b) expressions relatively global to it,

and

(c) expressions which declare the names they use (other than those of type (b)),

and if no calls on p are passed to p, then there is a straightforward conversion from P to an equivalent program P' in which p is replaced by several procedures, each of which has no parameters.

Thunkless call by name saves time because fewer parameters are passed whenever one of the replacements for p is called. The transformation also decreases the frequency of page faults in a virtual memory operating environment. Thunks can make the top of the runtime stack reference items near the bottom [4][5]; the memory management algorithm may have moved the lower parts of a deep stack out of fast storage by the time these references occur.

The cost of thunkless call by name is an increase in program size, and a combinatorial explosion can be arranged. Whether programs written to solve a problem will explode is another question. In a virtual memory system one can increase program size very cheaply so long as the system's slow storage capacity is not reached. Until then, the important space consideration is not program size but working set size [3, pp. 175, 180, 187]. Thunkless call by name can only improve the working set structure of a program.

5. FURTHER WORK.

The calling graph of a program has a node for each procedure and an arc from p to q whenever the body of p includes a call on q. In order to speed up the convergence of various iterations in an optimizer's analysis, it is helpful to transform a program so that its calling graph will have many and small (rather than few and large) strongly connected components. Maggiolo and Strong [7] show to do this with a sequence of macroexpansions.

Strong [11] and Walker and Strong [12] have developed ways to avoid unnecessary stacking in flowcharts that implement recursion equations. Parts of this work have been extended to procedural languages with only call by value and without pointer variables. It is our aim to extend this work to allow other calls and pointers as well. We are attempting to develop transformations which will replace all parameters and local variables by global variables, together with a minimal use of stacking in the sense of [11] and [12]. Theorem 4.4 is a positive result in this direction, stating that, under certain conditions, such transformations exist and avoid stacking altogether.

Of course, in any sufficiently powerful language (including all those discussed in this paper) we can avoid stacking via coding a simulated stack. Note that this kind of transformation alters the set of values contained in cells during computation; whereas, the transformation of Theorem 4.4 does not.

6. COMPARISONS WITH OTHER WORK IN SEMANTICS

Interpretive semantics comes in three main flavors, which we will call "specific", "metalinguistic", and "postulational". Notable examples of specific semantics are the ALGOL 60 report [8] and the definition of the contour model [6]. In the first case, the emphasis on the syntax of the language tends to overwhelm the other aspects of the interpreter, which are left tacit. In the second case, the emphasis on nested structures, cells whose values may be pointers to other cells, and scope rules for names tends to hide the fact that one specific language is being defined. The contour model language (let us call it CML) is very apt for compilative definitions of semantics:" a FUNGOL program computes whatever its image under the following compiler from FUNGOL to CML computes..." For our purposes CML is too close to machine language: what appears as a single macroexpansion in a dynamic program becomes a series of bookkeeping operations and jumps controlled by a static CML program. On the other hand, results like Theorem 4.4 clearly should not be proved separately for each procedural high level language.

Metalinguistic semantics specifies a metalanguage in which to write interpretive definitions of languages. If the metalanguage itself is very precisely defined, then the ambiguities that arise in informal interpreters (such as the one in [8]) can be avoided. One could try to build a semantic theory based on properties of the metalanguage or of some readily specifiable family of interpreters within the metalanguage.

The Vienna Definition Language is too flexible to constitute an implicit semantic theory. There is very little in common between, say, the Vienna interpreter for EPL [13, Sec. 4] and the Vienna interpreter for VDL itself [13, Sec. A]. If one tries to specify a family of Vienna interpreters to correspond to the family of procedural languages, one finds that the control structures and data structures in VDL are inappropriate.

A Vienna interpreter does not execute program statements, even after they have been expressed as abstract tree structures.

Instead the interpreter constructs a "control tree" whose nodes carry VDL instructions. The acyclic nature of the control tree and Vienna objects in general makes it impossible to follow a simple iterative loop without resorting to macroexpansion [13, Sec. 3.1]. Iterative source programs are ultimiately explained by recursion in the interpreter. For optimization we wish to go in the opposite direction, replacing recursion by iteration as much as possible. We need to clearly distinguish between recursion and iteration; no simple constraint on a Vienna interpreter will do this.

