TRANSFORMATIONAL PROGRAMMING-- APPLICATIONS TO ALGORITHMS AND SYSTEMS[1]

Robert Paige
Rutgers University
New Brunswick, NJ

I. Introduction and Backround

Ten years ago Cheatham and Wegbreit [4] proposed a transformational program development methodology based on notions of top-down stepwise program refinement first expressed by Dijkstra [10] and Wirth [45]. A schema describing the process of this methodology is given in fig. 1. To develop a program by transformation, we first specify the program in as high a level of abstraction and as great a degree of clarity as our programming language admits. This high level problem statement program P is proved correct semimechanically according to some standard approach (see Floyd and Hoare [15, 21]). Next, using an interactive system equipped with a library of encoded transformations, each of which maps a correct program into another equivalent program, we select and apply transformations one at a time to successive versions of the program until we obtain a concrete, low level, efficient implementation version P'.

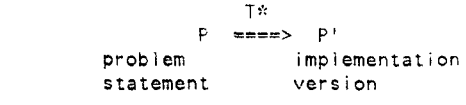

Fig. 1 Transformational Programming Schema

[1] This material is based in part upon work supported by the National Science Foundation under Grant No. MCS7905293. Part of this work was done while the author was visiting Stanford University.

The goals of transformational programming are to reduce programming labor, improve program reliability, and upgrade program performance In order for labor to be reduced, the effort required to obtain P, prove it correct, and derive P' by transformation should be less than the effort required to code P from scratch, and also to debug it Program reliability will be improved if P can be certified correct, and if each transformation preserves program meaning Finally, program performance will be upgraded if transformations are directed towards increased efficiency.

Experimental transformational systems that emphasize one or more aspects of the methodology outlined above have been implemented by Cheatham [5], Darlington [3], Loveman [27], Standish [41], Feather [14] Huet and Lang [11], and others However, all of these systems fall short of the goals, because of a number of reasons that include,

1 inability to mechanize the checking of transformation applicability conditions

2 reliance on large, unmanageable collections of low level transformations, and long arduous derivation sequences

3 dependency on transformations whose potential for improving program performance is unpredictable

4 use of source languages insufficiently high level to accomodate perspicuous initial program specifications and powerful algorithmic transformations

Yet, convincing evidence that this new methodology will succeed has come from recent advances in verification, program transformations, syntax directed editting systems, and high level languages These advances, discussed below, represent partial solutions to the problems stated above, and could eventually be integrated into a single system

1 The transformational approach to verification was pioneered by Gerhart [19] and strengthened by the results of Schwartz [39], Scherlis [36], Broy et al [2], Koenig and Paige [26, 31] Blaustein [1], and others Due mainly to improved technology for the mechanization of proofs of enabling conditions that justify application of transformations, this approach is now at a point where it can be effectively used in a system Such mechanization depends strongly on program analysis, and, in particular, on reanalysis after a program is modified Attribute grammars [24] have been shown to be especially useful in facilitating program analysis [23] Moreover, Reps [34] has discovered an algorithm that reevaluates attributes in optimal time after a program undergoes syntax directed editting changes (as are allowed on the Cornell Synthesizer [43]) He has implemented his algorithm recently, and has reported initial success

2 There are encouraging indications that a transformational system can be made to depend mainly on a small but powerful collection of transformations applied top-down fashion to programs specified at various levels of abstraction from logic down to assembler We envision such a system as a fairly conventional semiautomatic compiler in which classes of transformations are selected semimechanically in a predetermined order, and are justified by predicates supplied mechanically but proved semimanually Of particular importance is nondeterminism removal which as formulated by Sharir [40] could lead to a technique for turning naive, nondeterministic programs into deterministic programs with emergent strategies Such programs could then be transformed automatically by finite differencing [13, 16, 17, 18, 29 30, 31] and jamming [28, 31, 20] (which we have implemented) into programs whose data access paths are fully determined The SETL optimizer could improve these programs further by automatically choosing efficient data structure representations and aggregations

3 Of fundamental importance to the transformations just mentioned is the fact that they can be associated with speedup predictions Fong and Ullman [16] were the first to characterize an important class of algorithmic differencing transformations in terms of accurate asymptotic speedup predictions, e.g., they gave conditions under which repeated calculation of a set former {x in s|k(x)} could be computed in O(#s) + cost(k) steps By considering stronger conditions and special cases for the boolean valued subpart k, Paige [31] later gave sharper speedup predictions (e.g., either O(1) steps for each encounter of the set former or a cumulative cost of O(#s) steps for every encounter) associated with another differencing method Both Morgenstern [28] and Paige [31] prove constant factor improvements due to their jamming transformations (implemented by Morgenstern for the improvement of file processing, and by Paige for the optimization of programs) Constant factor speedup has also been observed for data structure selection by the method of basings but a supporting analytic study has not been presented [8, 37]

4 Essential to the whole transformational process is a wide spectrum programming language (or set of languages) that can express a program at every stage of development from the initial abstract specification down to its concrete implementation realization Since transformations applied to programs written at the highest levels of abstraction are likely to make the most fundamental algorithmic changes, it is important to stress abstract features in our language In addition to supporting transformations, the highest level language dictions should support lucid initial specifications, verification, and even program analysis Of special importance is SETL [38, 9], because its abstract set theoretic dictions can model data structures and

algorithms easily, because its philosophy of avoiding hidden asymptotic costs facilitates program analysis, because its semantics conforms to finite set theory and can accomodate a set theoretic program logic, and because it is wide spectrum. As is evidenced by the work of Schwartz, Fong, Paige, and Sharir, SETL is also a rich medium for transformation.

