EXPRESSION CONTINUITY AND THE FORMAL DIFFERENTIATION OF ALGORITHMS*

Bob Paige and J. T. Schwartz
Computer Science Department
Courant Institute of Mathematical Sciences
New York University

Abstract: This paper explores the technique of 'strength reduction' or 'formal differentation' in a set theoretic context, as recently introduced by Earley. We give pragmatic rules for the recognition and treatment of reasonably general cases in which the optimization is applicable, and consider some of the problems which arise in actually attempting to install this optimization as part of a compiling system.

1. Background.

Continued development of very high level languages depends in part on our ability to recognize common major aspects of programming style as resulting from the application of some standard technique of program improvement to an underlying program prototype. A technique of program improvement that we are able to perceive as general can become the basis for a general optimization method; and once this method is in hand, we can safely write programs in relatively simple unoptimized forms, since their more complex optimized forms will be seen as obvious improvements, derivable mechanically or semi-mechanically, from these simple forms. An interesting new high level optimization of this form has recently been described by Jay Earley [E1]. This optimization was applied by Earley to his proposed language VERS2 [E2]; but his ideas carry over easily to other set-theoretic languages such as SETL.

Earley's optimization technique, which he calls iterator inversion and which we shall prefer to call formal differentation, generalizes the classical method of 'reduction in operator strength', for which see [A1,C1,C2,K1-K5]. The basic formal idea of this technique can be put as follows. Suppose that an expression $C = f(x_1,\ldots,x_n)$ will be used repeatedly in a program region R, but that its calculation cannot be moved outside R because its parameters $x_1,\ldots,x_n$ are modified within R. If we make C available on each entry to R (by calculating it before entry) and keep C available within R by recalculating it each time one of its parameters is modified, then we may be able to avoid all full calculations of C within R.

For this approach to be reasonable, there must be some way of recalculating C more easily after its parameters are modified than by calculating C afresh each time it is required. For this to be the case, we are likely to require two conditions to be satisfied, which we may describe heuristically as follows:

(a) Within R, all changes $x_j = h(x_1,\ldots,x_n)$ to the parameters x_j should all be 'minor' or 'small'.

(b) For each such assignment, there should exist an update identity

$$f(x_1,\ldots,x_{j-1}, h(x_1,\ldots,x_n), x_{j+1},\ldots,x_n) = g(x_1,\ldots,x_n), f(x_1,\ldots,x_n))$$

*Work supported by NSF under Grant NSF-MCS76-00116.

which allows the new value f_{NEW} of f to be calculated from its old value f_{old} by an 'easy' calculation $f_{NEW} = g(x_1,\ldots,x_n,f_{OLD})$. If this is the case, then we say that the expression f is continuous in its parameters (relative to the modifications occurring within R).

The application of this idea in the set-theoretic setting was initiated by Earley and has been pushed further by Fong and Ullman [F1] who made the interesting observation that formal differentiation in a set-theoretic setting could actually improve the asymptotic behavior of an algorithm and that this fact could be used to develop a theoretical characterization of the situations in which this technique applied. In the discussion which follows, we shall pursue Earley's idea in a less formal sense than Fong and Ullman, aiming to state pragmatic rules for the discovery and treatment of reasonably general cases in which formal differentiation can be applied.

2. Initial Examples.

In SETL the computations s = s + {x} and s = s - {x} respectively add and delete the value of the element x from the set s. Both operations change s only 'slightly'. Similarly, if s and t are sets and the number of elements of Δ, #Δ, is much smaller than #s, then modifications of the form s = s ± Δ, represent 'slight' changes to s. If f is a set of pairs used as a SETL mapping, then the operation f(x) = z, which replaces all pairs whose first element is x by the pair <x,z> causes f to change slightly. If f is a set of (n+1)-tuples used as a multi-parameter mapping, then the indexed assignment $f(x_1,\ldots,x_n) = z$ alters f only slightly. If in a strongly connected region R all changes to variables which are sets or mappings are slight modifications of the kind just described, then these variables can be called induction variables within the region R.

