SETL Newsletter # 138

On Jay Earley's 'Method of Iterator Inversion'

J.T.Schwartz
and R. Paige

1. Introduction to Earley's Method.

In a recent Berkeley Technical Report (*High Level Iterators and a Method for Automatically Designing Data Structure Representation*, Memo ERL-M425, Feb. 1974) J. Earley has proposed an interesting optimization technique, which he calls 'iterator inversion'. This technique is applicable to languages of high level, and particularly to languages allowing explicit set-theoretical constructions. Iterator inversion is a method intermediate between the gathering of straightforward optimizations which can be applied routinely and an attempt to find recondite mathematical transformations of rare applicability. It is a general method, and suggests still further generalizations. In the present newsletter, we will describe Earley's method, discuss some of the problems bound to arise in its application, and explore a few of the generalizations which it suggests.

Earley's technique may be regarded as an extension of John Cocke's method of 'reduction in strength' to a set theoretic setting; for that reason we shall prefer to call it 'iterator reduction'. Suppose that a program P to be optimized contains an expression $E = E(a_1,\dots,a_n)$ involving n free parameters, and suppose that E represents an expensive enough calculation for it to be worth avoiding repeated evaluation of E. Suppose that E has the following property: if some of parameters of E, say for example $a_1,\dots,a_j$, are changed 'slightly' or 'differentially', then E changes slightly, in the sense that the new value of E can be obtained from its old value by some 'easy' calculation $E_{new} = f(E_{old})$.

In this case, we shall say (suggestively though only heuristically and of course not in the standard technical sense) that E is *continuous* in the free parameters $a_1,\ldots,a_j$. Suppose finally that, in some loop or more general program region R of interest, all the operations which change $a_1\ldots a_j$ change them only slightly. Then we can:

i. Keep E's current value $C(a_{j+1},\ldots,a_n) = E(a_1,\ldots,a_{j+1},\ldots a_n)$ available for appropriately chosen values of the parameters $a_{j+1},\ldots,a_n$. (This means that some of the values $C(a_{j+1},\ldots,a_n)$ must be stored; suppose for the moment that we store all previously calculated values $C(a_{j+1},\ldots,a_n)$, collecting these values together as a mapping C.)

ii. Use $C(a_{j+1},\ldots,a_n)$ rather than $E(a_1,\ldots,a_n)$ wherever the value of $E(a_1,\ldots,a_n)$ is required.

iii. For all values $a_{j+1},\ldots,a_n$ in the domain of the mapping C update $C(a_{j+1},\ldots,a_n)$ using an appropriate rule $C_{new} = f(C_{old})$ whenever one of the parameters $a_1,\ldots a_j$ is changed (these parameters will only change differentially).

iv. Update or calculate $C(a_{j+1},\ldots,a_n)$ using the rule

```
(1)      $C(a_{j+1},\ldots, a_{k-1}, b, a_{k+1},\ldots,a_n) =$
                 if $C(a_{j+1},\ldots,a_{k-1}, b, a_{k+1},\ldots,a_n)$ is val ne $\Omega$
                                                     then val
                      else $E(a_1,\ldots,a_{k-1}, b, a_{k+1},\ldots,a_n)$
                   /* obtained by evaluation of E */
```

whenever $k > j$ and a_k is changed by an assignment $a_k = b$.

Note that considerable memory may be needed to store values of C; thus in some cases we may prefer to keep only a small section $C'(a_j,\ldots,a_k) = C(a_j,\ldots,a_k\ldots,a_n)$, $k < n$, of $C(a_j,\ldots,a_k)$ available and to perform a full recalculation using the expression E each time one of the parameters $a_{k+1},\ldots,a_n$ is changed. In an extreme case, we may only keep one single value $C' = C(a_j,\ldots,a_n)$ available and will then recalculate C' whenever any of $a_{j+1},\ldots,a_n$ is changed.

Let us consider a few examples. The assignments

(2) $s = s + \{x\}$ and $s = s - \{x\}$

represent slight changes to the set s, as do

(2') $s = s + t$ and $s = s - t$

when the set t has only a few elements. If f is a set of pairs used as a mapping, then

(3) $f(x) = a$

causes f to change slightly. If f is a set of (n+1) - tuples used as a multi-parameter mapping, then

(3') $f(x_1,\ldots,x_n) = a$

is a slight change to f. The set theoretical operations

(4) $s + s_1$, $s - s_1$, $s * s_1$, and $f[s]$

are continuous in s, s_1, and f, as is shown by noting, for example, that before executing s = s - {x}, the expressions $C_1 = s + s_1$ and $C_2 = f[s]$ can be updated by executing

(5) $C_1 = C_1 -$ if $x \in s$ <u>and</u> x <u>n</u>$\in s_1$ then {x} else <u>nℓ</u>;

and

(6) $C_2 = C_2$ - if $x \in s$ then f{x} else <u>nℓ</u>;

respectively.

The operation $C_4 = f^{-1} [s]$, which can be written as

(7) $\{x \in$ <u>hd</u> $[f] | f(x) \in s\}$

is continuous in f, and before encountering f(x) = y is updated by

(8) $C_4 = C_4$ - if $f(x) \in s$ then {x} else <u>nℓ</u>
+ if $y \in s$ then {x} else <u>nℓ</u>.

However C_4 is discontinuous in s.
The operation f[s] is continuous in s even if f is a programmed function. The conditional expression 'if a then s_1 else s_2' is continuous in s_1 and s_2, but is, of course, discontinuous in its boolean parameter a.

The evaluation/retrieval operation f(x) is not continuous, since execution of $f(x_o) = y$ can give f(x) a completely new value. (On the other hand, if f is a programmed function and evaluation of $C_3 = f(x)$ is expensive it may be appropriate to avoid re-evaluation where possible, and to update C_3 using the statement

(9) C_3 = if x_o <u>ne</u> x then C_3 else f(x).)

Note that the ordinary rule 'continuous functions of continuous functions are continuous' can be applied to the set theoretical operations we have been considering.

As Earley has emphasized expressions involving set-formers provide more interesting examples of the phenomenon of 'continuity' which we have been discussing. The expression

(10) $C' = \{x \in s \mid f(x)\ \underline{eq}\ a\}$

is a prototypical example. This expression is continuous in s and f, but discontinuous in a. When s is changed by addition of an element y_o, then C' can be updated by executing

(11) $C' = C' + \text{if } f(y_o)\ \underline{eq}\ a \text{ then } \{y_o\} \text{ else } \underline{n\ell};$

and a similar rule applies for deletions. When f is changed by executing $f(y_o) = z$, then C' can be updated by executing

(12) $C' = \text{if } y_o \in s \text{ then } C' - (\text{if } f(y_o)\ \underline{eq}\ a \text{ then } \{y_o\} \text{ else } \underline{n\ell}) + \text{if } z\ \underline{eq}\ a \text{ then } \{y_o\} \text{ else } \underline{n\ell};$

(the update operation should be inserted just before the assignment $f(y_o) = z$). More insight is gained by writing (12) as

(13) $C' = C' - \{x \in \{u \in s \mid u\ \underline{eq}\ y_o\} \mid f(x)\ \underline{eq}\ a\} + \{x \in \{u \in s \mid u\ \underline{eq}\ y_o\} \mid z\ \underline{eq}\ a\};$

since (13) begins to suggest a rule for updating more general set-theoretical expressions than (10). For example, before changing f by $f(y_o) = z$ the set

(14) $C_1 = \{x \in s \mid g(f(x))\ \underline{eq}\ a\}$

can be updated by executing

(15) $$c_1 = c_1 - \{x\in\{u\in s \mid u \ \underline{eq}\ y_o\} \mid g(f(x)\ \underline{eq}\ a\} + \{x\in\{u\in s \mid u\ \underline{eq}\ y_o\} \mid g(z)\ \underline{eq}\ a\};$$

and the set

(16) $$c_2 = \{x\in s \mid f(g(x))\ \underline{eq}\ a\}$$

can be updated by executing

(17) $$c_2 = c_2 - \{x\in\{u\in s \mid g(u)\ \underline{eq}\ y_o\} \mid f(g(x))\ \underline{eq}\ a\} + \{x\in\{u\in s \mid g(u)\ \underline{eq}\ y_o\} \mid z\ \underline{eq}\ a\};$$

All the updating operations (13), (15), (17) are to be performed just prior to the change $f(y_o) = z$ for which they compensate. Next consider the case of a set-theoretic expression in whose defining condition f appears twice with different arguments, as for example

(18) $$c_3 = \{x\in s \mid f(g(x))\ \underline{eq}\ f(h(x))\}$$

or

(19) $$c_4 = \{x\in s \mid f(f(x))\ \underline{eq}\ a\}.$$

Before changing f by $f(y_o) = z$, we can update such sets using the following formal device: treat all the separate occurrences of f as if they were separate functions $f_1, f_2, \ldots$; and regard the change $f(y_o) = z$ as a series of modifications affecting each of these seperate functions, i.e., treat it as if it were $f_1(y_o) = z$, $f_2(y_o) = z$, etc.

