CHAPTER 3

SYMBOLIC ANALYSIS OF PROGRAMS WITH STRUCTURED DATA

3.0 Summary

We discuss the symbolic analysis of a class of programs such as those of LISP 1.0, which have a fixed interpretation for various operations on structured data including: operations for construction of structured objects (such as *cons* in LISP) and the selection of subcomponents (such as *car* and *cdr* in LISP), but no "destructive" operations (such as *replaca* or *replacd* in LISP 1.5). We continue to use the global flow model of Chapter 1, in which assignment statements are the only variety of statements and the program flow graph represents the flow of control.

A central problem here is the *propagation of selections*: the determination of the set SP of ordered pairs of selection operations and the objects which they may reference. The elements of SP are called *selection pairs*. We show that this propagation problem is at least as hard as transitive closure of a binary relation and we give an efficient algorithm, using bit vector operations, for computing SP. Schwartz[Sc2] requires the set SP for his method for the automatic construction of recursive type declarations, though he gave no explicit algorithm for propagating selections.

We consider further applications of propagation of selections including: the determination of selection operations that, when executed, always result in an error (i.e., they attempt to access non-existant subcomponents), the propagation of constants, and more generally the determination of covers (symbolic representations of values of text holding for all executions of the program). The methods of Chapter 2 for the determination of covers are improved so as to take into account reductions due to the selection of subcomponents of structured objects.

We apply these improved methods also to the construction of _type_ _covers_ which are representations of types (rather than values) of text expressions and hold for all executions of the program. Type covers are useful for the discovery of construction operations which are redundant in the sense that they have values of the same _type_ as values previously computed but are now dead (no longer referenced).

Finally, we discuss Schwartz's method of recursive type determination.

3.1 Introduction

This Chapter is concerned with the analysis of programs which manipulate structured data; for example:

the lists of LISP

the strings of SNOWBALL

and the arrays in FORTRAN, ALGOL, and PL/1.

Though we allow general operations for construction of structured objects and selection of subcomponents, our analysis is restricted to programs with no destructive operations: they must not modify subcomponents (i.e., install new sublists, insert or delete new characters of strings, or modify elements of arrays). Hence our methods are only applicable to a restricted subclass of the above programming languages with list, string, and array data structures. We believe our methods can be extended (with a certain increase in time and space cost) to programs which allow modification of subcomponents. (At any rate, there exist certain simple programing languages, such as LISP 1.0, which do not allow modification of subcomponents.)

As in the preceding Chapter, we discuss the analysis of a program P relative to a global flow model in which the flow of control is represented by the control flow graph F = (N, A, s) where the nodes of N correspond to contiguous sequences of assignment statements called blocks, the edges in A specify possible flow of control between blocks in N,

and all control flow begins at the <u>start block</u> $s \in N$.

Let $\Sigma = \{X,Y,Z,\ldots\}$ be the non-local program variables of P. For each $X \in \Sigma$ and block $n \in N-\{s\}$, we introduce the <u>input variable</u> $X^{\to n}$ denoting the value of X on input to block n. Also, for each $X \in \Sigma$, $X^{\to s}$ is a distinct constant sign denoting the value of X on input to the program P at the start block s. As in Chapter 1, we require a first order language without predicates to represent computations of P and their covers. Let EXP be the set of expressions built from input variables, and fixed sets of <u>constant signs</u> C and k-adic <u>function signs</u> Θ; here Θ is partitioned into the sets:

(1) OP, a set of <u>operator signs</u> used for elementary operations on atomic values,

(2) CONS, a set of <u>constructor signs</u> used to build up structured values,

(3) SEL, a set of 1-adic <u>selector signs</u> used to select subcomponents of structured values.

A <u>function application</u> is an expression of the form

$$\alpha = (\theta\ \alpha_1,\ldots,\alpha_k),$$

where θ is a k-adic function sign in Θ and $\alpha_1,\ldots,\alpha_k \in$ EXP. α is an <u>elementary operation</u> if θ is an operator sign op $\in$ OP, α is a <u>construction operation</u> if θ is a constructor sign cons $\in$ CONS, and α is a <u>selection operation</u> if $k = 1$ and θ is a selector sign sel $\in$ SEL. For each k-adic constructor

sign cons ϵ CONS and i, $1 \leq i \leq k$, there exists an unique selector sign sel ϵ SEL called the <u>i</u>th <u>selector of cons</u>.

As described in the examples below, in LISP there is a simple constructor (<u>cons</u>), two selectors (<u>car</u> and <u>cdr</u>), and elementary operations depending on the particular version of the language (none in LISP 1.0, arithmetic and logical operations in LISP 1.5).

SELECT(sel,α) gives the result of selection by a selector sign sel ϵ SEL on expression α ϵ EXP:

(1) if α is a construction operation

$$(\text{cons } \alpha_1 \ldots \alpha_k)$$

where sel is the ith selector of constructor sign cons, then SELECT(sel,α) = α_i.

(2) If α is a construction operation for which sel is <u>not</u> a selector, or α is a constant sign, or α is an elementary operation then SELECT(sel,α) = <u>error</u>, where <u>error</u> is a distinguished constant sign in C denoting an error condition.

(3) In all other cases (e.g., where α is itself a selection or an input variable) SELECT(sel,α) is left undefined.

For example, in LISP,

SELECT(cdr, (cons α_1 α_2)) = α_2.

We assume an <u>interpretation</u> (U, I) such that

(1) U is an <u>universe</u> of values consisting of

(a) ATOM, a set of atomic values (<u>atoms</u>), and

(b) <u>structured values</u> constructed by prefixing k-adic constructor signs in CONS to k-tuples in the universe U.

(2) I is a homomorphic mapping from EXP to U such that

(a) For each constant sign $c \in C$, $I(c) \in$ ATOM. We assume the constant signs in C are in one-to-one correspondence to atoms in ATOM. The distinguished constant sign <u>error</u> is also an atom and is freely interpreted: I(<u>error</u>) = <u>error</u>.

(b) For each k-adic function sign $\theta \in \boldsymbol{\theta}$, $I(\theta)$ is a partial mapping from U^k to U.

(i) Each k-adic operator sign op $\in$ OP is interpreted as a mapping I(op) from k-tuples of atoms into individual atoms (note that such mappings may <u>not</u> take structured objects as arguments).

(ii) k-adic constructor signs cons $\in$ CONS are freely interpreted:

$$I(cons)(z_1,\ldots,z_k) = (cons\ z_1,\ldots,z_k)$$

for all $z_1,\ldots,z_k \in U$.

(iii) Each selector sign sel $\in$ SEL is interpreted to map from expressions in the universe U to their corresponding subexpressions, or where this is not possible, to <u>error</u>. More formally, for each $\alpha \in$ EXP such that SELECT(sel,α) is defined:

$$I(sel)(I(\alpha)) = I(SELECT(sel,\alpha)).$$

Example 3A (LISP 1.0)

ATOM = {the empty list nil}

OP = the empty set {}

CONS = {the list constructor cons}

SEL = {car, the first selector of cons}

$\cup$ {cdr, the second selector of cons}

Example 3B

(similar to LISP 1.5 but without replaca and replacd)

ATOM = {the empty list nil}

$\cup$ {the integers}

$\cup$ {the boolean truth values truth and false}

OP = {and, or, plus, minus, mult, and div}

and and or are interpreted as logical conjuncton and disjunction; the other operator signs in OP are interpreted as the usual arithmetic operations. CONS, SEL are as in Example 3A.

Example 3C (Vectors of fixed length)

ATOM = $\{a_1, a_2, \ldots\}$

SEL = {the positive integers}

CONS = $\{vector^1, vector^2, \ldots\}$

where $vector^k$ is a k-adic constructor sign and the integer i is the ith selector of $vector^k$ for each $1 \leq i \leq k$. Note that the number of function places of each $vector^k$ is fixed; so it is not possible to construct variable length sequences. However we can easily extend the model to allow function signs with a variable number of arguments.

In Chapter 1 we defined a <u>constant reduction</u> on an expression in EXP to be the repeated substitution of constant signs for constant subexpressions (relative to a fixed interpretation); that is if $\alpha \in$ EXP contains a elementary operation

$$\alpha' = (\mathrm{op}\ c_1 \ldots\ c_k)$$

where op $\in$ OP and $c_1,\ldots,c_k \in$ C and there exists a constant sign $c \in$ C such that

$$I(c) = I(\mathrm{op})(I(c_1),\ldots,I(c_k)),$$

then we substitute c in the place of α'.

In addition, we define <u>selection reductions</u> to be the result of substituting SELECT(sel,α'), where it is defined, for each selection operation (sel α').

An expression is <u>reduced</u> by repeated constant and selection reductions.

For each program variable $X \in \Sigma$ defined (i.e., assigned to) at block $n \in N$, the <u>output variable</u> $X^{n\rightarrow}$ is a reduced expression in EXP for the value of X on <u>exit</u> from block n in terms of the constants and input variables at n. For example, $Y^{n\rightarrow}$ = (cons (car $X^{\rightarrow n}$) $Y^{\rightarrow n}$) in the program of Figure 3.1.

The <u>text expressions</u> of P are the output variables and their subexpressions. We assume the text expressions are reduced expressions. For each reduced expression $\alpha \in$ EXP

and control path p, EXEC(α,p) is intuitively a reduced expression in EXP for the value of α relative to p. For a more precise definition, see Section 1.3.

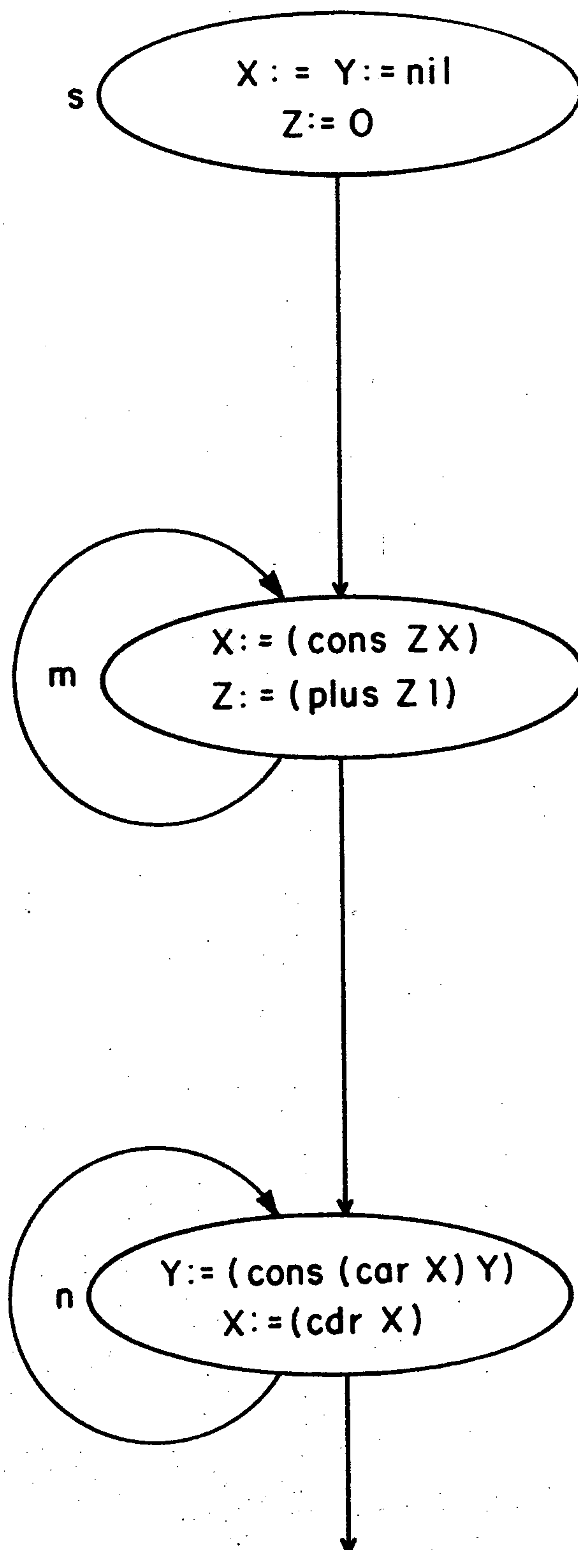

Figure 3.1. Reversal of a list in LISP.