Examples of SETL expressions continuous in all of their parameters are: set union s + t, set intersection s * t, and set difference s - t. For example, if we consider the expression s - t, then if s undergoes differential change s = s ± Δ , the value C of s-t can be restored by executing the corresponding code,

(1) C = C + (Δ - t), or C = C - Δ

Similarly, if t undergoes a small change t = t ± Δ , we can update C by performing

(2) C = C - Δ or C = C + (Δ * s).

As Earley has emphasized, expressions involving set-formers provide more interesting examples of this phenomenon of 'continuity'. The SETL expression

(3) C = {x ∈ s | f(x) eq q}

which computes the set of all values of the set s such that the boolean valued subexpression f(x) eq q holds is a prototypical example. This expression is continuous in s and f, but discontinuous in q. If s is varied slightly by s = s ± Δ then C can be updated by executing

(4) C = C ± {x ∈ Δ | f(x) eq q}

which represents a small change to C. When f is changed by executing the indexed assignment $f(y_0) = z$, then C can be updated by executing

(5) C = if y_0 ∈ s then C - (if f(y) eq q then {y_0} else nullset)
+ if z eq q then {y_0} else nullset;

just before the assignment $f(y_0) = z$.

3. Formal Differentiation of Set Theoretical Expressions Continuous in All of Their Parameters.

We now formulate a few general rules concerning the formal differentiation of set theoretical expressions continuous in all of their free parameters. It must 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 formal differentiation; 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 R and that the following conditions hold:

(i) The boolean valued subexpression K(x) contains m free occurrences 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 R; i.e., inside R, s is only changed by slight modifications of the form $s = s \pm \Delta$, where Δ 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 formally differentiate the expression (1) in the region R. 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 differentiate the expression (1) within a strongly connected region R, we begin by making it available on entrance to R. This is done by inserting the assignment $C = \{x \in s \mid K(x)\}$ into R's initialization block. Then , at each point p inside R 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 2 above): small changes in the set s, and changes to f which result from an indexed assignment.

Rule 1: Either before or after each occurrence in R of a small change to s, $s = s \pm \Delta$, we can insert the differential update operation

(2) $$C = C \pm \{x \in \Delta \mid K(x)\}$$

which preserves the value of (1) in C.

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 R in which the n-ary mapping f is changed slightly by an indexed assignment, make the following program transformation:

Relative Position	Original Code
P	$f(x_1,\ldots,y_n) = z$
	Differentiated Code
P-2	$s_2 = \{x \in s \; p_{11}(x) \; \underline{eq} \; y_1 \; \&\ldots\& \; p_{1n}(x) \; \underline{eq} \; y_n \; \underline{or}$ $\ldots \; \underline{or} \; p_{r1}(x) \; \underline{eq} \; y_1 \; \&\ldots\& \; p_{rn}(x) \; \underline{eq} \; y_n \}$
p-1	$C = C - \{x \in s_2 \mid K(x)\}$
p	$f(y_1,\ldots,y_n) = z$
p+1	$C = C + \{x \in s_2 \mid K(x)\}$

