SETL Newsletter # 130

J. Schwartz
May 29, 1974

Deducing relationships of
inclusion and membership in SETL programs.

1. Introduction. *A-Priori* plausibility of inclusion/membership relationships.

This newsletter takes up the 'high level optimisation' theme of NL.71, NL.118, NL.121, and of Aaron Tenenbaum's thesis (hereinafter referenced as TT). The generally good performance of Tenenbaum's 'type-finder' program suggests that it may be feasible to deduce deeper properties of the objects appearing in SETL programs. In the present newsletter, we will outline techniques which, building on the approach and result of the typefinder, allow useful relationships, among them relationships of inclusion and membership, to be established.

Before entering into technical detail, we make a few generalising remarks. Global optimisers are programs that prove, and then exploit, facts concerning other programs. Thus optimisers may be considered as country cousins of those still largely hoped for, more sophisticated routines which prove the correctness, in some appropriate formal sense, of these other programs. In contrast to full correctness-prover systems, which aim to prove a few major, hard, programmer-specified facts about a program, an optimiser aims to prove numerous small, easy facts about the programs which it analyses; moreover, optimisers themselves normally generate the surmises that they attempt to prove. We may put this comparison somewhat differently, by considering the nature of the theorem-proving algorithms which are typically employed by program analyser/optimisers on the one hand, and program-correctness verifiers on the other.

Theorem proving programs fall into two main families: on the one hand, those generically similar to the original 'geometry theorem prover'of Gelernter; on the other hand, those belonging to the resolution group. Provers of the first kind proceed very cautiously in generating objects not almost explicit in the situations with which they are presented. This limits very significantly the space of possibilities which such a prover needs to explore, and makes it possible for such provers to generate facts using a kind of transitive closure method; of course the closure-forming process employed may be optimised in various ways or steered by some heuristic. Provers of the second kind are more general, and in principle capable of reaching out much farther from an initially given set of hypotheses, largely because they have available, and are prepared to use, constructor mechanisms capable of generating all the objects of some full 'Herbrand universe'. However, their very generality confronts provers of the second type with the problem of searching rapidly growing, potentially infinite sets of possibilities, and at the present time provers of this second type generally founder amidst multitides of unexplored possibilities. We may therefore expect the fact-provers used implicitly in an optimiser to use transitive closure rather than resolution methods; and the technique used in connection with the optimisations described below confirms this general expectation.

An optimiser will only be useful if by examining the program P with which it is working, it can define a class of facts concerning P which are worth trying to prove. These facts must then be easy to prove, and, if proved, must yield valuable object-code improvements during the phases of compilation which follow code analysis.

For such an optimiser to be possible, it is probably necessary that rich semantic relationships should appear fairly explicitly in the source text with which an optimiser is required to work. This is to say that we expect deeper optimisations to be possible in a high-level language whose primitives are semantically rich, than in a language of lower level, whose primitives are of a more impoverished, hardware-like, character. The technical discussion which follows illustrates this general remark, and shows that a suitable optimiser will be able to guess and prove relationships of inclusion and membership between the objects of SETL programs. In a language of only slightly lower level, as for example a LISP-like language, it could be hard to formulate these relationships, much less to prove them efficiently.

2. An algebra of membership and inclusion relationships.

We suppose the SETL programs with which we work to be schematised in the manner described in TT, i.e., as a set of operator-argument tuples with designated target variables, arranged in a collection of basic blocks among which flow relationships are defined by a sucessor mapping. We shall when necessary write operator-operand-target tuples either as

(1a) $$targ = op(arg_1, arg_2, \ldots, arg_n)$$

as

(1b) $$targ = arg_1 \ \underline{op} \ arg_2,$$

or in whatever other notational form is convenient.

We suppose that John Cocke's 'use-definition chaining' process has been carried out; the algorithm to be described will use the output data of this procedure, which makes explicit the data flow relationships within a program P to be analysed.

We shall call the target variables of operator-operand tuples like (1a),(1b) *ovariables*. The analysis described in NL.71 and TT associates a type symbol with each ovariable. This type symbol classifies an ovariable, or rather its value as understood at compile time,as being one of the following: *elementary* (i.e. integer, real, string, etc.), *set*, *known-length-tuple*, or *unknown-legth-tuple*. Values of these last three kinds are described by type symbols of the structure {t}, $<t_1,\ldots,t_n>$, and [t] respectively, where t, $t_1,\ldots,t_n$ are themselves type symbols and give information concerning the type of a set's members and a tuple's components.

We now introduce a number of formal inclusion and membership relationships between ovariables appearing in a schematised SETL program P; the analysis which follows will base itself on these relations. Let o_1 and o_2 be ovariables of P. Then

i. $o_1 \in o_2$ means that (immediately after the definition of o_1) the value of o_1 is (necessarily) a member of the (current) value of o_2.

ii. $o_1 \subseteq o_2$ means that the value of o_1 is a subset of the value of o_2 (immediately after the definition of o_1; necessarily; where o_2 refers to its current value).

iii. $o_1 \in_n o_2$ means that o_1 is a tuple of known length this length being at least n, and that the n-th component of o_1 is a member of o_2.

iv. $o_1 \subseteq_n o_2$ means either that o_1 is a set of tuples of known length, this length being at least n, and that the n-th component of each member of o_1 is a member of o_2; or if n = 2 another meaning is possible, namely that o_1 is a tuple every one of whose components belongs to o_2.

v. $o_1 \in_\infty o_2$ means that o_1 is a tuple of unknown length, and that every component of o_1 is a member of o_2.

vi. $o_1 \subseteq_\infty o_2$ means that o_1 is a set of tuples of unknown length, and that every component of every member of o_1 is a subset of o_2.