3.2 Propagation of Selections

Our immediate goal is to determine all "*selection pairs*." Loosely speaking, these are pairs (w,u) where w is a selection operation (sel t) and u is a text expression whose value, relative to some execution of the program, may be obtained from the value of t by the use of the selector sign sel. More precisely, for text expressions t and t', let t' be *accessible from* t if EXEC(t,p) = t' for some control path p from loc(t') to loc(t). Note that selection sequences are generalizations of the value paths of Chapter 2. A *selection pair* is an ordered pair of text expressions (w,u) consisting of a selection w = (sel t) and u = SELECT(sel,t'), where SELECT(sel,t') is defined for some text expression t' accessible from t. We assume that the constant sign *error* is a text expression located at the start block, so t has a departing selection pair (t,*error*) if EXEC(t,p) = *error* for some control path p from s to loc(t). We also assume there are no selections at the start block s, so each selection in the text has at least one departing selection pair.

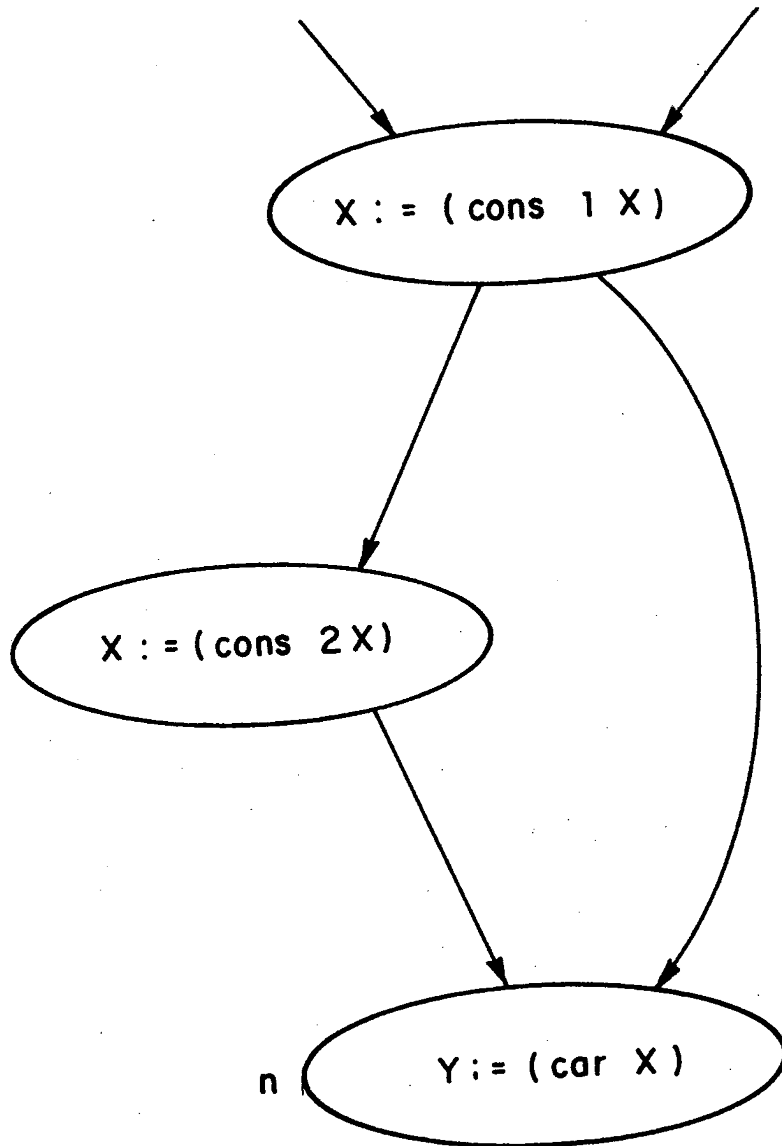

Figure 3.2. $(Y^{n+},1)$ and $(Y^{n+},2)$ are selection pairs.

Selection propagation is the task of discovering all selection pairs.

Theorem 3.2.1 Selection propagation in the interpretation of Example 3A (LISP 1.0) is at least as hard as computing the transitive closure of a binary relation.

Proof Let R be a binary relation on $\{n_1,\ldots,n_r\}$ and let $R^* = \{(n_{i_1},n_{i_k}) \mid (n_{i_1},n_{i_2}),\ldots,(n_{i_{k-1}},n_{i_k}) \in R,\ k \geq 1\}$ be the reflexive transitive closure of R. Consider the control flow graph $F_R = (N, A, s)$ of Figure 3.2 where

$$N = \{s=n_0,n_1,\ldots,n_{3r}\}$$

and the edge set A consists of

(1) R and

(2) for $i=1,\ldots,r$ edges (n_0,n_{r+i}), (n_{r+i},n_i), and (n_i,n_{2r+i}).

Let the text of n_0 be empty.

For $i = 1,\ldots,r$

(1) the text of n_i is empty

(2) the text of n_{r+i} is X := (cons nil X).

(3) the text of n_{2r+i} is the selection X := (cdr X).

It follows that $(n_i,n_j) \in R^*$

iff there is a value path from $X^{\to n_{2r+j}}$ to $X^{n_{r+i}\to}$

iff $(X^{n_{2r+j}\to}, X^{\to n_{r+i}})$ is a selection pair. □

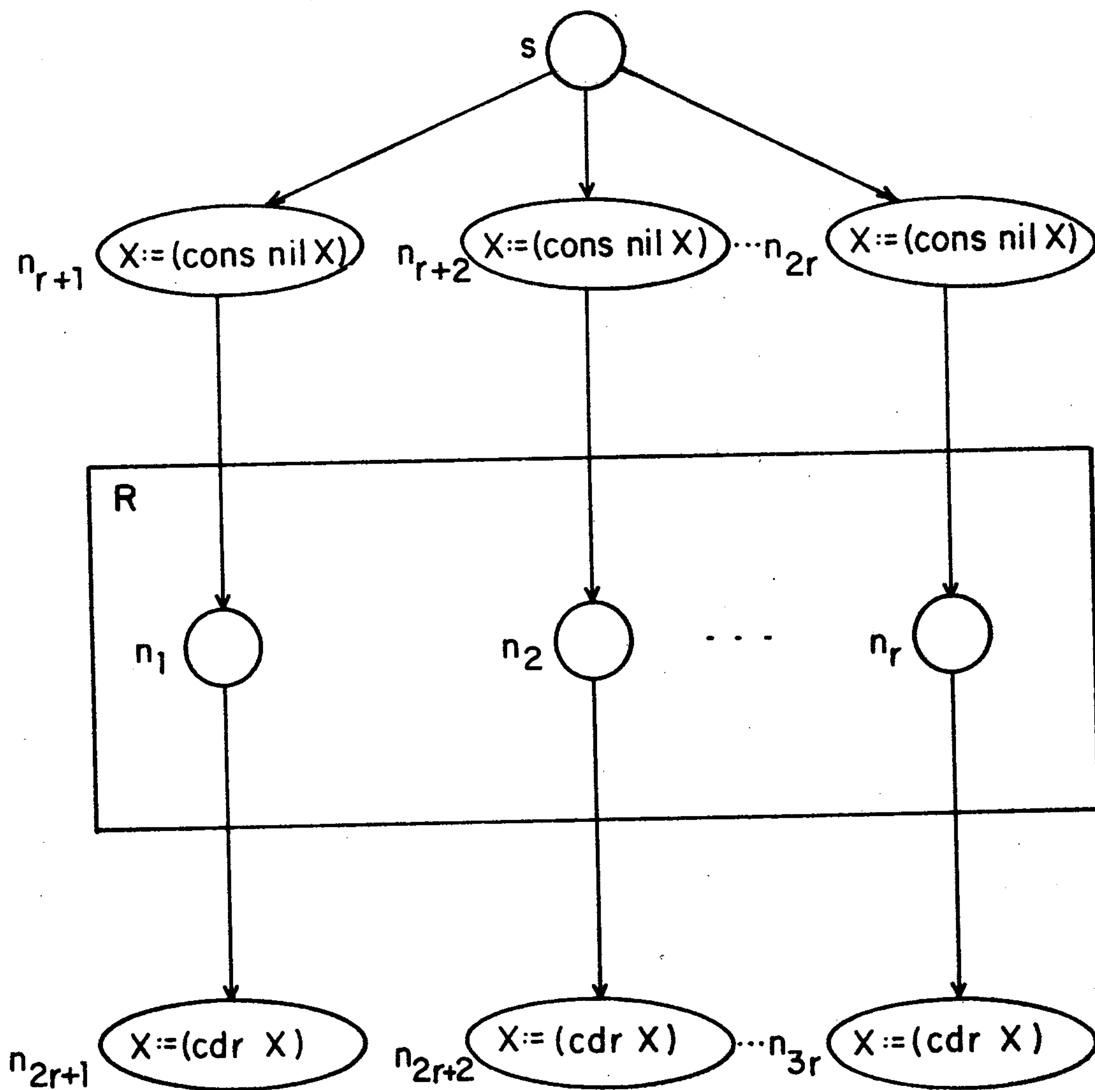

Figure 3.3. The control flow graph F_R.

As in Chapter 2, we use a _dag_ D(n) (an acyclic, oriented digraph) to represent computations local to a linear block of code $n \in N$. Each node of D(n) represents an unique text expression located at block n. A _global value graph_ GVG = (V, E, L) is a possibly cyclic, oriented digraph consisting of

(1) the dags of all the blocks in N, and

(2) a set of edges, called _value edges of GVG_, departing from nodes labeled with input variables. For each node $v \in V$ labeled with an input variable and control path p from the start block s to loc(v), there is a value edge (v,u) such that loc(u) is distinct from loc(v) and is contained in p. (the labeling L is consistent with that of the dags.)

