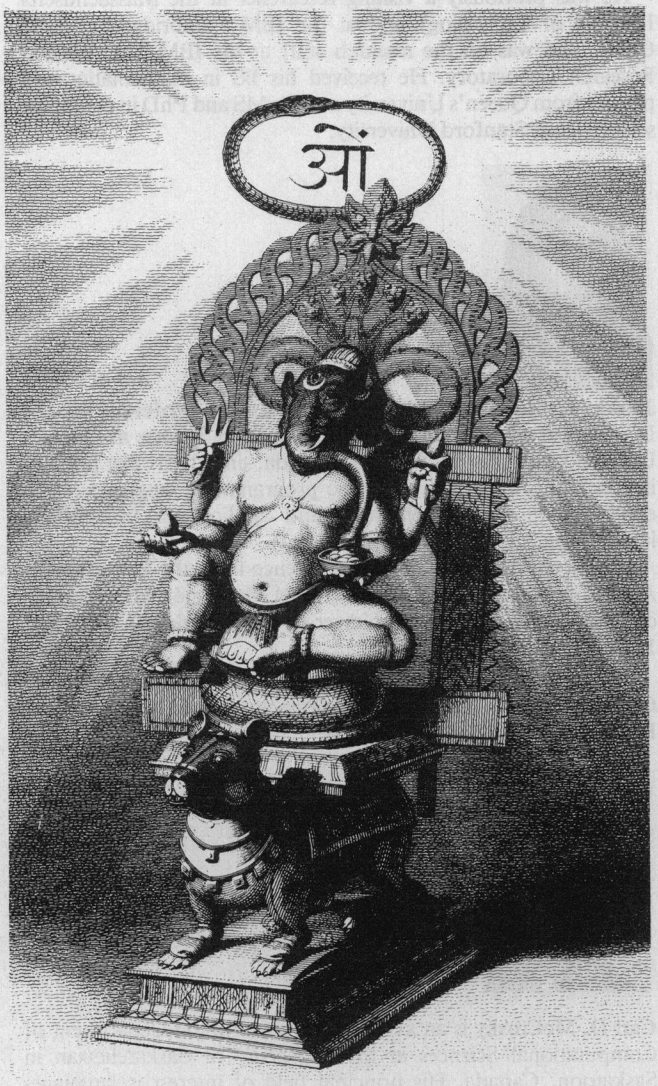
Programming with Invariants

Robert Paige, Rutgers University

The use of a restricted class of invariants as part of a language supports both the accurate synthesis of high-level programs and their translation into efficient implementations.



Ganesa, the Hindu god of prudence and policy, is represented with an elephant's head, an emblem of sagacity. He frequently rides a rat, whose conduct is esteemed for its wisdom and foresight.

Edward Moor, *Hindu Pantheon* (originally published in 1810), Philosophical Research Society, Los Angeles, 1976

Based on efficiency considerations, the most practical programming languages in current use foster programming in which low-level retrieval, modification, and control operations are interspersed. This confusing mixture makes programs hard to construct and difficult to understand, but easy to compile into efficient code. High-level languages such as APL or SETL propose to minimize this problem by making it possible to write programs with fewer and more abstract operations. Functional languages offer another solution: eliminating error-prone modification code so that programs can be written more uniformly in terms of retrieval and control operations. The combination of both these solutions has also been proposed.

However, these remedies sacrifice too much efficiency to make them competitive with languages such as Fortran. Furthermore, these approaches have not demonstrated convincingly that they make correct implementations of large and difficult programs, such as optimizing compilers or distributed operating systems, *substantially* easier to construct or understand.*

We propose another solution that exploits a simple semantic characterization of modification operations (sometimes called imperative code) to (1) reduce the occurrence of modification operations in programs, (2) make programming with modification operations easier, and (3) make programs easier to understand.

This solution is based on the observation that modification operations frequently serve to maintain specialized kinds of invariants that are essential to high program performance. These invariants are typically equalities of the form

(1) $E = f(x_1,\ldots,x_n)$

where f is a computationally expensive n-variate function and E is a variable uniquely associated with the value of f. Invariant (1) can be maintained within a program region B

*In computability theory, the step-counter predicate (which inputs a program P, data I, and count N, and decides whether P halts on input I before N computational steps) is far more compact and understandable in the form of conventional imperative code than as a primitive recursive function specification. Although the first certified Ada compiler was written in SETL, it was highly inefficient, and much more efficient Ada compilers were certified soon after.

A shorter version of this article appears in *Conf. Record HICSS-19,* Hawaii International Conference on System Sciences, January 8-10, Honolulu.

0740-7459/86/0100/0056$01.00 © 1986 IEEE

(where the value of f is required with high frequency) by updating E whenever any of the variables on which f depends is modified in B. Consequently, we can avoid recomputing f each time its value is needed inside B, since its value is kept stored in the variable E.

This approach, which is an elementary kind of finite differencing, can be usefully applied to a rich class of invariants that includes invariant (1) and the following type of existential form:

(2) $\exists y_1 \in f(x_1,\ldots,x_n),\ \exists y_2 \in y_1,\ldots,\ \exists y_k \in y_{k-1} \mid y_k = E$

When finite differencing maintains invariant (2) in a program region B, all occurrences of the indeterminate expression $\exists\exists..\exists f(x_1,\ldots,x_n)$ (involving k applications of the SETL choice operator*) are replaced by occurrences of E.

