Flow Analysis and Optimization of LISP-like Structures*

by

Neil D. Jones & Steven S. Muchnick
Department of Computer Science
The University of Kansas
Lawrence, Kansas 66045 USA

I. Introduction

In [12] the authors introduced the concept of binding time optimization and presented a series of data flow analytic methods for determining some of the binding time characteristics of programs. In this paper we extend that work by providing methods for determining the class of shapes which an unbounded data object may assume during execution of a LISP-like program, and describe a number of uses to which that information may be put to improve storage allocation in compilers and interpreters for advanced programming languages.

We are concerned chiefly with finding, for each program point and variable a finite description of a set of graphs which includes all the shapes of values the variable could assume at that point during the execution of a program. If this set is small or regular in structure, this information can be used to optimize the program's execution, mainly by use of more efficient storage allocation schemes.

In the first part we show how to construct from a program without selective updating a tree grammar whose nonterminals generate the desired sets of graphs; in this case they will all be trees. The tree grammars are of a more general form than is usually studied [8,19], so we show that they may be converted to the usual form. The resulting tree grammar could naturally be viewed as a recursive type definition [11] of the values the variables may assume. Further, standard algorithms may be employed to test for infiniteness, emptiness or linearity of the tree structure.

In the second part selective updating is allowed, so an alternate semantics is introduced which more closely resembles traditional LISP implementations, and which is equivalent to the tree model for programs without selective updating. In this model data objects are directed graphs. We devise a finite approximation method which provides enough information to detect cell sharing and cyclic structures whenever they can possibly occur. This information can be used to recognize when the use of garbage collection or of reference counts may be avoided.

The work reported in the second part of this paper extends that of Schwartz [17] and Cousot and Cousot [7]. They have developed methods for determining whether the values of two or more variables share cells, while we provide information on the detailed structure of what is shared. The ability to detect cycles is also new. It also extends the work of Kaplan [13], who distinguishes only binary relations among the variables of a program, does not handle cycles, and does not distinguish selectors (so that his analysis applies to nodes representing sets rather than ordered tuples).

II. Programs with Tree-like Data

In the first part of this paper, we shall carry out our analyses on a simple programming language called SL (Structure Language) whose syntax is as follows

```
program → {[label:] stmt}+
stmt    → assign | if | goto
assign  → var := exp | var := input |
          output := exp
if      → if test goto
test    → atom exp | null exp | var {=|≠} var
goto    → goto label
exp     → atom | var | var.sel |
          cons(exp {, exp}*)
```

We assume that instances of the syntactic classes var, sel, atom and label are members, respectively, of the sets Var, Sel, Atom and Label which are pairwise disjoint.

Informal Discussion

The language SL closely resembles LISP with the PROG feature but without functions or p-lists, and extended to allow arbitrary numbers of selectors. See Reynolds [16] for methods to handle recursively-defined functions.**

The semantics of SL are essentially those of LISP, with two minor exceptions: the customary uses of NIL (a special atom in LISP) must be done

*The work reported here was performed under the partial support of National Science Foundation grant MCS76-80269.

**Reynolds' work came to our attention after this development was completed. He treats a subset of LISP with recursive function calls and without sequential execution. It seems clear that the two methods could be combined.

via the empty or undefined data structure $\perp$; and any attempt to apply a selector (e.g., CAR, CDR) to an atom or $\perp$ will result in program abortion instead of being "undefined".

In LISP without selective updating operations it is natural to view the value of a variable as a tree without regard to cells, pointers, etc. Each internal node will have an edge labeled s leading to a subtree, for each s in the set of selectors $Sel = \{sel_1,\ldots,sel_m\}$. If $T_1,\ldots,T_m$ are trees, $cons(T_1,\ldots,T_m)$ denotes the tree consisting of a root node, with edges labeled $sel_1,\ldots,sel_m$ leading to the roots of $T_1,\ldots,T_m$, respectively. If T has this structure, then $T.sel_i$ denotes the subtree T_i.

All trees in examples in this paper will use the fixed set of selectors Sel = {hd,tl}. Trees will be given pictorially with the root at the top, leaves at the bottom, and atoms labeling the leaves. Selectors may be omitted for convenience, in which case the edge directed southwest (southeast) from a node goes to the "hd" subtree ("tl" subtree).

Semantics of SL

The value of an atom is an element of the set ATOM and is given by the function $\mathfrak{A}$: Atom → ATOM. We assume that atoms are otherwise unspecified simple data objects -- numbers, bounded-length character strings, booleans, etc.

Trees are defined formally by the sequences of labels encountered on paths from the root to the leaves. Such a sequence is written as $s_1.s_2.\ldots.s_n.a$ where $n \geq 0$, $s_1,\ldots,s_n \in Sel$ and $a \in ATOM$. The set of all such label sequences is naturally described by $Sel^* \times ATOM$.

By definition a *tree* is a finite subset T of $Sel^* \times ATOM$ such that if $s_1.\ldots.s_n.a \in T$, then $s_1.\ldots.s_n.s'_1.\ldots.s'_p.b \in T$ only if p = 0 and a = b. The null tree $T = \phi$ is written as $\perp$ (read "bottom").

The definition allows trees with "missing branches" such as {hd.1,tl.hd.2}. In diagrams the missing branches will be drawn, and filled in with $\perp$. Some examples are shown in Figure 1.

Figure 1

By definition if $T,T_1,\ldots,T_m$ are trees and $s \in Sel$ then

$$T.s = \{t_1\ldots t_n.atom \mid s.t_1\ldots t_n.atom \in T\}$$

and

$$cons(T_1,\ldots,T_m) = \bigcup_{i=1}^{m} \{sel_i.t_1\ldots t_n.atom \mid t_1\ldots t_n.atom \in T_i\}$$

The definitions are naturally extended as follows: Let $A,B_1,\ldots,B_m$ be sets of trees. Then

$$A.s = \{T.s \mid T \in A - ATOM - \{\perp\}\}$$

$$cons(B_1,\ldots,B_m) = \{cons(T_1,\ldots,T_m) \mid T_1 \in B_1,\ldots,T_m \in B_m\}$$

Following the style of denotational semantics [15,18], we define the meanings of the various constructs in terms of the domains ATOM, $VAL = 2^{Sel^* \times ATOM}$ and STORE = [Var → VAL] (a store $\sigma \in STORE$ is a function mapping each variable to its current value). We only define the meaning of expression evaluation and assignment statement execution. The other features of SL can be formally defined by well-known means, including continuations.

$\mathfrak{E}$: Exp → [STORE → VAL] is a partial function given by:

$\mathfrak{E}[\![atom]\!]\ \sigma = \{\mathfrak{A}[\![atom]\!]\}$

$\mathfrak{E}[\![var]\!]\ \sigma = \sigma(var)$

$\mathfrak{E}[\![var.s]\!]\ \sigma = \sigma(var).s$ [undefined if $\sigma(var) = \perp$ or is an atom]

$\mathfrak{E}[\![cons(e_1,\ldots,e_m)]\!]\ \sigma = cons(\mathfrak{E}[\![e_1]\!]\sigma,\ldots,\mathfrak{E}[\![e_m]\!]\sigma)$

$\mathfrak{AS}$: Assign → [STORE → STORE] is given by

$\mathfrak{AS}[\![var := exp]\!]\ \sigma = \lambda x \in Var.$ (*if* x = var *then* $\mathfrak{E}[\![exp]\!]\sigma$ *else* $\sigma(x)$)

Execution of a statement and the whole program will be aborted if

a) an expression which must be evaluated is undefined
b) an if statement compares two values either of which is a nonempty, nonatomic tree
c) a nonempty, nonatomic tree is read by a statement "var := *input*".