It can be shown that rule 2 is a corollary of rule 1. To see this, consider the set $D_{f_i} = \{\langle p_{i1}(x),\ldots,p_{in}(x)\rangle \mid x \in s\}$. Let p_i be the mapping whose domain is s and whose value is $p_i(x) = \langle p_{i1}(x),\ldots,p_{in}(x)\rangle$. Then for any n-tuple $\langle y_1,\ldots,y_n\rangle$, we have $p_i^{-1}(y_1,\ldots,y_n) = \{x \in s \mid p_{i1}(x) \; \underline{eq} \; y_1 \; \&\ldots\& \; p_{in}(x) \; \underline{eq} \; y_n\}$. 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 $\langle y_1,\ldots,y_n\rangle$. Moreover if s is modified by deletion of $\bigcup_{i=1}^{r} p_i^{-1}(y_1,\ldots,y_n)$, then the n-tuple $\langle y_1,\ldots,y_n\rangle$ 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 $\langle y_1,\ldots,y_n\rangle \notin \bigcup_{i=1}^{r} D_{f_i}$ just before point p, then the statement $f(y_1,\ldots,y_n) = z$ does not cause a significant change in any of the occurrences of f in (1). Consequently, C is not spoiled by assignment $f(y_1,\ldots,y_n) = z$ and hence, it remains available. Suppose now that in expression (1) C is available on entrance to the program point p. Then we could 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   s2 = {x ∈ s | p11(x) eq y1 &...& p1n(x) eq yn
                    or ... or pr1(x) eq y1 &...& prn(x) eq yn}
      p-2   s  =  s - s2
(3)   p-1   C  =  C - {x ∈ s2 | K(x)}
      p     f(y1,...,yn) = z
      p+1   s = s + s2
      p+2   C = C + {x ∈ s2 | K(x)}
```

In this code C is not spoiled by the statement $f(y_1,\ldots,y_n) = z$. 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 $R_1,\ldots,R_r$ are disjoint sets. Then $\{x \in s_2 \mid K(x)\}$ can be rewritten as $\bigcup_{i=1}^{r} \{x \in R_i \mid K(x)\}$. Suppose also that in each set $\{x \in R_i \mid K(x)\}$ $K(x)$ can be transformed (by elimination of redundant operations) into an equivalent but easier-to-evaluate expression $K_i(x)$. Then it may be worthwhile to work with the partition $\{R_i\}$ of s_2 instead of s_2 and to rewrite

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

As an example of this, observe that if we let $R_i = \{x \in (s - \bigcup_{k=0}^{i-1} R_k) \mid p_{i1}(x) \underline{eq} y_1 \;\&\ldots\&\; p_{in}(x) \underline{eq} y_n\}$, where $R_0 = \emptyset$, 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.

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

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

Suppose that the mapping f is changed slightly by an indexed assignment, $f(y_0) = z$ 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. (Note that this is not precisely the partition described above: rather, it arises from that partition by a further formal transformation, which for simplicity we omit.) 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 $R_1 = \{x \in s \mid (x + 1) \underline{eq} y_0\}$. Since the parameter part of f(f(x + 1) + 1) involves f(x + 1), we set

$$R_2 = \{x \in (s - R_1) \mid f(x + 1) + 1 \underline{eq} y_0\}$$

and similarly, we put

$$R_3 = \{x \in (s - (R_1 \cup R_2)) \mid f(f(x+1) + 1) \underline{eq} y_0\}.$$

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

