SETL Newsletter 181 W. H. Tsui

A Reformulation of Value-flow Analysis

March 1977

1. Introduction

In OVHL (II), relationships of inclusion/membership between objects of a SETL program are deduced by a process which also deduces relationships between parts of these objects. Relationships of this general nature are written in the form

$$O \; \eta_1 \; \eta_2 \cdots \eta_k \; \rho \; \mu_1 \; \mu_2 \cdots \mu_j \; O' \qquad (*)$$

where O and O' are symbols representing SETL objects, $\eta_1, \eta_2, \ldots, \eta_k$ and $\mu_1, \mu_2, \ldots, \mu_j$ are symbols representing monadic mappings on composite SETL objects, and ρ is a symbol representing a binary Boolean-valued operator on SETL objects. The relationship (*) holds if

$$(\eta_k \; \eta_{k-1} \cdots \eta_1 \; O) \; \rho \; (\mu_1 \; \mu_2 \cdots \mu_j \; O')$$

is true. Although only a few of these relationships are of practical interest, they are all incorporated into the deduction scheme so that the deduction process can be carried out efficiently by standard information propagation methods.

In the following, we will reformulate the problem of value-flow analysis along this line. The problem of value-flow analysis and the problem of inclusion/membership analysis bear strong resemblances to each other. By undertaking this reformulation, we hope not only to obtain a simpler and more efficient value-flow algorithm, but also hope that the reformulation will eventually lead to a unification of the two problems.

In order to express value-flow information in terms of relationships of the form (*), we introduce the relation $\leftarrow$, which intuitively corresponds to the mapping 'crthis' defined in OVHL(I). The relationship $O \leftarrow O'$ holds if the value of the object O' might possibly become the value of the object O during the execution of a program in which O and O' occur. The monadic operators we allow to appear in a $\leftarrow$ relationship are $\ni$, ∞, 1, 2, $\cdots$. As in OVHL(II), we use the symbol $\ni$ to denote the operator which selects an arbitrary element from a set, the symbol ∞ to denote the operator which selects an arbitrary component from a tuple, and n to denote the operator which relates the n-th component from a tuple.

EXAMPLE. In a program P with ovariable o and ivariable i, the relationship $i \ni \infty\, 2 \leftarrow o$ means that during the execution of P, if we apply the operations $\ni$, ∞, 2 sequentially to the current value of i, the value resulting from this sequence of operations might possibly come from some previous value of o. From another point of view, this relationship means also that the current value of o during an execution of P might possibly become the second component of some component of an element of some future value of i.

Let O_p = the set of all value-creating ovariables in a program P,

I_p = the set of all ivariables in P,

$\Sigma_+ = \{\ni, \infty, 1, 2, \cdots\}$,

Σ_+^* = the closure of Σ_+ , i.e. the set of all words over Σ_+ .

Then the relationship between the two formulations of value-flow analysis can be stated precisely as follows:

$$\begin{aligned}
\text{crthis}(q) &= \{o \in O_p \mid q \leftarrow o\} \\
\text{crmemb}(q) &= \{i \in I_p \mid q \ni \leftarrow i\} \\
\text{crsomcomp}(q) &= \{i \in I_p \mid q \infty \leftarrow i\} \\
\text{crcomp}(q,n) &= \{i \in I_p \mid q\, n \leftarrow i\} \\
\text{crpart}(q) &= \{o\colon o \in O_p,\ \gamma \in \Sigma_+^* \mid q\, \gamma \leftarrow o\}
\end{aligned}$$

for each occurrence q in P. This comparison makes it clear that we are interested primarily in relationships of the form $q\ \gamma \leftarrow o$. We will thereby confine our discussion to this type of relationship.

Before we proceed to describe an algorithm for deducing relationships $q\ \gamma \leftarrow o$ in a program, we will first consider an example which offers some insight into the principle that underlies the algorithm.

<u>EXAMPLE</u>

```
L1:  v = read;
L2:  t = <v>;
L3:  s = s + {t};
L4:  p = ∋ s;
L5:  y = p(1);
```

Generally speaking, in order to establish all relationships $q\ \gamma \leftarrow o$ for some fixed ovariable o, the value of o must be traced along all paths originating from o. In the above sequence of straight-line code, the only path from ovariable v_1 is (L1,L2,L3,L4,L5). (Note: We use the symbol v_i to denote the ovariable occurrence of v at Li, and the symbol $v_{i,j}$ to denote the j-th ivariable occurrence of v at Li). To trace the value of v_1, we consider each operation on this path that might transmit the value directly or indirectly. Starting from L1, we know the value of v_1 is created at L1 and then passed on to $v_{2,1}$. Hence we deduce the trivial relationships $v_1 \leftarrow v_1$ and $v_{2,1} \leftarrow v_1$. When L2 is executed, this value is incorporated as the first component of the value of t_2. Hence $t_2\ 1 \leftarrow v_1$ and consequently $t_{3,2}\ 1 \leftarrow v_1$. When L3 is executed, the value of $t_{3,2}$ is inserted into the value of s_3 as one of its elements. As a result, the value of v_1 is indirectly incorporated into the value of s_3 as the first component of one of its elements. Hence $s_3 \ni 1 \leftarrow v_1$ and $s_{4,1} \ni 1 \leftarrow v_1$. When L4 is executed, some random element is picked from the value of $s_{4,1}$ and transmitted to p_4. Since the element selected might be the one with the value of v_1 as its first component, the relationships $p_4\ 1 \leftarrow v_1$ and $p_{5,1}\ 1 \leftarrow v_1$ holds. The first component of this very same element is then extracted at L5. Since this component might possibly be the value of v_1 , we have $y_5 \leftarrow v_1$.

The above chain of reasoning is identical with the theorem generating process of an appropriately defined formal system. The axioms of this formal system include two types of relationships. Relationships of the first type are $v_{2,1} \leftarrow v_1$, $t_{3,2} \leftarrow t_2$, $s_{4,1} \leftarrow s_3$ and $p_{5,1} \leftarrow p_4$, representing the flow of data within the code sequence. Relationships of the second type are $t_2\ 1 \leftarrow v_{2,1}$, $s_3 \ni \leftarrow \ni s_{2,1}$, $s_3 \ni \leftarrow t_{3,2}$, $p_4 \leftarrow \ni s_{4,1}$ and $y_5 \leftarrow 1\ p_{5,1}$, representing the semantics of the instructions. The only inference rule in the formal system is:

$$\text{If } q'\ \alpha \leftarrow \beta\ q \text{ and } q\ \tilde{\beta}\ \gamma \leftarrow v_1 \text{ then } q'\ \alpha\ \gamma \leftarrow v_1 ,$$

where q and q' are occurrences; $\alpha,\beta,\gamma \in \Sigma_+^*$; and $\tilde{\beta}$ is the reversal of β. The theorems of this formal system are precisely those relationships relevant to the transmission of the value of v_1.

In the next section, this formal system will be recast in Kildall's lattice-theoretic framework (or rather, Tarjan's version of it).