A _value path of GVG_ is a path transversing only nodes linked by value edges.

In Section 2.1 we defined a special global value graph GVG^* = (V, E, L) with value edges defined so as to properly represent the flow of values of program variables between blocks of code; that is (1) the nodes in GVG^* are identified with the text expressions which they represent and (2) (t,t') is a value edge of GVG^* iff t is an input variable $X^{\rightarrow n}$ and t' is the output variable $X^{m \rightarrow}$ for some $(m,n) \in A$. This definition requires that the text expressions include all the input variables; for each input variable $X^{\rightarrow n}$ not originally a text expression at block $n \in N-\{s\}$, add a "dummy" assignment of the form

X := X.

Let V be the set of the resulting new text expressions corresponding to these dummy assignments.

An <u>access sequence</u> is a sequence of text expressions $(t_1,\ldots,t_k)$ such that for $1 \leq i < k$, each (t_i,t_{i+1}) is either a value edge of GVG* or a selection pair.

<u>Theorem 3.2.2</u> For all $t,t' \in V$, there is an access sequence from t to t' iff t' is accessible from t.

<u>Proof</u> Suppose there exists an access sequence $(t=t_1,\ldots,t_k=t')$. Then for $i = 1,\ldots,k-1$ whether (t_i,t_{i+1}) is a value edge or a selection pair, there is always a control path p_i from $loc(t_{i+1})$ to $loc(t_i)$ such that

$$t_{i+1} = EXEC(t_i,p_i).$$

Hence $t' = EXEC(t,p_{k-1}\cdot p_{k-2}\cdot\ldots\cdot p_1)$ so t' is accessible from t.

On the other hand, suppose there is a control path p of minimal length such that there exists text expressions t,t' such that p begins at loc(t') and ends at loc(t) and

$$t' = EXEC(t,p)$$

but there is no access sequence from t to t'. If t is an input variable, t has a departing value edge (t,τ) such that $loc(\tau)$ is distinct from loc(t) and $loc(\tau)$ is contained in p. If t is a selection, then there is a departing selection pair (t,τ) where $loc(\tau)$ is contained in p. In either case if $p = p_1\cdot p_2$ where p_2 is a subsequence of p from $loc(\tau)$ to

loc(t), then by the induction hypothesis

$$t' = EXEC(\tau, p_1)$$

so t' is accessible from τ and by the induction hypothesis there is an access sequence q from τ to t'. Hence, $(t,\tau)\cdot q$ is an access sequence from t to t', a contradiction. □

We now present an efficient algorithm for the discovery of all selection pairs.

Algorithm 3A

INPUT GVG* = (V, E, L) and V̄, the set of added text expressions corresponding to dummy assignments.

OUTPUT SP, the set of selection pairs of P.

```
begin
  declare for each t ε V
    VPt,ASt,ĀSt := sets of maximum size |V| each
      represented as bit vectors of length |V|
      (3|V| sets, initially all empty);
  procedure PROPAGATE(t,t'):
    begin
      add t' to ASt;
      add t to ĀSt';
      add (t,t') to Q;
    end;
  Q := the empty set {};
  Let VE be the edges in E departing from nodes labeled
    with input variables;
  Compute the transitive closure VE* of VE;
  VE* is represented by a family of sets {VPt|t ε V}
    where for t,t' ε V,  t' ε VPt iff there exists
    a value path in GVG* from t to t';
  for all t ε V do
    for all t' ε VPt do
      if t,t' ε V-V̄ then L0: PROPAGATE(t,t');
  untill Q = the empty set {} do
    begin
  L1: Choose some (t,t') ε Q and delete it from Q;
      for every selection w ε V where w = (sel t) do
        if u = SELECT(sel,t') is defined do
          begin
            add (w,u) to SP;
            L2: PROPAGATE(t,u);
          end;
      for all u ε ASt'-ASt do L3: PROPAGATE(t,u);
      for all w ε ĀSt-ĀSt' do L4: PROPAGATE(w,t');
    end;
  return SP;
end;
```

We require two Lemmas to demonstrate the correctness of Algorithm 3A.

<u>Lemma 3.2.1</u> On every execution of Algorithm 3A we have for all $t,t' \in V-\hat{V}$ at label L0:

(i) $t \in AS_{t'}$ iff $t' \in \overline{AS}_t$

(ii) if $(t,t') \in Q$ then $t \in AS_{t'}$

(iii) if $t \in AS_{t'}$ then there exists an access sequence from t to t'.

<u>Proof</u> by induction on the number of executions of the main loop of Algorithm 3A.

<u>Basis step</u> Initially, Q = the set of all pairs (t,t') such that $t,t' \in V-\bar{V}$ and there exists a value path from t to t', (i),(ii) hold by the calls to PROPAGATE(t,t') at L0, and since any value path is also an access sequence, (iii) also initially holds.

<u>Inductive step</u> Suppose (i),(ii), and (iii) have held over previous executions of the main loop of Algorithm 3A, and consider some (t,t') deleted from Q at L1. By (ii) and (iii), there is an access sequence from t to t'. Observe that if there is an access sequence from text expression y to some text expression z, then after any call to PROPAGATE(y,z), (i),(ii), and (iii) still hold, and for our purposes that call is considered <u>correct</u>.

<u>Case 1</u> If w is the selection (sel t) and u = SELECT(sel,t') is defined, then (w,u) is a selection pair, which is also an access sequence. Thus, the call to PROPAGATE(w,u) at L2 is correct.

Case 2 If $u \in AS_{t'} - AS_t$, then (ii) implies that there is an access sequence from t' to u, and hence there is an access sequence from t to u. Thus, the call to PROPAGATE(t,u) at L3 is correct.

Case 3 If $w \in \overline{AS}_t - \overline{AS}_{t'}$ then (iii) implies that $t \in AS_w$ and (iii) implies that there is an access sequence from t to w. Hence, there is an access sequence from w to t' and the call to PROPAGATE(w,t') at L4 is correct. □

Lemma 3.2.2 For all $t,t' \in V$, if there is an access sequence p from t to t' then (t,t') is eventually added to Q.

Proof by contradiction. Suppose (t,t') is not eventually added to Q, and let p be of minimal length. Note that if we have a call to PROPAGATE(t,t') then (t,t') is added to Q.

Case 1 If p is a value path from t to t' then there must be a call to PROPAGATE(t,t') at L0.

Case 2 If p is a selection pair then there exist text expressions y,z such that t is of the form t (sel y), t' = SELECT(sel,z), and furthermore there is an access sequence p' from y to z. Since p' is of length less than p, p' does not violate Lemma 3.2.2, so (y,z) is eventually added to Q, and hence there is a call to PROPAGATE(t,t') at L2.

Case 3 Otherwise $p = p_1 \cdot p_2$ where p_1 is an access sequence from t to y and p_2 is an access sequence from y to t'. Since p_1 and p_2 are of length less than p, Lemma 3.2.2 holds over p_1 and p_2, so both (t,y) and (y,t') are eventually added to (and later deleted from) Q.

Case 3a If (t,y) is deleted from Q after (y,t') then

$t' \in \overline{AS}y-\overline{AS}t$

and so there is a call to PROPAGATE(t,t') at L3.

Case 3b If (y,t') is deleted from Q after (t,y) then

$t \in \overline{AS}y-\overline{AS}t'$

and thus there is a call to PROPAGATE(t,t') at L4. □

Theorem 3.2.3 Algorithm 3A correctly computes SP in $O(\ell^2+|\Sigma||A|)$ bit vector operations, where $\ell = |V-\overline{V}|$ is the the number of original text expressions before the "dummy" assignments are added.

Proof Suppose (t,t') is a selection pair. By Lemma 3.2.2, (t,t') is added to Q at the call to PROPAGATE(t,t') at L2, and hence (t,t') is also added to SP at L2.

Now suppose that (t,t') is added to SP at L2. Then (t,t') is added to Q in the call to PROPAGATE(t,t') at L2, and by the proof of Lemma 3.2.1, (t,t') is a selection pair.

New we consider the lower time bounds of Algorithm 3A. The computation of VP by [T1] costs $O(|V|+|E|) = O(\ell+|\Sigma||A|)$ bit vector steps. Also, the processing associated with each (t,t') added and then deleted from Q is a constant number of bit vector operations. There may be $O(\ell^2)$ such pairs and no such pair is added to Q more than once. Hence, the total cost of Algorithm 3A is $O(\ell^2+|\Sigma||A|)$ bit vector operations. □

3.3 Constant Propagation and Covers of Programs with Structured Data.

Let P be a program with a fixed interpretation for the constructor and selector signs as in the Introduction of this Chapter. Here we wish to determine text expressions which are constant over all executions of P, and more generally we wish to determine covers: symbolic expressions in EXP for the value of text expressions which hold over all executions of the program. The main difference between the covers of this section and those of Chapter 2 is that here we define a *reduced expression* to be derived from repeated selection reductions of the sort described in Section 3.1, as well as the usual constant reductions. A reduced expression $\alpha \in$ EXP *covers* text expression t if

$$\mathrm{EXEC}(\alpha,p) = \mathrm{EXEC}(t,p)$$

for all control paths p from the start block s to loc(t), the block in N where t is located. A *cover of P* is a mapping ψ from the text expressions to EXP such that for each text expression t, $\psi(t)$ covers t.

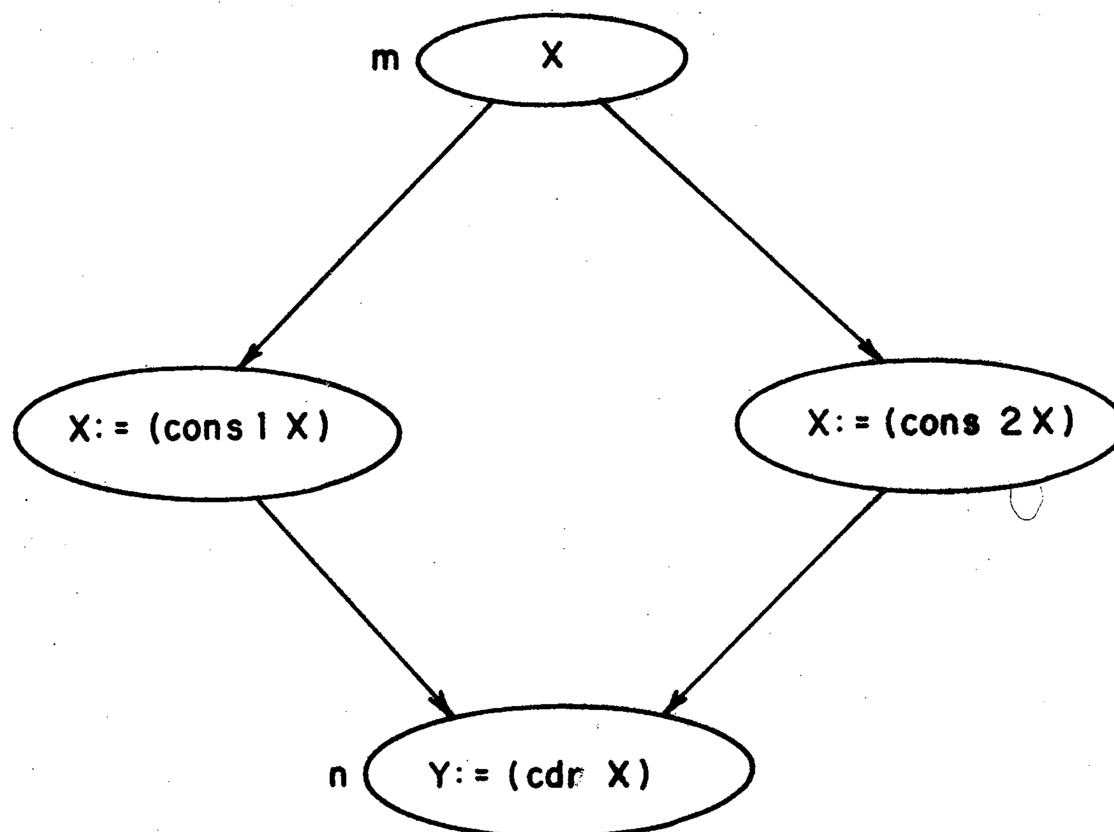

Figure 3.4. $Yn^{\rightarrow}$ = (cdr $X^{\rightarrow}n$) is covered by $X^{\rightarrow}m$.