Note that point (b) implies that these tests model EQ in LISP, rather than EQUAL.

III. Structure Shapes and Data Flow Equations

We now show how to construct a system of forward data flow equations from a program. Let X be a program variable in Var, and I a program point. The system will then have a variable F(I,X). The least fixed point solution of the system will associate with F(I,X) a set of tree shapes which includes all possible shapes X could have at point I in any possible execution of the program.

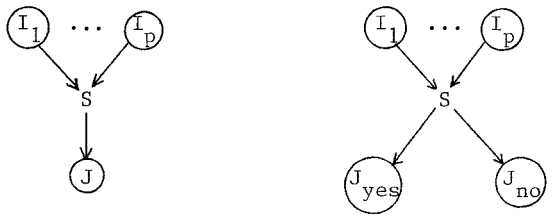
The set of shapes is defined simply by:

$$Shape = 2^{Sel^* \times \{o\}}$$

That is, we have replaced all elements of ATOM by the symbol o which represents an arbitrary atom. Next, we form the lattice 2^{Shape} of all subsets of

Shape, with the usual subset ordering. The variables in the equation system will have elements of 2^{Shape} as values.

As in [12] we first convert an SL program to a flowchart and annotate it with program points in the set $\mathfrak{pp} = \{0,1,\ldots,n\}$, one for each arc in the flowchart. Consider the flowchart segments of Figure 2.



(a) Non-*if* statement (b) *if* statement

Figure 2

The equations are formed as follows:

Form of S	Equation
X := atom or X := *input*	$F(J,X) = \{o\}$
X := Y	$F(J,X) = F(I_1,Y) \cup \ldots \cup F(I_p,Y)$
X := Y.s	$F(J,X) = F(I_1,Y).s \cup \ldots \cup F(I_p,Y).s$
X := *cons*($Y_1,\ldots,Y_m$)	$F(J,X) = \bigcup_{i=1}^{p} cons(F(I_i,Y_1),\ldots,F(I_i,Y_m))$
if atom X	$F(J_{yes},X) = \bigcup_{i=1}^{p} F(I_i,X) \cap \{o\}$ $F(J_{no},X) = \bigcup_{i=1}^{p} F(I_i,X) - \{o\}$
if null X	$F(J_{yes},X) = \bigcup_{i=1}^{p} F(I_i,X) \cap \{\perp\}$ $F(J_{no},X) = \bigcup_{i=1}^{p} F(I_i,X) - \{\perp\}$
if X = Y	$F(J_{yes},X) = \bigcup_{i=1}^{p} (F(I_i,X) \cap F(I_i,Y)) \cap \{o,\perp\}$ $F(J_{no},X) = \bigcup_{i=1}^{p} F(I_i,X) \cap \{o,\perp\}$ $F(J_{yes},Z) = F(J_{no},Z) = F(I_1,Z) \cup \ldots \cup F(I_p,Z)$ for $Z \neq X, Y$
other	$F(J,X) = F(I_1,X) \cup \ldots \cup F(I_p,X)$

Finally, the equation

$$F(I_0,X) = \{\perp\}$$

is included for the initial program point I_0 and each variable X.

Note that the only way in which statements constrain the shapes of values flowing to them is through the possibilities for abortion of execution. Taking these constraints into account through backward flow analysis, as discussed in our [12] or Kaplan and Ullman's [14], could provide more specific information about shapes. However, we ignore this possibility for the present since the extension is straightforward.

To obtain the maximal information available from forward flow analysis about the program's data values the F(I,X) sets should be as small as possible, as long as they include every value which may be computed. It is for this reason that the sets $\{o\}$, $\{\perp\}$ and $\{o,\perp\}$ appear in the equations -- to conclude as much as possible from the program's assumed correctness.

IV. Solving the Flow Equations

There are at least two methods available to solve the data flow equations. One is iteration in either its regular or chaotic form (see [12] and [6]) starting with every $F(I,X) = \phi$. It should be clear that the functions involved are all continuous, so solutions always exist.

This method is appropriate if the solution is finite. Unfortunately this is not generally the case for the systems under consideration here. Instead, we shall introduce here a method based on regular tree grammars which handles the finite and infinite cases equally well. The objective is to obtain a regular tree grammar such that the language it generates is a safe approximation to the minimal fixed point of the system of flow equations. This is useful, since tree grammars are a well-understood extension of regular string grammars; consequently existing algorithms can be used to test for finiteness, linearity, etc.

The approach is to form from each data flow equation a production in an extended regular tree grammar, which is then transformed into an ordinary tree grammar.

Tree Grammars

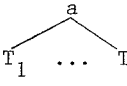
A *regular tree grammar* (see, for example, [8] or [19]) is a grammar $\langle N,\Sigma,P,S\rangle$ with N a finite set of nonterminal symbols, Σ a ranked alphabet of terminal symbols such that $N \cap \Sigma = \phi$, $S \in N$ is the initial nonterminal, and P is a finite set of productions of the form $A \to t$ where $A \in N$ and $t \in T_\Sigma(N)$. Here $T_\Sigma(N)$ is defined by

(i) $N \cup \Sigma_0 \subseteq T_\Sigma(N)$

(ii) if $k \geq 1$, $a \in \Sigma_k$ and $T_1,\ldots,T_k \in T_\Sigma(N)$, then $a[T_1,\ldots,T_k] \in T_\Sigma(N)$

(iii) nothing else is in $T_\Sigma(N)$

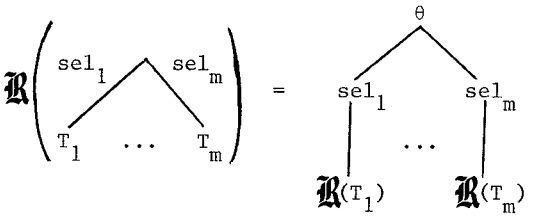
Note that the linear representation $a[T_1,\ldots,T_k]$ in (ii) corresponds to the tree



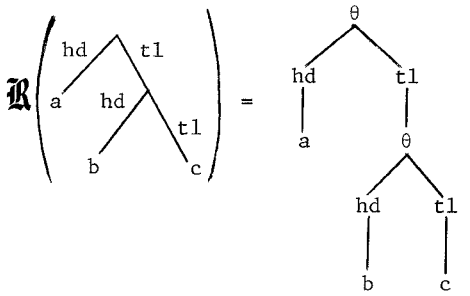
In order to describe our edge-labeled trees by tree grammars, we choose k = m and $\Sigma_0 = ATOM \cup \{\perp\}$, $\Sigma_1 = Sel$, $\Sigma_2 = \ldots = \Sigma_{m-1} = \phi$, $\Sigma_m = \{\theta\}$. If T is a tree as used in our description of SL, the corresponding element $\mathfrak{R}(T)$ of $T_\Sigma(N)$ is recursively defined by:

$\mathfrak{R}(atom) = atom$

$\mathfrak{R}(\perp) = \perp$

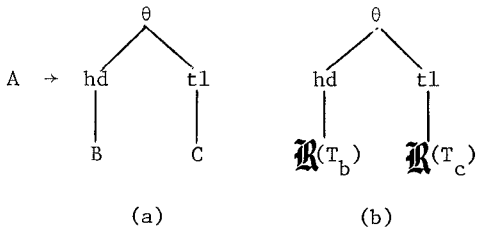


For example, we have the equation below:



We assume the semantics of a regular tree grammar is defined by least fixed points, in the same manner as was done by Ginsburg [9] for context-free languages. That is, nonterminals are interpreted as sets of trees, and the productions are viewed as a system of set equations. It should be clear that this gives the same generated set L(G) as the usual tree-rewriting semantics since the analogy between regular tree grammars and context-free grammars is very close.