2. Formalization of value-flow problems

DEFINITION Given a schematized SETL program P, and an ovariable o in P, the value-flow problem for o is a quintuple (L, F, G, f, a) where

(1) L is a semi-lattice whose elements are sets of words in Σ_+^* and whose binary meet operation is set union.

(2) F is a set of optimizing functions $f: L \to L$.

(3) G is the flow graph (N,E,n_0):
N = the set of nodes representing variable occurrences in P;
$E = E_1 \cup E_2$; E_1 is the set of edges (q,i) for all occurrences q and all ivariables i such that $q \in \text{bfrom}(i)$; E_2 is the set of edges (i,o') for all ovariables o' and all ivariables i in the same instruction as o';
n_0= the initial node representing the ovariable o.

(4) The mapping $f: E \to F$ associates each edge with an optimizing function. If $e_1 \in E_1$, then $f(e_1)(x) = x$ for all $x \in L$. For $e_2 \in E_2$, $f(e_2)$ depends on the instruction corresponding to e_2; e.g. in the instruction
$o = i_1$ <u>with</u> i_2 , we have
$f(i_1,o)(x) = \{\gamma \in x \mid \gamma(1)\ \underline{eq}\ \ni\}$,
$f(i_2,o)(x) = \{\ni\gamma : \gamma \in x\}$,
for all $x \in L$. (Note: $\ni\gamma$ denotes the concatenation of words $\ni$ and γ).

(5) $a: N \to L$ is the initializing function:

$$a(q) = \begin{cases} \{e\} & \text{if q is the ovariable o (e denotes the empty word)} \\ \phi & \text{otherwise} \end{cases}$$

The solution of this problem is defined to be the maximum solution to the set of equations:

$$x_o(q) = \bigwedge_{(q',q)\in E} f(q',q)(x_o(q')) \quad a(a) \;, \quad q \in N \;;$$

heuristically, $x_o(q)$ is a superset closely approximating $\{\gamma \in \Sigma_+^* \mid q\ \gamma \leftarrow o\}$.

The following is a list of representative SETL instructions and the optimizing functions associated with them:

(1) Transfer operation, $o = i_1$:

$f(i_1,o)(x) = x$ since $o \leftarrow i_1$.

(2) Inclusion operation, $o = \{i_1\}$:

$f(i_1,o)(x) = \{\ni \gamma : \gamma \in x\}$ since $o \ni \leftarrow i_1$.

(3) Extraction operation, $o = \ni i_1$:

$f(i_1,o)(x) = \{\gamma(2{:}) : \gamma \in x \mid \gamma(1)$ eq $\ni\}$ since $o \leftarrow \ni i_1$.

(4) Union operation, $o = i_1 + i_2$:

$f(i_1,o)(x) = f(i_2,o)(x)$

$= \{\gamma \in x \mid \gamma(1)$ eq $\ni\}$ since $o \ni \leftarrow \ni i_1$,
$o \ni \leftarrow \ni i_2$.

(5) Difference operation, $o = i_1 - i_2$:

$f(i_1,o)(x) = \{\gamma \in x \mid \gamma(1)$ eq $\ni\}$

$f(i_2,o)(x) = \emptyset$ since $o \ni \leftarrow \ni i_1$.

(6) Tuple-former, $o = \langle i_1,\ldots,i_m\rangle$:

$f(i_j,o)(x) = \{j\ \gamma : \gamma \in x\}$, $1 \le j \le m$ since $o\ j \leftarrow i_j$.

(7) Functional application, $o = i_1(i_2)$:

$f(i_1,o)(x) = \{\gamma(2{:}) : \gamma \in x \mid$ if known(i_2) is n ne Ω then $\gamma(1) \in \{\infty, n\}$ else $\gamma(1) \in (\Sigma_+ - \{\ni\})\}$
$+ \{\gamma(3{:}) : \gamma \in x \mid \gamma(1{:}2)$ eq $\ni 2\}$.

$f(i_2,o)(x) = \emptyset$.

The relatively complicated relationship appearing in (7) reflects the fact that this operation can represent one of three semantically different possibilities. If i_1 is a map, the operation extracts the second component of some element of i_1. Hence $o \leftarrow 2\ 1\ i$. If i_1 is a tuple, the operation extracts the n-th component of its value, n being the current value of i_2. (The mapping 'known' maps an occurrence to its known constant value). In case n is a compile-time constant, we have $o \leftarrow \infty\ i_1$ and $o \leftarrow n\ i_1$. Otherwise we have $o \leftarrow \infty\ i_1$ and $o \leftarrow m\ i_1$ for all positive integers m. If i_1 is a string, a completely new value is created and hence no additional possibilities need to be considered.

(8) Indexed assignment, $o = [i_1(i_2) \leftarrow i_3]$:

$$f(i_1,o)(x) = \{\gamma \in x \mid \text{if known}(i_2)\ \underline{\text{is}}\ n\ \underline{\text{ne}}\ \Omega\ \textbf{then}\ \gamma(1) \in (\Sigma_+ - \{n\})\ \text{else}\ \gamma(1) \in \Sigma_+\}$$

$$f(i_2,o)(x) = \{\ni 1\ \gamma : \gamma \in x\}$$

$$f(i_3,o)(x) = \{\ni 2\ \gamma : \gamma \in x\} + \text{if known}(i_2)\ \underline{\text{is}}\ n\ \underline{\text{ne}}\ \Omega\ \textbf{then}\ \{n\ \gamma : \gamma \in x\}\ \text{else}\ \{\infty\ \gamma : \gamma \in x\}.$$

The semantic grounds for (8) are as follows: If i_1 is a map, a pair is created with the value of i_2 as its first component and the value of i_3 as its second component and then the pair is inserted into the value of o together with elements of the value of i_1. Hence $o \ni 1 \leftarrow i_2$, $o \ni 2 \leftarrow i_3$ and $o \ni \leftarrow \ni i_1$. If i_1 is a tuple, the value of i_3 becomes the n-th component of o, with n being the current value of i_2. For the case that n is a compile time constant, we have $o\ n \leftarrow i_3$, $o\ \infty \leftarrow \infty\ i_1$ and $o\ m \leftarrow m\ i_1$ for all positive integers $m \neq n$. Otherwise, we have $o\ \infty \leftarrow i_3$ and $o\ a \leftarrow a\ i_1$ for all $a \in \Sigma_+ - \{\ni\}$.

(9) Tail extraction, $o = i_1\ (i_2:)$:

$$f(i_1,o)(x) = \{\infty\ \gamma(2:) : \gamma \in x \mid \gamma(1) \in (\Sigma_+ - \{\ni\})\}$$

$$f(i_2,o)(x) = \emptyset.$$

For the time being, we will ignore the case of i_2 being a compile-time constant. So we have $o\ \infty \leftarrow \infty\ i_1$ and $o\ \infty \leftarrow m\ i_1$ for all positive integers m.