Recall that the <u>origin</u> of an expression $\alpha \in$ EXP is intuitively the earliest point at which α is defined; formally origin(α) = s if α contains no input variables and otherwise origin(α) is the earliest block n $\in$ N (relative to the dominator ordering of the control flow graph F with the start block s first) such that an input variable $X^{\to n}$ appears in α (provided that this block is uniquely determined). Also, recall that $\overset{*}{\to}$ is the partial ordering of nodes in N by dominator relative of the control flow graph F = (N, A, s). We extend $\overset{*}{\to}$ to a partial ordering of covers. For covers ψ, ψ', $\psi \overset{*}{\to} \psi'$ iff origin(ψ(t)) $\overset{*}{\to}$ origin(ψ'(t)) for each text expression t. It follows from the results of Section 1.3 that if the program P is interpreted in the integer domain (i.e., ATOM is the set of natural numbers and the elementary operator signs in OP are interpreted as the usual arithmetic operations: addition, subtraction, multiplication, and division) then constant propagation is recursively unsolvable, and hence the determination of the covers minimal with respect to $\overset{*}{\to}$ is also impossible within the arithmetic domain.

Good, but not minimal, covers may be computed by an algorithm due to Kildall[Ki] (his algorithm is actually much more general; here we consider a specific application). After computing an approximate cover ψ_0, Kildall's algorithm iteratively compares the approximate covers of input variables to the approximate covers of the output

expressions of the corresponding variables at preceding blocks, and propagates the changes to succeeding blocks. In Chapter 2, we define the covers computed by his algorithm as fixed points of a functional Ψ. Here we define a similar functional Ψ'. For any mapping ψ from text expression to EXP, let $\Psi'(\psi)$ be the mapping from text expressions to EXP such that for each text expression t, $\Psi'(\psi)$ is derived from t by repeatedly

(1) substituting expression α for every input variable $X^{\rightarrow n}$ such that $\alpha = \psi(X^{m\rightarrow})$ for all $(m,n) \in A$.

(2) substituting the expression α for any selection u in t such that $\alpha = \psi(u')$ for all selection pairs (u,u').

(3) reducing (by both selection and constant reductions) the resulting expression.

We shall show, as we did for a similar functional Ψ in Chapter 2, that the fixed points of Ψ' are covers and that there exists a unique, minimal fixed point of Ψ'.

<u>Theorem 3.3.1</u> Each fixed point of Ψ' is a cover.

<u>Proof</u> by contradiction. Suppose ψ is a fixed point of Ψ' and ψ is not a cover. Let p be the shortest control path from the start block s to a block $n \in N$ containing a text expression t such that

$$\text{EXEC}(\psi(t),p) \not\equiv \text{EXEC}(t,p).$$

Furthermore assume for each proper subexpression t' of t,

$$\text{EXEC}(\psi(t'),p) = \text{EXEC}(t',p).$$

The case where t is an input variable was shown (in the

proof of Theorem 2.1) to be an impossible case. Otherwise, $\psi(t) = \psi(t')$ for all selection pairs (t,t'). But there is a selection pair (t,t') such that

$$EXEC(t,p2) = t'$$

for $p = p1 \cdot p2$ where p1 ends and p2 begins at loc(t').

Hence, $EXEC(t,p) = EXEC(t',p1)$

$= EXEC(\psi(t'),p1)$ by the induction hypothesis

$= EXEC(\psi(t),p1)$ since $\psi(t) = \psi(t')$

$= EXEC(\psi(t),p)$. □

Let GVG = (V,E,L) be an arbitrary global value graph as defined in Section 3.2. In Section 2.2 we also defined the set Γ_{GVG} of mappings from the nodes of GVG to EXP such that for each $\psi \in \Gamma_{GVG}$ and node $v \in V$ in GVG,

(1) If v is labeled with a constant sign c then $\psi(v) = c$.

(2) If L(v) is a k-adic function sign θ and $u_1,\ldots,u_k$ are the immediate successors of v in GVG then $\psi(v)$ is the expression derived by <u>constant</u> reductions from $(\theta\ \psi(u_1)\ldots\psi(u_k))$.

(3) If v is labeled with an input variable, then either $\psi(v) = X{\to}n$ or $\psi(v) = \alpha$ where $\alpha = \psi(u)$ for all value edges $(v,u) \in E$ departing from v.

Let Γ'_{GVG} be a set of mappings ψ from V to EXP such that for all $v \in V$, $\psi(v)$ satisfies cases (1), (2), (3), or the additional case

(3') L(v) is a selector sign and $\psi(v) = \alpha$ where $\alpha = \psi(u)$ for

all selection pairs (v,u) departing from v.

Note that the set of nodes satisfying cases (3) and (3') are sufficient to characterize an element of Γ'_{GVG}; and hence Γ'_{GVG} is finite.

Let GVG^* = (V, E, L) be the special global value graph defined in Section 2.2 where each node v ϵ V is identified with the text expression which it represents (hence, the node set V is considered to be the set of text expressions) and the value edges of GVG^* represent the flow of values through the program. Recall that (t,t') is a value edge of GVG^* iff t is an input variable $X^{\to n}$ and t' is the output expression X^m for some (m,n) ϵ A. For any text expression t ϵ V that is a selection, and $\psi \in \Gamma'_{GVG^*}$, if (3') holds for t then t is _simplifed_ by ψ. If in addition, $\psi(t) \neq$ _error_, then t is _properly simplified_ by ψ. Selection t is (_properly_) _simplifiable_ if t is (properly) simplified by some element of Γ'_{GVG^*}.

Our proof that selection simplifications actually improve elements of Γ'_{GVG^*} (Theorem 3.3.2) will allow us to show that Γ'_{GVG^*} is a semilattice with respect to the partial ordering $\overset{*}{\to}$ (Theorem 3.3.3). The unique minimal element of Γ'_{GVG^*} will then be shown in Theorem 3.3.4 to be the minimal fixed point of Ψ'. We require first some technical Lemmas.

Lemma 3.3.1 For each t ϵ V which is a selection or input

variable, and every control path from the start block s to loc(t), there is a maximal access sequence $(t=u_1,\ldots,u_k)$ such that $loc(u_1),\ldots,loc(u_k)$ are distinct blocks in p.

Proof by induction. We consider (t) to be a trivial access sequence. Suppose we have an access sequence $(t=u_1,\ldots,u_i)$ such that $loc(u_1),\ldots,loc(u_i)$ are distinct blocks in p. We further assume that $loc(u_i)$ occurs in p before $loc(u_1),\ldots,loc(u_{i-1})$. If u_i is neither a selection or input variable then $(t=u_1,\ldots,u_i)$ is a maximal access sequence. Otherwise, let p_i be the subsequence of p from s to the first occurrence of block $loc(u_i)$. Then there is a text expression u_{i+1} such that (1) $loc(u_{i+1})$ is contained in p_i and distinct from $loc(u_i)$ and (2) (u_i,u_{i+1}) is either a value edge (in the case u_i is an input variable) or a selection pair (if u_i is a selector sign). Hence $(t=u_1,\ldots,u_i,u_{i+1})$ is an access sequence and $loc(u_{i+1})$ is distinct from $loc(u_1),\ldots,loc(u_i)$. Since p is finite, we have our result. □

Lemma 3.3.1 will be used to construct maximal access sequences relative to fixed control paths. The next Lemma is analogous to Lemma 2.2.2 of Chapter 2.

Lemma 3.3.2 For each $\psi \in \Gamma'_{GVG^*}$ and $t \in V$, $origin(\psi(t)) \overset{*}{\rightarrow} loc(t)$.

Proof by contradiction. Suppose for some $t \in V$,

$$origin(\psi(t)) \overset{*}{\not\rightarrow} loc(t).$$

Then there must exist an input variable $X^{\rightarrow n}$ in $\psi(t)$ such

that $n \stackrel{*}{\not\to} loc(v)$, and hence there is an n-avoiding control path p from the start block s to loc(t). Also, there must exist an $u \in V$ also located at n such that $\psi(u) = X^{\to n}$. By Lemma 3.3.1, there is a maximal access sequence $(t=u_1,\ldots,u_k)$ such that $loc(u_1),\ldots,loc(u_k)$ are distinct blocks in p. Let j be the maximal integer $\leq$ k such that $\psi(u_1) = \ldots = \psi(u_j)$. If $L(u_j)$ is an input variable the $\psi(u_1) = \psi(u_j) = X^{\to n}$, so $loc(u_k) = n$ is contained on p, contradicting the assumption that p avoids n. Otherwise, if $L(u_j)$ is a function sign or constant sign, then $\psi(u) = \psi(u_k) \stackrel{+}{\to} X^{\to n}$, a contradiction with $\psi(u) = X^{\to n}$. □

The following Lemma shows that certain covers of simplifiable selection operations have a very special form.

Lemma 3.3.3 For each properly simplifiable selection $t \in V$, and $\psi \in \Gamma'_{GVG^*}$, if t is not simplified by ψ, then $\psi(t)$ is of the form $(sel_1 \ \ldots \ sel_k \ X^{\to n})$ where $sel_1,\ldots,sel_k$ are selector signs and $X^{\to n}$ is an input variable.

Proof by induction on subexpressions of $\psi(t)$.

Basis Step. By assumption t = (sel u) is not simplified by ψ, so $\psi(t)$ is of the form (sel $\psi(u)$). Also, note that since t is simplified by some element of Γ'_{GVG^*}, t has no departing selection pairs entering error.