We will also write tree productions in the SL notation for convenience. This does no harm since it is easily seen that $T_1 \Rightarrow T_2$ by production $A \to T_3$ iff $\mathfrak{R}(T_1) \Rightarrow \mathfrak{R}(T_2)$ by $A \to \mathfrak{R}(T_3)$. For example, the natural interpretation of $A \to$  is that for all $T_b \in B$, $T_c \in C$, the tree  is in A. Translating into T_Σ terminology, the production is as shown in Figure 3(a) and means that if $\mathfrak{R}(T_b) \in B$, $\mathfrak{R}(T_c) \in C$, then the tree in Figure 3(b) is in A.



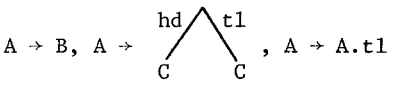
(a) (b)

Figure 3

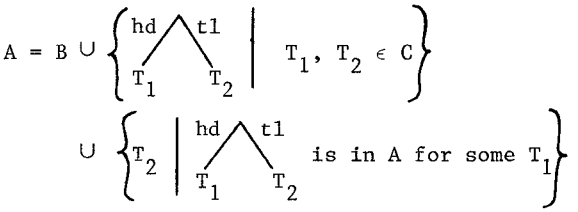
Having shown the connection to ordinary tree grammars, we now proceed to assume that all grammars are expressed and interpreted in terms of SL trees.

Definition An *extended regular tree grammar* is a quadruple G = <N,Σ,P,S> where N, Σ, P and S are as above, except that P is now allowed to contain productions of the form A → B.s where s ∈ Sel.

The semantics of such a production in terms of sets is simply the assertion that B.s ⊆ A. For example the three productions

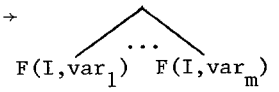


would correspond to the set equation



The new production type clearly gives rise to a continuous function, so the solution of the extended regular tree grammar may be found, as before, by least fixed points.

Examining the flow equations we see that they are nearly in the extended tree grammar form except for restrictions involving ○ and ⊥. Removing these is safe, since the result is only to enlarge the solution values. Referring to Figure 2(a), let I be any of the $I_1,\ldots,I_p$ preceding S; then the construction of the grammar can now be expressed by:

S	Production
var_1 := var_2	$F(J,var_1) \to F(I,var_2)$
var_1 := atom or var_1 := *input*	$F(J,var_1) \to ○$
var_1 := var_2.sel	$F(J,var_1) \to F(I,var_2).sel$
var_0 := *cons*(var_1, ...,var_m)	$F(J,var_0) \to$ 
otherwise	$F(J,var) \to F(I,var)$

An Example

Consider the program in Figure 4 which builds a linear tree X from input items, and then transfers them to Y so they appear on Y in their original order.

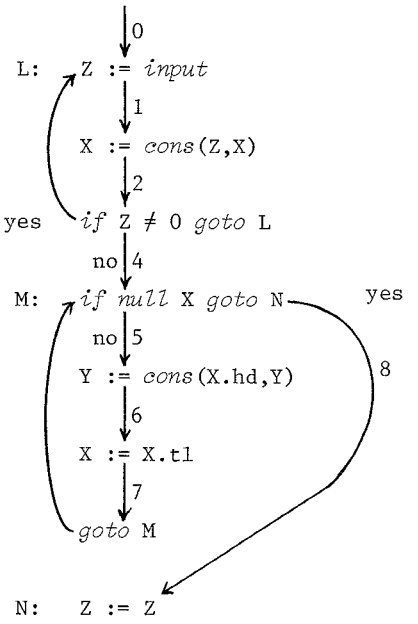
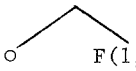
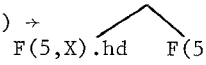


Figure 4

The productions obtained from this are (omitting those for Z):

$F(0,X) \to \perp$	$F(0,Y) \to \perp$
$F(1,X) \to F(0,X) \mid F(3,X)$	$F(1,Y) \to F(0,Y) \mid F(3,Y)$
$F(2,X) \to$ 	$F(2,Y) \to F(1,Y)$
$F(3,X) \to F(2,X)$	$F(3,Y) \to F(2,Y)$
$F(4,X) \to F(2,X)$	$F(4,Y) \to F(2,Y)$
$F(5,X) \to F(4,X) \mid F(7,X)$	$F(5,Y) \to F(4,Y) \mid F(7,Y)$
$F(6,X) \to F(5,X)$	$F(6,Y) \to$ 
$F(7,X) \to F(6,X).tl$	$F(7,Y) \to F(6,Y)$
$F(8,X) \to F(4,X) \mid F(7,X)$	$F(8,Y) \to F(4,Y) \mid F(7,Y)$

Simplifying this by compressing chains of productions and renaming, we get:

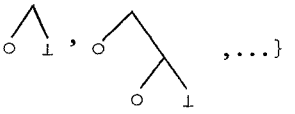
(A = F(1,X)): $A \to \perp \mid$ 

(B = F(5,X)): $B \to$  $\mid B.tl$

(C = F(5,Y)): $C \to \perp \mid$ 

(D = F(5,X).hd): $D \to B.hd$

The solutions are:

$A = B = C = \{\perp,$  $,\ldots\}$

$D = \{o\}$

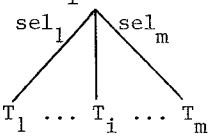
Note that the right linearity or finiteness of each variable is clearly evident.

Theorem If $G = \langle N,\Sigma,P,S\rangle$ is an extended regular tree grammar, there is an ordinary regular tree grammar $G' = \langle N,\Sigma,P',S\rangle$ with $L(G) = L(G')$.

Proof We give the construction, which uses Büchi's method of "derived rules" [4]. Define the relation $\sim$ on $N \cup \Sigma_0$ to be the smallest reflexive, transitive relation such that

(a) $A \to X$ implies $A \sim X$

(b) $A \to B.sel_i$, $B \sim C$ and

$C \to$  imply $A \sim T_i$,

provided $T_1,\ldots,T_m$ all derive nonempty sets of terminal trees.

Now define $G'' = \langle N,\Sigma,P'',S\rangle$ where

$P'' = P \cup \{A \to X \mid A \in N \text{ and } A \sim X\}$ and

$P' = P'' - \{A \to B.s \in P\}$.

Essentially the same theorem was proved in Reynolds [16] by another (more complex) method, so we omit the proof that $L(G) = L(G'') = L(G')$. □

The Example Revisited

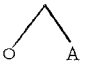
Elimination of productions with selectors on the right proceeds as follows:

1. $A \not\sim \perp$ and $C \sim \perp$ follow from the productions
2. $B \sim A$ follows from $B \to B.tl$, $B \sim B$ and

   $B \to$ 
3. $D \not\sim o$ follows from $D \to B.hd$ and

   $B \to$ 

The revised grammar has productions

$A \to \perp \mid$ 

$B \to$  $\mid A$

$C \to \perp \mid$ 

$D \to o$

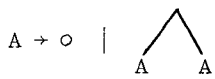
and it is easily checked that it has the same solution as the grammar with selectors on the right.