In the case of C_3 and C_4, this leads us to the following updating operations:

$$
\begin{aligned}
(20)\qquad C_3 = C_3 &- \{x\in\{u\in s \mid g(u)\ \underline{eq}\ y_o\} \mid f(g(x))\ \underline{eq}\ f(h(x))\} \\
&+ \{x\in\{u\in s \mid g(u)\ \underline{eq}\ y_o\} \mid z\ \underline{eq}\ f(h(x))\} \\
&- \{x\in\{u\in s \mid h(u)\ \underline{eq}\ y_o\} \mid (\text{if } g(x)\ \underline{eq}\ y_o \text{ then } z \text{ else } f(g(x)))\ \underline{eq}\ f(h(x))\} \\
&+ \{x\in\{u\in s \mid h(u)\ \underline{eq}\ y_o\} \mid (\text{if } g(x)\ \underline{eq}\ y_o \text{ then } z \text{ else } f(g(x)))\ \underline{eq}\ z\};
\end{aligned}
$$

$$
\begin{aligned}
(21)\qquad C_4 = C_4 &- \{x\in\{u\in s \mid f(u)\ \underline{eq}\ y_o\} \mid f(f(x))\ \underline{eq}\ a\} \\
&+ \{x\in\{u\in s \mid f(u)\ \underline{eq}\ y_o\} \mid z\ \underline{eq}\ a\} \\
&- \{x\in\{u\in s \mid u\ \underline{eq}\ y_o\} \mid (\text{if } f(x)\ \underline{eq}\ y_o \text{ then } z \text{ else } f(f(x)))\ \underline{eq}\ a\} \\
&+ \{x\in\{u\in s \mid u\ \underline{eq}\ y_o\} \mid (\text{if } z\ \underline{eq}\ y_o \text{ then } z \text{ else } f(z))\ \underline{eq}\ a\}
\end{aligned}
$$

Note that many of the updating operations shown above can be written in forms which make more explicit the fact that they can be performed efficiently. For example, (21) can be written as

$$
\begin{aligned}
(21')\qquad C_4 = C_4 &- \text{if } f(y_o)\ \underline{eq}\ a \text{ then } \{x\in s \mid f(x)\ \underline{eq}\ y_o\} \text{ else } \underline{n\ell} \\
&+ \text{if } z\ \underline{eq}\ a \text{ then } \{x\in s \mid f(x)\ \underline{eq}\ y_o\} \text{ else } \underline{n\ell} \\
&- \text{if } y_o \in s\ \underline{\text{and}}\ (\text{if } f(y_o)\ \underline{eq}\ y_o \text{ then } z \text{ else } f(f(y_o)))\ \underline{eq}\ a \text{ then } \{y_o\} \text{ else } \underline{n\ell} \\
&+ \text{if } y_o \in s\ \underline{\text{and}}\ (\text{if } z\ \underline{eq}\ y_o \text{ then } z \text{ else } f(z))\ \underline{eq}\ a \text{ then } \{y_o\} \text{ else } \underline{n\ell}\ .
\end{aligned}
$$

Next consider a set formed using a double iterator, as for example

$$(22)\qquad C_5 = \{e(x,y),\ x\in s,\ y\in t(x) \mid f(x)\ \underline{eq}\ g(y)\}.$$

If f is modified by $f(x_o) = z$, then C_5 can be updated by executing

(23) $$C_5 = C_5 - \{e(x,y),\ x\in\{u\in s \mid u\ \underline{eq}\ x_o\}, y\in t(x) \mid f(x)\ \underline{eq}\ g(y)\} + \{e(x,y),\ x\in\{u\in s \mid u\ \underline{eq}\ x_o\}, y\in t(x) \mid z\ \underline{eq}\ g(y)\};$$

which is of course the same as

(23') $$C_5 = \text{if } x_o \in s \text{ then } C_5 - \{e(x_o,y),\ y\in t(x_o) \mid f(x_o)\underline{eq}\ g(y)\} + \{e(x_o,y),\ y\in t(x_o) \mid z\ \underline{eq}\ g(y)\} \text{else } C_5$$

Similarly, if g is modified by $g(y_o) = z$, then C_5 can be updated by executing

(24) $$C_6 = C_6 - \{e(x,y),\ x\in s,\ y\in\{u\in t(x) \mid u\ \underline{eq}\ y_o\} \mid f(x)\ \underline{eq}\ g(y)\} + \{e(x,y),\ x\in s,\ y\in\{u\in t(x) \mid u\ \underline{eq}\ y_o\} \mid f(x)\ \underline{eq}\ z\ \};$$

which can also be written as

(25) $$C_6 = C_6 - \{e(x,y_o),\ x\in\{u\in s \mid y_o \in t(x)\} \mid f(x)\ \underline{eq}\ g(y_o)\} + \{e(x,y_o), x\in\{u\in s \mid y_o \in t(x)\} \mid f(x)\ \underline{eq}\ (z\ )\}.$$

Note that a first application of iterator reduction often paves the way for and suggests other iterator reductions. For example, after reducing (23) to (25), we may try to keep the current value of the set $\{u\in s \mid y_o\in t(x)\}$ available; and this may involve the insertion of additional updating operations.

2. General rules for the continuity of set-theoretical expressions.

We now formulate a few general rules concerning 'continuity' of set-theoretical expressions. We shall express these rules not in terms of the simple notion of continuity introduced in section 1, but in terms of the more detailed notion 'continuity relative to a particular, slight, modification of a given free parameter'. Note once more that for sets s we consider insertion, deletion, and addition or subtraction of a small set s_1 to be slight modifications; and for mappings f consider indexed assignments $f(x) = y$ or $f(x_1,\ldots,x_n) = y$ to be slight modifications. If in a loop all changes to variables which are sets or mappings are slight modifications of the kind just described, then these variables will be called induction variables of the loop. It must also be observed that none of the transformations which we are studying can safely be applied to expressions containing operations which cause side effects that are used; for which reason we shall always assume such operations to be absent in the expressions we treat. We also assume that typefinding is applied prior to any attempt to optimize by iterator reduction; so that object types are known during the analysis of a program for reduction. Consider the set-theoretic expression

(1) $$C = \{x \in s \mid K(x)\}$$

in which K(x) is any boolean-valued subexpression containing only free occurrences of the bound variable x, and containing no free instance of the set, s. Suppose that the expression (1) is used in a strongly connected region and that the following conditions hold:

i) The boolean valued subexpression K(x) contains m free occurrence of an n-ary mapping f (in which each such occurrence has at least 1 parameter expression involving x, the bound variable of the set former); all other free variables occurring in k are loop invariant.

ii) f and s are induction variables of the loop; i.e., inside the loop s is only changed by slight modifications of the form $s = s + s_1$, where s_1 is a small set in comparison with s, and f is only changed by indexed assignments of the form

$$f(y_1, y_2,\ldots,y_n) = z.$$

Then we can reduce the expression (1) inside this loop. Let C be a compiler-generated variable, to be associated with the value of the set former expression (1). We will say that C is *available on exit from* a program point p if C is equal to the value which the expression (1) would have if evaluated immediately after the statement at p is executed; C is *available on entrance* to p if C is available on exit from all predecessor points of p. If C is available on entrance to p, and if C is not available on exit from p (which will happen when execution of the statement at p changes the value of a parameter upon which the value of expression (1) depends), then we say that C is spoiled at P

To reduce the expression (1) within a loop L, we begin by making it available on entrance to L. This is done by inserting the assignment $C = \{x \in s | K(x)\}$ into the loop's initialization block. Then, at each point p inside L where the value of the induction variables s or f can change, the value of C (which could be spoiled at p) will be updated by inserting appropriate slight modifications $C = C \pm C_1$, where C_1 is a small set relative to C; this keeps C available after exit from p.

We shall now proceed systematically to discuss continuity properties of the expression (1) and associated update rules for C for two cases (illustrated by fragmentary examples in section i above) small changes in the set s, and changes to f which result from an indexed assignment.

Rule 1 (small changes in s). At each point p in L at which the set s is changed slightly (by addition or deletion of a set s_1 which is small in comparison with s) make the following program transformation:

Location	Original Code
P	$s = s + s_1$
	After Reduction
P	$s = s \pm s_1$
P + 1	$C = C \pm \{x \in s_1 \mid K(x)\}$

Then, if C is available on entrance to p and if the instruction at p kills C, the instruction inserted at P + 1 updates C and makes C available on exit from point P + 1. Moreover, since s_1 is small in comparison with s it will gererally be true that the modification to C is also small.

Rule 2. Suppose that the boolean subexpression K of (1) contains m free occurrences of the n-ary mapping symbol f. Suppose also that these m occurrences of f appear in r different terms,

$$f(p_{11}(x),\ldots,p_{1n}(x)),\ f(p_{21}(x),\ldots,p_{2n}(x)),\ldots,f(p_{r1}(x),\ldots,p_{rn}(x)),$$

where $p_{ij}(x)$ represents the j-th parameter expression (involving x which is the bound variable of the set former) of the i-th term. Then at each point p in the loop in which the n-ary mapping f is changed slightly by an indexed assignment, make the following program transformation:

Relative Position	Original Code
P	$f(y_1,\ldots,y_n) = z$
	After Reduction
P-2	$s_2 = \{x \in s \mid p_{11}(x)\ \underline{eq}\ y_1\ \&\ldots\&p_{1n}(x)\ \underline{\epsilon}\ y_n$ $\underline{or}\ \ldots\ \underline{or}$ $p_{r1}(x)\ \underline{eq}\ y_1\&\ldots\&p_{rn}(x)\ \underline{eq}\ y_n\}$

```
p-1          $C = C - \{x \in s_2 | K(x)\}$
p            $f(y_1,\ldots,y_n) = z$
p+1          $C = C + \{x \in s_2 | K(x)\}$
```

It can be shown that rule 2 is a corollary of rule 1. To see this, consider the set $D_{f_i} = \{<p_{i1}(x),\ldots,p_{in}(x)> | x \in s\}$ Let p_i be the mapping whose domain is s and where $p_i(x) = <p_{i1}(x),\ldots,p_{in}(x)>$. Then for any n-tuple $<y_1,\ldots,v_n>$, we have