Induction step. Suppose for some i, $1 \leq i \leq k$, $\psi(t)$ is of the form $(sel_1 \ \ldots \ sel_k \ \alpha)$. Consider any selector operation $t' = (sel_i \ u')$ such that $\psi(u') = \alpha$. We also assume in our induction hypothesis that t' has no departing

selection pairs entering <u>error</u>.

Suppose $\psi(u') = \alpha$ is not a selection operation or input variable.

<u>Case 1</u>. Suppose u' is an input variable. Let p be a control path from the start block s to loc(u'). By Lemma 3.3.1 we can construct a maximal access sequence from u' to some $\bar{u} \in V$. From this we can show that t' has a departing selection pair entering <u>error</u>, a contradiction with the induction hypothesis.

<u>Case 2</u>. Suppose u' is not an input variable. If u' is a construction operation for which sel_i is a selection, then t' is <u>not</u> a reduced expression, which is impossible. Otherwise, if u' is a constant sign or some other sort of function application other than a selection, then t' has a departing selection pair entering <u>error</u>, a contradiction with the induction hypothesis.

Hence $\psi(u')$ is either a selection or input variable. To complete our induction proof, for any selection $\bar{t}$ such that $\psi(\bar{t}) = \alpha$, if $\bar{t}$ has a departing selection pair entering <u>error</u>, then so does t', a contradiction. □

Now we show that simplification of selection operations always improves an element of Γ'_{GVG^*}.

<u>Theorem 3.3.2</u> For $\psi, \psi' \in \Gamma'_{GVG^*}$ and selection operation $t \in V$, if t is <u>not</u> simplified by ψ and t is properly simplified by ψ' then $\text{origin}(\psi'(t)) \overset{+}{\rightarrow} \text{origin}(\psi(t))$.

__Proof__. For any $N' \subseteq N$, let LCA(N') be the latest (furtherest from the start block s) common ancestor of the nodes in N' relative to the dominator tree of the control flow graph F. By Lemma 3.3.2, $\psi(t)$ is of the form ($sel_1 \cdots sel_k\ X \to n$). We proceed by induction on subexpressions of $\psi(t)$.

Suppose for some i, $1 \leq i \leq k$, if $i < k$, for every selection $\bar{t} \in V$ such that $\psi(\bar{t}) = (sel_{i+1} \ldots sel_k\ X^{\to n})$ then $LCA\{\bar{w} \mid (\bar{t},\bar{w})$ is a selection pair departing from $\bar{t}\} \overset{+}{\to} n$. Consider any selection $t' \in V$ such that $\psi(t') = (sel_i \ldots sel_k\ X \to n)$. Let u' be the immediate subexpression of t', so $origin(\psi(t')) = origin(\psi(u'))$. Then there exists a (possibly trivial) maximal access sequence from u' to some $\bar{t}$ such that $\psi(u') = \psi(\bar{t})$. By the induction hypothesis, $LCA\{\bar{w} \mid (\bar{t},\bar{w})$ is a selection pair departing from $\bar{t}\} \overset{+}{\to} n$. We can then show that $origin(\psi'(t)) \overset{*}{\to} LCA\{w' \mid (t',w')$ is a selection pair departing from $t'\} \overset{+}{\to} n$.

Since t is simplified by ψ', $\psi'(t) = \alpha$ where $\psi'(t') = \alpha$ for all selector pairs (t,t'). Hence,

$$origin(\psi'(t)) = origin(\alpha)$$
$$\overset{*}{\to} LCA\{w \mid (t,w) \text{ is a selection pair}\} \overset{+}{\to} n.$$
$$\overset{+}{\to} n = origin(\psi(t)). \quad \Box$$

In Section 2.2 we defined a partial function __min__ from EXP^2 to EXP; we extend __min__ to a partial mapping from $(\Gamma'GVG^*)^2$ to $\Gamma'GVG^*$ so that for each $\psi, \psi' \in \Gamma'GVG^*$, if $\bar{\psi}(t)$

$= \psi(t)$ min $\psi'(t)$ is defined for each text expression t, then ψ min $\psi' = \overline{\psi}$, and otherwise ψ min ψ' is undefined.

Theorem 3.3.3 $\Gamma'GVG^*$ forms a finite semilattice with respect to $\overset{*}{\rightarrow}$.

Proof. It is sufficient to show that min is well defined over $\Gamma'GVG^*$. Suppose for ψ', $\psi \in \Gamma'GVG^*$, ψ min ψ' is defined, so there is a text expression t such that $\psi(t)$ min $\psi'(t)$ is undefined but $\psi(u)$ min $\psi'(u)$ is defined for all u which are proper subexpressions of t' such that $\psi(t) = \psi(t')$. Thus t is either a selection operation or an input variable. Consider any control path p from the start block s to loc(t). By Lemma 3.3.1, we can construct a maximal access sequence $(t=u_1,\ldots,u_k)$ such that $loc(u_1),\ldots,loc(u_k)$ are unique blocks of p. Let j be the maximal integer $\leq k$ such that $\psi(u_1) = \ldots = \psi(u_j)$. By the proof of Theorem 2.2.1 of Section 2.2, we need only consider the case where t_j is a selection operation (sel u). Since j is maximal, t_j is not simplified by ψ and $\psi(t_j)$ = (sel $\psi(u)$). If t_j is also not simplified by ψ' then $\psi'(t_j)$ = (sel $\psi'(u)$) and by the induction hypothesis $\alpha = \psi(u)$ min $\psi'(u)$ is defined, so $\psi(t_j)$ min $\psi'(t_j)$ = (sel α). Otherwise, suppose t_j is simplified by ψ'. If $\psi'(t_j)$ = error then $\psi(t_j)$ min $\psi'(t_j)$ = error. If t is properly simplified by ψ' then by Theorem 3.3.2, origin($\psi'(t_j)$) $\overset{+}{\rightarrow}$ origin($\psi(t_j)$), so $\psi(t_j)$ min $\psi'(t_j) = \psi'(t_j)$. □

Theorem 3.3.4 Ψ' has a unique, minimal fixed point ψ^* which

is the minimal element of Γ'_{GVG^*}.

Proof Clearly, any fixed point of Ψ' is an element of Γ'_{GVG^*}. By Theorem 3.3.3, Γ'_{GVG^*} has a unique minimal element $\psi^* = \underline{\min}\ \Gamma'_{GVG^*}$. Let $\Psi^* = \Psi'(\psi^*)$. In proof of Theorem 2.2.1, we showed that $\Psi^*(X^{\rightarrow n}) = X^{\rightarrow n}$ for each input variable $X^{\rightarrow n}$ such that $\psi^*(X^{\rightarrow n}) = X^{\rightarrow n}$. Now suppose there is a selection $t \in V$ such that $\Psi^*(t) = \alpha$ where $\alpha = \psi^*(t')$ for all selection pairs (t,t'), but $\psi^*(t) \neq \alpha$. Let ψ be the mapping from text expressions to EXP such that for each text expression u, $\psi(u)$ is derived from $\Psi^*(u)$ by substituting α for each occurrence of $\Psi(t)$ in $\Psi^*(u)$, and reducing the resulting expression. Hence $\psi \in \Gamma'_{GVG^*}$ but by Theorem 3.3.2 origin($\psi(t)$) $\overset{+}{\rightarrow}$ origin($\psi^*(t)$), a contradiction with the assumption that ψ^* is the minimal element of Γ'_{GVG^*}. □

In the next section we describe a method for actually constructing ψ^*, the minimal element of Γ'_{GVG^*}.

3.4 The Computation of ψ^*, the minimal fixed point of Ψ'

Now we describe a method for actually constructing ψ^*, the minimal fixed point of Ψ' which was shown in Theorem 3.3.4 to be the minimal element of $\Gamma'GVG^*$. There are two main steps. We first reduce constant propagation with selection and constant reductions to constant propagation with only constant reductions; the latter problem is solved efficiently by the methods of Chapter 2. We then find ψ^* by constructing, by the methods of Chapter 2, the minimal element of ΓGVG_0, ΓGVG_1, ... ΓGVG_R where GVG_0, GVG_1, ..., GVG_R is a sequence of global value graphs derived from GVG^*.

Associate with each text expression t which is a selection (sel u), a new , distinct program variable SV_t called the selection variable of t. The corresponding input variable $SV_t^{\rightarrow loc(t)}$ will be unambiguously represented by dropping its superscript. The selection variable SV_t is installed in place of t in GVG^* by relabeling t with the selection variable SV_t, deleting the edge (t,u) originally departing from t and adding the selection pairs departing from t to the edge set. Conversely, the selection variable SV_t is replaced by t by reversing this process.

Let GVG be a labeled digraph derived from GVG^* by replacing any number of selections with their corresponding selection variables. Note that by definition of selection pairs, for any selection t relabeled in GVG with selection

variable SV_t, if p is a control path from the start block s to loc(t) then t has a departing selection pair (t,u), which is also a value edge of GVG, such that loc(u) is distinct from loc(t) and contained in p. Hence, GVG is a global value graph. Also, note that since the node set of GVG is V, the node set of GVG*, we continue to identify the nodes in V with text expressions. However selections in V may now be labeled in GVG with selection variables rather than selector signs. By Theorem 2.2.1, Γ_{GVG} has a unique, minimal element ψ. Chapter 2 gives an efficient method for the construction of ψ. Let us review these results.

GVG is <u>reduced</u> if $\psi(t)$ is the label of t in GVG for all $t \in V$ such that $\psi(t)$ is a constant sign. A reduced global value graph may be derived from GVG by the simple constant propagation algorithm presented in Section 2.3. We now assume GVG is reduced.

Recall that our method proceeds by induction on rank of text expressions. The <u>rank</u> of $t \in V$ labeled in GVG with a constant sign in GVG is 0. If t is labeled in GVG with a function sign θ, and $u_1,\ldots,u_k$ are the immediate successors of t in GVG, then the rank of t in GVG is

$$1+\text{MAX}\{\text{rank}(u_1),\ldots,\text{rank}(u_k)\},$$

and by definition of Γ_{GVG},

$$\psi(t) = (\theta\ \psi(u_1)\ \ldots\ \psi(u_k)).$$

Note that the rank induces a topological ordering (from

leaves to roots) of the dags of blocks from which GVG is built.

The case in which t is labeled with an input variable $X{\to}n$ is more difficult. Recall that a <u>value path</u> in GVG is a path p traversing only nodes linked by value edges and p is <u>maximal</u> relative to a fixed beginning node if p ends at a node with no departing value edges. The rank of t is

$$MIN\{rank(w) \mid w \text{ lies at the end of a maximal value path in GVG from } t\}.$$

This $t \in V$ is a <u>value source</u> relative to ψ if $\psi(t) = X{\to}n$.

We have from Chapter 2

<u>Theorem 2.4</u>. t is a value source of ψ iff there exist two maximal, almost disjoint (containing only one element in common) value paths in GVG from t to $u_1,u_2 \in V$ such that $\psi(u_1) \neq \psi(u_2)$. Furthermore, for each $t \in V$ labeled with an input variable $X{\to}n$, either

(1) $\psi(t) = \psi(u)$ for all u contained at the end of maximal value paths in GVG from t, or

(2) $\psi(t) = \psi(u)$ where u is the unique value source contained on all maximal value paths in GVG from t.