The type symbols associated with the ovariables of P tell us which of these relationships are *plausible*, i.e. possible *a priori*. Let t_1 and t_2 be the type symbols of o_1 and o_2 respectively. Then the relationship $o_1 \in o_2$ is only plausible if $t_2 = \{t\}$ and $t \leq t_2$ in the lattice of type symbols (cf. TT); $o_1 \subseteq o_2$ only plausible if $t_1 = \{t'\}$, $t_2 = \{t\}$, and $t' \leq t$ in the type symbol lattice. Moreover, $o_1 \in_n o_2$ is only plausible if $t_1 = \langle \bar{t}_1, \ldots, \bar{t}_k \rangle$ with $k \geq n$, $o_2 = \{t\}$, and $\bar{t}_n \leq t$ in the type symbol lattice; $o_1 \subseteq_n o_2$ is only plausible if either $t_1 = \{\langle \bar{t}_1, \ldots, \bar{t}_k \rangle\}$ and these same conditions hold; or $n = 2$, $t_1 = [t']$ and $t_2 = \{t\}$ with $t' \leq t$ in the lattice; or $n = 2$, $t_1 = \langle \bar{t}_1, \ldots, \bar{t}_k \rangle$, $t_2 = \{t\}$, and $t_j \leq t$ in the type lattice for all $1 \leq j \leq k$. The relationship $o_1 \in_\infty o_2$ is only plausible either if o_1 is of type $[t]$, $o_2 = \{t'\}$, and $t \leq t'$ in the type lattice, or if o_1 is of type $\langle t_1, \ldots t_n \rangle$ with $t_j \leq t'$ in the type lattice for all j. The relationship $o_1 \subseteq_\infty o_2$ is only plausible either if o_1 is of type $[\{t\}]$, $o_2 = \{t'\}$ and $t \leq t'$ in the type lattice, or if o_1 is of type $\langle \{t_1\}, \ldots, \{t_n\} \rangle$ with $t_j \leq t'$ in the type lattice for all j.

The set of all relationships plausible in this sense for a program P will be called the *maximum plausible set of relationships for P.*

3. Outline of an analysis algorithm.

Given a SETL program P, we wish to determine the set of all formal inclusion relationships which hold between its ovariables. To this end, we propose an algorithm which works 'downwards' (in the set of relationships described in the preceeding section), starting with the maximum plausible set of relationships for P, and systematically eliminating relationships which might be false until only relationships which are certainly true remain. The rules which apply are as follows.

a) Call the 'source' or 'input' variables of operator-operand tuples like (1a), (1b) *ivariables*. Each ivariable i will be *chained* to a set ud(i) of ovariables. We allow ivariables i to participate in inclusion/membership relationships $i \subseteq o$, $i \subseteq_n o$, etc. A relationship iRo can only hold if $o_1 R o$ holds for all $o_1 \in ud(i)$.

b) The family of true relationships oRo_1 and $o_1 R o$ in which an ovariable o participates depends on the nature of the operation defining o and on the relationships which hold for the input variables of this operation. To define this aspect of the analytic situation completely, a rule is required for every SETL primitive and every possible inclusion relationship. The SETL primitives are enumerated on page 4 of NL.71. We do not give a complete list of the rules showing the effect of SETL primitives on inclusion relations here, but only display a few typical cases.

Case 1: Relationship $o \subset o_1$. This is always false if the operation defining o is anything other than $\mathbb{P}$ i.nd i. $i_1(i_2)$, $i_1(i_2,\ldots,i_k)$. The rules for these seperate subcases are as follows:

Subcase(1a) $o = \mathfrak{P} i$. Then relationship $o \in o_1$, is false unless $i \subseteq o_1$.

(1b) $o =$ hd i. Then $o \in o_1$ is false unless $i \in_1 o_1$ or $i \in_\infty o_1$.

(1c) $o = i_1(i_2)$. Then either i_2 must be a known constant n and $i_1 \in_n o_1$; or i_1 must be of type $\langle t_1,\ldots,t_n\rangle$ and $i_1 \in_k o_1$ must hold for each $k \leq n$; or $i_1 \in_\infty o_1$ must hold; or i_1 must be of type $\{\langle t_1, t_2\rangle\}$ and $i_1 \subseteq_2 o_1$ must hold.

(1d) $o = i_1(i_2,\ldots,i_k)$. Then i_1 must be of type $\{\langle t_1,\ldots,t_{k+1}\rangle\}$ and $i_1 \subseteq_{k+1} o_1$ must hold.

Case 2: Relationship $o \subseteq o_1$. This is always false if the operation defining o is anything other than +, -, *, with, less, $i_1\ \{i_2\}$, $i_1\ \{i_2,\ldots i_k\}$, or $\{i_1\}$. The rules for these seperate subcases are as follows:

Subcase(2a) $o = i_1 + i_2$. Both $i_1 \subseteq o_1$ and $i_2 \subseteq o_1$ must hold. In the subcase $o = i_1$ with i_2, $i_1 \subseteq o_1$ and $i_2 \in o_1$ must hold.

(2b) $o = i_1 - i_2$, $o = i_1$ less i_2; then $i_1 \subseteq o_1$ must hold.

(2c) $o = i_1 * i_2$. then $i_1 \subseteq o_1$ or $i_2 \subseteq o_2$ must hold.

(2d) $o = i_1\ \{i_2\}$. Then i_1 must be of type $\{\langle t_1,t_2\rangle\}$, and $i_1 \subseteq_2 o_1$ must hold. We leave it to the reader to write the similar rule which applies to $i_1\ \{i_2,\ldots,i_k\}$.

(2e) $o = \{i_1\}$. Then $i_1 \in o_1$ must hold. Note that $o_1 \subseteq o_1$ is always true.

Case 3: $o \subseteq_n o_1$. This is always false if the operation defining o is anything other than +, -, *, with, less, $i_1\{i_2\}$, $i_1\{i_2,\ldots,i_k\}$, $\langle i_1, i_2,\ldots,i_k\rangle$, or $\{i_1\}$. Rules for these seperate subcases may readily be stated; most of these rules rather closely resemble the rules for the corresponding subcases of Case 2. For example, if $o = i_1 + i_2$, then $i_1 \subseteq_n o_1$ and $i_2 \subseteq o_n$ must hold. If $o = i_1\{i_2\}$, then i_1 must be of type $\{\langle t_1,\ldots,t_k\rangle\}$, and $i_1 \subseteq_{n+1} o_1$ must hold; the rule for $o = i_1\{i_2,\ldots,i_n\}$ is a straightforward generatisation of this. When $n = 2$ a few additional subcases arise. If $o = i_1 + i_2$, then $o \subseteq_2 o_1$ can hold if the types of i_1 and i_2 represent tuples (of known or unknown length) and $i_1 \subseteq_2 o_1$, $i_2 \subseteq_2 o_1$. If $o = \langle i_1,i_2,\ldots,i_k\rangle$, then $o \subseteq_2 o_1$ can hold if $i_j \subseteq_2 o_1$ for all $1 \leq j \leq k$.