```
R1 = {x ∈ s | x + 1 eq y0};
R2 =  x ∈ (s-R1) | f(x+1) +1 eq y0};
R3 =  x ∈ (s-(R1 * R2)) | f(f(x+1) + 1) eq y0};
C = C - {x ∈ R1 | f(f(f(y0) + 1)) eq f(f(y0) + 1)}
      - {x ∈ R2 | f(f(y0)) eq f(y0)}
      - {x ∈ R3 | f(y0) eq y0} ;
f(y0) = z;
C = C + {x ∈ R1 | f(f(z+1)) eq f(z+1)}
      + {x ∈ R2 | f(Z) eq z} + {x ∈ R3|z eq y0};
```

As noted by Earley, the method of formal differentiation which has been described can be extended in a useful way to apply to various SETL expressions that implicitly contain set formers. Among these are the forall iterator (i.e., (∀x ∈ s|K(x)) , block), the existential and universal quantifiers (i.e., ∃x ∈ s | K(x) and ∀x ∈ s | K(x)), and the compound operator (i.e.

[<binop>: x ∈ s | K(x)] e(x)).

To formally differentiate these expressions, we rewrite them by replacing the implicit set former subpart, $x \in s \mid K(x)$, which they contain with $x \in \{u \in s \mid K(u)\}$. The set former subexpression thus exposed can then be differentiated using Rules 1 and 2.

Let us now consider more closely the SETL compound operation

(5) $$C_1 = [\underline{binop}: x \in C]\ e(x)\ ,$$

an illustrative example of which $[+: x \in C]\ e(x)$ calculates the value $\sum_{x \in C} e(x)$. In general, $[\underline{binop}: x \in C]\ e(x)$ means $e(x_1)\ \underline{binop} \ldots \underline{binop}\ e(x_n)$ where $C = \{x_1, \ldots, x_n\}$. For the general case in which the binary operation binop has an appropriate inverse, inverse binop (e.g. arithmetic binary + with - as its inverse), we note that (5) is continuous relative to slight changes in C; i.e., before an occurrence of the code $C = C \pm \Delta$, C_1 can be updated by an appropriate inexpensive change, either

(6) $$C_1 = C_1\ \underline{binop}[\underline{binop}: x \in (\Delta - C)]\ e(x);$$

or

(6') $$C_1 = C_1\ \underline{inverse\ binop}[\underline{binop}: x\ (\Delta * C)]\ e(x);$$

Applying the heuristic rule 'continuous functions of continuous functions are continuous' to C_1 of (5) and C of (1) yields update identities for a more general compound operation form

(7) $$C = [\underline{binop}: x \in s \mid K(x)]\ e(x)\ .$$

In order to formally differentiate the expression (7) in a strongly connected region R, we require all the conditions imposed on (1) to hold, and also require that neither the set s nor the n-ary mapping symbol f occurring in K should appear in the subexpression e of (7). If all these conditions are met, we differentiate (7) by first making it available on entrance to R. This is accomplished by inserting the assignment (7) into R's initialization block. Next, within R at each point p where C can be spoiled by modifications to the induction variables s or f, we can apply the following continuity rules for (7) that parallel rules 1 and 2:

Rule 3: where s is modified in R by the code $s = s \pm \Delta$, the value of (7) can be maintained in C by executing

(7') $$C = C\ \underline{binop}[\underline{binop}: x \in (\Delta - s) \mid K(x)]\ e(x)$$

or

(7") $$C = C\ \underline{inverse\ binop}[\underline{binop}: x \in \Delta * s \mid K(x)]e(x)$$

respectively.

A similar rule analogous to Rule 2 can be stated to cover the case of changes to f.

These rules imply continuity properties for many other high level SETL operations. The counting operation applied to a set former; i.e., $\#\{x \in s \mid K(x)\}$ can be treated as $[+: x \in s \mid K(x)]\ 1$. When side effects of the existential and universal quantifiers can be ignored, then the corresponding SETL forms $\exists x \in s \mid K(x)$ and $\forall x \in s \mid K(x)$ can be rewritten as $[+: x \in s \mid K(x)]\ 1\ \underline{ne}\ 0$ and $[+: x \in s \mid \underline{not}\ K(x)]\ 1\ \underline{eq}\ 0$ respectively. Set inclusion (the predicate $R \supseteq S$) is continuous in both S and R since in SETL, R incs S can be handled as $[+: x \in S \mid x\ \underline{notin}\ R]\ 1\ \underline{eq}\ 0$.

4. Differentiation of Set-Formers Containing Parameters on which They Depend Discontinuously.

Most SETL expressions will not be continuous in all the parameters on which they depend. For example, the set former

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

is continuous in s and f, but it is discontinuous relative to changes in q. Suppose that the expression (1) occurs in a strongly connected region R, and suppose also that all changes to s and f are slight within R. Then if we know the set D_q of all values that q can take inside R, the computation (1) can be removed from R according to the following general formal differentiation scheme:

(a) define a map $C(q) = \{x \in s \mid f(x,q)$ gt $q\}$ for all values $q \in D_q$ on entrance to R;

(b) whenever differential changes to s or f occur in R, modify each stored set C(q) according to rules 1 and 2 (cf. section 3) for all values $q \in D_q$; e.g. after $s = s \pm \Delta$ we can perform the following update code:

$(\forall q \in D_q)\ C(q) = C(q) \pm \{x \in \Delta \mid f(x,q)$ gt $q\}$;

(c) whenever q changes in R nothing more is needed;

(d) replace all computations (1) in R by C(q).

Three major problems can easily make this approach infeasible:

(a) storage of all the sets C(q) may be too expensive;

(b) updating all the sets C(q) when a parameter upon which C depends continuously is modified may waste more time than is saved by avoiding the calculation of C;