II. Main Results

The original contributions of our work are listed below:

i. Our main result is the implementation of a prototype transformational programming system that incorporates several of the ideas mentioned above. This system, called RAPTS (Rutgers Abstract Program Transformation System) [32], supports the semiautomatic development of reliable and efficient software using source-to-source program transformations for an abstract variant of the SETL language. Like the prior transformational systems of Cheatham [4], Standish [41], Loveman [27], Darlington [3], and Feather [14], RAPTS has modules to perform parsing, unparsing (i.e., prettyprinting), search, and transformation application; it can manipulate libraries of transformations, source programs,and program development states; RAPTS provides a variety of user aids, also, global control flow and data flow analysis are used to prove the applicability conditions of our transformations automatically. However, our system emphasizes the strict stepwise refinement of programs by successive applications of powerful correctness preserving transformations that can be selected, justified, and applied with a much greater degree of mechanization than other systems.

We have used RAPTS for experimenting with algorithm derivation, system construction, and automated database processing. An important conceptual advantage in using SETL as both system implementation language and source language is that RAPTS can be used to improve itself, as was done for its dead code elimination procedure. In Appendix III we show how an inefficient but clear abstract specification of this procedure is transformed into a lower level SETL variant that runs in linear time with respect to the use-to-def links. Moreover, the transformational approach to verification together with appropriate assertion control could be used within RAPTS to prove itself correct.

ii. RAPTS uses a finite differencing method [33] that generalizes John Cocke's strength reduction [6], and provides an efficient implementation of a host of transformations including Jay Earley's 'iterator inversion' [13]. Our differencing algorithm is an outgrowth of less efficient and less general algorithms due to Cocke, Schwartz, and Kennedy [6, 7]. The reduction in strength algorithms found in [6, 7] execute in O(n) steps (where n is the number of nodes in the flow graph of a program loop) for a single pass. However, their algorithm and also the algorithm used by Paige and Koenig [31] takes O(n**2) steps in the worst case to compute, due to successive linear time passes over programs growing successively larger. Our new algorithm only requires a single pass, and executes in O(n) steps overall. We obtain this improvement by detection of all reducible expressions (that would be detected within multiple passes of the classical algorithms) in advance of any transformational steps. Our algorithm gains greater generality by accepting differencing transformations as input. Based on these differencing rules, we automatically determine categories of variable modifications upon which we can detect expressions amenable to reduction. Thus, differencing transformations can be applied over a wider range of data types than previously possible. However, in this paper we will stress the important application to set theoretic expressions, first observed by Earley [13].

Fong [17] first presented an algorithm to implement a subset of Earley's transformations, and her approach varied from his and Cocke's approach by using a deferred update strategy. She also gained more information by analyzing program paths instead of loops. However, her algorithm ran in time proportional to e log e bit vector operations, where e is the number of edges in the program flow graph. Furthermore, like the classical strength reduction algorithms, her algorithm must be reapplied over programs growing successively larger. (As in the case of the classical algorithms, this problem is due to the fact that reduction of one expression f can make another expression g(f), which depends on f, reducible; as was first noted by Cocke and Schwartz [6] and solved by Cocke and Kennedy [7], reduction of f can also introduce new auxiliary expressions that must be further reduced.) Even the improvement of Fong's algorithm by Tarjan [42] and Rosen [35] to almost linear time in e bit vector operations per pass fails to make it a viable competitor to the classical approach or our new improvement. It remains an interesting open problem whether the path analysis approach introduced by Fong can be modified into a single pass algorithm without losing asymptotic efficiency. We conjecture that this problem can be solved affirmatively. Further comparison of her work with ours can be found in [31].

Like Fong and Ullman [16] our application of set theoretic differencing is based upon reasonable conditions for ensuring asymptotic speedup. Although our current implementation includes about 50 groups of concrete differencing rules [31], recent theoretical improvements provide for a much more compact collection of meta-rules (see Appendix I) that are as easy to specify and implement as our current rules, and even more general than those proposed in [30].

iii. RAPTS can perform new and powerful set expression jamming transformations implemented by a linear time algorithm [30, 31]. A much more powerful algorithm than what is currently implemented and that yields 'optimal expression jamming' is found in [20].

iv. We have designed (but not yet implemented) a new way to mechanically estimate the asymptotic speed of an algorithm derived by transformation within RAPTS. Related to this is the earlier work of Wegbreit, who discussed a transformational system that mechanically analyzed the performance of Lisp programs as they were improved by transformation [44]. Because we are observing programs specified at a higher level of abstraction than Wegbreit, and because our transformations deal with more fundamental algorithmic program improvements, we obtain more global information.

v. We specify our initial abstract program at an unusually high level of abstraction beyond current standard SETL. Illustrations will be included in the next section and Appendix III.

vi. RAPTS incorporates an implementation of a class of abstract static to dynamic expression transformations that generalize Earley's iterator inversion. (See Appendix II for a sampling of these transformations.)

2. RAPTS Illustrations

It is, perhaps, most convenient to explain the transformational capabilities of RAPTS by example, using photo generated excerpts of an actual RAPTS derivation of topological sorting (an example first considered by Earley [13]).

Before proceeding, the reader may find it helpful to consult the brief description of SETL operations and their estimated computational costs (based on obvious hash table implementations for sets and maps) given in table 1.