Case 4: $o \in_n o_1$. This is always false if the operation defining o is anything other than +, $\langle i_1,\ldots,i_1\rangle$, $i_1(i_2)$, or $i_1(i_2,\ldots,i_k)$. Rules for the verious subcases are as follows:

Subcase(4a) $o = i_1 + i_2$. Then the type of both i_1 and i_2 must be tuple. If i_1 is of known length ℓ, and $\ell \leq n$, then $i_2 \in_{\ell-n} o_1$ must hold; if $\ell \geq n$ or if i_1 is of unknown length, then $i_1 \in_n o_1$ must hold.

(4b) $o = \langle i_1,\ldots,i_k\rangle$. Then $k \geq n$ must hold, and $i_n \in o_1$ must also hold.

(4c) $o = i_1(i_2)$. Then i_1 must be of type $\{<t_1,\ldots,t_\ell>\}$ with $\ell \geq n+1$, and $i_1 \in_{n+1} o_1$ must hold; the rule for $o = i_1(i_2,\ldots,i_k)$ is a straightforward generalisation of this. It must be provable in both these cases that o is not Ω.

Case 5: $o \in_\infty o_1$. This is always false if the operation defining o is anything other than +, $<i_1,\ldots,i_k>$, $i_1(i_2)$, or $i_1(i_2,\ldots,i_k)$. The rules applying to these various subcases are rather like those stated for the corresponding subcases of case 4; we leave it to the reader to supply all necessary details.

Case 6: $o \subseteq_\infty o_1$. This is always false if the operation defining o is anything other than +, -, *, with, less, $i_1\{i_2\}$, $i_1\{i_2,\ldots,i_k\}$, or $\{i_1\}$. The rules applying to these various subcases are rather like those stated for the various of case 3.

A few basic SETL constants will regularly enter into operations as ivariables. Each of these constants has properties which are to be exploited in applying the above rules or appropriate slight extensions of them. For example, the null set nl satisfies $nl \subseteq o_1$, $nl \subseteq_n o_1$, and $nl \subseteq_\infty o_1$ for every o_1; the null-tuple nult has properties which should be reflected in appropriate small extensions of the rules which have been stated.

The rules which have just been stated determine a mapping *musthold* from the class of all relationships oRo_1 to the power set of the class of all relationships iRo_1. Given that o is the target variable of the operation $op(i_1,\ldots i_R)$, the set $musthold(oRo_1)$ is defined as the set of all relationships iRo_1 which must hold according to the foregoing rules if oRo_1 is to hold. Using this map, we can describe our inclusion/membership finder as follows:

i. Given a schematised SETL program P, perform a type analysis for it and then determine the maximum plausible set S of relationships for P in the sense of section 2. Next reduce S by eliminating all relationships oRo_1 which are obviously impossible in view either of the operation defining o or of the types of the ivarables of this operation. This should leave a manageably small set of relationship symbols to be treated.

ii. After the preparatory steps just described, build up a map U which sends each ovariable o into the set {o'} of all ovariables o' which are chained to an ivariable of the operation defining o, and then iteratively remove relationships from S, as follows:
A workpile W is initialised to contain all ovariables o.
For each o in the workpile, and each relationship oRo_1 involving o and belonging to S, one checks to see if $musthold(oRo_1)$ is included in S. If not, oRo_1 is removed from S, and o is put in a set W_1 of modified variables.

When this process has been applied to all o ε W, W is reset to $U^{-1}[W_1]$, and the process repeats. Let S_∞ be the set of inclusion/membership relationships remaining in S when W becomes null. We call S_∞ the set of relationships *confirmed* by our analysis procedure; these relationships are necessarily true.

4. Generalised membership/inclusion relationships.

The relationships of membership and inclusion utilised in section 3 can be generalised substantially. The fuller set of relationships which we shall wish to consider is most adequately represented by composite symbols which we shall call *relation strings*. Let $\eta_1, \eta_2, \ldots, \eta_k$ be symbols representing monadic mappings on composite SETL objects, and let η be a symbol representing a binary truth-valued operator on SETL objects o and o_1. Then we write $o\ \eta_1\ \eta_2 \ldots \eta_k\ \eta\ o_1$ if the relation $(\eta_k\ \eta_{k-1} \ldots \eta_2\ \eta_1\ o)\ \eta\ o_1$ holds. The following examples will indicate the intent of these definitions. Let the symbol ∋ signify a monadic 'random membership choice' operation on sets, let the symbol n indicate the operation of choosing the n-th component of a tuple, let the symbol ∞ signify the operation of choosing a random component of a tuple, and let the symbol $\bar{n}$ signify the operation of choosing some random component, but not one of the first n-1 components, of a tuple. Let ε denote the boolean 'membership' relation, and let $>, \geq, <, \leq$ be comparison operators as usual.

Then the relationships $o\ni\in o_1$, $on\in o_1$, $o\infty\in o_1$, $o\ni n\in o_1$ and $o\ni\propto\in o_1$, are respectively the relationships $o\subseteq o_1$, $o\ \varepsilon_n\ o_1$, $o\ \varepsilon_\infty\ o_1$, $o\ \subseteq_n o_1$, and $o\ \subseteq_\infty o_1$ of section 3. The relationships $o\ni\ni\in o_1$, $o\ni n\ni\in o_1$, and $o\ni\infty\ni\in o_1$ are worth considering, as are $o\ni\ \bar{n}\ o_1$ and $o\ni.\bar{n}\ni\in o_1$. By working with arithmetic relationships_like $o > o_1$, $o\ \ni> o_1$, etc. one can hope to prove semantic facts like 'o is a set of non-negative integers/ which when known will permit useful optimisations.