(10) Concatenation operation, $o = i_1 \mid\mid i_2$:

$$f(i_1,o)(x) = f(i_2,o)(x)$$
$$= \{\infty\ \gamma(2:) : \gamma \in x \mid \gamma(1) \in (\Sigma_+ - \{\ni\})\}$$

since $o\ \infty \leftarrow \infty\ i_1$, $o\ \infty \leftarrow \infty\ i_2$, $o\ \infty \leftarrow m\ i_1$ and $o\ \infty \leftarrow m\ i_2$ for all positive integers m.

Even without giving a complete specification of f, we can convince ourselves by looking at these examples of optimizing functions that the value-flow problems satisfy all the criteria of a global flow problem as defined by Tarjan, with perhaps one exception, namely the boundedness condition. This observation means that the solution of a bounded value-flow problem can be obtained by any one of the standard methods for solving global flow problems, such as Tarjan's edge listing method or Kildall's workpile method. An algorithm based on the workpile method is given below.

3. An Algorithm for Solving Value-Flow Problems

Given a value-flow problem (L, F, G, f, a) for some ovariable o as input, the following algorithm traces the value of o. The output x is the solution of the value-flow problem in which $x(q) \supseteq \{\gamma \in \Sigma_+^* \mid q\ \gamma \leftarrow o\}$.

```
definef mfp(graph, f, a);
      <nodes, edges, o> = graph;
      (∀q ∈ nodes) x(q) = a(q); end ∀q;
      workpile = edges{o};
      (while workpile ne nℓ)
           <q1,q2> from workpile;
           y(q2) = x(q2);
           x(q2) = x(q2) meet f(q1,q2)(x(q1));
           if x(q2) ne y(q2) then
                workpile = workpile + edges{q2};
           end if;
      end while;
      return x;
end mfp;
```

4. A Concise Algebraic Specification of $f: E \to L$

Let $\Sigma_- = \{\ni^{-1}, \infty^{-1}, 1^{-1}, 2^{-1}, \cdots\}$ where $\ni^{-1}, \infty^{-1}, 1^{-1}, 2^{-1}, \cdots$ represent respectively the inverse of the operators $\ni$, ∞, 1, 2, $\cdots$ in Σ_+. E.g. heuristically speaking, $\ni^{-1}$ maps a value to all the sets containing it. Let $\Sigma = \Sigma_+ \cup \Sigma_-$. Our intention in this section is to represent $f: E \to L$ concisely in terms of a semi-group $(2^{\Sigma^*}, \circ)$. With this objective in mind, we define the binary operation $\circ$ in four steps, namely,

(1) Define $\circ: \Sigma \times \Sigma \to \Sigma^*$ as follows:

(i) For all $a \in \Sigma_+$ and $b \in \Sigma_+$, $a \circ b = ab$.

(ii) For all $a \in \Sigma_-$ and $b \in \Sigma_-$, $a \circ b = ab$.

(iii) $\ni^{-1} \circ \ni = e$ (e denotes the empty word)

$$\left.\begin{array}{l} n^{-1} \circ n = e \\ n^{-1} \circ \infty = e \\ \infty^{-1} \circ n = e \end{array}\right\} \quad \text{for all positive integers } n$$

$\infty^{-1} \circ \infty = e$

(iv) In all other cases, the result is undefined.

(2) Extend $\circ$ to $\Sigma \times \Sigma^* \to \Sigma^*$:

Let $a \in \Sigma$, $\gamma \in \Sigma^*$ and $\gamma = b_1 b_2 \cdots b_k$.

$$\text{Then} \quad a \circ \gamma = \begin{cases} (a \circ b_1) b_2 b_3 \cdots b_k & \text{if } \gamma \neq e \\ a & \text{if } \gamma = e \text{ and } a \in \Sigma_+ \\ \text{undefined} & \text{otherwise.} \end{cases}$$

(3) Extend $\circ$ to $\Sigma^* \times \Sigma^* \to \Sigma^*$:

Let $\gamma_1, \gamma_2 \in \Sigma^*$, $\gamma_1 = a_1 a_2 \cdots a_\ell$.

Then

$$\gamma_1 \circ \gamma_2 = \begin{cases} (a_1 \circ (a_2 \circ \ \ldots \ (a_\ell \circ \gamma_2) \ \ldots \ )) & \text{if } \gamma_1 \neq e \\ \gamma_2 & \text{otherwise} \end{cases}$$

(4) Finally, extend $\circ$ to $2^{\Sigma^*} \times 2^{\Sigma^*} \to 2^{\Sigma^*}$:
Let $x_1, x_2 \in 2^{\Sigma^*}$. Then

$$x_1 \circ x_2 = \{\gamma_1 \circ \gamma_2 : \gamma_1 \in x_1, \ \gamma_2 \in x_2 | \gamma_1 \circ \gamma_2 \neq \Omega\}.$$

It is straightforward to write f in the form of

$$f(i,o)(x) = T_{i,o} \circ x ,$$

with $T_{i,o}$ being a set of words over Σ. Here is a sample of $T_{i,o}$ corresponding to f(i,o) discussed in section 2.

(1) transfer operation, $o = i_1$:

$$T_{i_1,o} = \{e\}$$

(2) inclusion operation, $o = \{i_1\}$:

$$T_{i_1,o} = \{\ni\}$$

(3) extraction operation, $o = \ni i_1$:

$$T_{i_1,o} = \{\ni^{-1}\}$$

(4) union operation, $o = i_1 + i_2$:

$$T_{i_1,o} = T_{i_2,o} = \{\ni \ni^{-1}\}$$

(5) difference operation, $o = i_1 - i_2$:

$$T_{i_1,o} = \{\ni \ni^{-1}\}$$

$$T_{i_2,o} = \emptyset$$

(6) tuple-former, $o = \langle i_1, \cdots, i_m\rangle$:

$$T_{i_j,o} = \{j\} , \quad 1 \leq j \leq m$$

(7) functional application, $o = i_1(i_2)$:

$$T_{i_1,o} = \{2^{-1} \ni^{-1},\ \text{if known}(i_2)\ \underline{\text{is}}\ n\ \underline{\text{ne}}\ \Omega\ \text{then}\ n^{-1}\ \text{else}\ \infty^{-1}\}$$

$$T_{i_2,o} = \emptyset$$

(8) indexed assignment, $o = [i_1(i_2) \leftarrow i_3]$:

$$T_{i_1,o} = \{aa^{-1}\colon a \in \Sigma_+\} - \text{if known}(i_2)\ \underline{\text{is}}\ n\ \underline{\text{ne}}\ \Omega\ \text{then}\ \{n\ n^{-1}\}\ \text{else}\ \emptyset$$

$$T_{i_2,o} = \{\ni 1\}$$

$$T_{i_3,o} = \{\ni 2\ ,\ \text{if known}(i_2)\ \underline{\text{is}}\ n\ \underline{\text{ne}}\ \Omega\ \text{then}\ n\ \text{else}\ \infty\}$$

(9) tail extraction, $o = i_1(i_2{:})$:

$$T_{i_1,o} = \{\infty\ \infty^{-1}\}$$

$$T_{i_2,o} = T_{i_3,o} = \emptyset$$

(10) concatenation operation, $o = i_1 || i_2$:

$$T_{i_1,o} = T_{i_2,o} = \{\infty\ \infty^{-1}\}.$$

A complete specification of T for all SETL primitives is given in Appendix I.