This method does not yield a perfect solution to the original problem, for two reasons. First,

the flow analysis method associates with each node and each variable a set of values. While this makes grammatical analysis possible, it can lose some information, as in the following example:

```
      X := 1;
L:    X := cons(X,X);
      if ~ goto L
```

The values X may actually have at L are the complete binary trees of heights 0,1,2,.... However the method above leads to productions



which have all binary trees as solution.

The second reason is the restrictions concerning o and ⊥ in the flow equations which were ignored in constructing the productions. We conjecture that these restrictions do not destroy the regularity of the solutions, although they may increase the complexity of obtaining them.

V. Relating The Tree Grammars To Storage Allocation

A simple and fairly efficient implementation of SL may be organized as follows. Each internal node is represented by a record with fields $s_1,\ldots,s_m$, one for each selector in Sel. Any nonatomic tree is identified by a pointer to the record for its root node. Each program variable is bound to a *root word*, contained in a fixed runtime location, whose content is a pointer to the root record of its current value (or the value itself if atomic) thus an assignment X := Y merely copies one root word into another; X := Y.s copies the s field of Y's root record into X's root word; and X := *cons*($Y_1,\ldots,Y_m$) makes the root word of X point to a newly allocated record whose fields $s_1,\ldots,s_m$ are initialized to the values of $Y_1,\ldots,Y_m$. This method involves only a bounded amount of work for each statement type, and provides maximal natural storage sharing, i.e. all that can be achieved without the use of a hashing cons [10].

There are some obvious inefficiencies common to LISP-like languages which are amenable to data flow analysis. We now briefly discuss those which can be handled by use of the tree grammars just presented. The main tool used is the fact that familiar context-free and regular grammar algorithms generalize directly to tree grammars. In particular, infiniteness is easily decidable.

1. Let X be a variable, and consider $V(X) = \bigcup_{I \in pp} F(I,X)$, our upper bound on the set of values X may assume during execution.

   a) If V(X) contains at most one shape other than ⊥, a fixed location may be assigned to the root record of X, so its subfields may be addressed directly without need for the root word.

   b) If V(X) is finite, a storage area for X may be allocated statically for X before execution. This area need not participate in storage reclamation activities.

   c) Now consider $V(X).sel_i$. If this is empty, no record within a value of X needs to contain an sel_i field.

2. Let statement X := Y.s be preceded by program point I. If o or ⊥ is in F(I,Y), a runtime error is possible; if $F(I,Y) \subseteq \{o,\perp\}$, a runtime error will definitely occur.

More will be said about optimization of LISP-like programs in the second part of this paper, particularly concerning storage reclamation by reference counts and garbage collection, and the use of CDR-coding [1].

VI. Elimination of Reference Counts and Garbage Collection

In the remainder of this paper we assume an implementation like that described in the last section; in addition we extend the language (to a more powerful version called SUSL) by the addition of selective updating, in a manner similar to RPLACA and RPLACD in LISP.

Two standard methods for storage management are the use of reference counts and garbage collection. Garbage collection is the more powerful method, but the collection process is quite expensive and, in its classical forms, disruptive to the computation, especially in interactive and real-time contexts. When cyclic data structures cannot occur, as in SL, the method of reference counting may be used. However, this method requires both space overhead to store the counts and time to update them.

In this part of the paper we describe a method to reduce both types of overhead, often to zero, by a pre-execution program analysis. The analysis constructs finite approximations to the actual runtime data structures which may occur, and is guaranteed to detect cyclic structures and nodes with reference counts greater than one, if they can possibly exist. In this way, runtime data cells may be put into three classes:

1. Those whose reference counts never exceed one. These may be returned to free storage as soon as pointers to them are destroyed. No reference counts need be maintained.

2. Those which may not appear in cycles, but whose reference counts may exceed one. These may be allocated with reference count fields which are maintained during execution.

3. Other cells, which may appear in cycles. The overhead of reference counts may be avoided at the expense of using garbage collection.

In Clark & Green [5] it is observed that only 2% to 8% of LISP cells are ever pointed to more than once, so this optimization should result in substantial savings. Further, our method for detecting opportunities for optimization appears to be significantly more general than that of Barth [2].

Before proceeding to give an alternate semantics for SL based on the ideas sketched above and presenting methods for analyzing its storage allocation properties, we shall extend the language to include selective updating in a manner which models the functions RPLACA and RPLACD in LISP and assignment to records with pointers in languages such as PASCAL and PL/I. The new operation is written as X.s := Y and its intended effect is to replace the s-labeled edge from the root of X by an edge leading to the root node of Y. This selective updating operation makes it possible to create cyclic structures, as shown in Figure 5, where performing X.hd := X on the structure in (a) results in that shown in (b). Thus the language with selective updating is more powerful than without it. We call the language with selective updating SUSL (Selective Updating Structure Language).

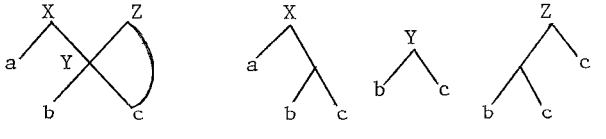
(a) (b)

Figure 5

We now give a semantics for SUSL which incorporates within it an alternate semantics for SL equivalent to that in Section II and based on the implementation ideas in Section V. To do this we first redefine the STORE to consist of all directed graphs of the following sort:

1. each internal node has one son for each selector in Sel
2. each leaf is labeled with an atom or $\perp$ (the null tree)
3. each variable in Var labels one and only one node
4. each node is accessible from a variable-labeled node
5. each node is a member of a universal set NODE of nodes

For example, the graph in Figure 6(a) is a store corresponding to the values of X, Y, and Z in (b).



(a) (b)

Figure 6

The following auxiliary function is used in the semantic definition:

node: Var → [STORE → NODE]

node vσ = the node in σ labeled v

In the example in Figure 6(a) *node* Zσ is the upper-rightmost node.

The effect of the assignment statement is a function

$\mathfrak{AS}$: Assign → [STORE → STORE],

defined below. In general, $\mathfrak{AS}[\![S]\!]\sigma$ is found by modifying σ (unless S aborts), as described in the following table. Afterwards, all nodes which are inaccessible from variables are removed from the new σ.

Form of S	$\mathfrak{AS}[\![S]\!]\sigma$
var := atom	add a new leaf node labeled $\mathfrak{A}$(atom); move var to the new node
var_1 := var_2	move var_1 to label *node* $var_2\sigma$
var_0 := *cons*(var_1, ...,var_m	make a new node n and move var_0 to label it; for i = 1,...,m, add an sel_i edge from n to *node* $var_i\sigma$
var_1 := var_2.sel	*if node* $var_2\sigma$ has an sel descendant n *then* move var_1 to node n *else* $\mathfrak{AS}[\![S]\!]\sigma$ is undefined
var_1.sel := var_2	*if node* $var_1\sigma$ has an sel edge from it *then* replace it by an sel edge leading to *node* $var_2\sigma$ *else* $\mathfrak{AS}[\![S]\!]\sigma$ is undefined

Execution is aborted in exactly the same situations as in the semantics of SL given in Section II.