For a relationship $on_1\ n_2\ldots n_k n o$ to be plausible, the type of o must be such as to allow $n_1,\ldots,n_k$ to be applied to o in sequence, and the type of the resulting quantity $o' = (n_k\ldots n_1\ o)$ must be such as to permit $o' n o$ to be true. Once the types of the objects appearing in a SETL program have been found, this restriction should serve to guarantee that the set of relationships which remain plausible is manageably small. A 'necessity rule' can then be given for every relationship $on_1 n_2\ldots n_k n o_1$ admitted into an analytic system, following which the analysis algorithm described in section 3 can be used with minimal changes to prove the validity of some subset of the collection of initially plausible relationships.

It is worth noting that by making use of the composite relationships which have just been defined we can actually establish more facts concerning the simpler relationships described in section 2, since the necessity rules stated in section 3 can be relaxed somewhat.

For example, $o \subseteq o_1$ can hold even if o is defined by an operation of the form $\exists i$, $i_1(i_2)$, or $i_1(i_2,\ldots,i_n)$, provided that the following conditions are satisfied:

Subcase(a) $o = \exists i$. Then $i\exists \exists \in o$ must hold.

(b) $o = i_1(i_2)$. Then i_1 must either be of type tuple and $i_1 \text{``}\exists \in o$ must hold, or of type $\{<t,\{t'\}>\}$, and $i_1 \exists 2 \exists \in o$ must hold.

(c) $o = i_1(i_2,\ldots,i_n)$. Then i_1 must have type $\{<t_1,\ldots,t_{n-1},\{t_n\}>\}$ and $i_1 \exists n \exists \in o$ must hold.

5. Information derivable from the presence of insertion and union operations.

Consider the code sequence

(1) $$\ldots\ s = \{n\};\ t = s_1 + s;\ldots$$

At a program point immediately subsequent to this sequence we can be sure that n is a member of s_1. But if taken in its simple form, the inclusion/membership analysis described in section 3 will remain unaware of, and so fail to exploit, this fact. Indeed, it will miss even the simpler fact that nes is certain to hold immediately after s = {n} has been executed.

However, a straightforward improvement of our approach can remedy this deficiency. Each appearance of the schematised version of an instruction

(2a) $t = s_1 + s;$

can in effect be replaced by an appearance of the sequence

(2b) $t = s_1 + s;\ s = s*t;$

each appearance of

(3a) $n \underline{in} s;$

by an appearance of

(3b) $n \underline{in} s;\ n = \exists \{ms \mid m \underline{eq} n\};$

each appearance of

(3a) $s = \{n\};$

by appearance of

(3b) $s = \{n\};\ n = \exists s;$

etc. Note that in the schematised program versions with which we work, the sequence (3b) can most appropriately be handled by treating it as if it read

(4) $n \underline{in} s;\ (n = \exists s;)$

where by writing $n = \exists s;$ we have indicated the presence of a 'pseudoassignment' forcing n to be a member of s, but where by placing this operation in parentheses we indicate that no other relationship involving n is spoiled by this (or, indeed, any other) pseudoassignment. Similarly the schematised form of (2b) may be treated in the manner which is suggested by the sequence

(5) $t = s_1 + s;\ (s = s*t;)$

It is occasionally possible to glean useful information from the form of the conditional transfers determining the successor relationships among the basic blocks of a SETL program P. Suppose, for example, that such a transfer has the form

(6) if s eq nl then go to *label*;.

then the program point *label* is reached via (6), we may be sure that s is nl.

To ensure that our analysis does not miss information of this kind, we apply the following treatment to every conditional transfer whose governing condition is simple enough to be worth bothering with. An auxiliary pseudo-block is generated from the transfer; the transfer is modified so as to jump to the pseudo-block, which in turn jumps to the original transfer destination. The pseudo-block contains a pseudoassignment which forces the condition appearing in the transfer to be true. For example, in treating (6), we modify it to read as

(6') if s eq nl then go to label';

where *label'* is a generated label prefixing the pseudo-block

(6) label': (s = nl;) go to *label*;

This pseudo-block insertion process gives P a somewhat different flow graph from that which it would otherwise have, which in turn changes the result obtained when data-flow analysis is applied to P. In the modified data-flow, some ivariables will be changed to ovariables appearing in pseudoassignments within pseudo-blocks.

This revised chaining makes explicit more precise information than would otherwise be available, and this enlarges the set of relationships which will be established by our analysis algorithm.

Note that it might be useful to allow SETL users to write pseudoassignments explicitly. A user-supplied pseudo-assignment would act as a kind of declaration, and could supply an optimising SETL compiler with information which it was unable to deduce, but in a form which it was easily able to use.

6. Information supplementary to inclusion/membership relationships

Once relationships of inclusion and membership between the objects of a SETL program P have been established, certain interesting optimisations come almost within reach. Our intent is to find cases in which a set s_1 included within a set s can do without explicit representation of its own; we hope merely to issue a 'serial number' to each element of s, and then to represent s_1 by a bit-vector, the n-th bit of the vector signifying whether the n-th element of s does or does not belong to s_1. We may also try to find cases in which a map f known to have domain included in s can be represented either by a vector of pointers or by a collection of pointers attached to the elements x of s, where each pointer determines one value f(x).

For representations of this kind to be within reach, it is clearly necessary that the analysis described in section 3 and 4 should confirm the relationships $s_1 \subseteq s$ and $f \subseteq_1 s$. We must also be sure that s_1 (resp. f) is not set up when the set s has one particular value and then used after elements have been removed from s.

If s_1 is itself made part of some composite object o, either as a set member or as a tuple component, additional complications arise. For representation of s_1 as an s-based bitvector to remain desirable in this case, it is necessary that the following condition should be satisfied:

(C1) o is dead at each program point at which s is diminished.