(c) storage of the set D_q may be too expensive.

Nevertheless, these problems can be overcome in cases in which we know that when an expression E must be stored as a map, the map will be continuous relative to differentia changes in the continuity parameters of E (i.e., update operations are only required ove a small part of the domain of the map). Fortunately, this property holds for a few special cases of common occurrence in SETL programs. One such example is

(2) $C_1 = \{x \in s \mid f(x)$ eq $q\}$

If, on entrance to R, C_1 is stored as a map $C_1(q)$ defined on D_q, and if s and f are induction variables of R, then (2) can be differentiated profitably. After the occurrence of $s = s \pm \Delta$, we can invoke the update rule

(3) $(\forall x \in \Delta \mid f(x) \in D_q)\ C_1(f(x)) = C_1(f(x)) \pm \{x\}$;;

Whenever the indexed assignment $f(x_0) = z$ occurs, the following inexpensive code can be executed:

(4) if $x_0 \in s$ and $f(x_0) \in D_q$ then $C_1(f(x_0)) = C_1(f(x_{0-})) - \{x_0\}$;;
$f(x_0) = z$;
if $x_0 \in s$ and $f(x_0) \in D_q$ then $C_1(f(x_0)) = C_1(f(x_0)) + \{x_0\}$;;

Example (2) above typifies the treatment of a somewhat broader class of expressions that can often be differentiated profitably. Within this class we consider the set formers

(5) $C = \{x \in s \mid K_1(x)$ 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 (5) is a subexpression only involving x, parameters upon which (5) 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 (5) is assumed to be a subexpression only involving the parameters $q_1,\ldots,q_t$ on which C depends discontinuously, and

also the maps f_i.

We assume that expressions estimating $D_{q_i},\ldots,D_{q_t}$ are either available at compile time or are computed dynamically and are available on entrance to R. We can simplify (5) 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 $b \in D_b$.

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

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

where the free variable q is a set. Recall from section 3 that (6) 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

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

will often represent a small change to C. However, because the set former in (7) still requires an iteration over s, this update computation will often be too expensive to allow profitable differentiation of (6).

For this reason, it is appropriate in handling (6) 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 (7) can be replaced by the less expensive code

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

Set formers involving boolean valued subexpressions involving comparison operations such as

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

can sometimes be treated as special cases of (6). 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 (9) 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

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

or

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

as appropriate.

Another class of special cases derives from

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

a set former which despite its close resemblance to (6) must be handled very differently. Whereas (6) is continuous in all of its parameters, (11) 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 (11) we derive the update computation

(12) $$(\forall q \in D_q)\ C(q) = C(q) \pm \{x \in \Delta \mid q \in f(x)\};.$$

When D_q is small, (12) can be expected to be inexpensive. When D_q is large, we may wish to extend the iteration (12) not over all of D_q but only over the smaller set

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

which can be rewritten equivalently as

(14) $$C' = [+: x \in \Delta]\ f(x) * D_q\ .$$

5. A Few Preliminary Remarks on Implementation

To implement formal differentiation rules sketched in the preceding pages, 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 profitably be formally differentiated.

(2) Formalize rules (as we began to do in sections 3 and 4) for updating all basic differentiable 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 them at all. These transformations must in effect match nonelementary SETL expressions to basic elementary patterns, and must then carry out appropriate 'symbolic calculations'.

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

(a) it is continuous in all the parameters changed within some strongly connected region R; or

(b) it is discontinuous in some parameters which vary within R 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 objectins (a) and (b) of the previous section). Since the transformations which are actually applied will 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 formal differentiation scheme.

(4) Select the most profitable of the program versions which result from application of the transformations (3). In the next section some of our implementation ideas will appear implicitly in our manual optimization of a simple program.

6. How Automatically Can Formal Differentiation be Applied? Study of an Example.

To come to terms with the above 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 = nultuple;
        (while ∃a ∈ s | sp{a} * s eq nullset)
            t = t + <a>;     /* tuple concatenation */
            s = s - {a};
        end while;
```