5. Unbounded value-flow problems

As mentioned earlier, the algorithm of section 3 does not necessarily converge. Consider the following program:

```
L1:  s = nℓ;
L2:  (while ...)
L3:      s = {s};
L4:  end while;
```

The relationship $s_3 \ni^i \leftarrow s_1$ holds for all $i \geq 1$, meaning that we can be confronted with a solution of $x_{s_1}(s_3)$ that is infinite, containing words $\ni$, $\ni\ni$, $\ni\ni\ni$, ... of arbitrary length. The algorithm shown above can never arrive at this solution. It simply keeps on incrementing the value of $x_{s_1}(s_3)$ forever. In general, the algorithm fails to terminate whenever its input is a program in which $o\ \gamma \leftarrow o$ holds for some ovariable o and some non-empty word $\gamma \in \Sigma_+^+$. We will call these culprits self-dependent ovariables. Once we have recognized its source, the problem of divergence can be avoided, for example, by either of the following modifications of our earlier method:

Method I (a la Fong, Kam and Ullman)

Change the initialization of a value-flow problem to $a(o') = \Sigma_+^*$ for each self-dependent ovariable o'. (Note: Σ_+^* is the zero-element of the semi-lattice L). This will force the value of $x_o(o')$ to remain at Σ_+^* and hence the algorithm will terminate. By underestimating $x_o(o')$ this way, value-flow information concerning o' will be lost. Nevertheless, this underestimation has no effect on other occurrences whose values do not depend on o'.

Use of this method makes it necessary to determine whether an ovariable is self-dependent or not. Fortunately, there are many good criteria for deciding which ovariable might possibly be self-dependent. For instance, the algorithm itself might be able to make such a decision. Recall that the algorithm always approaches its solutions from below, incrementing the value of $x_o(q)$ iteratively. As $x_o(q)$ grows larger, we can be more and more certain that its solution is an infinite set. Therefore, we can fix some arbitrary limit on the size of each $x_o(q)$ and whenever that limit is exceeded, set $x_o(q)$ to Σ_+^*.

Method II (a la Tenenbaum)

The basic idea of this method is to limit the length of γ in a relationship $q\ \gamma \leftarrow o$. Let $\pi = \Sigma_+ \cup \{\omega\}$ and ω be a monadic operation which maps a value to all its parts. Let $L' = 2^{\pi^*}$. Define the operation $\circ'$: $2^{\Sigma^*} \times L' \rightarrow L'$ the same way $\circ$ is defined, except that

(1) $a \circ' \omega = \omega$ for all $a \in \Sigma$, and

(2) given $a \in \Sigma$, $\gamma \in \pi^*$, then

$$a \circ' \gamma = \text{if } |a\circ\gamma| \leq \ell \text{ then } a\circ\gamma$$
$$\text{else } (a\circ\gamma)(1{:}\ell)\ ||\ \omega,$$

where ℓ is some fixed number.

By replacing the operation $\circ$ by $\circ'$ and the semi-lattice L by L', the algorithm will converge for all value-flow problems. This modification will retain considerably information on self-dependent ovaraibles. However, this is done at the expense of losing information concerning other variable occurrences. Since this method keeps track of values in a composite object only up to a certain level of depth, some amount of information on occurrences with deeply nested values is bound to be lost.

6. Backward Value Tracing

So far we have only discussed the problem of tracing a value in the forward direction. We have seen how relationships $q\ \gamma \leftarrow o$ are deduced for each ovariable o. It seems appropriate at this time to consider the problem of tracing a value in the backward direction and seeing how relationships $q \leftarrow \gamma\ o$ can be deduced for each occurrence q. Because of the symmetric nature of our formulation, these two problems are almost identical. To deduce all $q \leftarrow \gamma\ o$ for some fixed q, we can use the same algorithm we used before to find all $q' \leftarrow \gamma\ q$, except that we must first reverse all the edges in the flow graph, and also make use of the following $\tilde{f}$ in place of the former f:

$$\tilde{f}(o,i)(x) = T_{o,i}\ \tilde{\circ}\ (x)\ , \quad \text{for all}\ \ x \in L\ ,$$

where

$$T_{o,i} = \{a_{\ell}^{-1}\ a_{\ell-1}^{-1} \cdots a_{1}^{-1} : a_1 a_2 \cdots a_{\ell} \in T_{i,o}\}\ ,$$

and

$$x_1\ \tilde{\circ}\ x_2 = \{\widetilde{(\gamma_1 \circ \tilde{\gamma}_2)} : \gamma_1 \in x_1,\ \gamma_2 \in x_2 | \gamma_1 \circ \tilde{\gamma}_2 \neq \Omega\}.$$

(Note: $\tilde{\gamma}$ is the reversal of γ.)

The ability to trace values in the backward as well as the forward direction is important. With both options, the algorithm can be used to compute each value of the value flow functions (i.e. the cr functions) efficiently independent of any other value. Preferably, forward tracing is used to compute $\text{crthis}^{-1}(q)$, $\text{crmemb}^{-1}(q)$, $\text{crpart}^{-1}(q)$, etc. and backward tracing is used to compute crthis(q), crmemb(q), etc. This allows an implementor the freedom of not storing the entire cr functions, but computing a value each time it is needed. This is particularly significant if the optimizer requires only a few values of the cr function. As it now stands, the optimizer uses value-flow information only in the determination of control flow and the evaluation of the destructive use condition. In the first application, the optimizer need only know $\text{crthis}^{-1}(o)$ for the definition o of each procedure. In the second application, the optimizer need only know crthis(i) for each ivariable i in a modification instruction and $\mathrm{crpart}^{-1}(o)$ for each o in some of these crthis(i).

7. Evaluation of the Destructive Use Condition

In OVHL(I), the destructive use condition for an ivariable i is stated in terms of the set $\ell(i)$ of all ovariables whose current value might possibly incorporate the value of i:

An ivariable i may be used destructively if its value is dead, i.e. for all ovariables o in $\ell(i)$, there does not exist a path from o to some ivariable of v(o) through i that is clear of any ovariable occurrence of v(o). (Note: v(o) denotes the variable of o.)

The set $\ell(i)$ is estimated according to the formula:

$$\ell(i) = \mathrm{crpart}_{p_i}^{-1}[\mathrm{crthis}(i)]$$

where p_i = exsinthis(i) is the set of instructions which might have been executed between the time a value is created and the time it reappears as the value of i; and $\mathrm{crpart}_{p_i}^{-1}(o)$ is the subset of $\mathrm{crpart}^{-1}(o)$ containing all ovariables o' such that the value created at o might possibly be incorporated into the value of o' along some path lying entirely in p_i.