$p_i^{-1}(y_1,\ldots,y_n) = \{x \in s | p_{i1}(x) \ \underline{eq}\ y_1 \ \&,,,\&\ p_{in}(x)\ \underline{eq}\ y_n\}$.
If s changes by deletion of $p_i^{-1}(y_1,\ldots,y_n)$, then D_{f_i} changes by deletion of the n-tuple $<y_1,\ldots,y_n>$. Moreover if s is modified by deletion of $\bigcup_{i=1}^{r} p_i^{-1}(y_1,\ldots,y_n)$, then the n-tuple $<y_1,\ldots,y_n>$ is removed from the domain of all the f terms occurring in (1). Next we observe that if C is available on entrance to p (i.e., is available just prior to the modification to f by the indexed assignment $f(y_1,\ldots,y_n) = z$), and if $<y_1,\ldots,y_n> \notin \bigcup_{i=1}^{r} D_{fi}$ just before point p, then the statement $f(y_1,\ldots,y_n) = z$ does not change any of the occurrences of f in (1). Consequently, C is not spoiled by the indexed assignment, and it remains available. Suppose now that in expression (1) C is available on entrance to the program point p. Then we would proceed as follows 1) at p-3, put s_2 equal to the set $\bigcup_{i=1}^{r} p_i^{-1}(y_1,\ldots,y_n)$ 2) at p-2 delete s_2 from s; 3) at p-1 update C in accordance with rule 1, 4) at p+1 add s_2 back to s; and 5) at p+2, use rule 1 again to update C. This would give us the following code:

```
     p-3      $s_2 = \{x \in s \mid p_{11}(x)$ eq $y_1$ &...& $p_{in}(x)$ eq $y_n$ or
                             ... or
                             $p_{r1}(x)$ eq $y_1$ &...& $p_{rn}(x)$ eq $y_n\}$
(3)  p-2      $s = s - s_2$
     p-1      $C = C - \{x \in s_2 \mid K(x)\}$
     p        $f(y_1,\dots,y_n) = z$
     p+1      $s = s + s_2$
     p+2      $C = C + \{x \in s_2 \mid K(x)\}$
```

In this code C is not spoiled by the statement $f(y_1,\dots,y_n) = z$. Hence, if C is available on entrance to p-3, then by Rule 1 we know that C remains available on exit from p+2. And now finally, since in (3) the value of the set s is the same before p-2 as after p+1, and because s is not used between p-2 and p+1, the code (3) is equivalent to that shown in Rule 2.
The assumption that at least one of the parameters in each f term in K involves x (the bound variable of the set former) will usually cause the set s_2 to be small in comparison with s. Then,by the continuity properties stated in Rule 1, we can conclude that the modification to C appearing in Rule 2 is small compared with the set C itself.

The code generated by Rule 2 can be improved by eliminating redundancies in the expression $\{x \in s_2 \mid K(x)\}$ which appears at locations p-1 and p+1. Suppose we know that $s_2 = \bigcup_{i=1}^{r} R_i$, where $R_1,\dots,R_r$ are disjoint sets. Then $\{x \varepsilon\ s_2 \mid K(x)\}$ can be rewritten as $\bigcup_{i=1}^{r} \{x \varepsilon\ P_i | K(x)\}$. Suppose also that in each set $\{x \varepsilon\ R_i | K(x)\}$ $K(x)$ can be transformed (by elimination of redundant operations) into an equivalent but easier-to-evaluate expression $K_i(x)$.

Then it may be worth while to work with the partition $\{R_i\}$ of s_2 instead of s_2 and to rewrite

$$\{x \in s_2 \mid K(x)\} \text{ as } \bigcup_{i=1}^{r} \{x \in R_i | K_i(x)\}.$$

As an example of this, observe that if we let $R_i = \{x \in (s - \bigcup_{k=o}^{i-1} R_k) | P_{i1}(x)$ eq y_1 &...& $P_{in}(x)$ eq $y_n\}$, where $R_o = \emptyset$, then $R_1,\ldots,R_r$ form a partition of s_2. Moreover, on the set R_i we can replace the term $f(p_{i1}(x),\ldots,p_{in}(x))$ which appears in the expression K at location p-1 of the code generated by Rule 2 by $f(y_1,\ldots,y_n)$ (cf.(3) above) . This can lead to a version of line p-1 of (3) which is relatively easy to evaluate, and it is therefore tempting to apply the same transformation to line p+2 of (3). However, at location p+2 we cannot, even after breaking up $\{x \in s_2|K(x)\}$ into $\bigcup_{i=1}^{r} \{x \in R_i|K(x)\}$, simply replace each term $f(p_{i1}(x),\ldots,p_{in}(x))$ in K by z. This is because the indexed assignment appearing in line p of (3) changes f and may therefore cause some parameter $p_{ij}(x)$ appearing in $\{x \in R_i|K(x)\}$ within (3) and containing an occurrence of f to have a value different from y_j. When dealing with cases complicated enough for this problem to arise, we can make use of a second, finer, partition $R_j',\ldots,R_r'$ of s_2 defined as follows: First set $R_o' = \emptyset$ as before. Next find all f terms $f_{i_1},\ldots,f_{i_{r_o}}$ whose parameter expressions involve no f term, and put

$$R'_1 = \{x \in s \mid p_{i_1 1}(x) \ \underline{eq}\ y_1 \ \& \ldots \& \ p_{i_1 n}(x) \ \underline{eq}\ y_n\},$$
$$R'_2 = \{x \in (s - R'_1) \mid p_{i_2 1}(x) \ \underline{eq}\ y_1 \ \& \ldots \& \ p_{i_2 n}(x) \ \underline{eq}\ y_n\}, \ldots,$$
$$R'_{r_o} = \{x \in (s - \bigcup_{k=0}^{r_o - 1} R'_k) \mid p_{i_{r_o} 1}(x) \ \underline{eq}\ y_1 \ \& \ldots \& \ p_{i_{r_o} n}(x) \ \underline{eq}\ y_n\}.$$

After this, find all f terms $f_{i_{r_{o+1}}}$, $f_{i_{r_{o+2}}}, \ldots, f_{i_{r_1}}$ which do not belong to the set $F_1 = \{f_{i_1}, \ldots, f_{i_{r_o}}\}$ but whose parameter expressions only contain f terms which do belong to F_1. Define sets $R'_{r_o+1}, \ldots, R'_{r_1}$ by writing

$$R'_\ell = \{x \in (s - \bigcup_{k=o}^{\ell-1} R'_k) \mid p_{i_\ell 1}(x) \ \underline{eq}\ y_1 \ \& \ldots \& \ p_{i_\ell n}(x) \ \underline{eq}\ y_n\}.$$

Iterating this procedure sufficiently often we will obtain a partition $\{R'_1, \ldots, R'_r\}$ which can be used to eliminate redundant calculations of $f(p_{i1}(x), \ldots, p_{in}(x))$ at both p-1 and p+1. More specifically, if we let $K(x)_{s_1, \ldots, s_n}[t_1, \ldots, t_n]$ denote the result of substituting the terms $t_1, \ldots, t_n$ for the terms $s_1, \ldots, s_n$ occurring in K(x), we can replace the code occurring at location p-1 (in Rule 2) by

$$C = C - \bigcup_{i=1}^{r} \{x \in R'_i \mid K(x)_{p_{i1}(x), \ldots, p_{in}(x)} [y_1, \ldots, y_n]\}$$

and the code occurring at p+1 by

$$C = C + \bigcup_{i=1}^{r} \{x \in R'_i \mid (K(x)_{f(p_{i1}(x), \ldots, p_{in}(x))}[Z])_{p_{i1}(x), \ldots, p_{in}(x)} [y_1, \ldots, y_n]\}.$$

(Note that the immediately preceeding formula describes two successive steps of substitution.)

This general method allows the code used to reduce various set former expressions in examples (13) - (25) of section 1 above to be generated automatically.

As an example of the redundancy elimination method just outlined, consider the following example:

(4) $C = \{x \varepsilon s | f(f(f(x+1) + 1)) \ \underline{eq}\ f(f(x+1) + 1)\}$.

Suppose that the mapping f is changed slightly by an indexed assignment, $f(y_o) = Z$ which occurs at a program point p. Then to update the value of (4) we proceed as follows. First a partition R_1, R_2, R_3 is computed. Observe that this partition contains three sets because only three different f terms occur in the boolean subexpression in (4): these are $f(x + 1)$, $f(f(x+1)+1)$, and $f(f(f(x+1) + 1))$. Since $f(x+1)$ is the only f term of (4) whose parameter expression involves no f term, we put $R_1 = \{x \varepsilon s | (x+1)\ \underline{eq}\ y_o\}$. Since the parameter part of $f(f(x+1) + 1)$ involves $f(x+1)$, we set

$$R_2 = \{x \varepsilon (s-R_1) \ | \ f(x+1) + 1 \ \underline{eq}\ y_o\} \text{ and}$$

$$R_3 = \{x \varepsilon (s-R_1 \cup R_2) \ | \ f(f(x+1) + 1)\ \underline{eq}\ y_o\}.$$

The code generated to update (4) is then as follows:

```
$R_1 = \{x \varepsilon s \ | \ x + 1 \ \underline{eq}\ y_o\}$;
$R_2 = \{x \varepsilon (s-R_1) \ | \ f(x+1) + 1 \ \underline{eq}\ y_o\}$;
$R_3 = \{x \varepsilon (s-R_1 U R_2) \ | \ f(f(x+1) + 1)\ \underline{eq}\ y_o\}$;
$C = C - \{x \varepsilon R_1 \ | \ f(f(f(y_o) + 1)) \ \underline{eq}\ f(f(y_o) + 1)\}$
        $- \{x \varepsilon R_2 \ | \ f(f(y_o)) \ \underline{eq}\ f(y_o)\}$
        $- \{x \varepsilon R_3 \ | \ f(y_o) \ \underline{eq}\ y_o\}$;
$f(y_o) = Z$;
$C = C + \{x \varepsilon R_1 \ | \ f(f(z+1)) \ \underline{eq}\ f\ (z+1)\}$
        $+ \{x \varepsilon R_2 \ | \ f(Z) \ \underline{eq}\ Z\} + \{x \varepsilon R_3 | Z \ \underline{eq}\ y_o\}$;
```

Note that computing the partition, R_1, R_2, R_3 is about as expensive as computing the set

$$s_2 = \{x \in s | x+1 \ \underline{eq}\ y_o \ \underline{or}\ f(x+1) + 1 \ \underline{eq}\ y_o \ \underline{or}\ f(f(x+1) + 1) \ \underline{eq}\ y_o\}$$

of Rule 2. Thus, the use of partitions in the manner just described can be expected to improve the code produced by direct application of Rule 2.