Operation	Remarks	Estimated Cost
s with:= x	element addition	O(1)
s less:= x	element deletion	O(1)
x in s	set membership	O(1)
s +:= delta	set addition	O(#delta)
s -:= delta	set deletion	O(#delta)
f(x) := y	indexed map assignment	O(1)
f(x1, ..xn)	function retrieval	O(n)
(forall x in s)	forall loop	O(#s * cost(Block))
Block(x)		
end forall		
{x in s | k(x)}	set former	O(#s * cost(k))
exists x in s | k(x)	existential quantifier	O(#s * cost(k))
forall x in s | k(x)	universal quantifier	O(#s * cost(k))
s + t	set union	O(#s + #t)
s * t	set intersection	min(O(#s),O(#t))
s - t	set difference	O(#s)
f[s]	image set	O(#f)

TABLE 1. Complexity Estimates of Setl Operations

Our initial algorithm specification inputs a set s and a set of pairs sp representing an irreflexive transitive predecessor 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: sp maps each element x of s into the set sp{x} of predecessor elements. The algorithm proceeds by repeatedly searching for the minimal elements of the partially ordered set s, adding such elements to the end of t, and then removing them from s. Prettyprinted in RAPTS, our initial program is,

```
    program topsort ;
1     read ( sp ) ;
2     print ( sp ) ;
3     t := [ ] ;
4     s := domain sp + range sp ;
5     ( while exists a in s | ( ( sp { a } ) * s ) = { } )
6       t with := a ;
7       s less := a ;
      end while ;
8     if s = { } then
9       print ( t ) ;
      else
10      print ( 0 ) ;
      end if ;
    end program ;
```

The running time of the initial program is slow, essentially O(#E**2), which is due to repeated search for the minimal elements of s each cycle through the while loop. Speeding up this program entails searching for the minimal elements only once, and maintaining the set of minimal elements by inexpensive 'differential' computations within the while loop as s decreases. This strategy for program improvement is captured by a basic program optimization method we call finite differencing.

In order to facilitate finite differencing, we must first turn the code above into a normal form in which set update operations are implemented in terms of element additions and deletions, set intersections and deletions are rewritten as set formers, etc. For our example, the system will carry out several local transformations (selected from a production system of simple rewrite rules). Application of each transformation is justified by an assertion specified by a SETL predicate. If the system can simplify the predicate to 'true', the transformation is applied automatically. Otherwise, the system asks the user to confirm the partly simplified predicate. (In all of our examples presented here, the system has proved these predicates.)

The essential fragment of the normal form of the topological sorting procedure appears just below.

```
( while exists a in { set141 in s | # { set142 in sp { set141 } |
  set142 in s } = 0 } )
  t with := a ;
  s less := a ;
end while ;
```

Finite differencing will automatically transform the normal form algorithm into an equivalent but more efficient algorithm that uses the speedup strategy stated earlier. In rough terms, differencing will perform the following three steps based on Cocke's reduction in strength schema [6].

i. Just before the while loop, insert code that evaluates the set of minimal elements

```
{set141 in s|#{set142 in sp{ set141 }|set142 in s}= 0}          (1)
```

and stores it into the variable 'minset'. We call this code the *initialization* for minset.

ii. Within the while loop where s is modified, insert code that recalculates minset from its old value so that it always stores the value of the set of minimal elements at the point (line 5) where it is computed. We call the code that updates minset the *difference* of minset with respect to the modification *s less:= a*. When the difference code is executed just prior to the modification, it is called *predifference* code; when it is executed just after the modification, it is called *postdifference* code.

iii. At line 5 replace the minimal set, which is made redundant by steps (i) and (ii), with the variable minset.

For this approach to improve program performance, the overall computational cost of calculating the initialization and difference code in the transformed program must be less than the cost of repeated calculations of the minimal set in the unoptimized program. Our system makes this analysis based on classical code motion assumptions (based on Cocke and Schwartz [6, 31]) and mechanical examination of the minimal set (1) and the while loop within the normal form of our algorithm before differencing is applied. For this example, the system will predict that differencing will yield asymptotic improvement in the cost of computing (1). (Note that Fong and Ullman [16] relied on weaker assumptions; see [31] for a comparison.)

The intuitive ideas behind the analysis are based on a decision procedure for a class of expressions for which the cost of computing difference code relative to certain kinds of parameter modifications is asymptotically less expensive than the cost of full expression evaluations. We say that expressions belonging to this class are *differentiable*. To define the class of differentiable expressions, we first define a finite collection of 'elementary' differentiable expressions and their associated difference code blocks whose computational cost is comparatively small. As is shown in [31], the full class of differentiable expressions is

formed from composition of the elementary expressions and parameter substitution This extended definition is justified by a formal calculus that constructs inexpensive difference code for a nonelementary differentiable expression by combining difference code for the elementary differentiable expressions out of which it is formed

We now apply the preceding analysis to the minimal set (1) using the collection of basic set theoretic differentiable expressions found in Appendix I. Examination of the minimal set calculation detects three potentially differentiable subexpressions,

```
newpred{setl41} = {setl42 in sp {setl41} |setl42 in s }
numpred(setl41) = # newpred {setl41 }
minset = {setl41 in s | numpred(setl41) = 0}
```

that might permit efficient differencing for the minimal set Unfortunately, neither newpred{setl41} nor numpred(setl41) are differentiable, because we cannot form efficient difference code for them relative to the arbitrary modifications in the free variable setl41 that occur within the while loop of the normal form

However, we can overcome this problem using transformations (listed in Appendix II) that handle dynamic expression formation, a generalization of Earley's iterator inversion Application of transformation (4) of Appendix II converts newpred{setl41} into the following differentiable expression

```
newpred = {[x,y] in sp| y in s}
```

which removes the troublesome free variable setl41, and stores values of newpred{setl41} for all relevant values of setl41. Likewise, transformation (26) turns numpred(setl41) into the following expression that can be maintained dynamically at low cost,

```
numpred = {[x, #newpred{x}]: x in domain newpred}
```

Supported by the elementary differentiable expressions, newpred, numpred, and minset, the minimal set (1) is seen to be differentiable, and our system can proceed to carry out the main transformational steps that will speed up the normal form of the toplological sort, i.e.,

i. Store initial values into newpred, numpred, and minset on entry to the while loop

ii Update newpred, numpred, and minset just prior to line 7 where s is modified in order to make the computation of the minimal set at line 5 redundant

Consistent with previous discussion, we refer to the update code involved in task (ii) as the difference of newpred, numpred, and minset with respect to the element deletion **s less:= setl1**, and we form this difference code using a kind of 'chain rule' that combines the separate rules for forming difference code first for newpred, then numpred, and finally minset (i.e., from inner to outer subexpression of the minimal set).