Relatively few complications will be caused by insertion into o of bitvectors representing subsets of s if the following condition is also satisfied:

(C2) All the elements of the composite object o are subsets of s.

Condition (C1) is not quite sufficient to ensure the desirability of representing s_1 by an s-based bitvector. To state a sufficient condition, we must first make some appropriate definitions. Given an ovariable o or an ivariable i of a SETL program P, we define the following functions.

By *crthis(o)* resp. *crthis(i)* we mean the set of all ovariables which can create an object which at some moment in the execution of P becomes the current value of o (resp. i): If the value of o or i can be a set, then by *crmemb(o)* (resp. *crmemb(i))* we mean the collection of all ivariables j whose values become incorporated as members into a set which at some moment in the execution of P becomes the current value of o (resp. i) If the value of o or i can be a vector, then by *crsomcomp(o)* (resp. *crsomcomp(i))* we mean the collection of all ivariables j whose values become incorporated as components into a vector which at some moment in the execution of P becomes the value of o (resp. i). By *crpart(o)* (resp. *crpart(i))* we mean the collection of all ivariables whose value might either be equal to or become incorporated, either as members, members of members, components, members of components, components of members, etc. into a composite object which at some moment in the execution of P becomes the value of o (resp. i). Methods for calculating these functions are described in Newsletter 131.

Using these functions, we make the following

Definition: Let s, s_1 and t be ovariables of a program P, we say that t is *superior to* s_1 if s_1 belongs to

[+: ie crpart(t)] crthis(i).

We say that s_1 is a *dependent subset* of s if $s_1 \subseteq s$ (in the sense of section 3) and if the value of every object superior to s_1 is dead at each operation which might remove an element from s. The map f is *domain dependent* on s (resp. range dependent on s) if $f \subseteq_1 s$ (resp. $f \subseteq_2 s$) and if the value of every object superior to f is dead at each operation which might remove an element from s.

To indicate subset dependency, domain dependency, and range dependency we write $s_1 \subseteq: s$, $f \subseteq:_1 s$, $f \subseteq:_2 s$ respectively. Note that an analogous notion $o\ n_1 \ldots n_k\ n:\ o_1$ may be defined for each of the relations $o\ n_1 \ldots n_k\ n\ o_1$ introduced in section 4 above.

If $s_1 \subseteq: s$, then s_1 can, in the manner described at the beginning of the present section, be represented by a vector of bits. This vector can be inserted, in lieu of s_1, into each composite object of which s_1 is to become a part. Similarly, if $f \subseteq:_1 s$, then f can be represented by a vector or some other suitable collection of pointers, which can be inserted, in lieu of f, into each composite object of which f is to become a part.

Note that once P has been analysed for inclusion/membership relationships, we can use something very close to a standard live-dead analysis to tell which of the more precise relationships $s_1 \subseteq: s$, $f \subseteq:_1 s$, etc. hold.

We have noted above that insertion into a composite object o of bitvectors representing subsets of s will be least problematical when the part of o into which this insertion is made can only contain subsets of s. The methods which have already been described allow just such properties of objects o to be established: in the notation introduced in section 4, the fact to be proved is $o \ni \ni \in s$ if o is a set, $on \ni \in s$ or $o \, \omega \ni \in s$ if o is a tuple. Additional details concerning the manner in which we propose to treat bitvectors representing subsets of s (and vectors of pointers defining functions with domain contained in s) will be found in the following section.

7. Optimisations which inclusion/membership information makes possible.

Once the inclusion/membership relationship and other forms of information described in the preceeding sections have been made available, one has developed a basis upon which optimisation is possible. Some of the optimisations which come within reach are global in character, and relate to the question of data-structure choice. Others are simple but useful peephole optimisations. One such local optimisation is the following: if $s_1 \subseteq s_2$ is known to hold, then the equality test s_1 eq s_2 can be done as $(\#s_1)$ eq $(\#s_2)$, which is very simple and can be compiled in-line.

The global optimisations which can be based on the analysis presented in the preceeding pages are more numerous, interesting, and significant. Let s and s_1 be variables appearing in a SETL program P, and suppose that $s_1 \subseteq: s$ has been established. Then,as noted in the preceeding section, we can treat s_1 (at the implementation level) as a bitvector, each bit position corresponding to some element of s, and the associated bit/value determining membership/nonmembership in s_1. This implies that the elements of s have been assigned serial numbers; this can be done simply by issuing serial numbers sequentially to elements as they are added to s. The bitvector representing s_1 can be carried with two auxiliary fields, one determining the number of bits in the block which represents it, the other representing the number of elements present in s_1. Elements x for which we can establish $x \in: s$ can be represented as implementation-level pairs consisting of an integer (the serial number of x as a member of s) and a supplementary root-word (giving x in some more explicit way; perhaps in its standard SETL representation, perhaps, if for example x is known to stand in the relationship $x \subseteq: \bar{s}$ to some other set $\bar{s}$, in a bit-vector representation determining the elements of $\bar{s}$ which belong to x). Note that the second component of this pair can in some cases be seen to be unnecessary. If an optimising compiler decides to use these representations, we shall write $s_1 \subseteq::s$ in the first case, $x \in::s$ in the second.

Mappings f for which f $\subseteq:_1$ s can be established can be represented either by a vector v of pointers, v(n) being the value f(v) for the xεs with serial number n, or, if f is never made part of a composite object, by a family of pointers stored directly in the s- representing hashtable. If one of these representations is used, we shall write f $\subseteq_1$::s. If an optimising SETL compiler decides to use these representations (and the number of possibilities among which it must decide will be nicely limited by the set of relationships o $\eta_1 \ldots \eta_k$ η: o_1 which it has been able to verify) then quite a number of code improvements will become possible. Let us examine a few typical cases.

If $s_1 \subseteq$::s and $s_2 \subseteq$::s, equality tests become bit-equality tests, unions and intersections become boolean operations on bit-vectors. If in addition xε::s, then the test x ε s_1 becomes a bit-condition test. Even if yε::s is false, indeed, even if the membership relation yεs remains uncertain, the test yεs_1 can be transformed into the code sequence conveyed by

(1) if $serial_s$(y) is sery eq Ω then f else sery εs_1.