Let σ be an acyclic STORE graph and X a variable occuring in σ. Define *tree* Xσ to be the tree which results from performing node splitting on the directed acyclic graph comprising all nodes and edges reachable from *node* Xσ. For example if σ is shown in Figure 6(a) then *tree* Zσ is the tree labeled Z shown in Figure 6(b).

Theorem Let assign be any SL assignment statement (or, equivalently, and SUSL assignment statement other than a selective updating operation) and σ an acyclic SUSL store with variables $X_1,\ldots,X_n$. Then

$$\mathfrak{AS}_{SL}[\![\text{assign}]\!]\ \{X_1 \mapsto \textit{tree}\ X_1\sigma,\ldots,X_n \mapsto \textit{tree}\ X_n\sigma\}$$
$$= \{X_1 \mapsto \textit{tree}\ X_1(\mathfrak{AS}_{SUSL}[\![\text{assign}]\!]\sigma),\ldots,$$
$$X_n \mapsto \textit{tree}\ X_n(\mathfrak{AS}_{SUSL}[\![\text{assign}]\!]\sigma)\}$$

where $\{a_1 \mapsto b_1,\ldots,a_n \mapsto b_n\}$ denotes the finite function f: $\{a_1,\ldots,a_n\} \mapsto \{b_1,\ldots,b_m\}$ satisfying $f(a_i) = b_i$ for i = 1,...,n.

Thus the two languages are semantically equivalent if we ignore the selective updating operation. We omit the proof of this since it just amounts to showing that the usual LISP implementation strategy is valid.

Define a node in a STORE graph to be *shared* if there are two or more distinct paths from variables (or possibly from the same variable) to the node, and to be *cyclic* if it is contained within a cycle in the graph.

VII. Modeling the Sharing Semantics

As is usual in flow analysis, our approach is to define a system which is finite and whose solution in effect symbolically executes the program in parallel over all possible execution paths. The structures just described may grow unboundedly in two ways: in depth (i.e. path length from a variable to a leaf); and there may be an unbounded number of inaccessible (garbage) nodes. To remedy this we discard inaccessible nodes, and consider only bounded approximations to the graphs (annotated with sharing and circularity information to aid in the reference count analysis).

Define a directed graph to be k-*limited* if each node is accessible from a node labeled with a variable by a selector-labeled path of length $\leq k$. Then the flow analysis lattice Share is the set of all sets of directed k-limited graphs of the following form:

1. each variable X $\in$ Var labels one node, denoted *node* X, and each node may be labeled by one or more variables
2. there are two sorts of nodes: *unknown*, labeled ? and *known*, not labeled ?(an unknown node represents a set of nodes whose internal structure is not represented in the k-limited approximation)
3. unknown nodes may be labeled with either of the following (and possibly a variable):
   - s indicating that the unknown structure represented by the node may contain sharing
   - c indicating that the unknown structure may contain a cycle
4. each leaf is labeled with o (indicating an atom), $\perp$ (indicating the null tree), or ? and possibly s or c
5. each known node has one outgoing solid edge $\xrightarrow{sel}$ for each sel $\in$ Sel
6. each unknown node may have any number of outgoing unlabeled dotted edges --->, each going to a different node

The lattice operations are set union (join) and intersection (meet).

Given a fixed set of selectors and a fixed set of variables, the number of k-limited graphs with no inaccessible nodes is clearly finite. As an example of a k-limited graph, consider the 3-limited graph in Figure 7.

A node in a Share graph is defined to be *shared* if there are two or more distinct paths from variables (or possibly the same variable) to the node, or if it is accessible from a node labeled s, or if it is itself labeled c. It is *cyclic* if it is included in a cycle or labeled c.

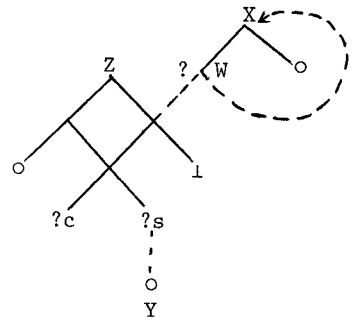


Figure 7

VIII. Constructing the Data Flow Equations

Our approach is to associate with each program point I a set of k-limited graphs F(I), each graph modelling a store resulting from one or more execution paths. Consider the flowchart segments of Figure 2.

The equations are formed as follows, where

$$D = F(I_1) \cup \ldots \cup F(I_p)$$

The functions *clean* and *next* are defined below.

Form of S	Equation
assignment	$F(J) = \bigcup_{\bar{\sigma} \in D} clean(next[\![S]\!]\bar{\sigma})$
if atom var	$F(J_{yes}) = \{\bar{\sigma} \in D \mid$ *node* var $\bar{\sigma}$ is labeled o or ?$\}$ $F(J_{no}) = \{\bar{\sigma} \in D \mid$ *node* var $\bar{\sigma}$ is unknown, or is known and not labeled o$\}$
if null var	$F(J_{yes}) = \{\bar{\sigma} \in D \mid$ *node* var $\bar{\sigma}$ is labeled $\perp$ or ?$\}$ $F(J_{no}) = \{\bar{\sigma} \in D \mid$ *node* var $\bar{\sigma}$ is unknown, or is known and not labeled $\perp\}$
if $var_1 = var_2$ *if* $var_1 \neq var_2$	$F(J_{yes}) = F(J_{no}) = \{\bar{\sigma} \in D \mid$ *node* $var_1\bar{\sigma}$ and *node* $var_2\bar{\sigma}$ are both labeled o or both labeled $\perp\}$
other	$F(J) = D = F(I_1) \cup \ldots \cup F(I_p)$

To define *clean* and *next*, let XShare be the set of all sets of graphs satisfying conditions 1 through 6 of the definition of Share; however, they need not be k-limited. The functionalities are now *next*: Assign $\rightarrow$ Share $\rightarrow$ XShare and *clean*: XShare $\rightarrow$ Share. The idea is that *next* applies the statement, and *clean* makes the resulting graph(s) k-limited.

The function *node* can be carried over to the graphs in Share and XShare naturally.

We now explain how to compute *next* $[\![S]\!]\ \bar{\sigma}$ for an assignment statement S and set of graphs $\bar{\sigma}$.

First,

$$next\ [\![S]\!]\ \bar{\sigma} = \bigcup_{\bar{\gamma} \in \bar{\sigma}} next\ [\![S]\!]\ \{\bar{\gamma}\}$$

If $\{\bar{\gamma}\} \in$ Share, *next* $[\![S]\!]$ $\{\bar{\gamma}\}$ will normally consist of one graph, obtained by modifying $\bar{\gamma}$ as described in the table below. However *next* $[\![S]\!]$ $\{\bar{\gamma}\}$ will be empty if S aborts, and may have more than one element if a variable is moved to a descendent of an unknown node.