The method of reduction which has been described can be extended in a useful way to apply to various SETL expressions that implicitly contain set formers. Amoung these are the forall iterator (i.e., $(\forall\ x \in s|K(x))$, *block*), the existential and universal quantifers (i.e., $\exists x \in s\ |K(x)$ and $\forall x \in s\ |K(x)$), and the compund operator (i.e.

$$[\text{<binop>}: x \in s \ | \ K(x)] \ e\ (x) \ ).$$

To apply the method of reduction to these expressions, we rewrite them by replacing the implicit set former subpart, $x \in s|K(x)$, which they contain with $x \in \{ u \in s \ | \ K(u)\}$. The set former subexpression thus exposed can then be reduced using Rules 1 and 2.

We will now discuss the compound operator

(5) $\qquad C = [\text{<binop>}: x \in s \ | \ K(x)] \ e\ (x)$

which, like the set former, has relevance to vari-.s other SETL operators.

Here we require that the binary operator, 'binop', have an appropriate 'inverse' (which we will call 'inverse binop'). The arithmetic binary operator + (with - as its inverse) is an example. (5) is then continuous in s and f, and the following reduction rules which parallel rules 1 and 2 apply: Rule 3. at each point p in the loop at which the set s is changed slightly (by addition or deletion of a set s_1 that is small in comparison with s) make the following program transformations:

Relative Location	Original Code
p	$s = s - s_1$
	After Reduction
p-1	$s_3 = s_1 \cap s$
p	$s = s - s_1$
p+1	C = C <inverse binop> [<binop>: $x \varepsilon s_3 \mid K(x)$] e (x)

where <inverse binop> is the appropriate inverse binary operator of <binop>. If C is available on entrance to p in the original code, then after reduction takes place C is restored and is available on exit from p+1. A similar rule applies for updates after $s=s+s_1$.

Rule 4. (Analogous to Rule 2). Suppose that the Boolean expression K(x) of (5) contains m free occurrences of the n-ary mapping f. Let us use the same notation as in Rule 2. Then at each point p in the loop at which f is changed by an indexed assignment, the following code transformation should be made:

Relative Position	Original Code
p	$f(y_1,\ldots,y_n) = z$
	After Reduction
p-2	$s_2 = \{x \ \varepsilon\ s \mid p_{11}(x)$ <u>eq</u> y_1 &...& $p_{1n}(x)$ <u>eq</u> y_n <u>or</u> ... <u>or</u> $p_{r1}(x)$ <u>eq</u> y_1 &...& $p_{rn}(x)$ <u>eq</u> $y_n\}$
p-1	C = C <inverse binop> [<binop>: $x \varepsilon s_2 \mid K(x)$] e (x)
p	$f(y_1,\ldots,y_n) = z$

p+1 $\quad C = C$ <binop> [<binop>: $x\varepsilon s_2 | K(x)$] $e(x)$

It is easily seen that Rule 4 follows from Rule 3 in much the same way that Rule 2 follows from Rule 1.

We can extend the applicability of reduction rules 3 and 4 by treating the counting operation applied to a set former, # $\{x\varepsilon s | K(x)\}$, as [+: $x\varepsilon s | K(x)$] 1. Code transformation can be especially profitable in this case since two operations, set former and counting, can be eliminated. We can treat the existential and universal quantifiers for the case when 'locating' or other side effects of these quantifiers can be ignored similarly. (This observation, like much of the other content of this newsletter, is due to Earley _op cit_.) Specifically, an existential expression $\exists x\varepsilon s | K(x)$ can be rewritten as #$\{x\varepsilon s | K(x)\}$ _ne_ 0, and then the size operation #$\{x\varepsilon s | K(x)\}$ can be handled in the manner described just above. A universal expression $\forall x\varepsilon s | K(x)$ can be transformed first into the equivalent _not_ $\exists x\varepsilon s$ | _not_ $K(x)$ which can then be treated as an existential, specifically as _not_ #$\{x \varepsilon s$ | _not_ $K(x)\}$ _ne_ 0 which can at once be reduced to

([+: $x\varepsilon s$ | _not_ $K(x)$] 1) _eq_ 0.

3. _Set-formers containing parameters on which they depend discontinuously._

We continue to consider set-formers

(1) $$C = \{x \in s \mid K(x)\}$$

involving one bound variable x and a number of free variables. Suppose that the expression (1) appears within a loop L in which one free variable q upon which (1) depends discontinuously is changed.

Then to avoid re-evaluating (1) within L, we can keep the value $C = C(q)$ of (1) available for every value that the free variable q can assume inside L (we will refer to this set as D_q). Although such an approach is often feasible, it can easily be made infeasible by one of three factors

(a) Storage of all the sets C(q) can require too much space.

(b) Updating all the sets C(q) whenever a parameter on which C depends continuously is modified may waste more time than is saved by avoiding the calculation of (1).

(c) Storage of the set D_q may require too much space.

As an example of objection (a), consider

(2) $$C(q) = \{x \in s \mid f(x,q) \ \underline{gt}\ q\}.$$

For a 'randomly' chosen f some large percentage of all the $x \in s$ will belong to the set (2) for many q. Hence, the sets C(q) will be large for many q. These sets will often overlap, and storing them will undoubtedly require much more space than is required by s. On the other hand, when the sets C(q) are disjoint, as in

(3) $$C(q) = \{x \in s \mid f(x) \ \underline{eq}\ q\}$$

it is possible that no extra space will be required for storing C(q). This will be true when explicit representation of s can be suppressed, and when s can be represented by a partition that includes the collection of sets C(q).

Objection (a) to storage of the values C(q) becomes weaker under the following 3 conditions:

1) When the amount of overlapping of the C(q) is small,
2) When the number of stored sets C(q) is small,
3) When the size of each stored set is small.

Note that when C is discontinuous in more than one free variable and a map $C(q_1,\ldots,q_t)$ is maintained then the number of sets maintained (in addition to s) is $\prod_{i=1}^{t} \# D_{q_i}$.

Thus, if there is any overlapping at all among the stored sets it can easily be quite costly to reduce (1) if there are several parameters upon which (1) depends discontinuously. One way of diminishing the number of sets $C(q_1,\ldots,q_t)$ that must be stored in such cases is to group the parameters $q_1,\ldots,q_t$ into subexpressions. For example, to deal with an expression

(3) $C(q_1,q_2) = \{x \in s \mid f(x)\ \underline{eq}\ (q_1 + q_2)\}$

we can make use of the substitution $b = q_1 + q_2$ and store only

(3') $C(b) = \{x \in s \mid f(x)\ \underline{eq}\ b\}$

where $D_b = \{q_1 + q_2,\ q_1 \in D_{q_1},\ q_2 \in D_{q_2}\}$. Since the inequality

$\# D_b \leq (\# D_{q_1}) \times (\# D_{q_2})$ always holds, the number of sets

C(b) needed will never be greater than the number of sets $C(q_1,q_2)$.

Let us now go on to consider objection (b) and certain methods for ameliorating it. Suppose we want to reduce the set former (1) and suppose that the value of (1) is kept in the map $C(q_1,\ldots,q_t)$, where $q_1,\ldots,q_t$ are parameters upon which (1) depends discontinuously. Then whenever a variable upon which (1) depends continuously undergoes a small change in L, it may be necessary to update all the values $C(q_1,\ldots,q_t)$ for all necessary values of $q_1 \in D_{q_1},\ldots,q_t \in D_{q_t}$, by making appropriate small changes. The code generated by application of Rule 1 of the preceeding section to (1) could for example be the following:

(4) /* after the operation */ $s = s \pm s_1$;

$$(\forall q_i \in D_{q_1}, \ldots, q_t \in D_{q_t})\ C(q_1,\ldots,q_t) = C(q_1,\ldots,q_t) \pm \{x \in s_1 | K(x)\};;$$

and the code generated by the corresponding Rule 2 could be as follows:

(4') $$(\forall q_1 \in D_{q_1}, \ldots, q_t \in D_{q_t})\ s_2(q_1,\ldots,q_t) =$$

$$\{x \in s\ | p_{11}(x,q_1,\ldots,q_t)\ \underline{eq}\ y_1 \ \&\ldots\&\ p_{1n}(x,q_1,\ldots,q_t)\ \underline{eq}\ y_n$$
$$\underline{or}\ldots\underline{or}\ p_{r1}(x,q_1,\ldots,q_t)\ \underline{eq}\ y_1 \ \&\ldots\&\ p_{rn}(x, q_1,\ldots,q_t)\ \underline{eq}\ y_n\};$$
$$C(q_1,\ldots,q_t) = C(q_1,\ldots,q_t) - \{x \in s_2(q_1,\ldots,q_t)|K(x)\};;$$
$$f(y_1,\ldots,y_n) = z;$$
$$(\forall q_1 \in D_{q_1}, \ldots, q_t \in D_{q_t})\ C(q_1,\ldots,q_t) = C(q_1,\ldots,q_t) + \{x \in s_2\ (q_1,\ldots,q_t)\ |\ K(x)\};;$$

Observe from (4) and (4') above that the cost of updating the sets $C(q_1,\ldots,q_t)$ is directly proportional to the number of these sets stored. Hence, the technique of reducing the number of stored sets by regrouping (discussed previously as a way of avoiding objection (a)) will also be useful in ameliorating objection (b).