The method described in OVHL(I) for computing the function exsinthis is extremely and unnecessarily expensive. It solves a set of equations that are almost identical to the equations for crthis. This seemingly redundant computation is necessary because the cr functions do not contain enough information from which exsinthis can be directly evaluated. In this section, we will examine a simple and fast technique for computing exsinthis(i) from the $\leftarrow$ relationships.

Given an ivariable i and an ovariable o $\in$ crthis(i). We say an occurrence q <u>transmits</u> the value of o to i if during an execution of the program, the value of o might be incorporated into the value of q and then subsequently it is extracted from q and becomes the value of i, i.e. $q\ \gamma \leftarrow o$ and $i \leftarrow \tilde{\gamma}\ q$ for some $\gamma \in \Sigma_+^*$. It is clear that an ovariable other than o transmits the value of o to i only if one of the ivariables in the same instruction also transmits the value of o to i. It is also clear that to find exsinthis(i), it suffices to find the set of all occurrences that transmit the value of some ovariable in crthis(i) to i. The argument for the latter claim goes as follows:

By the definition of exsinthis, an instruction belongs to exsinthis(i) if and only if there exists a path p from some ovariable o $\in$ crthis(i) to i along which the value of o is transmitted to i. On such a path p, there must exist a sequence of instructions L, L2, ..., Ln such that

(1) For $1 \leq j < n$, the ovariable o_j of instruction Lj transmits the value of o to i; and

(2) For $1 < j \leq n$, there is an ivariable $i_j \in du(o_{j-1})$ in the instruction Lj that transmits the value of o to i. ($o_1 = o$ and $i_n = i$.)

$$\begin{tikzcd}
\text{L1:} & o \arrow[drr, bend right, "du"] & = op_1(\cdots) & & & /\!*\ o_1 = o\ */ \\
\text{L2:} & o_2 \arrow[drr, bend right, "du"] & = op_2(\cdots, & i_2 & , \cdots) & \\
\text{L3:} & o_3 & = op_3(\cdots, & i_3 & , \cdots) & \\
 & & \vdots & & & \\
\text{L(n-1):} & o_{n-1} \arrow[drr, bend right, "du"] & = op_{n-1}(\cdots, & i_{n-1} & , \cdots) & \\
\text{Ln:} & o_n & = op_n(\cdots, & i, & \cdots) & /\!*\ i_n = i\ */
\end{tikzcd}$$

<u>Figure 1</u>. The path p = (L1,$\cdots$,L2,$\cdots$,L3,$\cdots$,L(n-1),$\cdots$,Ln)

Finding exsinthis(i) is simply a matter of enumerating all such value-transmitting paths. The above criteria (necessary but not sufficient) for paths that transmit the value to i tell us how the enumeration can be done statically. First find the set C(i) of all ivariables that transmit the value of some $o \in$ crthis(i) to i, and then link these ivariables up by data flow information. Specifically, we can use the formula:

$$(*) \quad \text{exsinthis}(i) = \bigcup_{i' \in C(i)} \text{chainback}(i') .$$

Note that chainback(i') is, as in OVHL(I), the set of all instructions which lie along some v(i')-clear path beginning at a definition of v(i') and terminating at i'.

In our formulation, the relationships $i \leftarrow \gamma\ i'$ can be deduced by tracing backward the value of i and the relationships $i'\ \tilde{\gamma} \leftarrow o$ can be deduced by tracing forward the value of each $o \in$ crthis(i). The set C(i) of ivariables i' transmitting the value of some $o \in$ crthis(i) to i can thus be found. Hence exsinthis(i) can be computed efficiently according to formula (*).

In fact, an improvement on (*) can be made:

$$(**) \quad \text{exsinthis}(i) = \bigcup_{\substack{\text{all ivariabes } i' \text{ such that} \\ i \leftarrow \gamma\ i' \text{ for some } \gamma \in \Sigma_+^*}} \text{chainback}(i').$$

The use of (**) will eliminate the need for forward tracing. It will also solve one of the most annoying technical problems in value flow analysis -- the handling of multi-value creating instructions.

In general, more than one value (pointer) can be created at a SETL primitive instruction. In order for the destructive use condition to be valid, we must find a way of registering the creation of all these values. In the formulation of OVHL(I), we can either insert auxiliary instructions to account for all these values or regard all of them as values of the ovariable in the instruction.

In the extreme situation, we have the read operation o = read which creates an indefinite number of values (pointers). Consider the following example:

```
L1:  s = read;
L2:  x = ∋s;
L3:  x = x with y;
L4:  t = s + s';
```

Here we have $\text{crthis}(x_{3,1}) = \{x_2\}$ and consequently $\ell(i) = \text{crpart}_{p_i}^{-1}[\text{crthis}(x_{3,1})] = \{x_2\}$. Since the variable x is redefined at L3, we come to the wrong conclusion that $x_{3,1}$ may be used destructively, while in fact the value of $x_{3,1}$ is a member of the current value of s_1, which is very much alive at L3.

To avoid this error, we can either insert a sequence of auxiliary instructions before L1, simulating the SETL input routine, or we can modify the equations for the cr functions so that

$$\begin{aligned} \text{crthis}(s_1) &= \{s_1\}, \\ \text{crmemb}(s_1) &= \{s_1\}, \text{ and} \\ \text{crsomcomp}(s_1) &= \{s_1\}, \end{aligned}$$

treating the read operation as if all the values it creates are stored in s one time or another.

Similar adjustments should be made for other multi-value creating instructions. In the case of the power set instructions o = pow(i) and o = npow(n,i), the power set o is formed by first creating the subsets of i. In order to account for the creation of these subsets, we can use a dummy temporary t and expand o = pow(i) to

```
    o = pow(i)
    t = aux___arb(i)                /* t = ∋ i */
    t = aux___set(t)                /* t = {t} */
    t = aux___set(t)                /* t = {t} */
    o = aux___or(o,t)               /* o=if junk then o else t*/
```

(Similar expansion for o = npow(n,i)). Or if we want to avoid this expansion, we can modify the equations by setting crmemb(o)= crthis(i) for the instructions o = pow(i) and o = npow(n,i). This modification is necessary regardless of whether the value of i is always copied before inserting into the power set, or in the case of o = npow(n,i), whether the size of the set i is known to differ from n.

Indexed assignments such as f(x) = y are also multi-value creating instructions. Semantically, when f is a map, the instruction f(x) = y first creates a pair <x,y> and then inserts this pair into the map f, replacing any old pair <x,z>. To account for the creation of the pair <x,y>, f(x) = y can be expanded to

```
    f(x) = y
    t    = aux___pair(x,y)          /* t = <x,y> */
    f    = f aux___with t           /* f = f + {t} */
```

(The effect on indexed assignment is not as significant as in the case of read operations and pow set operations, since it is very unlikely that a map is implemented as a set of pairs).