Form of S	*next* $[\![S]\!]$ $\{\bar{\gamma}\}$
var := atom or var := *input*	Add a new leaf labeled ○ to $\bar{\gamma}$; Move var to the new node
var_1 := var_2	Move var_1 to label *node* $var_2\bar{\gamma}$
var_0 := *cons*(var_1, ...,var_m)	Make a new node n and move var_0 to label it; for i = 1,...,m add an sel_i edge from n to *node* $var_i\bar{\gamma}$
var_1.sel := var_2	*case* *node* $var_1\bar{\gamma}$ has an sel edge: replace it by an sel edge leading to *node* $var_2\bar{\gamma}$ *node* $var_1\bar{\gamma}$ is labeled ○ or ⊥: *next* $[\![S]\!]$ $\{\gamma\} = \phi$ *node* $var_1\bar{\gamma}$ is unknown: Add an edge -----> from *node* $var_2\bar{\gamma}$ (if not present)
var_1 := var_2.sel	*case* *node* $var_2\bar{\gamma}$ has an sel descendent n: move var_1 to node n *node* $var_2\bar{\gamma}$ is known but has label ○ or ⊥: *next* $[\![S]\!]$ $\{\bar{\gamma}\} = \phi$ *node* $var_2\bar{\gamma}$ is unknown with immediate descendants $n_1,...,n_r$: *next* $[\![S]\!]$ $\{\bar{\gamma}\} = \{\bar{\gamma}_0,\bar{\gamma}_1,...,\bar{\gamma}_r\}$ where: $\bar{\gamma}_0,\bar{\gamma}_1,...,\bar{\gamma}_r$ are $\bar{\gamma}$ with var_1 moved to nodes *node* $var_2\bar{\gamma}$, $n_1,...,n_r$ respectively.

In Figure 8 we illustrate *next* $[\![S]\!]$ $\bar{\sigma}$ for several statements S, where $\bar{\sigma}$ contains only the single 3-limited graph of Figure 7.

The purpose of the function *clean* is to restore the k-limited character of the graphs in *next* $[\![S]\!]$ $\bar{\sigma}$. We first define for any XShare $\bar{\gamma}$, a subgraph $\mathcal{G}(\bar{\gamma})$ which consists of all nodes n which are not accessible from any variable-labeled node by a selector-labeled path of length k - 1 or less, together with all edges in $\bar{\gamma}$ between such nodes.

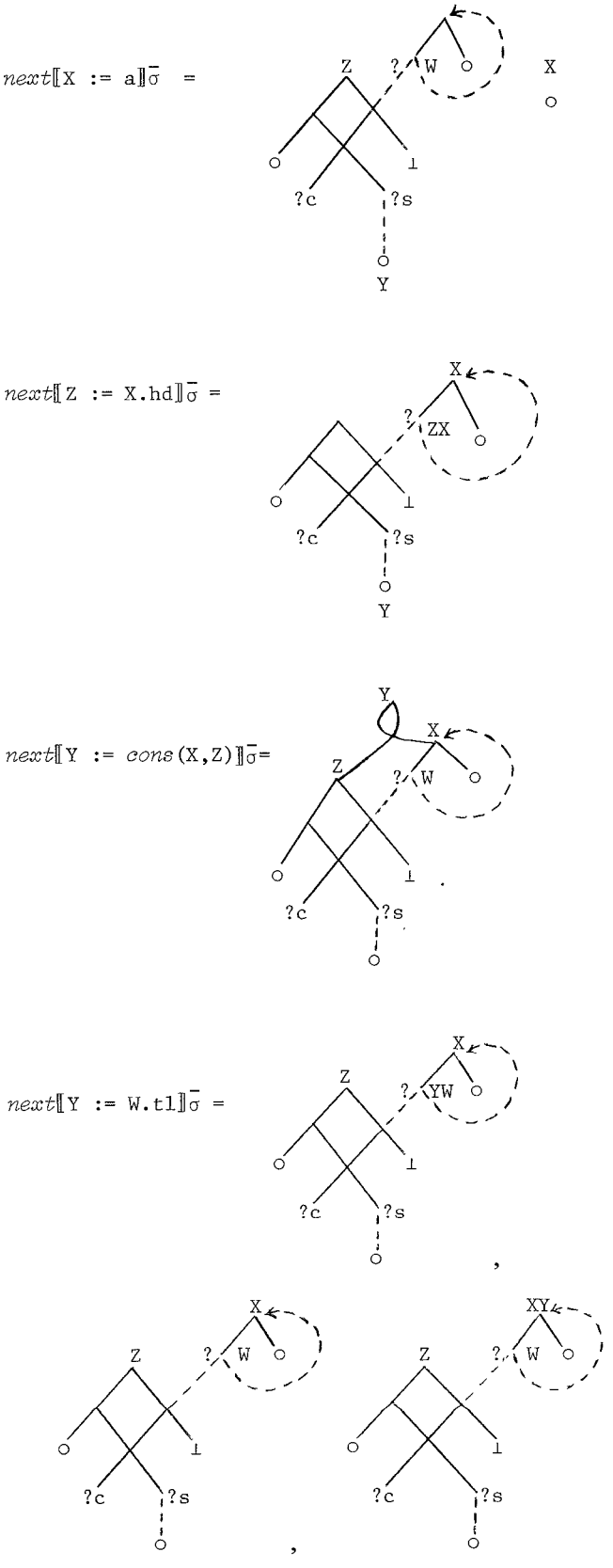


Figure 8

Now *clean* is the set of k-limited graphs resulting from applying the following transformation to each graph $\bar{\gamma}$ in $\bar{\sigma}$:

1. Remove all nodes which are inaccessible from variables
2. Partition $\mathfrak{C}(\bar{\gamma})$ into strongly connected components $C_1, C_2, \ldots$
3. *for* each C_i *do*
   if C_i contains at least one edge
   then coalesce C_i into a single unknown node, labeled c
4. Let the resulting graph be called $\bar{\gamma}'$. Partition $\mathfrak{C}(\bar{\gamma}')$ into undirected connected components $C_1', C_2', \ldots$
5. *for* each C_i' *do*
   if C_i' contains more than one node
   then coalesce C_i' into a single unknown node $\bar{n}$;
   if C_i' contains a node labeled c
   then label $\bar{n}$ with c
   else if C_i' contains a shared node or a node labeled s
   then label $\bar{n}$ with s

The "coalescing" operation above is done by merging the nodes of C into a single node $\bar{n}$, preserving incoming and outgoing edges and variable labels within C. More precisely,

1. Create a new node $\bar{n}$
2. Label it with ? and with all variables labeling nodes in C
3. Redirect any edge coming into C to point to $\bar{n}$
4. Replace any edge coming out of C by a dotted edge from $\bar{n}$ ---> to the same endpoint, provided such an edge does not already exist
5. Delete all nodes of C and edges between them.

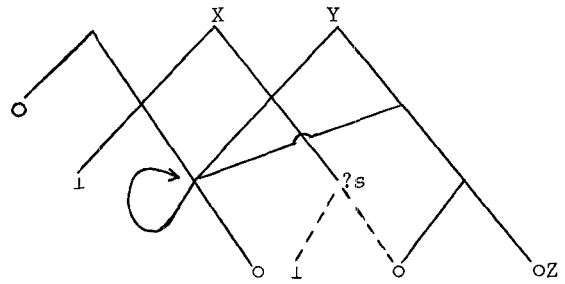
As an example of *clean*, suppose we start with the graph in Figure 9(a). Steps 1 through 5 result in the graphs in Figure 9(b) through (f), assuming the resulting graph is to be 2-limited.

Note that our comments about backward flow analysis in Section III apply here as well.