By using an operator-operand tree together with possibly cyclic web arcs, we allowed control to flow through the program itself, just as in a flowchart or an ALGOL 60 program. Without web arcs we would need repetitions of arbitrary names to serve the purposes of DCL and JUMP arcs. Spelling out names leads to constant worry about capture of free variables, programs which are the same except for choices of names, and so on. As soon as one tries to prove anything, such considerations become extremely tedious and error prone.

Culik [2, Section 6] eliminates names in favor of "pointers" between tree nodes that correspond to our DCL and JUMP arcs. His success in concisely and readably formalizing a substantial portion of ALGOL 60's semantics [2, Table 2] encouraged us to deal with web structure directly rather than with names. We have gone farther in this direction by adding other control cursors and shadows that carry CONTINUE, so that all control flow could be expressed directly by moving CONTROL along arcs in PROG. Culik simulates normal flow by adding symbols like EXEC and END to the operators at nodes [2, Section 6] and by manipulating these symbols with rules based on programmed grammars [2, Section 3].

Sections 2 and 3 here exemplify the third flavor of semantics: abstract postulates about interpreters and the mathematical consequences of the postulates. We illustrated the postulates with fragments of specific interpreters. In discussing VDL we hinted at the implications of the postulates for a metalanguage for defining procedural languages. The three flavors can and should be blended, but they should not be confused.

In trying say just what procedural languages have in common, we have been guided primarily by technical considerations. We knew that Lemmas 3.6 and 3.7 were true of languages like ALGOL 60 long before we could say why they were true without becoming mired in the syntax of that particular language. We knew that these lemmas led to Theorem 4.4 long before we could state the presuppositions expressed in the Program Locality Principle (Post. 3.5) and Declaration Principle (Post. 3.8). Another important consideration was that the postulates should actually be true for the interpreters described informally by existing successful definitions of specific languages. The semantic portions of the ALGOL 60 report [8] and the standard semantics for flowcharts were especially pertinent.

REFERENCES

1. Constable, R. L. and Gries, D., On classes of program schemata. SIAM J. Computing 1 (1972), 66-118.
2. Culik, K., A model for the formal definition of programming languages. CSRR 2065, Dept. of Appl. Analysis and Computer Sci., U. of Waterloo, Waterloo, Ont., June, 1972.
3. Denning, P. J., Virtual memory. Computing Surveys 2 (1970), 153-189.
4. Ingerman, P.Z., Thunks. Comm. ACM 4 (1961), 55-58.
5. Irons, E. T. and Feurzig, W., Comments on the implementation of recursive procedures and blocks in ALGOL 60. Comm. ACM 4 (1961), 65-69.
6. Johnston, J.B., The contour model of block structured processes. SIGPLAN Notices 6 (February, 1971), 52-82.
7. Maggiolo-Schettini, A. and Strong, H. R., A graph-theoretic algorithm with applications for transforming recursive programs. AICA Convegno di Informatica Teoria, Pisa, March, 1973.
8. Naur, P. (Ed.), Revised report on the algorithmic language ALGOL 60. Comm. ACM 6 (1963), 1-17.
9. Parnas, D. L., On the criteria to be used in decomposing systems into modules. Comm. ACM 15 (1972), 1053-1058.
10. Rosen, B. K., Tree-manipulating systems and Church-Rosser theorems. J.ACM 20 (1973), 160-187.
11. Strong, H. R., Translating recursion equations into flowcharts. JCSS 5 (1971), 254-285.
12. Walker, S. A. and Strong, H. R., Characterization of flowchartable recursions. JCSS (to appear).
13. Wegner, P., The Vienna definition language. Computing Surveys 4 (1972), 5-63.
14. Wirth, N., and Hoare, C.A.R., A contribution to the development of ALGOL. CACM 9 (1966), 413-431.
15. Wirth, N., and Weber, H., EULER: a generalization of ALGOL, and its formal definition. CACM 9 (1966), 13-25 and 89-99.
16. Wulf, W., et al., Bliss/11 Reference Manual. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Penn., March, 1972. [See also BLISS: a language for systems programming, CACM 14 (1971), 780-790.]

Acknowledgment

We thank K. C. Fanti for preparing the manuscript under very difficult circumstances.