Neither of the above two solutions is clean, especially when auxiliary instructions are involved. On the other hand, the difficulty of handling multi-value creating instructions is totally avoided in a backward oriented method which uses formula (**) to compute P_i = exsinthis(i) and the formula

(***) $\quad \ell(i) = C'(i)$

to compute $\ell(i)$, where C'(i) is the set of all ovariables o such that for some ovariable o' and some $\gamma,\gamma' \in \Sigma_+^*$,

$i \leftarrow \gamma' \, o'$ and $o \, \gamma \leftarrow \gamma' \, o'$ within P_i. Since C'(i) not only accounts for ovariables in crthis(i) but all other possible sources of the value of i as well, there is no need to require all values to be created explicitly at some ovariable.

E.g. in the example given above for the read operation, the backward tracing alone will discover that $x_{3,1} \leftarrow \ni s_{2,1}$ and of course $x_{3,1} \leftarrow x_{3,1}$. By (**), exsinthis$(x_{3,1})$ = {L1,L2,L3}. By (***), $\ell(i) = \{s_1, x_2\}$. Though x is dead, s is still alive when L3 is executed. Hence we come to the right conclusion that $x_{3,1}$ cannot be used destructively.

8. Conclusion

We have formulated the problem of value flow analysis in terms of the formal lattice-theoretic framework of Kildall. Besides its theoretical significance, we believe the algorithm based on this reformulation has practical value. In particular, it computes the individual values of the cr functions separately and efficiently, which will often make it unnecessary to store the entire function. We also collect considerably more information than is immediately available in the cr functions of OVHL. Using this information, the function 'exsinthis' can be evaluated by a simple and fast algorithm. Last but not least, we are optimistic about the possibility of using the technique that we have sketched to unify the three major forms of analysis performed by the SETL optimizer, and we see our reformulation as a step toward this goal.

Appendix I. Specification of $f: E \rightarrow F$

In this appendix we specify the function $f: E \rightarrow F$ by tabulating the matrix T. (Note $f(i,o)(x) = T_{i,o} \circ x$). For an instruction $o = op(i_1,\ldots,i_j,\ldots,i_m)$, the value of $T_{i_j,o}$ depends mainly on the opcode op and the operand number j.

opcode	operand No.	list of elements in $T_{i_j,o}$
op ⌉ add	1	$\ni\ni^{-1}$
	2	$\ni\ni^{-1}$
op ⌉ cc	1	$\infty\infty^{-1}$
	2	$\infty\infty^{-1}$
op ⌉ less	1	$\ni\ni^{-1}$
op ⌉ lessf	1	$\ni\ni^{-1}$
op ⌉ mod	1	$\ni\ni^{-1}$
	2	$\ni\ni^{-1}$
op ⌉ mult	1	$\ni\ni^{-1}$
	2	$\ni\ni^{-1}$
op ⌉ repl	2	$\infty\infty^{-1}$
op ⌉ sub	1	$\ni\ni^{-1}$
op ⌉ with	1	$\ni\ni^{-1}$
	2	$\ni$
op ⌉ arb	1	$\ni\ ^{-1}$
op ⌉ dom	1	$\ni 1^{-1}\ni^{-1}$
op ⌉ range	1	$\ni 2^{-1}\ni^{-1}$
op ⌉ rand	1	$\ni^{-1}, \infty^{-1}$
op ⌉ end	1	$\infty\infty^{-1}$
op ⌉ subst	1	$\infty\infty^{-1}$
op ⌉ set 1	1	$\ni$
op ⌉ pair	1	1
	2	2
op ⌉ next	1	$\ni^{-1}$
op ⌉ nextd	1	$1^{-1}\ni^{-1}$
op ⌉ nxtinc	1	$1^{-1}\ni^{-1}$

opcode	operand No.	List of elements in $T_{i_j,o}$
op ¬ of	2	$2^{-1}\ni^{-1}$, if known(i_1) <u>is</u> n <u>ne</u> Ω then n^{-1} else ∞^{-1}
op ¬ oft	2	if known(i_1) <u>is</u> n <u>ne</u> Ω then n^{-1} else ∞^{-1}
op ¬ ofa	2	$\ni 2^{-1}\ni^{-1}$
op ¬ ofb	2	$\ni 2^{-1}\ni^{-1}$
op ¬ pow	1	$\ni\ni\ni^{-1}$
op ¬ npow	2	$\ni\ni\ni^{-1}$
op ¬ argin	1	e
op ¬ argout	1	e
op ¬ asn	1	e
op ¬ fval	1	e
op ¬ retasn	1	e
op ¬ sof	0	e
	1	1
	2	$\ni 2$, if known(i_1) <u>is</u> n <u>ne</u> Ω then n else ∞
op ¬ sofm	0	$\ni\ni^{-1}$
	1	$\ni 1$
	2	$\ni 2$
op ¬ sofa	0	$\ni\ni^{-1}$
	1	$\ni 1$
	2	$\ni 2\ni^{-1}$
op ¬ send	0	e
	2	$\infty\infty^{-1}$
op ¬ ssubst	0	e
	3	$\infty\infty^{-1}$
aux ¬ arb	1	$\ni^{-1}$
aux ¬ asn	1	e
aux ¬ or	1	e
	2	e

opcode	operand No.	list of elements in $T_{i_j,o}$
aux ⌝ dis	1	e
	2	e
aux ⌝ with	1	$\ni\ni^{-1}$
	2	$\ni$
op ⌝ set	1	$\ni$
op ⌝ tup	1	(arg 1 of i_1)
op ⌝ ofn	2	$\underbrace{2^{-1}\ 2^{-1}\ \ldots\ 2^{-1}}_{\text{arg2 times}}\ \ni^{-1}$

Appendix II. A More Sophisticated Approach to the Nonconvergence Problem

Neither of the two methods described in section 5 for solving the nonconvergence problem is an intellectually satisfying solution in the sense that they both lose information that is otherwise obtainable by the algorithm of OVHL(I). Despite the fact that the information lost is for all practical purposes extremely insignificant, we wish to consider other alternatives. In this appendix, we will examine one alternative that makes use of the fact that each $x_o(q)$ in the solution of a value-flow problem is a regular set and thus can be represented by a regular expression. The claim that $x_o(q)$ is a regular set can be verified by constructing a finite automaton for $x_o(q)$.