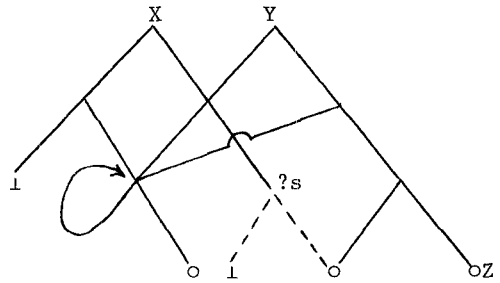
IX. Solution of the Flow Equations

Note that Share is finite. It is not hard to see that F( ) is monotonic, so the minimal fixed point solution may be obtained by regular or chaotic iteration.

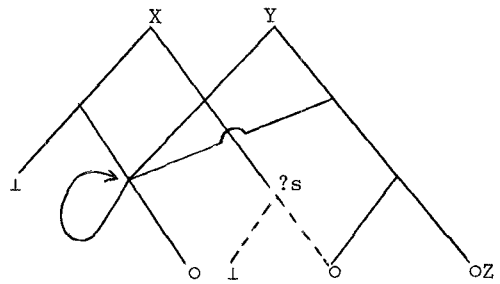
As an example of the kind of information that can be obtained from the equations, suppose no node accessible from *node* $X\sigma$ in any Share graph in the solution is in a cycle or labeled c. Then no node accessible from X in any computation can be part of a cycle, so the descendants of X need not be managed by garbage collection. Similar remarks apply to sharing: a non-shared node can be deallocated as soon as any reference to it is destroyed.



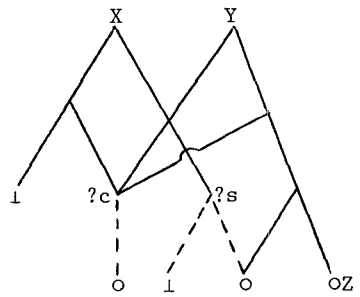
(a)



(b)

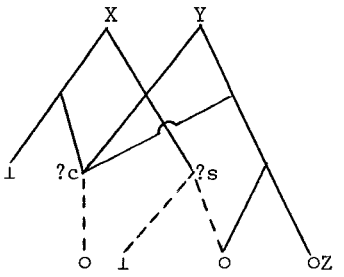


(c)

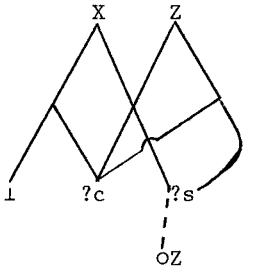


(d)

Figure 9



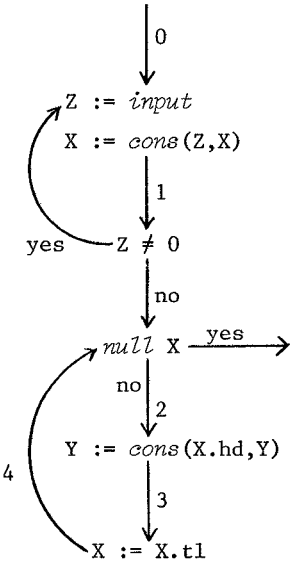
(e)



(f)

Figure 9 (continued)

Theorems establishing these facts will be proved in the next section. First we give an example of the flow equations and their solution, using again the program in Figure 4. For illustrative purposes we compress this program a bit and insert a few program points to obtain the flowchart and forward flow equations in Figure 10.



$$F(0) = \left\{ \begin{matrix} X & Y \\ \perp & \perp \end{matrix} \right\}$$

$$F(1) = F(0) \cup \overline{\mathfrak{AS}}[\![X := \overset{o}{\underset{X}{\wedge}}]\!]\ F(0)$$

$$F(2) = F(1) \cup F(4)$$

$$F(3) = \overline{\mathfrak{AS}}[\![X := \overset{X.hd}{\underset{o}{\wedge}}\ \overset{Y}{\underset{o}{\wedge}}]\!]\ F(2)$$

$$F(4) = \overline{\mathfrak{AS}}[\![X := X.tl]\!]\ F(3)$$

where $\overline{\mathfrak{AS}} = clean \circ next$

Figure 10

To solve the equations we proceed by the method of chaotic iterations, iterating F(1) to stability and then in turn iterating F(2), F(3) and F(4) until the whole system stabilizes. The solution for F(1) and F(2) with k = 2 is indicated by the table in Figure 11. No shared or cyclic structures occur, so the simplest storage allocation method may be used. Further, X := X.tl frees one cell which can be used immediately by the Y := *cons*(X.hd,Y).

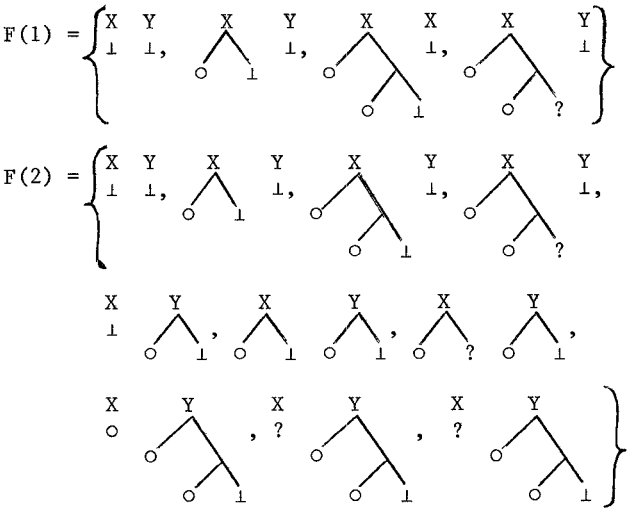


Figure 11

X. Theorems on Detection of Sharing and Cycles

We show in this section that the Share model is capable of detecting any sharing or cycling which may occur in the data structures of a SUSL program. Of course, since the model is finite and based on conservative assumptions, it may also indicate the possibility of sharing or cycling where none occurs in the actual program.

To state the results we first need to define a compatibility relation between STORE and Share graphs which will embody the intuitive notion that, if a STORE σ results from a SUSL computation leading to program point I, then the set of Share graphs F(I) contains a graph γ representing σ. For example, for the STORE graph in Figure 12(a), the Share graph in (b) is compatible, while that in (c) is not.

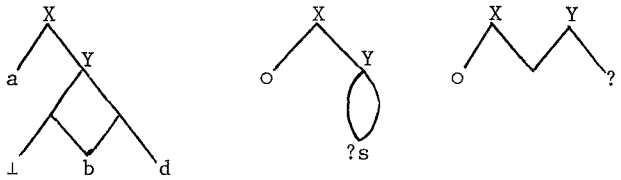


Figure 12

Let *nodes*(σ) be the set of nodes in graph σ. By definition an *admissible node correspondence* from σ to $\bar{\gamma}$ is any function Γ: *nodes*(σ) → *nodes*($\bar{\gamma}$) such that

1. Γ(*node* Xσ) = *node* X$\bar{\gamma}$ for all X ∈ Var
2. Let there be an edge in σ from n to n_1 with selector label sel. Then
   (a) if Γ(n) is known, there is an sel-labeled edge in $\bar{\gamma}$ from Γ(n) to Γ(n_1)
   (b) if Γ(n) is unknown, either Γ(n) = Γ(n_1) or there is a dotted edge from Γ(n) to Γ(n_1)

Further, σ and $\bar{\gamma}$ are *compatible* (written σ ~ $\bar{\gamma}$) iff

1) there is an admissible node correspondence Γ from σ to $\bar{\gamma}$
2) if node n is shared in σ then either Γ(n) is shared in $\bar{\gamma}$ or Γ(n) is accessible in $\bar{\gamma}$ from a node labeled s or c
3) if node n is contained in a cycle in σ then either Γ(n) is contained in a cycle in $\bar{\gamma}$ or Γ(n) is labeled c

Thus it is easy to see that the graph in Figure 12(c) is not compatible with that in Figure 12(a) because, among other reasons, the tail descendant of *node* Xσ is *node* Yσ, while the tail descendant of *node* X$\bar{\gamma}$ is not *node* Y$\bar{\gamma}$.