There is another technique which is worth mentioning despite the difficulties its use can involve, because it is likely to accomplish the same aim more effectively than the method just described. Instead of iterating over the cross product of all the sets D_{q_i} as in (4) and (4'), we can consider the set D_q of all t-tuples $q = \langle q_1,\ldots,q_t\rangle$ of values that the parameters $q_1,\ldots,q_t$ can collectively have, and can then iterate over this set. However, predetermination of the set D_q would in most cases not be easy.

Execution of the general update code (4) and (4') can be quite costly, and the methods presented so far for avoiding objection (b) are not terribly powerful. For general expressions (1) that depend discontinuously on several parameters we would expect objection (b) to make reduction infeasible. Nevertheless, a few important special cases of common occurrence in SETL programs can be reduced profitably by supplementing the methods just described with a few additional techniques.

To discuss one such special case we consider example (3) once more. Applying Rule 1 to (3), we see that the computation required to update $\{x \in s \mid f(x)\ \underline{eq}\ q\}$ after $s = s \pm s_1$ is

(5) $\qquad (\forall q \in D_q)\ C(q) = C(q) \pm \{x \in s_1 | f(x)\ \underline{eq}\ q\}.$

As noted in the preceeding discussion of objection (b), we know that if $\#\ D_q$ is too large the computation (5) will be costlier than a full recalculation of (3). However, it is only necessary to apply Rule 1 to those sets C(q) that actually change. But C(q) will not change if $\{x \in s_1 |\ f(x)\ \underline{eq}\ q\}$ is empty. Thus the set D_q appearing in (5) can be replaced by

(6) $\qquad C' = \{q \in D_q \mid (\exists\ x \in s_1 \mid f(x)\ \underline{eq}\ q)\},$

which is usually smaller. The set C' can be rewritten as

(6') $\qquad C' = \{f(x), x \in s_1 \mid$ [illegible]$x) \in D_q\}.$

Note that (6') is continuous in both the parameters s_1 and f (we assume that D_q is a region constant), and that when it is profitable to reduce (6') or when D_q is large in comparison with s_1 it becomes profitable to replace (5) by

(5') $\quad (\forall\ b \in \{f(x), x \in s_1 | f(x) \in D_q\})\ C(b) = C(b) \pm \{y \in s_1 | f(y)\ \underline{eq}\ b\};;$

If the map f of (3) is changed within L by an indexed assignment

(7) $$f(x_o) = z$$

then Rule 2 of the previous section applies to the sets C(q) in the following way:

(8)
$(\forall q \in D_q)\ s_2(q) = \{x \in s \mid x\ \underline{eq}\ x_o\};$
$C(q) = C(q) - \{x \in s_2(q) \mid f(x)\ \underline{eq}\ q\};;$
$f(x_o) = z;$
$(\forall q \in D_q)\ C(q) = C(q) + \{x \in s_2 \mid f(x)\ \underline{eq}\ q\};;$

Since $s_2(q)$ does not change within the iterations (8), it is better to rewrite this as

(8')
$s_2 = \{x \in s \mid x\ \underline{eq}\ x_o\};$
$(\forall q \in D_q)\ C(q) = C(q) - \{x \in s_2 \mid f(x)\ \underline{eq}\ q\};$
$f(x_o) = z;$
$(\forall q \in D_q)\ C(q) = C(q) + \{x \in s_2 \mid f(x)\ \underline{eq}\ q\};$

Applying the same transformation that was used to derive (5') from (5), we can transform (8') still further into

(8")
$s_2 = \{x \in s \mid x\ \underline{eq}\ x_o\};$
$(\forall\ b \in \{f(x),\ x \in s_2 \mid f(x) \in D_q\})\, C(b) =$
$C(b) - \{y \in s_2 \mid f(y)\ \underline{eq}\ b\};;$
$f(x_o) = z;$
$(\forall\ b \in \{f(x),\ x \in s_2 \mid f(x) \in D_q\})\ C(b) =$
$C(b) + \{y \in s_2 \mid f(y)\ \underline{eq}\ b\};;$

Finally, because s_2 contains either no or one element, we can eliminate the iterations of (8") completely, writing (8") as

```
(8"')    if $x_o \in s$ & $f(x_o) \in D_q$ then $C(f(x_o)) = C(f(x_o)) - \{x_o\}$;;
         $f(x_o) = z$;
         if $x_o \in s$ & $z \in D_q$ then $C(z) = C(z) + \{x_o\}$;;
```

which shows that objection (b) does not apply to example (3). This example typifies the treatment of a somewhat broader class of expressions in which the objection (b) can be avoided.

One such expression is

(9) $$C = \{x \in s \mid f(x)\ \underline{eq}\ g(q)\};$$

more generally, we can consider

(10) $$C = \{x \in s \mid K_1(x)\ \underline{eq}\ K_2(q_1,\ldots,q_t)\},$$

where $q_1,\ldots,q_t$ are free variables upon which C depends discontinuously. We assume that K_1 of (10) is a subexpression only involving x, parameters upon which (10) depends continuously, and maps f_i upon which C can depend discontinuously but whose occurrences in K_1 all have parameters depending on x. K_2 of (10) is assumed to be a subexpression only involving the parameters $q_1,\ldots,q_t$ on which C depends discontinuously, and also on the maps f_i.

We assume that expressions estimating $D_{q_1},\ldots,D_{q_t}$ are available at compile time. We can simplify (10) by substituting a new free variable b for $K_2(q_1,\ldots,q_t)$; then we compute D_b and keep the value $C = C(b)$ available for every value $b \in D_b$. A reduction procedure for (10) which incorporates these ideas can be described as follows:

1) On entrance to L define initial values for D_b and $C(b)$:

(11) $$D_b = \{\ K_2(q_1,\ldots,q_t),\ q_1 \in D_{q_1},\ldots,q_t \in D_{q_t}\};$$

$$(\forall\ b \in D_b)\ C(b) = \{x \in s \mid K_1(x)\ \underline{eq}\ K_2(q_1,\ldots,q_t)\};;$$

2) changes of $q_1,\ldots,q_t$ inside L do not require updates of C or D_b.
3) After a change to s(as in (4))perform the following update computation,

(12) $$(\forall\ u \in \{K_1(x),\ x \in s_1 \mid K_1(x) \in D_b\})\ C(u) = C(u) \pm \{y \in s_1 \mid K_1(y)\ \underline{eq}\ u\};;$$

4) Suppose that the subexpressions K_1 and K_2 of (10) respectively contain r_1 and r_2 free occurrences of the n-ary mapping f.

Using the same notation as was used in Rule 2 of section (ii), we denote the r_2 different occurrences of f in K_1 by

$$f_1(p_{11}(x),\ldots,p_{1n}(x)),\ldots,f_{r_1}\ (p_{r_1 1}(x),\ldots,p_{r_1 n}(x))$$

and denote all the r_2 different occurences of f in K_2 by

$$f_{r_1+1}\ (u_{11}(q_1,\ldots,q_t),\ldots,\ u_{1n}\ (q_1,\ldots,q_t)),\ldots,$$
$$f_{r_1+r_2}\ (u_{r_2 1}(q_1,\ldots,q_t),\ldots,u_{r_2 n}(q_1,\ldots,q_t)).$$

Then at each point p in L at which f is changed by an indexed assignment

(13) $$f(y_1,\ldots,y_n) = z$$

the code to update the value C = C(b) appearing in (10) is as follows;

(14) $s_2 = \{x \in s \mid p_{11}(x)\ \underline{eq}\ y_1\ \&\ldots\&\ p_{1n}\ (x)\ \underline{eq}\ y_n\ \underline{or}$
$\ldots\ \underline{or}$
$p_{r_1 1}(x)\ \underline{eq}\ y_1\ \&\ldots\&\ p_{r_1 n}(x)\ \underline{eq}\ y_n\}$;

$(\forall\ b \in \{K_1(x),\ x \in s_2 | K_1(x) \in D_b\})\ C(b) =$
$C(b) - \{y \in s_2 | K_1(y)\ \underline{eq}\ b\}$;;

$D_2 = \{<q_1,\ldots,q_t>,\ q_1 \in D_{q_1},\ldots,q_t \in D_{q_t} | u_{11}(q_1,\ldots,q_t)\ \underline{eq}\ y_1$
$\&\ldots\&\ u_{1n}(q_1,\ldots,q_t)\ \underline{eq}\ y_n\ \ \underline{or}\ \ \ldots\ \ \underline{or}$

$u_{r_2 1}(q_1,\ldots,q_t)\ \underline{eq}\ y_1\ \&\ldots\&\ u_{r_2 n}\ \ (q_1,\ldots,q_t)\}$;