Here, $serial_s$(y) is an implementation-level mapping which transforms each y into its serial number as a member of s, if y if found to be a member of s; otherwise $serial_s$(y) is Ω. If s must support the mapping $serial_s$, the integer values required can be stored directly with the hash-table representing s, so that $serial_s$(y) available wherever the

dynamic test yεs is made. Note that the calculation implied by (1) is no faster than the standard SETL test $y \varepsilon s_1$; however, by keeping s_1 in bitvector form we speed up the tests $x \varepsilon s_1$ when $x \varepsilon :: s$, and can also hope to save space since s_1 is represented in a highly condensed way.

If $s_1 \subseteq :: s$ but $s_2 \subseteq :: s$ is false, then to form the union $s_1 + s_2$ we will have to transform s_1 back into a form compatible with s_2; perhaps the standard SETL form, perhaps some other. The code sequence which results can be that suggested by

(2) $z = \text{copy}(s_2)$; $(\forall\, x \in s \mid \text{serial}_s(x) \in s_1)$ x in z;;

We emphasise that (2) may be no faster than the standard SETL union-forming operation. Indeed, it may be slower, since it involves an iteration over s (rather than over s_1, which is smaller). Nevertheless, keeping s_1 as a bit-vector may yield a worth-while saving in space. And by modifying our implementation technique, we can avoid the loss of speed which the full iteration over s appearing in (2) seems to imply. An approach can be employed which is useful wherever a set $s_1 \subseteq :: s$ appears, explicitly or implicitly, in an iterator. Suppose that this is the case. Then, if s_1 is relatively 'dense' in s, i.e., if $(\# s_1)/(\# s)$ is expected to exceed 10% approximately, then an iterator $(\forall x \varepsilon s_1)$ can without grave inefficiency generate the sequence of a serial numbers n for which the n-th position in the s_1-representing bit-vector is 1, simply by locating nonzero bits by a fast machine-level

process. Suppose next that (# s_1)/(# s) is considerably smaller than this. Then to represent s_1 we can use a list L (of serial numbers) and a bitvector V in combination; the bitvector as before, the list chaining together all those integers which correspond to '1' bits in the bitvector. Then iteration over s_1 can be iteration over this list. If deletions from s_1 must sometimes be made, then L can be a two-way list; alternatively, one may delete the element x with serial number n simply by turning off the n-th bit-position in V, but leaving n in L until the next iteration over L, at which time n can be removed. Note that to choose the most advantageous way of representing L one requires density information of a type hardly likely to be deducible automatically. This information can be elicited interactively or made available through programmer-supplied declarations.

If serial numbers n assigned to elements of s must ever be converted back into standard SETL representations of the objects x which they represent (e.g., on assignment y = x of an xs::s to a variable y not possessing this property) then as noted above x may be represented by a pair whose first component is a serial number and whose second component points to the standard SETL representation of x. If no such conversion is necessary, then this second component can be omitted.

If the set s_1 satisfies $s_1 \subseteq: s$, then the elements of s_1 can be represented simply by bitvectors, possibly supplemented by lists. If s_1 can assume values which are not subsets of s, then we may have to attach a compile-time 'type' field to each value of s_1. This field will define the manner in which the value of s_1 is to be interpreted: whether this be as a bitvector defining a subset of s, a SETL object in its standard representation, or whatever. If f is a mapping or a tuple satisfying $f \subseteq:_2 s$ then f can be treated as a set of ordered pairs in which the second element of each pair is a serial number. If this is done, we shall write $f \subseteq::_2 s$. If s_1 is a set of sets satisfying $s_1 \ni \ni \in: s$, then the elements of s can be represented by bitvectors; if then $s_2 \subseteq: s_1$, we can issue a serial number for each bitvector, and represent the set-of-sets s_2 as a bitvector or as a bitvector supplemented by a list.

8. Proving inequalities among integer-valued variables.

Let o and o_1 be two ovariables of a SETL program P, and suppose that both are known to be of type integer. Then relationships $o \geq 0$, $o_1 \leq 0$, etc. may be provable. In addition to relations of this simple, essentially unary, form, we may also hope to prove binary relationships of the form $o \geq o_1$ etc. Proved assertions ensuring that particular integers appearing in P are necessarily short can be of considerable value, since they can allow these integers to be held in LITTLE rather than in SETL form, possibly yielding great improvements in the speed of arithmetic and indexing.

If o is the integer output of an operation $o = i_1 + i_2$ then $o \geq 0$ will hold if $i_1 \geq 0$ and $i_2 \geq 0$ hold. By using this rule in connection with the analysis algorithm described in section 3, we will be able to prove 'monadic' inequality relationships $o \geq 0$, $o_1 \leq 0$, etc. The proof of binary relationships $o \leq o_1$ will generally depend on the improvements to the analysis algorithm which are outlined in section 6, and, in particular, require use of information obtained the form of the conditions appearing in conditional transfers such as

(1) if $i \leq j$ then go to *label*;

It is instructive to consider the example

(2)
```
...
j = # s;
i = 0;
```

```
label:  (uses of i, but no operations modifying i)
        i = i + 1;
        (more uses of i, but no operations modifying i)
        if i < j then go to label;
        ...
```

The technique outlined in section 6 changes this to what is essentially

```
(2')            ...
                j = # s;
                i = 0;
        label:  (uses of i)
                i = i + 1;
                (more uses of i)
                if i < j then go to label';
                ...
        label': (i = ∃{k < j};) go to label;
```

In this modified program the first group of uses of i chains to the two assignments i = 0 and i = ∃{k < j}. Thus we can be sure that i < j for these uses of i. It follows that i $\leq$ j after i = i + 1 is executed, and hence that i $\leq$ j is valid for all uses of i in the second group.

In a program containing many integers, one may not wish to investigate all possible binary relationships i $\leq$ j. It would then be reasonable to include i $\leq$ j in the set of initially plausible relationships only if a pseudo-assignment

(i = ∃{k < j};) appears in the modified program source. More generally, we can choose to regard certain kinds of relationships R as 'marginally likely apriori' and investigate the truth of such relationships only when this is suggested by some explicit feature of the source code being analysed.