Given a path $p = (q_1, q_2, \ldots, q_n)$ in a flow graph with $q_1 = o$, the initial node, and $q_n = q$, we call a word α in Σ^* a label of the path p if $\alpha = \alpha_1\alpha_2 \cdots \alpha_{n-1}$, where $\alpha_i \in T_{q_i,q_{i+1}}$, $1 \leq i \leq n$. Suppose we also label each individual edge (q,q') in the flow graph by the words in $T_{q,q'}$. Then obviously the set $y_o(q)$ of labels of all paths from o to q is regular, since the labeled flow graph can easily be transformed into the state diagram of a nondeterministic finite automaton (NFA) for $y_o(q)$, by label splitting and e elimination. The reversal $\tilde{y}_o(q)$ of $y_o(q)$ is also regular. Furthermore, $\gamma \in x_o(q)$ iff $\gamma \in \Sigma_+^*$ and γ can be obtained from a word in $y_o(q)$ by applying repeatedly the cancellation rules:

$$\ni^{-1}\ni = e$$
$$\infty^{-1}\infty = e$$
$$\left.\begin{array}{l} \infty^{-1}\, n = e \\ n^{-1}\, n = e \\ n^{-1}\, n = e \end{array}\right\} \quad \text{for all positive integers } n.$$

Hence the state diagram for $x_o(q)$ can be obtained from the state diagram for $\tilde{y}_o(q)$ by (i) adding label e for edge (q_1,q_3) whenever there are edges (q_1,q_2) and (q_2,q_3) with labels a and b respectively such that ab = e in accordance with the cancellation rules; (ii) applying the standard e elimination procedure, and then (iii) eliminating all labels in $\{e\} \cup \Sigma_-$.

The above construction can be summarized in the following procedure:

```
(1)  Consider each edge $(q_1,q_2)$ in the flow graph.
     For each word $\gamma = a_1\ a_2 \cdots a_\ell$ in $T_{q_1,q_2}$ :
     if $|\gamma| \leq 1$, label the edge $\gamma$;
     otherwise, create new nodes $n_1,n_2,\cdots,n_{\ell-1}$
          and edges $(q_1,n_1),(n_1,n_2),\cdots, (n_{\ell-2},n_{\ell-1}),(n_{\ell-1},q_2)$
          and label these edges $a_\ell,a_{\ell-1}, \cdots a_1$ respectively.
(2)  If there is a pair of edges $(q_1,q_2)$, $(q_2,q_3)$ with labels
     a and b such that $b \circ a = c$ and $c \in \{e\} \cup \Sigma_+$ , label the
     edge $(q_1,q_3)$  c.  (If the edge $(q_1,q_3)$ does not exist,
     introduce it.)
(3)  Repeat (2)  until no new label can be generated.
     (Note: An edge can have more than one label.)
(4)  If there is a cycle along which all edges have label e,
     merge all nodes on this cycle.
(5)  Delete all labels in $\{e\} \cup \Sigma_-$ from the graph, except the
     label e on (o,q) if there is one.
(6)  Delete all edges with no label.
(7)  Reverse all the edges; exit.
```

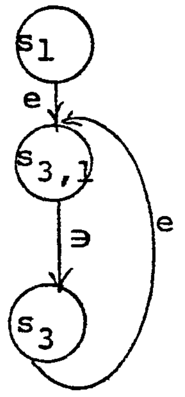
The graph obtained by this procedure is the state diagram representing an NFA for $x_o(q)$ with initial state q and final state o.

EXAMPLE. Consider the example given in section 5.

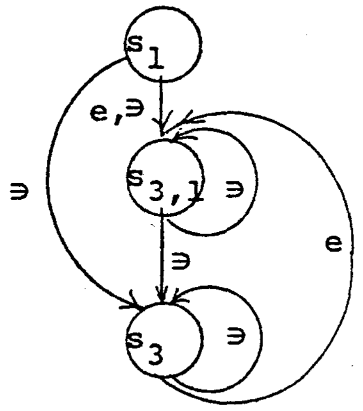
```
L1:  s = nℓ;
L2:  (while ...)
L3:     s = {s};
L4:  end while;
```

The NFA for $x_{s_1}(s_3)$ can be constructed in these steps according to the above procedure:

(1) The flow graph for tracing the value of s_1 is labelled as follows:

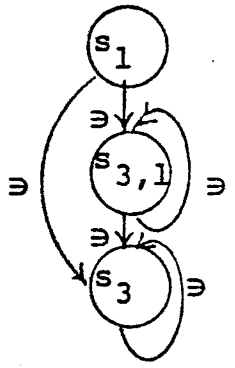


(2) and (3). Apply the rules repeatedly until convergence:

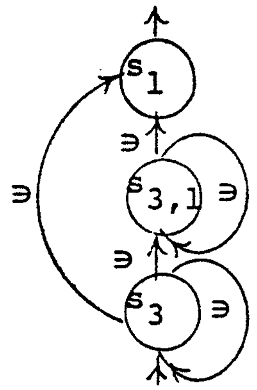


(4) Step (4) is not applicable since there is no cycle with all its edges labelled e.

(5) and (6). Delete labels e and then delete edges with no labels:



(7) Reverse the edges; then we have the NFA for $x_{s_1}(s_3)$:



The NFA's for $x_{s_1}(s_1)$, $x_{s_1}(s_{3,1})$, $x_{s_1}(s_1)$, $x_{s_3}(s_{3,1})$, $x_{s_3}(s_3)$ can be constructed analogously.

We now proceed to construct a global flow problem $(\bar{L}, \bar{F}, G, \bar{f}, \bar{a})$ isomorphic to a given value-flow problem (L, F, G, f, a). Since two regular expressions are said to be equal iff they represent the same regular set, there is a 1-1 correspondence between regular expressions and regular sets. More specifically, we let $r(x) = \bar{x}$ iff $\bar{x}$ is a regular expression representing the regular set x. We will build the semi-lattice $\bar{L}$ and subsequently the global flow problem $(\bar{L}, \bar{F}, G, \bar{f}, \bar{a})$ on this 1-1 correspondence. Let $\bar{L}$ be the semi-lattice whose elements are the regular expressions over Σ_+ and where the meet operation $|$ is defined as:

$$R_1 \mid R_2 \equiv \begin{cases} R_1 + R_2 & \text{if } R_1, R_2 \neq \emptyset, \\ R_1 & \text{if } R_2 = \emptyset \quad , \\ R_2 & \text{otherwise} \quad , \end{cases}$$

for $R_1, R_2 \in \bar{L}$. Let $\bar{F}$ be the set of functions $f\colon \bar{L} \to \bar{L}$. Let $\bar{a}(q) = r(a(q))$ for all nodes q in G. Since $a(q)$ equals either $\{e\}$ or $\emptyset$, $\bar{a}(q)$ is either e or $\emptyset$. It remains to find $\bar{f}$ such that $\bar{f}(q,q')(r(x)) = r(f(q,q')(x))$ for all edges (q,q') and for all $x \in L$. So we let $\bar{f}(q,q')(R) = T_{q,q'} \bar{\circ} R$ for all $R \in \bar{L}$ where $\bar{\circ}$ is defined similarly to $\circ$ as follows:

(1) Define $\bar{\circ} : \Sigma \times (\Sigma \cup \{e,\emptyset\}) \to \bar{L}$:

For $a \in \Sigma_+$, $b \in \Sigma$, $a \bar{\circ} b = ab$

For $a \in \Sigma_+$, $a \bar{\circ} \emptyset = \emptyset$, $a \bar{\circ} e = a$

For $a \in \Sigma_-$, $b \in \Sigma_-$, $a \bar{\circ} b = ab$

For $a \in \Sigma_-$, $a \bar{\circ} e = \emptyset$,

$a \bar{\circ} \emptyset = \emptyset$,

and $\ni^{-1} \bar{\circ} \ni = e$

$$\left.\begin{aligned} n^{-1} \bar{\circ}\, n &= e \\ n^{-1} \bar{\circ}\, \infty &= e \\ \infty^{-1} \bar{\circ}\, n &= e \end{aligned}\right\} \text{ for all positive integers } n,$$

$\infty^{-1} \bar{\circ} \infty = e$.