$f(y_1,\ldots,y_n) = z$;

$(\forall\ b \in \{K_1(x),\ x \in s_2 | K_1(x) \in D_b\})\ C(b) = C(b)\ +$
$\{y \in s_2 \mid K_1(y)\ \underline{eq}\ b\}$;;

$$(\forall\, b \in \{K_2(q(1),\ldots,q(t)), q \in D_2 \mid \underline{\text{not}}\ b \in D_b\})$$
$$C(b) = \{x \in s \mid K_1(x)\ \underline{\text{eq}}\ b\};\ D_b = D_b + \{b\};;$$

If any of the restrictions we have imposed on (10) are lifted, the profitablity of reducing (10) will generally be lowered. Suppose, for example, that we allow variables $q_1,\ldots,q_t$ on which K_1 depends discontinuously to occur in K_1 of (10). Suppose that b_o is substituted for K_2 and that $b_1,\ldots,b_v$ arise from $q_1,\ldots,q_t$ by appropriate regrouping and substitution. Then the value $C = C(b_o,b_1,\ldots,b_v)$ would have to be kept available for every value $b_o \in D_{b_o},\ldots,b_v \in D_{b_v}$. The number of stored sets $C(b_o,\ldots,b_v)$ equals $\prod_{i=o}^{v} \# D_{b_i}$, and this many steps will be required for each iteration occurring in (11), (12), and (14). Moreover, the set s_2 appearing in the first line of (14) can now depend on $b_1,\ldots,b_v$, so that it may have to be defined as a map $s_2(b_1,\ldots,b_v)$.

The preceeding results apply in an interesting way to a class of set formers typified by

(15) $$C = \{x \in s \mid f(x) \in q\},$$

where the free variable q is a set. Recall from section (1) that (15) is continuous relative to small changes in s and relative to indexed assignments to f. If q is changed by a computation $q = q \pm q_1$ where $\# q_1 << \# q$, then the corresponding update correction

(16) $$C = C \pm \{x \in s \mid f(x) \in q_1\}$$

will often represent a small change to C. However, because (16) still requires an iteration over s, this update computation will often be too expensive to allow profitable reduction of (15).

For this reason, it is appropriate in handling (15) to use the identity

$$\{x \in s \mid f(x) \in q_1\} = \bigcup_{b \in q_1} \{x \in s \mid f(x) \ \underline{eq}\ b\}.$$

The sets $C' = \{ x \in s \mid f(x) \ \underline{eq}\ b\}$ which then appear can be treated by the methods sketched earlier in the present section, which require that we store a map $C'(b)$ for all b in an appropriate domain set D_b. Then the update operation (16) can be replaced by the less expensive code

(16') $$C = C \pm [+: b \in q_1]\ C'(b).$$

Set formers involving boolean valued subexpressions based on comparison operations such as

(17) $$C_1 = \{x \in s \mid f(x) < q\}$$

can sometimes be treated as special cases of (15). To see this, let M be the largest q value that needs to be considered, and let m be the minimum value of $\{f(x), x \in s\}$ over all f and s that can appear. Putting $sq = \{b, m \le b < q\}$, we see that (17) is equivalent to $\{x \in s \mid f(x) \in sq\}$.

If q changes slightly by $q = q \pm q_1$, then sq changes, also slightly, by

$$sq = sq + \{b, q \le b < q + q_1\}$$

or by

$$sq = sq - \{b, q - q_1 < b \le q\}.$$

Thus to update C_1 we can simply execute

(18) $$C_1 = C_1 + [+: q \le b < q + q_1]\ C'(b)$$

or $$C_1 = C_1 - [+: q - q_1 \le b < q]\ C'(b)$$

as appropriate.

Another class of special cases derives from

(19) $$C = \{x \in s \mid q \in f(x)\}$$

a set former which despite its close resemblance to (15) must be handled very differently. While (15) is continuous in all of its parameters, (19) is discontinuous in q. Thus we must save the value of C in a map C(q) defined for all values $q \in D_q$. Applying Rule 1 of the last section to (19) we derive the update computation

(20) $$(\forall q \in D_q)\ C(q) = C(q) \pm \{x \in s_1 \mid q \in f(x)\}.$$

When D_q is small, (20) can be expected to be inexpensive. When D_q is large, we may wish to extend the iteration (20) not over all of D_q but only over the smaller set

(21) $$C' = \{q \in D_q \mid (\exists x \in s_1 \mid q \in f(x))\}$$

which can be rewritten equivalently as

(21') $$C' = [+\colon x \in s_1]\ f(x) * D_q.$$

The techniques and concepts described in this section can be used to extend the basic continuity Rules 1 and 2 to generalized set formers involving multiple iterators, i.e.

$$(22)\quad C = \{e(x_1,\ldots,x_q),\ x_1 \in t_1, x_2 \in t_2(x_1),\ldots,x_q \in t_q(x_1,\ldots,x_{q-1}) \mid K(x_1,\ldots,x_q)\}.$$

Here we assume that K does not contain any free occurrences of any free parameters appearing in the set expressions $t_1,\ldots,t_q$. In order to reduce the total expression (22), we must first be able to reduce all the subexpressions t_j upon which (22) depends. Let us suppose for the sake of simplicity that each expression t_j in (22) is continuous in all of its parameters other than $x_1,\ldots,x_{j-1}$ (which we will treat as 'discontinuity parameters'). Then if the parameters on which t_j depends continuously undergo only small changes in L, we know from preceeding analysis that t_j is reducible; to reduce it we store its values as a map $\bar{t}_j(x_1,\ldots,x_{j-1})$. Since for $2 \leq j \leq q$, the range of values of x_j depends on the values of $x_1,\ldots,x_{j-1}$, it is convenient to consider the set

$$D_{<x_1,\ldots,x_{i-1}>} = \{<x_1,\ldots,x_{i-1}>,\ x_1 \in t_1,\ldots,x_{i-1} \in t_{i-1}(x_1,\ldots x_{i-2}$$

as the domain of the map $\bar{t}_i(x_1,\ldots,x_{i-1})$. Whenever a paramete in which t_i is continuous changes differentially, the values of $\bar{t}_i(x_1,\ldots,x_{i-1})$ must be updated for all necessar values of $<x_1,\ldots,x_{i-1}> \in D_{<x_1,\ldots,x_{i-1}}$ (these sets of values are defined by rules like those discussed earlier in the present section); the actual update operations to be employed will be defined by rules like those of section 2. Moreover, since any change to $t_i(x_1,\ldots,x_{i-1})$ can also change the domains $D_{<x_1,\ldots,x_j>}$ for $j > i$, and in particular can cause them to increase, it will sometimes be necessary to calculate additional map values

$\bar{t}_{i+1}(x_1,\ldots,x_i),\ldots,\ \bar{t}_q(x_1,\ldots,x_{q-1})$ when $t_i(x_1,\ldots,x_{i-1})$ changes. (Note that similar recalculations appear in the final lines of (14).) Once the $t_1,\ldots,t_q(x_1,\ldots,x_{q-1})$ are known to be reducible, then (22) can be reduced by replacing the subexpressions $t_1,\ldots,t_q(x_1,\ldots,x_{q-1})$ by the maps $\bar{t}_1,\ldots,\bar{t}_q(x_1,\ldots,x_{q-1})$ and evaluating the result on entrance to loop L. Then directly after any of the maps $\bar{t}_i(x_1,\ldots,x_{i-1})$ are updated within L we can insert code (derived in part from the code to update $\bar{t}_i$) which updates (22).

As an example, suppose that $t_1,\ldots,t_q(x_1,\ldots,x_{q-1})$ are all reducible to maps $\bar{t}_1,\ldots,\bar{t}_q(x_1,\ldots,x_{q-1})$, all these maps having domains as described just above, and suppose that a particular set expression t_I has the form

$\{x \in s \mid K'(x,x_1,\ldots,x_{I-1})\}$. If s is changed slightly within L by an assignment $s = s - \Delta$, then the appropriate updating operations are

$$(23)\quad (\forall x_1 \in \bar{t}_1,\ldots,\forall x_{I-1} \in \bar{t}_{I-1}(x_1,\ldots,x_{I-2}))\ \bar{t}_I(x_1,\ldots,x_{I-1}) = \bar{t}_I(x_1,\ldots,x_{I-1}) - \{x \in \Delta \mid K'(x,x_1,\ldots,x_{I-1})\};;$$

$$C = C - \{e(x_1,\ldots,x_q), x_1 \in \bar{t}_1,\ldots,x_{I-1} \in \bar{t}_{I-1}(x_1,\ldots,x_{I-2}), x_I \in \{x \in \Delta \mid K'(x,x_1,\ldots,x_{I-1})\}, x_{I+1} \in \bar{t}_{I+1}(x_1,\ldots,x_I),\ldots, x_q \in \bar{t}_q(x_1,\ldots,x_{q-1}) \mid K(x_1,\ldots,x_q)\};$$

However in the case of a differential modification $s = s + \Delta$ of s, the situation is not so fortunate. In this case the necessary update operations are described by the following much more complicated nested loop which updates old values of $\bar{t}_I$ and calculates new values of $\bar{t}_{I+1},\ldots,\bar{t}_q$:

(23') $(\forall x_1 \in$ [illegible] $t_{I-1}(x_1,\ldots,x_{I-2}))$ $\bar{t}_I(x_1,\ldots$ [illegible] $=$
$\bar{t}_I(x_1,\ldots,x_{I-1}) + \{x \in \Delta \mid K'(x_1,\ldots$ [illegible]$)\}$

$(\forall x_I$ [illegible] t_{I-1} [illegible] $\bar{t}_{I+1}(x_1,\ldots,x_I) = t_{I+1}(x_1,\ldots,x_I)$

$(\forall x_{I+1} \in \bar{t}_{I+1}(x_1,\ldots,x_I))$ $\bar{t}_{I+2}(x_1,\ldots,x_{I+1}) = t_{I+2}(x_1,\ldots,x_{I+1}$

$\ldots(\forall x_{q-1} \in \bar{t}_{q-1}(x_1,\ldots,x_{q-2}))$ $\bar{t}_q(x_1,\ldots,x_{q-1}) = t_q(x_1,\ldots,x_{q-1}$

<u>end</u> $\forall x_1,\ldots,$ <u>end</u> $\forall x_{q-1}$;

$C = C + \{e(x_1,\ldots,x_q),\ x_1 \in t_1,\ldots,x_{I-1} \in t_{I-1}(x_1,\ldots,x_{I-2}),$
$x_I \in$ [illegible] $(x_1,\ldots,x_{I-1}),\ x_{I+1} \in \bar{t}_{I+1}(x_1,\ldots,x_I),\ldots,$
$x_q \in \bar{t}_q(x_1,\ldots,x_{q-1}) \mid K(x_1,\ldots,x_q)\}$; .