We will illustrate the chain rule by proceeding with this example The predifference of newpred relative to the modification s less:= a is

```
(Forall setl48 in { x in domain sp | a in sp { x } })          (2)
        newpred{setl48} less:= a;
end forall ;
```

Observe that the predifference code (2) contains a costly embedded expression

```
succ{a} =  { x in domain sp | a in sp { x } }
```

that we do not want to compute. However, the system will recognize that this expression can itself be reduced by 'second' differencing. At the same time that the three other differentiable expressions are detected, the system will recognize that differentiation of the dynamic expression

```
succ = {[y,x]: x in domain sp, y in sp{x}}
```

can efficiently eliminate the costly, static expression occurring within the difference code (2).

The predifference code for numpred relative to the change in newpred within (2) is simply

```
numpred(setl48) -:= 1;                                          (3)
```

The final step of the chain rule involves forming the difference of minset relative to modifications in both of its parameters, s and numpred. These predifference blocks are

```
comment:  relative to changes in s                              (4)
      if numpred ( a ) = 0 then
        minset less := a ;
      end if ;
```

and

```
comment: relative to changes in numpred                    (5)
        if set148 in s then
          if numpred ( set148 ) = 0 then
            minset less := set148 ;
          elseif numpred ( set148 ) = 0 + 1 then
            minset with := set148 ;
          end if ;
        end if ;
```

respectively. The chain rule combines the preceding blocks of difference code to form the following collective predifference of newpred.succ, numpred, and minset with respect to s less:= a

```
        ( forall set148 in succ { a } )                    (6)
          if set148 in s then
            if numpred ( set148 ) = 0 then
              minset less := set148 ;
            elseif numpred ( set148 ) = 0 + 1 then
              minset with := set148 ;
            end if ;
          end if ;
          numpred ( set148 ) - := 1 ;
          newpred { set148 } less := a ;
        end forall ;
        if numpred ( a ) = 0 then
          minset less := a ;
        end if ;
```

Analysis of the overall cost of executing the block (6) rests on three easy observations.

i. Based on the monotonically decreasing set s within the while loop, we estimate that (6) is executed O(#s) times, where s is the initial value.

ii. Based on the complexity estimates stated in Table 1, the difference blocks (3), (4), and (5) involve only constant factor costs. Such costs are subsumed by the costs of surrounding code and can be ignored. These examples illustrate the following general property.

Definition An expression E = f(s) is *strongly continuous* with respect to modifications of the form ds to s if the cost of the difference code for E with respect to ds is O(1).

Thus, minset is strongly continuous with respect to indexed assignments to numpred and element deletions to s. Also, numpred is strongly continuous with respect to element additions and deletions to newpred.

iii. Repeated execution of the difference code for newpred.

```
(Forall set148 in succ{a})
        newpred{set148} less:= a;
end forall ;
```

relative to each distinct element 'a' removed from the monotonically decreasing set s, has an overall asymptotic cost no worse than a single calculation of **newpred** = {[x,y] **in sp** | y **in s**} at the initial value of s; i.e., O(#sp). This example illustrates the following general property.

Definition. An expression E = f(s) is *weakly continuous* with respect to modifications ds to s if for every minimal length sequence of operations ds1,ds2,...,dsn (of the form ds) that constructs the final value s2 from the initial value s1, the cumulative cost of all difference code for E with respect to all of the operations ds1,...,dsn is O(max(cost(f(s1)),cost(f(s2))) + n). (Note that all of our speed estimates are based on the heuristics given in Table 1.)

Thus, newpred is weakly continuous with respect to element additions to s.

Based on the preceding analysis, the asymptotic cumulative cost of (6) is estimated at O(#sp) which is an order of magnitude better than the overall cost of the minimal set computation (1) in the normal form algorithm.

Based on Table 1 and the assumption that initialization of newpred, succ, numpred, and minset can be achieved by the straightforward assignments

```
newpred := {[x,y] in sp| y in s};                                    (7)
succ := {[y,x]: x in domain sp, y in sp{x}};
numpred := {[x, #newpred{x}]: x in domain newpred};
minset := {set141 in s | numpred(set141) = 0};
```

we estimate the preprocessing costs to be O(#sp), which justifies our prediction of asymptotic speedup However, we gain a constant factor improvement over this naive initialization by jamming the implicit loops within these set formers [31] The jamming algorithm implemented in RAPTS constructs newpred and numpred in a single loop A deeper investigation of this important transformation and an improved algorithm is found in [20]

The speedup prediction just presented is based on analysis of the normal form algorithm, so that it can be determined whether differentiation is profitable prior to any differencing transformations are applied. By analysis of the normal form, it is sometimes also possible to estimate the asymptotic speed of the transformed algorithm Based on the Table 1 estimates and detection of the presence of monotonic set growth within 'while' loops (which can provide an estimate for the loop repetition frequency), it can be determined that after the minimal set computation is replaced by 'minset', all code other than that which has been introduced by differencing and initialization contributes no more than O(#sp) in overall cost Adding in our estimates for cumulative differencing and initialization costs gives us an overall estimate of O(#sp) in running time for the transformed algorithm.