9. Proving 'Nonduplication' of Elements Added to a Set; Single-Valuedness of Maps.

If one knows (by operator-operand tracing) that a SETL program P adds elements to a set s, and removes unspecific elements (essentialy by the from operator); if one knows in addition that s is never used for a membership test x∈s; and if one knows finally that none of the elements added to s are members of s at the moment at which addition is attempted, then we may say that s is being used as a 'simple workpile'. In this case it is very reasonable to implement s either as a pushdown stack or as a list of pointers used in pushdown fashion. In the present section, we shall show how to establish those assertions of the form x ∉ s which are needed to justify such a representation. We achieve this by quite a straightforward adaptation of the methods described in sections 2 and 3. It is convenient to proceed by introducing an explicit notation for set complements into the algebara of relationships given in section 2. Suppose that we denote the complement of s by the symbol $\bar{s}$. Then proceeding much as in section 2 we can define relationships $o_1 \in \bar{o}_2$, $o_1 \subseteq \bar{o}_2$, $o_1 \in_n \bar{o}_2$, $o_1 \subseteq_n \bar{o}_2$, $o_1 \in_\infty \bar{o}_2$, and $o_1 \subseteq_\infty \bar{o}_2$. Moreover, generalised relationships $o_1\ \eta_1 \ldots \eta_k \in \bar{o}_2$ like those introduced in section 4 can also be defined.

The rules defining the effect on these relationships of the various SETL primitives very much resemble those described in sections 2 and 4. As an example, note that the relationship $o \in \bar{o}_1$ will fail if the operation defining o is anything other than $\exists i$, hd i, $i_1(i_2)$, $i_1(i_2,\ldots,i_k)$ or newat. The rules for the first few of these cases are the same as those stated in subcases 1a-1d of section 2. If o is defined by o = newat, then $o \in \bar{o}_1$ is certainly satisfied. Note also that, if additional relationships are carried, then $o \in \bar{o}_1$ may be found to hold in a few additional cases. For example, if in the notation of section 4 we have $o_1 \ni \leq i$ (resp. $o_1 \ni \geq i$) and if o is defined by o = i + 1 (resp. o = i - 1) then $o \notin \bar{o}_1$ will hold.

The relationship $o \subseteq \bar{o}_1$ can hold if o is defined by +, -, *, with, less, $i_1\{i_2\}$, $i_1\{i_2,\ldots,i_k\}$, $\{i_1\}$, $i_1(i_2)$ or $i_1(i_2,\ldots,i_k)$. The rules controlling the validity of $o \subseteq \bar{o}_1$ in the case of defining operation +, *, with, less, $i_1\{i_2\}$, $i_1\{i_2,\ldots,i_k\}$ $\{i_1\}$, $i_1(i_2)$, and $i_1(i_2,\ldots,i_k)$ are very much like those stated in cases 2a and 2c-2e of section 2 (as well as the remarks concerning $o \subseteq o_1$ made in section 4) and will not be restated. Concerning the case in which o is defined by $o = i_1 - i_2$, we note that $o \subseteq \bar{o}_1$ will remain valid if either $i_1 \subseteq \bar{o}_1$ or $o_1 \subseteq i_2$.

Observe that we will need to investigate relationships like $o \in \bar{o}_1$ or $o \subseteq \bar{o}_1$ only if $o \in o_1$ (resp. $o \subseteq o_1$) is plausible. If $o \in o_1$ is implausible, then of course $o \in \bar{o}_1$ is true.

Information useful in deducing relationships $o \in \bar{o}_1$, $o \subseteq \bar{o}_1$, etc. will sometimes be derivable from the form of the conditional transfers which appear in P. A conditional transfer of the form

(*) if x ∈ s then go to label;

can, in much the manner described in section 5, be rewritten as

(2)
```
if x ε s then go to label';
(x = ∋ s̄;)
...
```

which makes the fact s ε s̄ available along the non-transfer branch.

As an example of the foregoing, we may consider the following transitive closure routine

(3)
```
definef tranc(f,s);
new = s; all = s;
(while new ne nl)
        x from new;
L1:     newer = f{x} - all;
L2:     all = all + newer;
L3:     new = new + newer;
end while;
return all;
end tranc;
```

Applying the methods that have been outlined above one will deduce that new ⊆ all, and hence that in line L1 newer ⊆ $\overline{\text{all}}$. From this it follows that newer ⊆ $\overline{\text{new}}$ in line L3. From this an optimising compiler could go on to the decision to represent the set *new* as a simple list.

In dealing with maps it is generally important to know when they are single-valued. When true, this fact can often be proved by a simple method, which as the reader will see is a straightforward variant of the techniques used earlier in the present newsletter. Specifically, we introduce a family of monadic assertions concerning a set f of ordered n-tuples; these assertions are written as $\alpha_n f$, where $n \geq 1$. The assertion $\alpha_n f$ signifies that f is a single-valued mapping of n parameters. This relationship is only plausible if f is a set of (n+1)-tuples or if f is the null set. The necessity rules for $\alpha_n f$ are as follows: the assignment f = nl, or any assignment of the form

$$f = \{\langle x_1,\ldots,x_n,\ e(x_1,\ldots,x_n)\rangle,\ x_1 \in s_1,\ x_2 \in s_2(x_2),\ldots \mid C(x_1,\ldots x_n)\}$$

confirms $\alpha_n f$. For $\alpha_n f$ to hold after an assignment $f(x_1,\ldots,x_n) = e$, it must hold before this assignment. For $\alpha_n f$ to hold after f = f with $\langle x_1,\ldots,x_n,y\rangle$, one of the assertions $x_j \tilde{\in}_j f$ must be provable.