Despite the complexity of (23'), it is not difficult to see that the work implied by the differential update calculations (23) and (23') is much less than the work necessary to perform the total calculation (22).

[illegible] differential changes to every value [illegible] the map $\bar{t}$ [illegible] x_{j-1}) can take [illegible] continuous [illegible] of the parameters except for [illegible] we can [illegible] continuous relative to all of the [illegible] variables of [illegible].

[illegible] and (23'), we note [illegible] of all of the computed set expressions t_j of (22) cannot be reduced profitably (because the space required by the maps $\bar{t}_j$ is excessive), we can sometimes reduce (22 without reducing the maps t_j. This is done by replacing (23) and/or (23') by the code

(24) $C = C \pm \{e(x_1,\ldots,x_q),\ x_1 \in t_1,\ldots,x_{I-1} \in t_{I-1}(x_1,\ldots,x_{I-2}),$
$x_I \in \{x \in \Delta \mid K'(x,x_1,\ldots,x_{I+1})\},\ x_{I+1} \in t_{I+1}(x_1,\ldots,x_I),\ldots,$
$x_q \in t_q(x_1,\ldots,x_{q-1}) \mid K(x_1,\ldots,x_q)\};$

to be executed within L after $s = s \pm \Delta$. Note that this code will often be inferior to (23) and/or (23') for cases in which (23)/(23') can be used, since in (24) the expression t_j must be recalculated repeatedly; however, since the set $\{x \in \Delta | K'(x,x_1,\ldots,x_{1-1})\}$ will generally be much smaller and easier to calculate than $\{x \in s \mid K'(x,x_1,\ldots,x_{1-1}\}$, the update operation (24) may be much less burdensome than the full calculation (22).

Rule 2 generalizes to (22) much more smoothly than Rule 1. Whenever an n-ary map f(all of whose occurrences in K of (22) have at least one parameter involving a bound variable x_i) is changed within L by an indexed assignment $f(y_1,\ldots,y_n) = z$,

the code required to update the value of (22) is

(25) $s_2 = \{<x_1,\ldots,x_q>,\ x_1 \in t_1,\ldots,x_q \in t_q\ (x_1,\ldots,x_{q-1})$
$\mid p_{11}(x_1,\ldots,x_q)\ \underline{eq}\ y_1\ \&\ldots\&\ p_{1n}(x_1,\ldots,x_q)\ \underline{eq}\ y_n\ \underline{or}$
$\ldots\ \underline{or}\ p_{r1}(x_1,\ldots,x_q)\ \underline{eq}\ y_1\ \&\ldots\&\ p_{rn}(x_1,\ldots,x_q)\ \underline{eq}\ y_n\};$
$C = C - \{e(x(1),\ldots,x(q)),\ x \in s_2 | K(x(1),\ldots,x(q))\};$
$f(y_1,\ldots,y_n) = z;$
$C = C + \{e(x(1),\ldots,x(q)),\ x \in s_2 | K(x(1),\ldots,x(q))\};$

where the notation used is like that which we have employed in the preceeding discussion of Rule 2. Much the same technique suffices to handle cases in which f can occur in the expression e of (22) as well as in K.

4. A Few Preliminary Remarks on Implementation

To implement set-theoretic strength reduction, we will need to perform the following steps:

1) develop an algorithm which, given parsed SETL code P plus possible additional information including use definition chains, type analysis, declarations describing the relative sizes of sets and maps, etc., finds all the expressions $E = E(x_1,\ldots,x_n)$ in P which can be reduced;

2) formalize rules (as we began to do in section 2 and 3) for updating all basic reducible forms of SETL expressions.

3) program the transformations (of parsed code P) which apply these update rules in several possible ways, one of which is not to apply reduction at all. These transformations must in effect match nonelementary SETL expressions to basic elementary reducible patterns, and must then carry out appropriate 'symbolic calculations'.

To avoid involvement with unprofitable cases, we suggest the following heuristic: reduce an expression $E = E(x_1,\ldots,x_n)$ only if either

a) it is continuous in all the parameters changed within some important loop L; or

b) it is discontinuous in some parameters which vary within L, but the map $\bar{E}$ needed to store its values is continuous in all the parameters x_j in which E is continuous; i.e., only a few values of $\bar{E}$ need to be changed when x_j is changed slightly (recall the discussion of reduction objections 1 and 2 of the previous section). (Since the reducing transformations which are actually applied leave behind large numbers of expressions which can be simplified very greatly by the application of constant folding, dead code and redundant expression elimination etc., it is important to incorporate these cleanup optimizations into any generalized strength reduction program.

4) select the most profitable of the program versions which result from application of the transformations (3). Work is currently in progress on implementing generalized reduction in strength as part of an experimental SETL optimizer system. While we do not intend to give details now, in the next section some of our implementation ideas will be implicit in our manual reduction of a simple program.

5. Is Set Theoretical Strength Reduction Apt to be Practical ?

To come to terms with this question we consider a simple example - Knuth's Topological Sort (this example is also studied in Earley, op.cit). The input assumed by this algorithm is a set s and a set of pairs SP representing an irreflexive transitive relation defined on s; as output, it produces a tuple t in which the elements of s are arranged in a total order consistent with the partial order sp. A concise SETL form of the algorithm is as follows:

```
(1)                  t = nult;
                     (while ∃ A ∈ s | sp {A} * s eq nℓ)
                         t = t + <A>;
                         s = s - {A};
                     end while;
```

The while loop L of (1) contains only one non-embedded expression, the existential quantifier

(2) $\exists\ A \in s \mid sp\,\{A\}\ \underline{eq}\ \underline{n\ell}$

which is not already in a 'most reduced' form which might be found in a table of such forms. Since use-definition analysis will reveal that A, the bound variable of the quantifier, is used within L, we transform (2) into

(3) $\exists\ A \in \{x \in s \mid sp\,\{x\}\ \underline{eq}\ \underline{n\ell}\}.$

This prepares for an attempt to reduce the setformer expression $\{x \in s \mid sp\ \{x\}\ \underline{eq}\ n\ell\}$, whose value we will call ZRCOUNT. The elementary reducible form to which ZRCOUNT belongs is $\{x \in s | K(x)\}$ where $K(x)$ is a boolean valued map defined on s. In ZRCOUNT s is already in reduced form, but matching shows us that $K(x)$ is to be taken as the subexpression $sp\ \{x\} \cap s\ \underline{eq}\ n\ell$; for the expression ZRCOUNT to be reducible, we require that its subexpression $K(x)$ should be reducible to a map $\bar{K}(x)$ continuous with respect to differential changes in all the parameters (besides x) upon which $K(x)$ depends continuously. To reduce $K(x)$ we first rewrite it as $((sp\ \{x\} \cap s) \subseteq \underline{n\ell})\ \&\ (\underline{n\ell} \subseteq (sp\ \{x\} \cap s))$, which simplifies to $(sp\ \{x\} \cap s) \subseteq \underline{n\ell}$. This last expression is in turn transformed into $[+: y \in (sp\ \{x\} \cap s) \mid \underline{not}\ y \in \underline{n\ell}]\ 1\ \underline{eq}\ 0$, and then again into $[+: y \in sp\ \{x\} \cap s]\ 1\ \underline{eq}\ 0$. To reduce integer equalities, we will always require that both arguments of $\underline{eq}$ be reducible. The parameter 0 of the preceding expression is elementary; the second parameter $K_2 = [+: y \in sp\ \{x\} \cap s]\ 1$ of the immediately preceeding equality is reducible only if the subexpression $K_3 = sp\ \{x\} \cap s$ is reducible. We observe that K_3 is continuous with respect to the induction parameter s but not with respect to x or sp. However, K_3 reduces to a map $\bar{K}_3\ (x)$ which is continuous relative to small changes to sp (note here that sp is a region constant of L). Furthermore $\bar{K}_3$ is continuous relative to small changes in s; i.e., before or after performing $s = s \underline{+}\ \Delta$, $\bar{K}_3$ can be updated by executing the code

(4) $(\forall\ y \in [+: x \in \Delta]\ succ\ (x))\ \bar{K}_3(y) = \bar{K}_3(y)\ \underline{+}\ sp\ \{y\} \cap \Delta;$

where $succ\ (x) = \{y \in D_x | x \in sp\ \{y\}\}$ is an auxilliary map

(defined for all elements of D_x) which, as part of our general reduction procedure, we introduce in making the reduction (4). In (4) it can be seen that the domain D_x over which x varies is simply the set s.