The problem of discovering the value sources of ψ is reduced in Section 2.6 to the computation of dominator trees, for which there is an efficient algorithm due to Tarjan[T4].

The next Theorem reduces constant propagation with selection and constant reductions, to constant propagation

with only constant reductions.

For this we will require two special mappings

M1(GVG): Γ_{GVG} to Γ'_{GVG^*}

M2(GVG): Γ'_{GVG^*} to Γ_{GVG}

For any $\psi \in \Gamma_{GVG}$, let M1(GVG)(ψ) be the mapping ψ_1 from V to EXP such that for all $t \in V$, $\psi_1(t)$ is derived from t by repeatedly

(1) substituting (sel ψ(u)) for each selection w = (sel u) such that $\psi(w) = SV_t$ is the selection variable of t.

(2) substituting ψ(u) for each $u \in V$ labeled in GVG with an input variable.

Observe that $\psi_1 \in \Gamma'_{GVG^*}$.

Let ψ_2 = M2(GVG)(ψ) be the mapping from V to EXP such that for each $t \in V$,

(1) t' is derived by substituting selection variable SV_u for each nonsimplifiable selection $u \in V$ such that u is labeled in GVG with the selection variable SV_u.

(2) $\psi_2(t)$ is derived from t' by substituting $\psi^*(u)$ for each u labeled with an input variable in GVG and such that $\psi^*(u')$ = $\psi(u')$ for each $u' \in V$ such that $\psi^*(u')$ is a proper subexpression of $\psi^*(u)$.

Observe that $\psi_2 \in \Gamma_{GVG}$.

<u>Theorem 3.4.1</u> If $\overline{GVG}$ is derived from GVG* by substituting selection variables for <u>all</u> selections, and $\overline{\psi}$ is the minimal element of $\Gamma_{\overline{GVG}}$, then for each $t \in V$, $\psi^*(t)$ is a constant

sign c iff $\overline{\psi}(t) = c$.

<u>Proof IF</u>. Suppose $\overline{\psi}(t)$ is a constant sign c, but $\psi^*(t) \neq c$ Let $\psi_1 = M1(\overline{GVG})(\overline{\psi})$. Hence $\psi_1(t) \overset{*}{\rightarrow} \psi^*(t)$ and $\psi_1 \in \Gamma'_{GVG^*}$, contradiction with the assumption that ψ^* is the minima element of Γ'_{GVG^*}.

<u>ONLY IF</u>. Suppose $\psi^*(t)$ is a constant sign c, but $\psi'(t) \neq c$ Let $\psi_2 = M2(\overline{GVG})(\overline{\psi})$. Then $\psi_2(t) \overset{*}{\rightarrow} \overline{\psi}(t)$ and $\psi_2 \in \Gamma_{\overline{GVG}}$, contradiction with the assumption that $\overline{\psi}$ is the minima element of $\Gamma_{\overline{GVG}}$. □

We now define a sequence of global value graphs

$GVG_0, GVG_1, \ldots$

derived from $\overline{GVG}$. GVG_0 is the reduced graph derived fro $\overline{GVG}$. For $r = 0,1,\ldots$ let NSS(r) be the set of selection of rank r which are not simplifiable and let GVG_{r+1} b derived from GVG_r by restoring each $t \in NSS(r)$; i.e., if t (sel u) then the label of t is set to sel, all selectio pairs departing from t are deleted, replaced by the origina edge (t,u). Let ψ_r be the minimal element of Γ_{GVG_r}. Let = MAX{r | GVG_r contains a node of rank r}.

<u>Theorem 3.4.2</u> $\psi_R = \psi^*$.

<u>Proof</u> Observe that each selection $t \in V$ is labeled with selection variable SV_t iff t is not simplifiable. Alsow have $\psi^* \in \Gamma_{GVG_R}$ which implies that $\psi^* \overset{*}{\rightarrow} \psi_R$, and we have ψ_R Γ'_{GVG^*} which implies that $\psi_R \overset{*}{\rightarrow} \psi^*$. Hence $\psi^* = \psi_R$. □

The remaining problem is the determination o

simplifiable selections in V.

<u>Theorem 3.4.3</u> For all selections $t \in V$ of rank r and labeled in GVG_r with a selection variable, $t \in NSS(r)$ iff t is a value source relative to ψ_r.

<u>Proof IF</u>. Suppose $t \in NSS(r)$ but t is <u>not</u> a value source of ψ_r, so $\psi_r(t) = \alpha$ where $\alpha = \psi_r(t')$ for all selection pairs (t,t'). Let $\psi_r' = M1(GVG_r)(\psi_r)$. Then $\psi_r' \in \Gamma_{GVG^*}$ and t is simplified by ψ_r', so by Theorem 3.3.2, $origin(\psi_r'(t)) \overset{+}{\rightarrow} origin(\psi^*(t))$, a contradiction with the assumption that ψ^* is the minimal element of Γ_{GVG^*}.

<u>ONLY IF</u>. Suppose t is simplified by ψ^*, so there exists an expression α such that $\psi^*(t) = \psi^*(t') = \alpha$ for all selection pairs (t,t'). Let $\hat{\psi}_r = M2(GVG_r)(\psi_r)$. Then $\hat{\psi}_r(t) = \hat{\psi}_r(t') = \alpha$ for all selection pairs (t,t') and $\hat{\psi}_r \in \Gamma_{GVG_R}$. If t is a value source of ψ_r, then by Theorem 3.3.2, $origin(\hat{\psi}_r) \overset{+}{\rightarrow} origin(\psi_r)$, a contradiction with the assumption that ψ_r is the minimal element of Γ_{GVG_r}. Thus t is not a value source of ψ_r. □

Let a <u>trivial value path</u> be of the form (t) where $t \in V$ is a node labeled with either a constant or function sign.

<u>Corollary 3.4.1</u> For each $t \in V$, t is of rank r in GVG_R iff there is a (possibly trivial) maximal value path in GVG_r from t to a node of rank r in GVG_r and such that p avoids all elements of NSS(r).

<u>Proof</u> Observe that for any $t \in V$ labeled in GVG_r with a constant or function sign, t is of rank r in GVG_r iff t is

of rank r in GVG_R. Otherwise, suppose t ϵ V is labeled with an input variable in GVG_r.

Suppose t is of rank r in GVG_R. Then there is a maximal value path p from t to some t' ϵ V such that all nodes of p are of rank r in GVG_R. Hence, p avoids all elements of NSS(r), and p is a maximal value path in GVG_r. Since t' is labeled with a constant or function sign, t' is also of rank r in GVG_r.

Suppose, on the other hand, that there is in GVG_r a maximal value path p from t to some t' or rank r in GVG_r such that p avoids all elements of NSS(r). It is always possible to find such a p containing only nodes of rank r in GVG_r. Hence, p is a maximal value path of GVG_r and t is of rank r in GVG_R. □

Our algorithm for computing ψ^*, the minimal fixed point of Ψ', is summarized below.

<u>Algorithm 3B</u>.

<u>INPUT</u> GVG*.

<u>OUTPUT</u> ψ^*, the minimal fixed point of Ψ'.

```
begin
  Discover all selection pairs by Algorithm 3A;
  Let GVG‾ be derived from GVG* by installing a seletion
  variables in the place of each selection;
  Apply Algorithm 2A to construct GVG0, the reduction of
  GVG‾;
  for r := 0 by 1 to oo do
    begin
    Apply Algorithm 2B to construct Vr, the set of all text
    expressions of rank r in GVGr;
    if Vr is empty then return ψ*;
    Compute by Algorithm 2C, ψr, the minimal element of
    rGVGr;
    Let NSS(r) be the set of selection of Vr which are value
    sources relative to ψr;
    comment By Theorem 3.4.3, NSS(r) is the set of
    nonsimplifiable selections of rank r in GVGr;
    for all t ε Vr contained on a (possibly trivial) maximal
    value path in GVGr avoiding all elements of NSS(r) do
      begin
        comment By Corollary 3.4.1, t is of rank r in GVGR;
        ψ*(t) := ψr(t);
      end;
    Let GVGr+1 be derived from GVGr by replacing each
    selection variable SVt in VSr with the original
    selection t;
    end;
end;
```

Let ℓ be the length of the text of program P and recall that GVG* is of size $O(|V|+|E|) = O(\ell+|\Sigma||A|)$.

<u>Theorem 3.4.4</u> Algorithm 3B is correct and costs $O(\ell^2+|\Sigma||A|)$ bit vector and $O(\ell(\ell+|\Sigma||A|))$ elementary operations.

<u>Proof</u> The correctness of Algorithm 3b follows directly from Theorems 3.4.1-3.4.3.

By Theorem 3.2.2, the computation of all selector edges by Algorithm 3A costs $O(\ell^2+|\Sigma||A|)$ bit vector operations. For each $r = 1,2,...$ the computation of Ψ_r may cost $O(\ell+|\Sigma||A|)$ elementary operations by the results of Chapter 2. Since the maximum r such that V_r is not empty is $\leq \ell$, the total time cost of Algorithm 3b is $O(\ell^2+|\Sigma||A|)$ bit vector and $O(\ell(\ell+|\Sigma||A|))$ elementary operations. □