Further improvement in time and especially space can be realized by performing dead code elimination, which exploits the increase in data independence resulting from differencing and jamming. Based on an algorithm due to Kennedy [23, 31], our dead code elimination procedure detects all assignments to newpred as superfluous The result of this final step is,

```
    program topsort ;
1     read ( sp ) ;
2     print ( sp ) ;
3     t := [ ] ;
4     s := domain sp + range sp ;
5     succ := { } ;
6     ( forall set149 in domain sp , set150 in sp { set149 } )
7       succ { set150 } with := set149 ;
      end forall ;
8     numpred := { } ;
9     ( forall [ set143 , set144 ] in sp )
10      if set144 in s then
11        numpred ( set143 ) + := 1 ;
        end if ;
      end forall ;
12    minset := { } ;
13    ( forall set154 in s )
14      if numpred ( set154 ) = 0 then
15        minset with := set154 ;
        end if ;
      end forall ;
16    ( while exists a in minset )
17      t with := a ;
18      ( forall set148 in succ { a } )
19        if set148 in s then
20          if numpred ( set148 ) = 0 then
21            minset less := set148 ;
            elseif numpred ( set148 ) = 0 + 1 then
22            minset with := set148 ;
            end if ;
          end if ;
23        numpred ( set148 ) - := 1 ;
        end forall ;
24      if numpred ( a ) = 0 then
25        minset less := a ;
        end if ;
26      s less := a ;
      end while ;
27    if s = { } then
28      print ( t ) ;
```

```
      else
29      print ( 0 ) ;
      end if ;
    end program ;
```

Our current implementation outputs the code just above after mechanical application of the preparatory transformations, dynamic expression formation, finite differencing, jamming, and dead code elimination. As is evident from our example, these transformations treated together function primarily to automate the formation of data access paths. Since the length of such paths traversed during execution is strongly related to the asymptotic running time of an algorithm, it is not surprising that finite differencing and its ancillary transformations yield asymptotic speedup. In addition to speedup, the process just illustrated supports verification. The soundness of our transformations, along with a standard correctness proof of the initial abstract algorithm, proves the correctness of the less perspicuous but more efficient equivalent algorithm above.

Further improvement will result from manually initiated assertion propagation and easy syntactic transformations. For example, we can introduce the assertion assert setl44 in s just before line 10 in order to eliminate the extraneous membership test at line 10. It is also worthwhile to place the statements assert setl48 in s prior to line 19, assert numpred(setl48) /= 0 just before 20, and assert numpred(a) = 0 immediately before 24, and then exploit these assertions in obvious ways.

Further automatic improvement by a large constant factor may be achieved by data structure selection and aggregation [8, 37], transformations that should eventually be integrated into RAPTS. However, considerable extensions to the referenced method are needed to obtain the most desirable data structures for our example (see Knuth [25]).

A careful semiautomatic approach to select data structures for topological sorting has been worked out by Katzenelson [22], who used clusters of abstract data types. Katzenelson observed that the most difficult transformational step involves showing how numpred and minset can share the same space. To solve this problem, we use the following transformation which can be justified mainly on syntactic grounds. Since numpred is pointwise monotonically decreasing to 0 within the while loop from lines 16 to 26, and since numpred is only referenced when it is nonzero just before line 22, we can release its space when it goes to 0. Note, however, that when numpred(x) becomes 0 is exactly when minset = {x in s| numpred(x) = 0} is augmented by x. The result of all these transformational steps yields the following data structures:

base(i.e.,universal set) numpred,
domain sp + range sp sp minset

x	pointer to sp(x)	numpred(x) or pointer to next element of minset queue

sp(x) list

pointers to base

It is worthwhile to elaborate on the approach used to estimate asymptotic program performance for the topological sorting example. Most differentiable expressions in Appendix I are either strongly or weakly continuous with respect to element additions or deletions to set or map valued parameters. For example, all elementary differentiable expressions except for (8) and (10) in Appendix I are strongly continuous with respect to the set S, strong continuity is also exhibited by expression (1) with respect to the set Q and by expressions (3), (4), (7), and (10) with respect to the function F. Weak continuity can be observed in Expressions (5) and (6) with respect to the set Q and in expression (10) with respect to S. Note that the expression associated with newpred in the topological sort example is of the form (5) of Appendix I, and exhibits both strong and weak continuity. Some properties of continuity are formalized in the theorem below.

Theorem: i. Strong continuity is closed under arbitrary composition.

ii. Let E = f(s) be weakly continuous with respect to changes ds to s, and let ds1,...,dsn be any minimum length sequence of operations of the form ds that constructs s2 from s1. If all difference code for E with respect to ds1,...,dsn forms a minimum length sequence of operations of the form dE that constructs f(s2)

from f(s1), and if g(E) is weakly continuous with respect to dE, then g(f(s)) is weakly continuous with respect to s. The cumulative cost all the difference code for g(f(s)) is O(max(cost(f(s1)),cost(f(s2)) + max(cost(g(f(s1)),cost(g(f(s2))))).

iii If f(s) is strongly continuous with respect to ds, it is also weakly continuous with respect to ds

For an example of composition of weakly continuous expressions, consider nested image sets. The image set expression E = f[s] is weakly continuous with respect to element additions and deletions to s, and in a program loop where s is monotonically increasing (resp decreasing), so is the value of E after differencing is applied. Thus if we consider the nested expression h[g[f[s]]] in such a loop in which h,g, and f are invariant, the cumulative cost of executing difference code within the loop after it is optimized is estimated to be O(#h+#g+#f).