Once this reduction has been applied to K_3, we can go back and attempt reduction of the expression $K_2 = [+: y \in \bar{K}_3(x)]\ 1$ which led us to consideration of K_3.

The expression K_2 is discontinuous with respect to changes to x and indexed assignments to $\bar{K}_3$, but continuous (cf. Rule 3 of section 2) with respect to differential changes to each set $\bar{K}_3(x)$. However, K_2 can be reduced to a map $\bar{K}_2(x)$ (defined over all $x \in s$) which is continuous with respect to indexed assignments to $\bar{K}_3$, and also with respect to differential changes in the sets $\bar{K}_3(y)$. The update rules that must be applied to $\bar{K}_2(y)$ when $\bar{K}_3(y) = \bar{K}_3(y) \pm \Delta$ is executed are as follows:

(5) $\bar{K}_2(y) = \bar{K}_2(y) + [+: w \in (\Delta - \bar{K}_3(y))]\ 1$ and
$\bar{K}_2(y) = \bar{K}_2(y) - [+: w \in (\Delta \cap \bar{K}_3(y))]\ 1$ respectively.

From this it can be seen that the boolean quantity $K = \bar{K}_2(x)$ eq 0 reduces to a map $\bar{K}(x)$ (defined on s) which is continuous with respect to small changes to $\bar{K}_2$. Using this last fact, we can then go on to note that, ZRCOUNT = $\{x \in s \mid \bar{K}(x)\}$ is continuous with respect to small changes to the induction parameters s and $\bar{K}$, and to derive an update rule for ZRCOUNT.

Some subexpressions among the 5 reducible expressions and subexpressions, ZRCOUNT, K, K_2, K_3, and succ can be reduced together with their outer expressions in order to conserve space. Starting with the innermost expressions, we see that K_3 and K_2 both reduce to maps defined over the domain of the same discontinuity parameter, and hence that they can be condensed into a single map COUNT(x) = $[+: y \in sp\{x\} \cap s]\ 1$. Indeed, since $\bar{K}_3$ depends only on $\bar{K}_2$, we need only combine update rules (4) and (5) to derive this fact. When s is changed by $s = s - \Delta$, COUNT can be updated by executing the following statement just prior to this change:

(6) $(\forall y \in [+: x \in \Delta]\ succ(x))$ COUNT(y)=COUNT(y)$-[+: w \in (sp\{y\} \cap \Delta \cap s)]\ 1$;;

Here succ is a map whose reduction is governed by the discontinuity rule (4) above. More specifically, succ depends continuously on s, and rule (20, 21') of section 3 states that succ is to be updated after $s = s - \Delta$ by executing

(7) $(\forall b \in [+: x \in \Delta]\ sp\{x\} \cap s)$ succ(b) = succ(b) $- \{x \in \Delta \mid x \in sp\{x\}\}$;;

Once COUNT is reduced, we can reduce ZRCOUNT = {x ∈ s| COUNT(x) eq 0} by immediate application of Rules 1 and 2 of section 2. Hence, we can entirely avoid involvement with the auxiliary map $\bar{K}$. The transformations that have been applied lead to the following much improved form of the topological sort (1)

```
          t = nult;
          (∀ A ∈ s) COUNT(A) = [+: y ∈ sp {A} * s] 1;
                    succ (A) = {y ∈ s | A ∈ sp {y}};;
          ZRCOUNT = {x ∈ s | COUNT(x) eq 0};
          (while ∃ A ∈ ZRCOUNT)
                    t = t + <A>;
(8)                 (∀y ∈ succ(A)) COUNT(y) = COUNT(y)
                                 - [+: w ∈(sp {y} * {A} * s)] 1;
                                   ZRCOUNT = ZRCOUNT
                               + if COUNT(y) eq 0 then  {y} else  nℓ;;
                    s = s - {A};
(9)                 (∀b ∈ sp {A} * s) succ(b) = succ(b)-{x ∈ {A} |
                                                           x ∈ sp {x}};;
(10)                ZRCOUNT = ZRCOUNT - {x ∈ {A}| COUNT(x) eq 0};
                    end while;
```

A very good optimizer might determine that the expression

[+: w ∈ (sp {y} * {A} * s)] 1 of (8) is just the constant 1, that sp{A} * s of (9) is nℓ, and that {x ∈ {A}| COUNT (x) eq 0} of (10) is simply {A}. Also, s can be eliminated as a dead variable inside L. With these improvements, a final version of the toplogical sort would be written as follows:

```
          t = nult;
          (∀ A ∈ s) COUNT(A) = [+: y ∈ sp {A} * s] 1 ;
                    succ(A) = {y ∈ s | A ∈ sp {y}};;
          ZRCOUNT = {x ∈ s | COUNT(x) eq 0};
          (while ∃ A ∈ ZRCOUNT)
                    t = t + <A>;
                    (∀y ∈ succ(A)) COUNT(y) = COUNT(y) - 1;
                          ZRCOUNT = ZRCOUNT + if COUNT(y) eq 0 then
```

```
                                   {x} else nℓ;;
                         ZRCOUNT = ZRCOUNT - {A};
                         end while;
```

If properly implemented, this final version of the topological sort algorithm will run in a number of cycles proportional to the number *nsp* of elements in the map *sp*. The original form (1) of the algorithm will require something like *nsp* * (# *s*) * (# *s*) cycles, which can be much larger. However, the chain of symbolic transformations which leads from (1) to (4) is quite long, and it appears doubtful that an automatic optimizer will be able to traverse this chain unguided, especially since in this case, and still more so in more general cases, there exist competing transformations whose application an automatic system would have to consider. Thus it appears likely that the answer to the question prefixed to the present section is 'probably not'. However, it may be practical to design a semi-automatic system, whose user may interatively signify that he wishes a particular subexpression of a program to be reduced in one of several possible ways. This may make it possible to derive efficient program v rsions with more certainty and less labor than would be typical if the final program version had to be worked out in an entirely manual way.

6. A few General Remarks on 'Continuity' in High-Level Programming

Something close to the heuristic notion of 'continuity' suggested in section 1 of the present newsletter often seems to play an important role in algorithm design. In newsletter 135A, we noted that programs will commonly be structured as nests of loops; many of the loops in such a structure realise some set-theoretical expression $E = E(a)$ by applying a map $M = M_a$ repeatedly until E emerges as a fixed point of M. The efficiency of programs having this structure can often be improved by noting that within an 'outer' loop L_{out} which contains an 'inner' loop L_{in} producing the value E(a), the parameters a of E(a) are varied only slightly. An observation of this kind often allows one to restructure L_{in} for efficiency by calculating E using its available previous value, which calculation can of course be substantially more rapid than calculation of E 'from scratch' would be. This line of thought makes it clear that an algorithm for evaluating $E = E(a)$ will be of particular interest if it has good continuity properties. Suppose for example that E(a) is calculated as the fixed point of a transformation M_a. There will in general be many transformations M_a M'_a, M''_a,... all of which have the value E(a) as fixed point; among these transformations will often be particularly interested in those M_a for which the sequence E(a), $M_{\bar{a}}(E(a))$, $M_{\bar{a}}(M_{\bar{a}}(F(a)))$,... leads after comparatively few iterations to the fixed point $E(\bar{a})$ of $M_{\bar{a}}$ (where we assume that the parameter values a and $\bar{a}$ differ only slightly). This line of thought points up problem area in algorithmic analysis which has not yet been explored systematically.

It is instructive to consider one or two cases in which algorithms or data structures having useful properties of continuity are known or can be devised. First consider sorting, and the problem of maintaining the sorted form of a set s to which modifications are continually being made by addition and deletion. If there are n elements in s, the bubble sort will correct for an insertion or deletion in approximately n/2 steps. However, if the sorted form of s is kept as a balanced tree, one can connect for an insertion or deletion in log n steps.

Next consider the minimum *min* of a set s of integers. After an insertion s = s + {x} one can update *min* by executing

```
min = if x lt min then x else min;
```

and after a deletion s = s - {x} by executing

```
min = if x ne min then min else (sort x)(1).
```

Since in many situations the minimum of s will rarely be deleted, it will rarely be necessary in using this procedure to generate the sorted form of x. On the other hand, if the minimum of s is used in a process, as for example a selection sort, which invariably deletes the minimum, then one wants an algorithm which has good 'worst case' rather good' typical case' continuity properties. In such a situation, it is reasonable to arrange s as a vector v = tree s having the implicit tree property, i.e. v(n) < v(2*n) and v(n)<v(2*n+1). Then the minimum of s is necessarily v(1), i.e. can be expressed as (tree s)(1). Note that in approaching the quantity min s in this way, we have essentially factored the function min into the product of two functions, of which the first,

tree, is continuous, while the second (indexing) can be performed rapidly.

Algorithm continuity becomes particularly important in connection with procedures which attempt to optimize large combinatorial structures S by applying local transformations to them, especially if the transformations to be applied interact in a way which makes the applicability of each transformation depend on the effects of the transformation which preceeds it. Interaction of this kind quite typically occurs when we apply optimizing transformation to programs, since in this case the analysis which must be carried out to determine whether a given transformation can be applied is complex and expensive. It is therefore important to find methods which allow the results of a prior analysis to be updated rapidly to give a new analysis when a program structure S is modified. If available, such techniques would make it possible for an optimization algorithm to explore spaces of program transformations as freely as a manual programmer does; until such techniques are developed, our approach to optimization is bound to remain 'stiff' and somewhat cautious.