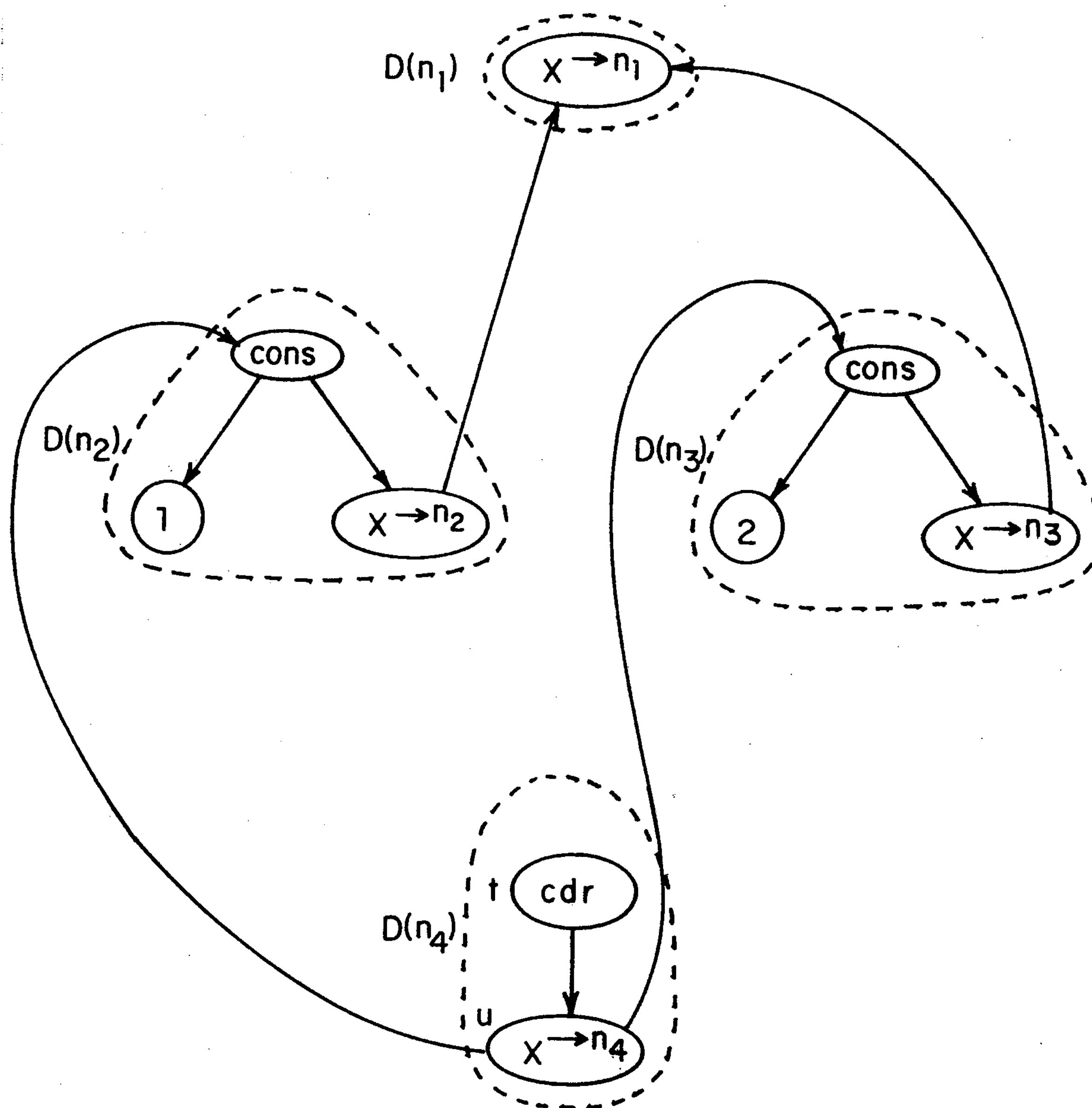

Figure 3.5. The global value graph GVG* for the program of Figure 3.4.

3.5 Type Covers and Type Declarations.

Types are expressions used to specify the shape of structured objects. A type cover of text expression t is a closed form expression for the type of t which holds on all executions of the program P. We show that the methods of the last section may be applied to the construction of type covers. Type covers have applications analogous to the usual sort of covers used to represent values of text expressions. For example, if the type cover of text expression t is a constant type, then the value of t has a fixed type over all executions of P. Text expressions which have the same type cover have values of the same shape on each execution of P (whereas, text expressions given the same type declaration may have different values of different shape over particular executions of P; see the latter part of this Section). A text expression t which is a construction operation is redundant if (1) every execution of P from the start block s to loc(t) passes thru a block containing a text expression t' with type cover common to t and furthermore, (2) the structured object computed by t' is dead (not referenced) on every execution path following loc(t) (in other words, the storage allocated for t' could be used to store t).

A type declaration of program P is used to specify, for each text expression t, the set of all types of values that

t may evaluate to, over all executions of P. A recursive type declaration uses recursion to specify infinite sets of types. In the latter part of this Section we discuss methods due to Tennenbaum[Te] and Schwartz[Sc2] for the automatic construction of type declarations for "type-free" programs (programs written without explicit type declarations). The method due to Schwartz is direct (noniterative) and more powerful than Tennenbaum's iterative method since it results in recursive type declarations for text which may have an infinite set of types (whereas, the method of Tennenbaum results in weaker, non-recursive type declarations).

We shall observe that the set of all possible types of a given text expression, over all executions of the program P, need not be a context-free language although the type declaration facilities of most programming languages are essentially context-free grammars. Hence, it is not possible to construct "tight" (exact) type declarations within most programming languages.

Fix (U,I) as an interpretation of program P as described in Section 3.1. Recall that the universe of structured values U is built from a set of atoms in the fixed set ATOM and k-adic constructor signs in CONS. Also, recall that EXP is the set of expressions built from input variables (representing the value of program variables on

input to blocks in N), constant signs in C, and k-adic function signs in θ (including operator signs in OP, constructor signs in CONS, and selector signs in SEL).

Let τ be a mapping initially of domain ATOM $\cup$ Σ into EXP such that

(1) For each $a \in$ ATOM, $\tau(a)$ is a symbol denoting the type of a.

(2) For each program variable $X \in \Sigma$, there exists an unique variable $\tau(X)$ = TX.

Extend τ to a homomorphic mapping from EXP to EXP thusly:

(a) for each constant sign $c \in C$, if c is of the form $X^{\to s}$ (representing the value of program variable X on input to the start block s) let $\tau(X^{\to s}) = \tau(X)^{\to s} = TX^{\to s}$. Otherwise, let $\tau(c) = \tau(I(c))$.

(b) for each input variable $X^{\to n}$, $\tau(X^{\to n}) = \tau(X)^{\to n} = TX^{\to n}$.

(c) τ distributes over function applications thus:

$$\tau(\theta\ \alpha_1\ \ldots\ \alpha_k) = (\theta\ \tau(\alpha_1)\ \ldots\ \tau(\alpha_k)).$$

Also, extend τ to subsets S of EXP and U:

$$\tau(S) = \{\tau(\alpha) \mid \alpha \in S\}.$$

A <u>type cover</u> of program P is a mapping ψ from the text expressions of P to τ(EXP) such that for each text expression t of P,

$$\text{EXEC}(\psi(t),p) = \tau(\text{EXEC}(t,p))$$

for all control paths p from the start block s to loc(t).

For example, consider the control flow graph of Figure 3.7. Let $\tau(1)$ = int. Note that $Z^{n\rightarrow}$ and $X^{m\rightarrow}$ do not have the same covers but do have the same type cover (cons int $TY^{\rightarrow m}$).

Let P_τ be the program derived from P by substituting $\tau(t)$ for each text expression t. Fix $(\tau(U), I_\tau)$ be the interpretation of P_τ where I_τ is the identity mapping over τ(ATOM), and for each k-adic elementary operation sign op ϵ OP (recall that I(op) is a mapping from $ATOM^k$ to ATOM) and $a_1, \ldots, a_k \epsilon$ ATOM,

$$I_\tau(op)(\tau(a_1), \ldots, \tau(a_k)) = \tau(I(op)(a_1, \ldots, a_k)).$$

Theorem 3.5.1 ψ is a cover of P_τ iff ψ is a type cover of P.

Proof Consider any text expression t and control path from the start block s to loc(t). By the fact that τ is a homomorphism over EXP,

$$EXEC(\tau(t),p) = \tau(EXEC(t,p)).$$

If ψ is a cover of P_τ then

$$EXEC(\psi(t),p) = EXEC(\tau(t),p),$$
$$= \tau(EXEC(t,p)).$$

On the other hand, if ψ is a type cover of P then

$$EXEC(\psi(t),p)) = \tau(EXEC(t,p))$$
$$= EXEC(\tau(t),p). \quad \square.$$

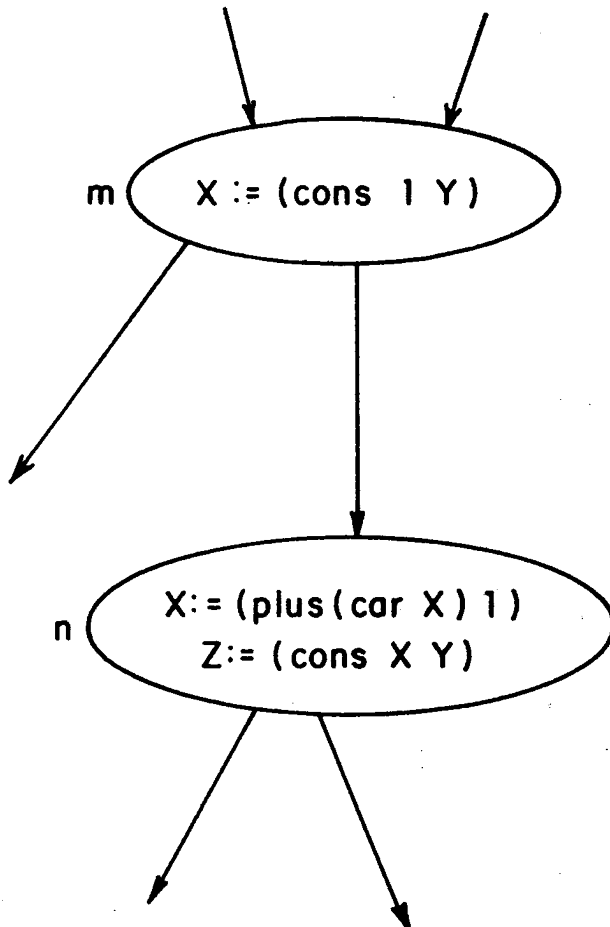

Figure 3.7. The control flow graph of a program P in LISP.

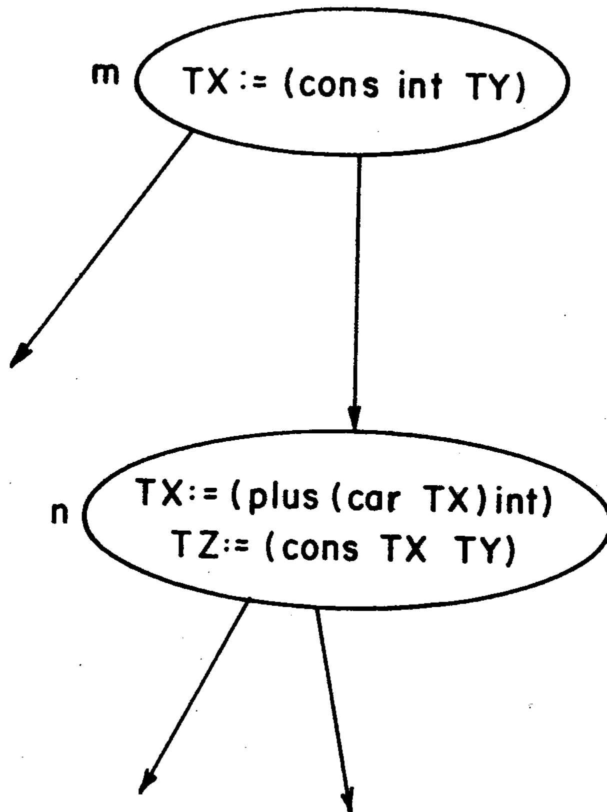

Figure 3.8. The type program P_τ derived from the program P of Figure 3.7.

Let TV = $\{T_1, T_2, \ldots\}$ be a set of _type variables_; in the following we assume TV and the special symbol _oneof_ is distinct from the elements of τ(ATOM) and CONS. We distinguish the type variable _any_ ϵ TV which will represent the set of all types. Let TEXP be the set of expressions built from τ(ATOM), TV, and CONS. We shall assume that for a fixed program P, CONS and τ(ATOM) are finite sets.