We have used the preceding mechanical asymptotic time estimates successfully on several algorithms that include finding connected components in a graph, finding the center of a free tree [31], finding all nonterminals that derive the empty string in a context free grammar, computing attribute closure, and partitioning a flow graph into intervals

The implementation of finite differencing within RAPTS uses three main algorithms

i. definition of variable modification categories based on finite differencing rules;

ii detection of variables and differentiable expressions (in a program loop) that fall into the categories defined in step i.

iii finite differencing of the differentiable expressions with respect to the program loop

Details of these algorithms can be found in [33]. Step ii which is logically similar to the procedure presented in [30], passes through a postordering of a parse tree form of the program loop. At each node a value number is computed [6, 12] to determine whether an expression is differentiable, and also whether it has been encountered before The goal of this step is to determine all differentiable expressions. Some of these are detected directly in the loop, while others occur as auxiliary expressions within differencing rules The time complexity is linear in the sum of the parse tree sizes for the loop and the differentiable expressions. Step iii uses two lists produced by step ii a list of places in the loop where each differentiable expression occurs, and a list of places each variable is modified. It rapidly replaces each differentiable expression f by its associated variable name E within the loop by a straightforward bottom up procedure For each modification dx to a variable x on which some differentiable expression depends, the collective pre- and postdifference blocks are formed with respect to dx and inserted around dx The time complexity for this procedure is linear in the sum of the sizes of the inserted difference code and the code which is eliminated as redundant

Conclusion

Interactive syntactic editting systems such as the Cornell Synthesizer have successfully demonstrated a program construction methodology that mitigates compile time error The Synthesizer speeds the process of program construction by dynamically monitoring syntax and, to some extent, semantics while the program is entered interactively. Transformational programming is a proposed methodology that aims to eliminate run time error, so that debugging would be unnecessary It seeks to speed the programming process by interactively monitoring program correctness and efficiency during program construction

RAPTS is a novel implementation of a prototype transformational system that represents a synthesis of old and new ideas. It incorporates new algorithms (its differencing algorithm is an improvement over classical strength reduction used in conventional compiling systems) and new transformations. We have used RAPTS to derive many simple algorithms such as the one just presented For a more complicated example see Appendix III. We have introduced a straightforward mechanism for estimating the speedup that results from finite differencing and the speed of a differentiated algorithm prior to differentiation. Important followup work to this would be to determine conditions under which these initial performance estimates are preserved by a conventional complexity measure after conventional data structures are chosen to implement the sets and maps occurring within the differentiated algorithm.

Appendix I. Differentiable Set Expressions

Listed below is a small, but fairly complete, collection of elementary set-theoretic meta-expressions that can be maintained efficiently by differencing. We assume that each set former in this collection can also be expressed in terms of multi-iterators, which generalize cartesian product, e.g.,

```
{e(X,Y): X in S, Y in T(X) | K(X,Y)}
```

Although the difference rules associated with each elementary expression are not shown, they follow

easily from standard distributive laws. Out of these meta-expressions and corresponding difference rules, the more concrete and numerous elementary expressions and efficient difference rules found in [31] can be derived.

```
1.   S + Q
2.   { X in S | B(X) }
3.   {X in S | F(X) = T} where T is an integer valued
       constant, F is integer valued.
4.   {X in S | F(X) /= T}  where T is an integer valued
       constant, F is integer valued.
5    {X in S | e(X) in Q}
6.   {X in S | e(X) notin Q}
7.   {X in S | F(X) relop R} where F[S] is dense on the interval of integers
               containing the range of R values;  F must be integer valued;
               relop can be any of the comparisons <,>,<=,>=
8    {X in S | F(X) relop R} where F[S] is sparse on the interval of integers
               containing the range of R values, or when F and R can be
               real;  in this case, we also maintain the following two
               auxiliary expressions:
               V = SORTED(F[S]) AND
               K = MIN/{I in [1..#V + 1] | NOT(V(I) < R)}
9.   {e(X): X in S}
10   F[S]
11.  #S
12.  +/S  where + represents arithmetic sum.
```

Appendix II. DYNAMIC EXPRESSION FORMATION

Below we present rules, based on Earley's iterator inversion [13] and Paige's method of discontinuity removal [30], for transforming static set formers and other set theoretic expressions into a form suitable for efficient dynamic modification. Each basic expression f given below depends on free variables q,q1,q2,.. that can undergo such modifications that disallow efficient dynamic maintenance of the value of f. However, f can be profitably maintained dynamically by eliminating its free variables and using a dynamic expression f' associated with f in the table below. Note that f' stores the values of f{q} for all useful instantiations of q.

Static Expression	Dynamic Expression
1. {X in G{q} | B(X)}	{[Y,X] in G | B(X)}
2. {X in S | F(X) = q}	{[F(X),X]: X in S}
3. {X in G{q2} | F(X) = q1}	{[[F(X),Y],X]: [Y,X] in G}
4. {X in G{q} | X in Q}	{[Y,Y] in G | Y in Q}
5. {X in S | X in F{q}}	{[X,Y] in F | Y in S}
6. {X in G{q1} | X in F{q2}}	{[[x,y],z]: [x,z] in G, y in domain F | z in F{y}}
7. {X in G{q} | F(X) in Q}	{[Y,X] in G | F(X) in Q}
8. {X in S | F(X) in H{q}}	{[y,x]: y in domain H, x in domain F | F(x) in H{y}}
9. {X in G{q1} | F(X) in H{q2}}	{[[y,z],x]:[y,x] in G, z in domain H | F(x) in H{y}}
10. {X in G{q} | X notin Q}	{[X,Y] in G | Y notin Q}
11. {X in S | X notin F{q}}	{[y,x]: y in domain F, x in S | x notin F{y}}
12. {X in G{q1} | X notin F{q2}}	{[[y,z],x]: [y,x] in G,z in domain F |x notin F{z}}
13. {X in G{q} | F(X) notin Q}	{[Y,X] in G | F(X) notin Q}
14 {X in S | F(X) notin H{q}}	{[y,x]: y in domain H, x in S | F(x) notin H{y}}
15 {X in G{q1}|F(X) notin H{q2}}	{[[y,z],x]:[y,x] in G,z in domain H|F(x) notin H{z}}
16. {X in S | q in F{X}}	{[Y,X]: X in S, Y in F{X}}
17. {X in G{q2} | q1 in F{X}}	{[[U,X],W] [W,X] in G, U in F{X}}
18. {X in G{q} | X < R}	{[X,Y] in G | Y < R}
19 {X in S | X < F{q}}	{[y,x]: y in domain F, x in S | x < F{y}}
20. {X in G{q1} | X < F{q2}}	{[[y,z],x]: [y,x] in G, z in domain F | x < F{z}}
21. {X in G{q} | F(X) < R}	{[X,Y] in G | F(Y) < R}
22. {X in S | F(X) < H{q}}	{[y,x]: y in domain H, x in S | F(x) < H{y}}
23. {X in G{q1} | F(X) < H{q2}}	{[[y,z],x]. [y,x] in G, z in domain H | F(x) < H{z}}
24. {F(X): X in G{q}}	{[y,F(x)]: [y,x] in G}
25. F[G{q}]	{[y,x]: [y,z] in G, x in F{z}}

```
26.#F{q}                          {[y, #F{y}], y in domain F}
27.+/F{q}                         {[X, +/F{X}], X in DOMAIN F}
where + represents arithmetic sum
```