10. Examples

In order to assess the improvements likely to be attained by the optimisation algorithms described in the the proceeding section, we shall consider a few examples. The following small 'transitive closure' routine bears examination:

```
definef tranclose(f, startset);
/* f is a mapping from a set s to itself; startset a subset of s */
tranc = startset; new = startset;
(while new ne nl)
      new = f[new] - tranc; tranc = tranc + new;
end while;
return tranc;
end while;
```

We assume that an optimising compiler is able to recongnise that the return statement should be written as *return copy(tranc);* then the relationships *tranc* $\subseteq$*:s* and *new* $\subseteq$*:s* can be deduced, and hence compilation can proceed on the basis *tranc* $\subseteq$*:: s*, *new* $\subseteq$*::s*. This will lead to the use of a pure bit-vector representation for *tranc*, and a bit-vector-plus-list representation for *new* (since an iteration over *new* is implicit in the operation *f[new]*.) With these representations, the set difference and union operations appearing inside the *while*-loop will be performed either as boolean operations or as a sequence of set-bit/drop-bit steps.

Next we consider one of the 'Huffman code' routines described in O.P.II, p.149. This routine, with a few small changes, is as follows (changes shown in italics):

```
definef huftables(chars, freq);
basis = copy(chars); work = basis; wfreq = copy(freq); k=nl, x=nl;
(while # work gt 1)
```

```
    c1 = getmin work; c2 = getmin work; newat is n in basis;
    ℓ(n) = c1, r(n) = c2;
    wfreq(n) = wfreq(c1) + wfreq(c2); n in work;
end while;
code = nℓ; seq = nulb;  walk(∃ work is top);
return <code, ℓ, r, top>;
end huftables;
definef getmin set;  /* wfreq is global */
<keep, least>= <∃ set is x, wfreq(x)>;
(∀x∈ set)
        if wfreq(x) lt least then <keep, least> = <x, freq(x)>;
end ∀x;
keep out set; return keep;
end getmin;
```

We shall suppose in the discussion which follows that the types of all the quantities used in the above code have been determined. We also assume that an analysis like that described in the proceeding sections but applying even across subprocedures has been applied to the code shown above. Then by applying the techniques described earlier in the present newsletter one will establish the following relationships: work $\subseteq$:basis, c1 $\in$:basis, c2 $\in$:basis, r $\subseteq_1$ basis, ℓ $\subseteq_2$ basis, r $\subseteq_2$ basis, ℓ $\subseteq_2$ basis, wfreq $\subseteq_1$ basis. The technique used in the preceeding section will establish the relationship n $\in$ work for the ivariable n which appears in the statement n in work. These facts can lead an optimising SETL compiler to the data representations work $\subseteq$:basis, c1 $\in$:basis, c2 $\in$:basis, wfreq $\subseteq$:basis. Moreover, since the set *work* is used only to support iterations, and since in particular no membership test x in work ever needs to be performed, *work* can be represented by a simple list. An optimised translation of the *huftables* code might therefore store the values of *wfreq* in fields directly associated with the elements of basis, and represent work as a list of root words.

This would yield code substantially better than that produced from the preceeding program by unoptimised translation. Note however that a programmer implementing this same algorithm might choose to represent the tree functions l and r by arrays. Of course, this can only be done if compensating changes are introduced into whatever code calls the *huftables* routine; and such changes may be beyond the capability of the automatic optimiser we envisage.

As a final example, we consider a portion of the 'interval finder' routine described in O.P.II, pp. 269-270.

The code in question is as follows:

```
definef interval (nodes, x);
/* npreds, followers, and cesor are assumed to be global */
npreds = {<x,0>  x∈ nodes};
(∀ x ∈ nodes, y ∈ cesor(x))
          npreds(y) = npreds(y) + 1;;
int = nult;follows = {x}; count = {<y,0>, y∈nodes};
count(x) = npreds(x);
(while {y ∈ followers | npreds(y) eq count(y)} is newin ne nl)
       (∀ z ∈ newin)
             int(# int + 1) = z;
             z out followers;
             (∀ y ∈ cesor(z)|y ne x)
                    count(y) = count(y) + 1;
                    y in followers;
             end ∀y;
       end ∀z;
end while;
return int;
end interval;
```

```
definef intervals(nodes, entry);
/*followers,follow, intov are all assumed to be global */
ints = nl; seen = {entry}; follow = nl; intov = nl;
(while seen ne nl)
        node from seen;
        interval (nodes, node) is i in ints;
        follow(i) = followers;
        ( 1 ≤ ∀k ≤ # i) intov(i(k)) = i;;
        seen = seen + followers;
end while;
return ints;
end intervals;
```

Once more we suppose that the types of all quantities used in the above code have been determined, and that an analysis like that described in the preceeding sections but applying even across subprocedures has been applied to this code. Then, applying the techniques used in the preceeding section, one will establish the following relationships: npreds $\subseteq:_1$ nodes, int $\in_\infty$ nodes, followers $\subseteq:$ nodes, count $\subseteq:_1$ nodes, newin $\subseteq:$ nodes, xs: nodes, ys: nodes, seen $\subseteq:$ nodes, i $\in:$ nodes, ints $\ni \in_\infty$ nodes; follow $\subseteq:_2$ nodes, i $\subseteq:$ nodes, follow $\ni \in:_2$ nodes, intov $\subseteq_1$ nodes, intov $\subseteq_2$ ints. Moreover, the only sets which must support iterations are *nodes*, *followers*, *newin* and *seen*. An optimising SETL compiler could therefore generate a translation in which the elements of *nodes* and of *ints* carried serial numbers, and in which *x* and *y* were represented by serial numbers, *i* and *int* by a vector of serial numbers, and *followers*, *newin*, and *seen* by list/bitvector combinations. The values of the maps *npreds*, *count* and *follow* can be stored in a group of three fields associated with each *n* ∈ *nodes*: the value of *follow* will be a bitstring.

This gives much better code than that which results from unoptimised translation of the preceeding SETL source. A still more penetrating and global optimiser might be able to deduce that the values *intov* can be stored within a fourth field associated with each $n \in nodes$, and that each value can be represented by a serial number designating some $int \in ints$. This degree of optimisation would come close to matching the code likely to result from manual transcription of the preceeding SETL code into a language of the PL/1 level.