We next show that the transition functions $\mathfrak{AS}(\ )$ and $\overline{\mathfrak{AS}}(\ )$ preserve compatibility.

Theorem: Let assign be an SUSL assignment statement, σ a STORE graph and $\bar{\gamma}$ a Share graph such that σ ~ $\bar{\gamma}$. Then there exists $\bar{\gamma}'' \in \overline{\mathfrak{AS}}[\![\text{assign}]\!]\{\bar{\gamma}\}$ such that $\mathfrak{AS}[\![\text{assign}]\!]\sigma \sim \bar{\gamma}''$.

Proof: The proof (omitted for brevity) compares the effects of $\mathfrak{AS}$, *next* and *clean* on σ and $\bar{\gamma}$, by an enumeration of cases, to show that the diagram in Figure 13 commutes.

In the diagram σ' = $\mathfrak{AS}[\![\text{assign}]\!]\sigma$, Γ is the admissible node correspondence given by σ ~ $\bar{\gamma}$, $\bar{\gamma}'$ is a graph in *next* [[assign]]{$\bar{\gamma}$}, $\bar{\gamma}''$ = *clean* $\bar{\gamma}'$, and Γ' and Γ'' are appropriate admissible node correspondences. □

To relate the above theorem to the identification of situations where storage management methods simpler and more efficient than gargabe collection can be used, we first define *graph* X$\bar{\gamma}$ for a variable X and a Share graph $\bar{\gamma}$ to be the subgraph of $\bar{\gamma}$ comprising all nodes reachable from *node* X$\bar{\gamma}$ and all edges between them. We then have the following two corollaries.

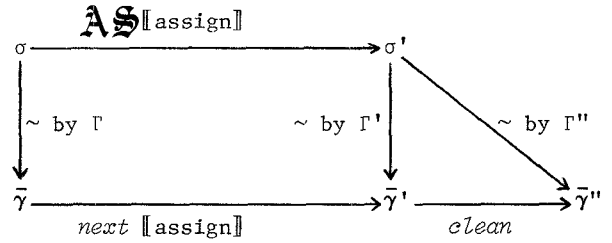


Figure 13

Corollary If *graph* X$\bar{\gamma}'$ contains no shared nodes for all $\bar{\gamma} \in F(I)$ for all program points I, then variable X requires neither reference counting nor garbage collection, i.e., any node reachable from the root of X may be deallocated immediately when a reference to it is removed.

Proof By the theorem, all sharing which can occur in the STORE semantics is recognized in the compatible Share graphs. If no node in *graph* X$\bar{\gamma}$ is ever shared in any possible $\bar{\gamma}$, then nodes reachable from X can never be shared. □

Corollary If *graph* X$\bar{\gamma}$ contains no cyclic nodes for all $\bar{\gamma} \in F(I)$ for all I, then variable X requires no garbage collection, i.e., management of storage for nodes reachable from the root of X may be done by reference counting.

Proof Similar to that for preceding corollary. □

The above results are global in two respects--they concern the behavior of a variable throughout the execution of a program and concern all nodes reachable from it. The information present in the F(I) is sufficient, however, to obtain results which are local in both senses. This could be applied to make very efficient use of a dynamic storage management system in which cells are divided into three types: those which are immediately deallocated, those which are reference counted and those which are garbage collected. Then a particular cell which can be identified as never being shared in the future can be allocated as the first type, one which may be shared but will never be cyclic as the second type, and the remainder as the last type. We leave the detailed development and analysis of this approach to later work.

It should be noted in closing that Baker [1] has studied the situation in which reference counting is an appropriate method for dynamic storage management. His findings indicate that our intended use is an appropriate one.

REFERENCES

1. Baker, Henry G., Jr., List Processing in Real Time on a Serial Computer, *CACM*, vol. 21, no. 4, 1978, pp. 280 - 294.

2. Barth, J. M., Shifting Garbage Collector Overhead to Compile Time, *CACM*, vol. 20, no. 7, 1977, pp. 513 - 519.

3. Brainerd, W. S., Tree Generating Regular Systems, *Information and Control*, vol. 14, 1969, pp. 217 - 231.

4. Büchi, J. R., Regular Canonical Systems, *Archiv f. Math. Logik und Grund.*, vol. 6, 1964, pp. 91 - 111.

5. Clark, D. W. and C. C. Green, An Empirical Study of List Structure in LISP, *CACM*, vol. 20, no. 2, 1977, pp. 78 - 87.

6. Cousot, Patrick and Rhadia Cousot, Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations, Proc. ACM Symp. on Artif. Intel. and Prog. Lang., *SIGPLAN Notices*, vol. 12, no. 8, August 1977, pp. 1 - 12.

7. Cousot, Patrick and Rhadia Cousot, Static Determination of Dynamic Properties of Generalized Type Unions, *SIGPLAN Notices*, vol. 12, no. 3, March 1977, pp. 77 - 94.

8. Engelfriet, J., Tree Automata and Tree Grammars, DAIMI Report FN-10, Department of Computer Science, University of Aarhus, Denmark, 1975.

9. Ginsburg, Seymour, *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York, 1966.

10. Goto, E., Monocopy and Associative Algorithms in an Extended LISP, University of Tokyo, Japan, May 1974.

11. Hoare, C. A. R., Recursive Data Structures, *Inter. J. Comp. and Sys. Sci.*, vol. 4, no. 2, 1975, pp. 105 - 132.

12. Jones, N. D. and S. S. Muchnick, Binding Time Optimization in Programming Languages: Some Thoughts Toward the Design of an Ideal Language, *Proc. 3rd ACM SIGACT - SIGPLAN Symp. on Princ. of Prog. Lang.* January 1976, pp. 77 - 94.

13. Kaplan, Marc, Relational Data Flow Analysis, Technical Report 243, Dept. of Elec. Eng. and Comp. Sci., Princeton University, April 1978 (revised).

14. Kaplan, Marc and J. D. Ullman, A General Scheme for the Automatic Inference of Variable Types, *Conf. Record of 5th ACM Symp. on Princ. of Prog. Lang.*, Tucson, AZ, January 1978, pp. 60 - 75.

15. Milne, Robert and Christopher Strachey, *A Theory of Programming Language Semantics*, Chapman and Hall, London; Halsted Press, John Wiley, New York, 1976.

16. Reynolds, John C., Automatic Computation of Data Set Definitions, *Proc. of IFIP Congress 68*, August 1968, pp. B69 - B73.

17. Schwartz, J. T., Optimization of Very High Level Languages - I: Value Transmission and Its Corollaries, *Computer Languages*, vol. 1, 1975, pp. 161 - 194.

18. Stoy, Joseph E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.

19. Thatcher, J., Tree Automata: An Informal Survey, in Aho, Alfred (ed.), *Currents in the Theory of Computing*, Prentice-Hall, 1973, pp. 143 - 172.