Appendix III. Differencing Applied to Dead Code Elimination Within RAPTS

1. Below is an initial abstract algorithm specifying a portion of the dead code elimination procedure used within RAPTS. The set crit is the set of critical statements (initially defined to be the print statements of a program). The algorithm works by repeatedly adding to crit the set of instructions that can affect the value of variable uses within crit until crit no longer grows.

```
iuses{q} is the set of variable uses within statement q
usetodef{u} is the set of all variable definitions that can reach
        variable use u
instof(d) is the statement associated with a variable definition d
compound(q) is the compound statement immediately containing statement
        q

    program dead ;
1     read ( instof , usetodef , iuses , compound , crit ) ;
2     ( converge )
3       crit + := ( instof [ usetodef [ iuses [ crit ] ] ] + compound [
          crit ] ) ;
      end ;
4     print ( crit ) ;
    end ;
```

2. It is within the normal form below that 14 differentiable expressions are detected, including 1st and 2nd difference expressions. Analysis determines that the maps instof, usetodef, iuses, and compound are all weakly continuous with respect to element additions in their set valued arguments, that weak continuity is closed for these expressions, and finally, that the cumulative cost of difference code is estimated to be

$$O(\#instof+\#usetodef+\#iuses+\#compound),$$

which is dominated by O(#usetodef). This estimate is the same for initialization costs. It is easy to see that after differencing, the remaining costs are proportional to the sum of the input and output sizes.

```
    program dead ;
1     read ( instof , usetodef , iuses , compound , crit ) ;
2     ( while exists set11 in { set10 in ( instof [ usetodef [ iuses [
        crit ] ] ] + compound [ crit ] ) | set10 notin crit } )
3       crit with := set11 ;
      end while ;
4     print ( crit ) ;
    end ;
```

3. After differencing and dead code elimination, the main loop of the algorithm appears below. Note that 4 out of the 14 differentiable expressions have been eliminated as useless. The passage from step 1 to 3 is done completely automatically within RAPTS.

```
35    ( while exists set11 in newinsts )
36      ( forall set15 in iuses { set11 } | nusepred ( set15 ) = 0 )
37        ( forall set118 in usetodef { set15 } | ndefpred ( set118 ) =
```

```
          0 )
38        ( forall set123 in instof ( set118 ) | ninstpred ( set123 )
            = 0 )
39          if set123 notin comps then
40            if set123 notin crit then
41              newinsts with := set123 ;
              end if ;
42            instpnts with  = set123 ;
            end if ;
43          insts with := set123 ;
          end forall ,
44        ( forall set130 in instof ( set118 ) )
45          ninstpred ( set130 ) + := 1 ;
          end forall ;
        end forall ;
46      ( forall set127 in usetodef ( set15 ) )
47        ndefpred ( set127 ) + := 1 ;
        end forall ,
      end forall ,
48    ( forall set110 in compound ( set11 ) | ncompred ( set110 ) = 0
        )
49      if set110 notin insts then
50        if set110 notin crit then
51          newinsts with  = set110 ;
          end if ;
52        instpnts with  = set110 ;
        end if ;
53      comps with := set110 ;
      end forall ;
54    ( forall set114 in compound ( set11 ) )
55      ncompred ( set114 ) +  = 1 ;
      end forall ;
56    ( forall set139 in iuses ( set11 ) )
57      nusepred ( set139 ) +  = 1 ,
      end forall ;
58    if set11 in instpnts then
59      newinsts less  = set11 ;
      end if ,
60    crit with  = set11 ;
    end while ;
```

References

1. Blaustein, Barbara T. Enforcing Database Assertions Techniques and Applications Tech Rept TR-21-81, Center for Research in Computing Technology, Harvard University, Aug. 1981

2. Broy, M., Partsch, H., Pepper, P., and Wirsing, M "Semantic Relations in Programming Languages' *Information Processing 80* (1980)

3. Burstall, R. M., and Darlington, J "A Transformation System for Developing Recursive Programs ' *JACM 24*, 1 (Jan 1977)

4. Cheatham, T. E., and Wegbreit, Ben. A Laboratory for the Study of Automating Programming Proc AFIPS 1972 Spring Joint Computer Conf., 1972

5. Cheatham, T E, Holloway, G H, Townley, J A Program Refinement by Transformation Proc 5th Int Conf on Software Engineering, Mar, 1981

6. Cocke, John and Schwartz, J T. *Programming Languages and Their Compilers.* CIMS, New York University, 1969

7. Cocke, John and Kennedy, Ken "An Algorithm for Reduction of Operator Strength" *CACM 20*, 11 (Nov 1977)

8. Dewar, Robert B. K., Grand, Arthur, Liu Ssu-Cheng Schwartz, Jacob T.,and Schonberg, Edmond. "Program by Refinement, as Exemplified by the SETL Representation Sublanguage." *TOPLAS 1*, 1 (July 1979).