The while loop L of (1) contains only one nonembedded expression which is not already in a 'most reduced' form such as might be found in a table of such forms. This is the existential quantifier

```
(2)     ∃a ∈ s | sp{a} * s eq nullset
```

Since use-definition analysis will reveal that a, the bound variable of the quantifier, is used within L, we transform (2) into

```
(3)     ∃a ∈ {x ∈ s | sp{a} * s eq nullset}.
```

This prepares for an attempt to differentiate the set former expression {x ∈ s | sp{a} * s eq nullset},whose value we will call ZRCOUNT. The elementary pattern describing ZRCOUNT is {x ∈ s|K(x)}, where K(x) is the subexpression sp{x} * s eq nullset. In ZRCOUNT s is already in a reduced form. Thus, to reduce the expression ZRCOUNT, we must reduce its subexpression K(x). To reduce K(x) we first rewrite it as ((sp{x} * s) ⊆ nullset)& (nullset ⊆ (sp{x} * s)), which simplifies to (sp{x} * s) ⊆ nullset. This last expression is in turn transformed into [+: y ∈ (sp{x} * s) | not y ∈ nullset] 1 eq 0, and then again into [+: y ∈ sp{x} * s] 1 eq 0. To reduce integer equalities, we will always require that both arguments of eq be reducible. The parameter 0 of the preceding expression is elementary; the second parameter K_2 = [+:y ∈ sp{x} * s] 1 of the immediately preceding equality is reducible only if the subexpression K_3 = sp{x} * s is reducible. We observe that K_3 is continuous with respect to the induction variable s but not with respect to x or sp. However, sp is a region constant of L, and hence, K_3 can be reduced to a map $\bar{K}_3(x)$ depending only on the single discontinuity parameter x. Furthermore $\bar{K}_3$ is continuous relative to small changes in s; i.e., at each small change s = s ± Δ , K_3 can be updated by executing the code

```
(4)     (∀y ∈ [+: x ∈ Δ] succ(x)) K̄3(y) = K̄3(y) ± sp{y} * Δ;;
```

where succ(x) = {y ∈ s | x ∈ sp{y}} is an auxiliary map introduced when we apply the general differentiation rules sketched in the preceding pages.

Once this transformation has been applied to K_3 , we can go back and attempt to differentiate the expression K_2 = [+: y ∈ $\bar{K}_3(x)$] 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 3) with respect to differential changes to each set $\bar{K}_3(x)$. Thus K_2 can be reduced to a map $\bar{K}_2(x)$ (defined over all x ∈ s) depending only on the discontinuity parameter x. The

update rules which must be applied to $\overline{K}_2(y)$ when $\overline{K}_3(y) = \overline{K}_3(y) \pm \Delta$ is executed are respectively as follows:

(5) $\overline{K}_2(y) = \overline{K}_2(y) + [+: w \in (\Delta - \overline{K}_3(y))]\ 1$

and $\overline{K}_2(y) = \overline{K}_2(y) - [+: w \in (\Delta * \overline{K}_3(y))]\ 1$

Since $\overline{K}_3$ plays no direct but only a subsidiary role in our calculations, we can eliminate it by combining update rules (4) and (5). This leads to the following update rule for K_2, which we shall henceforward call by the more heuristic name COUNT:

(6) (∀y ∈ [+: x ∈ Δ] succ(x))COUNT(y)=COUNT(y)-[+: w ∈ (sp{y}*Δ*s)] 1;;

Here succ is as in(4) above. Note that succ depends continuously on s, and that by applying the general rules of section 3 we see that succ is to be updated after s = s - Δ by executing

(7) (∀b ∈ [+: x ∈ Δ] sp{x} * s) succ(b) = succ(b) - {x ∈ Δ|x ∈ 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 3; this eliminates the auxiliary map $\overline{K}$. Combining all these transformations and applying appropriate cleanup, we obtai the following much improved form of the topological sort (1)

```
     t = nultuple;
     (∀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 nullset;;
          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 that the constant 1, that sp{a} * s of (9) is nullset, 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 topological sort could be written as follows:

```
     t = nultuple;
     (∀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 {y} else nullset;;
          ZRCOUNT = ZRCOUNT - { a} ;
     end while;
```

This final version of the topological sort algorithm will run in a number of cycle 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. However, it may be practical to design a semi-automatic system, whose user may interactively signify that he wishes a particular subexpression of a program to be differentiated in one of several possible ways. This may make it possible to derive efficient program versions 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.

7. Conclusion

Work is currently in progress to produce working SETL algorithms that generalize and realize the techniques sketched in the preceding pages. Algorithms which perform auxiliary functions such as preparatory or cleanup transformations are being studied. We expect cleanup to be a major problem. Our goal is to implement formal differentiation as a semi-manual extension of the optimized SETL system currently under development at N.Y.U. [B1, B2, Sch3, St1, St2].

Formal differentiation has great potential for transforming very high level code to reasonably efficient low level code. As a SETL to SETL optimization it works best when a SETL program is written in a very 'high' style (in which iterative operations abound).* The result of formal differentiation is then highly efficient low style SETL. As is pointed out in [Sch2], program validation can be expected to apply most easily to very high level programs since the required correctness proofs can be expected to be shorter.

*In this regard, see the remarks in [L1], regarding a simpler experiment on source-to-source transformation in SETL.

References

[A1] Allen, Frances F., Cocke, John, and Kennedy, Ken, "Reduction of Operator Strength," Rice University Tech. Rep. 476-093-6, August 1974.

[B1] Balzer, Robert, Goldman, Neil, and Wile, David, "On the Transformational Implementation Approach to Programming," UCS/Information Sciences Institute, Marina del Rey, Calif., April 1975.

[B2] Burstall, R. M., and Darlington, J., "A Transformation System for Developing Recursive Programs," D.A.I. Research Report No. 19, University of Edinburgh, March 1976.

[C1] Cocke, John and Kennedy, Ken, "An Algorithm for Reduction of Operator Strength," Rice University Technical Report 476-093-2.

[C2] Cocke, John and Schwartz, J. T., Programming Languages and Their Compilers, Courant Institute of Mathematical Sciences, New York University, 1969.

[E1] Earley, Jay, "High Level Iterators and a Method for Automatically Designing Data Structure Representation," Dept. of Elec. Engr. & Computer Sciences, and the Electronic Research Lab., Univ. of California, Berkeley, Calif., February 1974.

[E2] Earley, Jay, "High Level Operations in Automatic Programming," Dept. of Elec. Engr. & Comp. Sciences and the Electronics Research Lab., University of California, Berkeley, Calif., October 1973.

[F1] Fong, Amelia C. and Ullman, Jeffrey, D., "Induction Variables in Very High Level Languages," Proc. Third ACM Symposium on Principles of Programming Languages, January 1976.

[K1] Kennedy, Ken, "Reduction in Strength Using Hashed Temporaries," SETL Newsletter No. 102, March 12, 1973.

[K2] Kennedy, Ken, "Linear Function Test Replacement," SETL Newsletter No. 107, May 20, 1973.

[K3] Kennedy, Ken, "Global Dead Computation Elimination," SETL Newsletter No. 111, August 7, 1973.

[K4] Kennedy, Ken, "An Algorithm to Compute Compacted Use Definition Chains," SETL Newsletter No. 112, August 14, 1973.

[K5] Kennedy, Ken, "Variable Subsumption with Constant Folding," SETL Newsletter No. 123, February 1, 1974.

[L1] Liu, S-C., "An Experiment with Peephole Optimization in SETL," to appear in the, Proceedings of the September 1976 Moscow Conference on Very High Level Languages.

[S1] Schonberg, Ed, "The VERS2 Language of J. Earley Considerd in Relation to SETL," SETL Newsletter No. 124, January 30, 1974.

[Sch1] Schwartz, J. T., On Programming: An Interim Report on the SETL Project. Installments I, II, Courant Institute of Mathematical Sciences, New York University, New York, 1974.

[Sch2] Schwartz, J. T., "General Comments on High Level Dictions, and Specific Suggestions Concerning 'Converge' Iterators and Some Related Dictions," SETL Newsletter No. 133B, January 29, 1975.

[Sch3] Schwartz, J. T., "Introductory Lecture at the June 28 'Informal Optimization Symposium'," SETL Newsletter No. 135, July 1, 1974.

[Sch4] Schwartz, J. T., "Structureless Programming, or the Notion of 'Rubble', and the Reduction of Programs to Rubble," SETL Newsletter No. 135A, July 12, 1974.

[Sch5] Schwartz, J. T., "A Framework for Certain Kinds of High Level Optimization," SETL Newsletter No. 136, July 16, 1974.

[Sch6] Schwartz, J. T. and Paige, R., "On Jay Earley's 'Method of Iterator Inversion'," SETL Newsletter No. 138, April 19, 1976.

[St1] Standish, Thomas, "An Example of Program Improvement using Source-to-Source Transformations," Dept. of Information and Computer Science, University of California at Irvine, Irvine, Calif., February 11, 1976.

[St2] Standish, Thomas, et al., "The Irvine Program Transformation Catalogue," Dept. of Information and Computer Science, University of California at Irvine, Calif., January 7, 1976.