A _type declaration_ for input variable $X^{\to n}$ consists of a statement of the form

declare $X^{\to n}$ _type_ α

where $\alpha \in$ TEXP.

A type declaration is interpreted in the context of a _type variable definition block_ TDEF, consisting of a sequence of statements of the form

$T =$ _oneof_ $\{\alpha_1, \ldots, \alpha_k\}$

(or just $T = \alpha_1$ if $k=1$) where $T \in$ TV-{any} and $\alpha_1, \ldots, \alpha_k \in$ TEXP. We assume no $T \in$ TV occurs more than once on the left hand side of a statement in TDEF.

We now construct a set of productions (in the sense of formal language theory) by substituting for each statement

$T =$ _oneof_ $\{\alpha_1, \ldots, \alpha_k\}$

of TDEF, the context-free productions

$T \to \alpha_1, T \to \alpha_2, \ldots, T \to \alpha_k$.

Also, for the special symbol any we have the productions

any $\to \tau(a)$

for each a ϵ ATOM, and

any $\rightarrow$ (cons any ...k-times... any)

for each k > 0 and k-adic constructor sign cons ϵ CONS. For each T ϵ TV, let TDEF[T] be the context-free language generated by these productions with T considered to be the start symbol, the type variables as nonterminals, and the terminal symbols are taken from CONS $\cup$ τ(ATOM). Note that TDEF[T] is a subset of τ(U) and TDEF(any) = τ(U). Also, for each α ϵ TEXP, let TDEF[α] be the set {α' ϵ τ(U) | α' is derived from α by substituting some element of TDEF[T] for each type variable T occurring in α}.

For each α ϵ τ(U), let EXPAND(α) = {α' ϵ τ(U) | α' is derived from α by substituting some element of τ(U) for each constant sign of the form $X^{\rightarrow}s$}. For each input variable $X^{\rightarrow}n$, let TYPES($X^{\rightarrow}n$) = {α | α ϵ EXPAND(τ(EXEC($X^{\rightarrow}n$,p)) and such that p is some control path from the start block s to n}.

Consider again the type declaration

<u>declare</u> $X^{\rightarrow}n$ <u>type</u> α.

This type declaration is <u>proper</u> in the context of TDEF if

TYPES($X^{\rightarrow}n$) $\subseteq$ TDEF[α]

and is <u>tight</u> if

TYPES($X^{\rightarrow}n$) = TDEF[α]

For example, a proper type declaration for input variable $Xn^{\rightarrow}$ of Figure 3.7 is

declare $X^{\to n}$ *type* (cons int any).

Although the type definition facilities of many programming languages employ essentially the above scheme, it is interesting to note the scheme is not even powerful enough to give tight type definitions of programs without selection operations. Let f, g, and h be constructor signs of arity 1,1, and 3 respectively. Also, let $\tau(0)$ = int. In Figure 3.9,

$$TYPES(Zm^{\to}) = TYPES(Z^{\to n})$$
$$= \{h(f^k(int), g^k(int), f^k(int)) \mid k \geq 1\}$$

which is clearly not a context-free language language and hence is not definable by the above type declaration scheme.
□

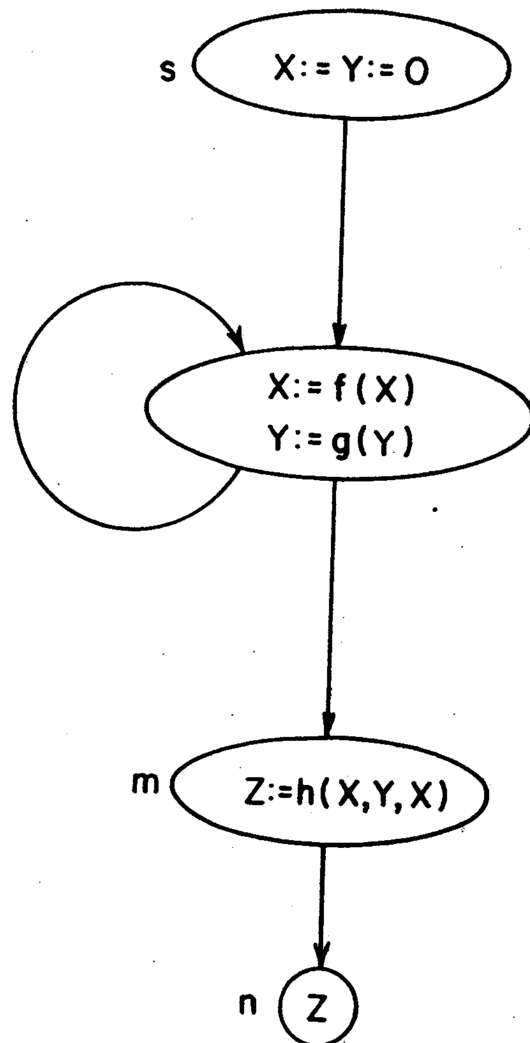

Figure 3.9. There is no tight type declaration for input variable $Z^{\to n}$.

We now describe a simplified version of the method of Schwartz[Sc2] for constructing proper (but not necessarily tight) type declarations. We require the special global value graph GVG* and selection pairs of Section 3.2. To simplify the method, we assume that for each k-adic elementary operation sign op ϵ OP, there exists a unique α_{op} ϵ τ(U) such that $\alpha_{op} = \tau(I(op)(a_1,\ldots,a_k))$ for all $a_1,\ldots,a_k$ ϵ ATOM.

For each text expression t which is a selection, let SV_t ϵ TV be the unique <u>selection variable</u>. Let $\hat{\tau}$ be the mapping from text expressions to TEXP such that,

(1) For each constant sign c ϵ C,

  (a) if c is of the form X→s (representing the value of program variable X on input to the start block s) then $\hat{\tau}(c)$ = any,

  (b) and otherwise, let $\hat{\tau}(c) = \tau(c)$.

(2) For each input variable X→n, $\hat{\tau}$(X→n) = TX→n.

(3) For each function application t = (θ $\alpha_1 \ldots \alpha_k$),

  (a) if θ is a elementary operator op ϵ OP then

$$\tau(t) = \alpha_{op}.$$

  (b) if θ is a constructor sign cons ϵ CONS then $\tau(t)$ = (cons $\hat{\tau}(\alpha_1)\ldots\hat{\tau}(\alpha_k)$).

  (c) if θ is a selector sign in SEL then

$$\hat{\tau}(t) = SV_t,$$

  the unique selection variable associated with t.

We assume that $TX^{\to n} \in TV$, for each input variable $X^{\to n}$. Consider the special type variable definition block $TDEF^*$ such that for each input variable $X^{\to n}$, we have the statement:

$$TX^{\to n} = \underline{oneof}\{\hat{\tau}(t) \mid (X^{\to n},t) \text{ is a value edge of } GVG^*\},$$

and for each text expression t which is a selection, there is a type declaration statement:

$$SV_t = \underline{oneof}\{\hat{\tau}(u) \mid (t,u) \text{ is a selection pair}\}.$$

Theorem 3.5.2 For each text expression t and each control path from the start block s to loc(t), $EXPAND(\tau(EXEC(t,p)))$ is contained in $TDEF^*[\hat{\tau}(t)]$.

Proof Let p be the shortest control path from the start block s to some block n containing a text expression t such that $EXPAND(\tau(EXEC(t,p)))$ is not contained in $TDEF^*[\hat{\tau}(t)]$. Clearly, $n \neq s$. We proceed by induction on subexpressions of t.

Consider a constant sign c. If c is of the form $X^{\to s}$ then $\hat{\tau}(X^{\to s})$ = any and so $EXPAND(\tau(EXEC(X^{\to s}))) = \tau(U) = TDEF[any]$. Otherwise $\tau(EXEC(c,p)) = \tau(c) = \hat{\tau}(c)$ and so $EXPAND(\tau(EXEC(c,p))) = \{\tau(c)\} = TDEF^*[\hat{\tau}(c)]$.

For each input variable $X^{\to n}$, $EXEC(X^{\to n},p) = EXEC(X^{m\to},p')$ where $p = p' \cdot (m,n)$. By the induction hypothesis, $EXPAND(\tau(EXEC(X^{m\to},p')))$ is contained in $TDEF^*[\hat{\tau}(X^{m\to})]$. By

definition, $TDEF^*[\tau(X^{\to n})]$ contains $TDEF^*[\hat{\tau}(X^{m\to})]$. Henc EXPAND($\tau$(EXEC($X^{\to n}$,p))) = EXPAND($\tau$(EXEC($X^{m\to}$))) is containe in $TDEF^*[\hat{\tau}(X^{\to n})]$.

If u is a selection contained within t, then u has a departing selection pair (u,u') such that EXEC(u,$\bar{p}$) = EXEC(u',$\bar{p}$) for some control path $\bar{p}$ which is a subsequence of p starting at s. By definition, $\hat{\tau}(u)$ is the selection variable SV_u and $TDEF^*[SV_u]$ contains $TDEF^*[\hat{\tau}(u')]$. Also, by the induction hypothesis EXPAND(τ(EXEC(u',$\bar{p}$))) is contained in $TDEF^*[\hat{\tau}(u')]$. Hence, EXPAND(τ(EXEC(u,p))) is contained in $TDEF^*[\hat{\tau}(u)]$.

Now suppose t is a function application (θ $t_1 \ldots t_k$) such that θ is not a selection and τ(EXEC(t_i,p)) is contained in $\hat{\tau}(t_i)$ for i = 1,...,k. But

τ(EXEC(t,p)) = EXEC((θ $\tau(t_1) \ldots \tau(t_k)$),p)

= (θ τ(EXEC(t_1,p))...τ(EXEC(t_k,p)))

Hence τ(EXEC(t,p)) is contained in $\hat{\tau}(t)$ = (θ $\hat{\tau}(t_1) \ldots \hat{\tau}(t_k)$), a contradiction. □.

This immediately implies that

Corollary 3.5 For each input variable $X^{\to n}$,

declare $X^{\to n}$ **type** $TX^{\to n}$

is proper, relative to type variable definition block $TDEF^*$.

The above method for constructing type declarations is due to Scwartz[Sc2]. We conclude by listing $TDEF^*$ for the program of Figure 3.1. Let τ(nil) = null, τ(0) = int, t =

(car $X^{\to n}$), and t' = (cdr $X^{\to n}$).

TDEF* = ($TX^{\to m}$ = <u>oneof</u>{null, (cons int $TX^{\to m}$)},
$TY^{\to m}$ = null,
$TZ^{\to m}$ = int,
$TX^{\to n}$ = $SV_{t'}$,
$TY^{\to n}$ = <u>oneof</u>{$TY^{\to m}$, (cons SV_t $TY^{\to n}$)},
SV_t = <u>oneof</u>{error, int},
$SV_{t'}$ = <u>oneof</u>{error, null, $TX^{\to m}$})