9. Dewar, Robert. The SETL Programming Language. Manuscript

10. Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, 1976.

11. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B Programming environments based on structured editors: the Mentor Experience Tech. Rept Rapport de Recherche No 26, INRIA, Rocquencourt, France, July, 1980.

12. Downey, Peter, Sethi, Ravi, and Tarjan, Robert. "Variations on the Common Subexpression Problem" *JACM 27*, 4 (Oct 1980).

13. Earley, Jay. 'High Level Iterators and a Method for Automatically Designing Data Structure Representation' *Journal of Computer Languages 1* (1976), 321-342

14. Feather, Martin S *A System for Developing Programs by Transformation*. Ph.D. Th., U. of Edinburgh, 1979

15. Floyd, Robert W. Assigning Meaning to Programs. Proceedings of Symposia on Applied Mathematics Vol XIX, American Mathematics Society, Providence, R. I., 1967.

16. Fong, Amelia C. and Ullman, Jeffrey D. Induction Variables in Very High Level Languages. Proc. Third ACM Symp on Principles of Programming Languages, Jan, 1976

17. Fong, A. C. Elimination of Common Subexpressions in Very High Level Languages. Proc. 4th ACM Symposium on Principles of Programming Languages, Jan, 1977.

18. Fong, A. C. Inductively Computable Constructs in Very High Level Languages Proc 6th ACM Symposium on Principles of Programming Languages, Jan, 1979.

19. Gerhart, S Correctness Preserving Program Transformations. Proc Second ACM Symposium on Principles of Programming Languages, 1975

20. Goldberg, Allen, Paige, Robert. Loop Fusion. unpublished manuscript

21. Hoare C. A. R. "An Axiomatic Basis for Computer Programming" *CACM 12*, 10 (1969) 576 - 581

22. Katzenelson, J. "Clusters and Dialogues for Set Implementations." *IEEE Trans. on Software Engineering SE-5*, 3 (May 1979).

23. Kennedy, Ken. A Survey of Compiler Optimization Techniques. In *Program Flow Analysis*, Muchnick, S., Jones, N., Eds., Prentice Hall, 1981, pp. 5-54.

24. Knuth, D. E. "Semantics of Context-free Languages' *Mathematical Systems Theory 2*, 2 (1968).

25. Knuth, D. E.. *Fundamental Algorithms*. Addison-Wesley, 1968.

26. Koenig, Shaye. A Transformational Framework for Automatic Derived Data Control and Its Applications in an Entity-Relationship Data Model Tech Rept LCSR-TR-23, Rutgers University, Dept of Computer Science. 1981. New Brunswick, N. J

27. Loveman, D. B. "Program Improvement by Source to Source Transformation' *JACM 24*, 1 (Jan 1977).

28. Morgenstern, Matthew. *Automated Design and Optimization of Management Information System Software* Ph.D Th., MIT, Laboratory for Computer Science, Sep 1976.

29. Paige, R., and Schwartz, J. T. Expression Continuity and the Formal Differentiation of Algorithms Proc Fourth ACM Symp. on Principles of Programming Languages, Jan, 1977

30. Paige, Robert *Formal Differentiation*. UMI Research Press, 1981. Revision of Ph.D. thesis, NYU. June 1979

31. Paige, Robert, and Koenig, Shaye. "Finite Differencing of Computable Expressions." *ACM TOPLAS 4*, 3 (July 1982)

32. Paige Robert. RAPTS - The Rutgers Abstract Program Transformation System Compiler Demonstration, Symp on Compiler Construction, Boston

33. Paige, Robert. An Efficient Implementation of Finite Differencing Dept of Computer Science, Rutgers University, Dec, 1982

34. Reps, Thomas Optimal-time Incremental Semantic Analysis for Syntax-directed Editors. Proc. Ninth ACM Symp. on Principles of Programming Languages, Jan, 1982

35. Rosen, B. K. Degrees of Availability In *Program Flow Analysis,* Muchnick, S., Jones, N., Eds., Prentice Hall, 1981, pp 55 - 76.

36. Scherlis, William L Program Improvement by Internal Specialization. 8th POPL, Jan, 1981.

37. Schonberg, Schwartz, and Sharir Automatic Data Structure Selection in SETL. Proc. Sixth ACM Symp on Principles of Programming Languages, Jan, 1979

38. Schwartz, J T.. *On Programming; An Interim Report on the SETL Project, Installments I and II.* CIMS, New York Univ., New York, 1974.

39. Schwartz, J T Correct Program Technology. Tech Rept. Courant Computer Science Report Num. 12. New York University, Dept of Computer Science, Sep, 1977

40. Sharir, M. Some Observations on Formal Differentiation. New York University, Dept of Computer Science, 1980.

41. Standish, Thomas An Example of Program Improvement Using Source-to-Source Transformations Univ. of Cal at Irvine, Dept of Information and Computer Science, Feb, 1976.

42. Tarjan, R. E "A Unified Approach to Path Problems." *JACM 28,* 3 (July 1981)

43. Teitelbaum, T. and Reps, T. "The Cornell Program Synthesizer: a syntax-directed programming environment." *CACM 24,* 9 (Sep 1981)

44. Wegbreit, B "Goal-directed program transformation" *IEEE Trans. Software Engineering SE-2,* 2 (June 1976)

45. Wirth, N "Program Development by Stepwise Refinement" *CACM 14,* 4 (April 1971), 221-227.