(2) Extend $\bar{\circ}$ to $\Sigma \times \bar{L} \to \bar{L}$:

Given $R \in \bar{L}$, $a \in \Sigma$, then

(i) if $R = R_1 R_2 \cdots R_k$,

$$a \bar{\circ} R = (a \bar{\circ} R_1)R_2 \cdots R_k + (a \bar{\circ} R_2)R_3 \cdots R_k + \ldots + (a \bar{\circ} R_j)R_{j+1} \cdots R_k$$

where j is the smallest positive integer such that $e \notin r^{-1}(R_j)$.

(ii) if $R = R_1 + R_2 + \ldots + R_k$,

$$a \bar{\circ} R = (a \bar{\circ} R_1) + (a \bar{\circ} R_2) + \cdots + (a \bar{\circ} R_k).$$

(iii) if $R = (R_1)^*$,

$$a \bar{\circ} R = (a \bar{\circ} R_1)R_1^* + a \bar{\circ} e \ .$$

(3) Extend $\bar{\circ}$ to $\Sigma^* \times \bar{L} \to \bar{L}$:

Given $\gamma = a_1 a_2 \cdots a_\ell$; $a_i \in \Sigma$, $1 \le i \le \ell$ and $R \in \bar{L}$

$$\gamma \bar{\circ} R = \begin{cases} (a_1 \bar{\circ} (a_2 \bar{\circ} \cdots (a \ \bar{\circ} R) \cdots)) & \text{if } \gamma \neq e \\ R & \text{otherwise.} \end{cases}$$

(4) Finally, extend $\bar{\circ}$ to $2^{\Sigma^*} \times \bar{L} \to \bar{L}$:

Given $x = \{\gamma_1, \gamma_2, \cdots, \gamma_k\}$, $\gamma_i \in \Sigma^*$ and $R \in \bar{L}$,

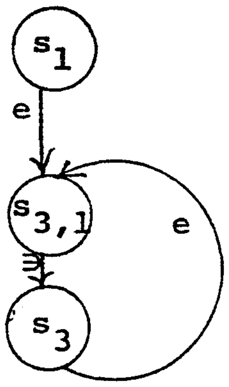
$$x \bar{\circ} R = (\gamma_1 \bar{\circ} R) + (\gamma_2 \bar{\circ} R) + \cdots + (\gamma_k \bar{\circ} R).$$

The upshot of this development is that the solution of the global flow problem $(\bar{L}, \bar{F}, G, \bar{f}, \bar{a})$ represents the solution of the value-flow problem (L,F,G,f,a). That is, $\bar{x}_o(q) = r(x_o(q))$ for all variable occurrences q. It is true that, because of this isomorphism, the algorithm of section 3 will not converge for $(\bar{L}, \bar{F}, G, \bar{f}, \bar{a})$ if it fails to converge for (L,F,G,f,a). But the main difference is that $\bar{x}_o(q)$ always has a finite representation. We can generate a closure of regular expressions by oracles such as Rudin's rule, i.e. $x = ax + b \Rightarrow x = a^*b$. Hence the problem of nonconvergence will be avoided if the algorithm of section 3 is modified as follows: Whenever the algorithm updates the value of $\bar{x}_o(q)$ for some occurrence q, it should also check whether q may be a self-dependent ovariable, and if so, update $\bar{x}_o(q)$ further by setting it to $\bar{x}_q(q) \ || \ \bar{x}_o(q)$, i.e. the concatenation of regular expressions $\bar{x}_q(q)$ and $\bar{x}_o(q)$. The regular expression $\bar{x}_q(q)$ will contain at least one closure symbol * if q is indeed a self-dependent ovariable. To find $\bar{x}_q(q)$, we can construct a NFA for it, as described earlier, and then construct a regular expression from the NFA, for instance, by algorithm 2.1 on page 106 in Aho and Ullman, Volume I.

EXAMPLE Let us again consider the example given in section 5.

```
L1:  s = nℓ;
L2:  (while ... )
L3:     s = {s};
L4:  end while;
```

The flow graph for tracing the value of s_1 in the forward direction is:



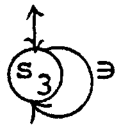
To trace the value of s_1, the modified algorithm proceeds as follows: (for the sake of convenience, we write x(q) for $\bar{x}_{s_1}(q)$)

(1) Initialize $x(s_1) = e$, $x(s_{3,1}) = \emptyset$, $x(s_3) = \emptyset$

(2) $x(s_{1,1}) = x(s_{3,1}) \mid \{e\} \bar{\circ}\, x(s_1) = \emptyset \mid e = e$.

(3) $x(s_3) = x(s_3) \mid \{\ni\} \bar{\circ}\, x(s_{3,1}) = \emptyset \mid \ni = \ni$.

(4) Since s_3 may be a self-dependent ovariable, construct the NFA for $\bar{x}_{s_3}(s_3)$:



and obtain $\bar{x}_{s_3}(s_3) = \ni^*$.
Hence $x(s_3) \geqq \bar{x}_{s_3}(s_3) \,||\, x(s_3) = \ni^*\ni$.

(5) $x(s_{3,1}) = x(s_{3,1}) \mid \{e\} \bar{\circ}\, x(s_3) = e + \ni^* \ni.$

(6) $x(s_3) = x(s_3) \mid \{\ni\} \bar{\circ}\, x(s_{3,1})$

$$= \ni^*\ni \mid \{\ni\} \bar{\circ}\, (e + \ni^*\ni)$$

$$= \ni^*\ni + \ni + (\ni\ni^* + \ni)\ \ni + \ni\ni.$$

(7) Since $\ni^*\ni = \ni^*\ni + \ni + (\ni\ni^* + \ni)\ \ni + \ni\ni$, the procedure converges.

One difficulty of this approach is obvious from the above example. The algorithm is liable to generate long and unintelligible regular expressions. Since deciding the equality of two regular expressions is an exponentially difficult problem, this approach might not be feasiable in implementation terms. (The next step to be taken in this direction is of course to look for canonical forms for regular expressions that are suitable for our purposes.)

Note in passing that the crthis, crmemb, crpart functions can be computed directly from the NFA's constructed from the flow graphs. E.g. $q \leftarrow o$ (resp. $q \ni \leftarrow o$, ...) if there is an edge from q to o with label e (resp. $\ni$, ...). So we have yet another algorithm for solving value-flow problems. In fact, this is preferable to the regular expression approach if there is a substantial number of self-dependent ovariables in SETL programs.