We prefer to specify both invariants (1) and (2) more uniformly in terms of the reduction notation,

$E \rightarrow f(x_1,\ldots,x_n)$
and
$E \rightarrow \exists\exists\ldots\exists f(x_1,\ldots,x_n)$,

because it actively conveys our particular interest in using invariants for performing substitutions. With respect to the invariant lhs→rhs, we say that rhs is the *replaceable* term, and lhs is the *replacing* term.

The imperative code that maintains these invariants (by updating E) is very difficult to compose and debug because (1) the programmer is not aware of the invariants being implemented, (2) the invariants are too numerous, and (3) the interlocking dependencies among the invariants complicate the code to maintain them.

There is some evidence that a major part of the hand crafting that goes into the construction of high-performance optimizing compilers consists of this very sort of imperative code** (see the useless code elimination algorithm discussed in Goldberg and Paige[1]). This could explain why optimizing compilers are so costly to construct. However, our previous investigations[2] suggest that, while this imperative code itself contains a high level of structural detail and complexity, the process of constructing it could be quite simple. If this hypothesis is correct, then the problem to be coded (perhaps an optimizing compiler) is not so complex, and its implementation should not be so labor-intensive.

The basic idea behind our solution is stated as follows. We regard an efficient program P as arising naturally from a less efficient but more perspicuous program P' by maintaining and exploiting a collection of well-chosen invariants within regions of P. To exploit this idea, we incorporate invariants as a primitive programming language construct with user facilities to define them, bind them to programs, and delineate program regions where they are maintained.

A user-defined invariant encapsulates the way an invariant is established and maintained by itself, separately from the way it depends on other invariants. The more difficult task of generating efficient code to establish and maintain a whole collection of interdependent invariants is done automatically by the system. The implementation is based on the transformational techniques of finite differencing and stream processing.[1,2]

This approach allows the dynamic part of a program to be separated from the functional part in a way that fosters both program comprehension and efficiency. The imperative code to establish and maintain invariants is captured within the invariant definitions. The actual invariants (1 and 2) can be regarded as dynamic functions or relations.

Previously, we reported on more restricted but fully automatic versions of finite differencing[2,3] in which the invariants and the code regions where they are maintained are discovered automatically. The papers just cited illustrate a fully automatic finite differencing applied to such examples as topological sorting, graph reachability, the bankers algorithm, the available expressions problem, useless code elimination, attribute closure, and many more. The semiautomatic application of finite differencing discussed in this article allows invariants to be used in many more contexts.

This article describes and illustrates an extension of SETL[4] with invariants and an implementation method based on a partial difference calculus that improves the calculus described in Paige and Koenig.[2] The material is presented in three main sections. The first section gives the definition and implementation design of a single invariant $E \rightarrow f(x_1,\ldots,x_n)$. The second section presents a chain rule mechanism for maintaining collections of interdependent invariants. The third section explains how to maintain invariants $E \rightarrow f(x_1,\ldots,x_n)$ in which the same variable is substituted for more than one of the parameters $x_1,\ldots,x_n$.

These three sections also illustrate the partial difference calculus with examples that show how invariants can facilitate the use of dynamic data structures in programming, the interfacing of system modules, and the introduction of access paths. Relevant background material on programming methodology, compiler methodology, and related work is presented in the sidebar on pp. 58-61.

The use of invariants described here is a central feature of a prototype transformational programming system called RAPTS that has been operational for more than four years.[5,6]

*If s is a set, the SETL choice operation $\exists s$ yields an arbitrary element chosen from s.

**In combining useless code elimination with four other conventional global optimizations, Jiazhen Cai and I have observed that the code involved in maintaining invariants predominates over all other code.

Individual invariants: Example 1—from selection sort to heap sort

It is often possible to obtain efficient algorithms by using customized dynamic data structures to avoid costly recomputations. Such data structures can be seen as implementations of invariants. For example, consider a heap data structure: a complete binary tree with a key value at each node in which the key value at every nonleaf node x must not exceed the key values of any of the children of x (which implies that the minimum key is stored in the root).

The tree is represented as a vector v in which the parent of each component $v(i)$, $i>1$, is stored in $v(\text{floor}(i/2))$ and the root is stored in $v(1)$. This heap data structure can be used to maintain the minimum value of a set s dynamically relative to additions of new values into s and deletions of the minimum value from s. Heaps illustrate a most basic kind of invariant that can be used to transform a simple selection sort directly into a heap sort.

Selection sort. Consider the following selection sort routine, which sorts a set of numbers q into a tuple t:

```
(3)  read(q);                  $Read a set of numbers q, and
     t:=[ ];                   $initialize t to the empty tuple.
       (while q≠{ })           $While q is nonempty, find its
          t with:=min/q;       $minimum value, add it to the end
          q less:=min/q;       $of t, and delete it from q.
       end while;
     print(t);
```

Because this routine repeatedly searches for the minimum value of q, it has a nonoptimal running time of $O(n^2)$, where n is the initial number of elements in q. But since the search operation $\min/q$ occurs within a region of code in which q is modified only by deletion of its minimum element, we can obtain a speedup by maintaining q as a heap.

Based on this idea, heap sort can be seen as arising directly from selection sort by a transformation that maintains the equality $E=\min/q$ as an invariant at the two program points where $\min/q$ is computed. Such an invariant would make these occurrences of $\min/q$ redundant and replaceable by occurrences of E.

Background

Our treatment of invariants and their implementation by finite differencing unifies two different developments. One development has to do with programming methodology: the informal but essential principles that facilitate the manual construction of programs and the synthesis of algorithms. Another has to do with compiler methodology: the formal techniques of syntactic analysis and symbolic manipulation by which programs can be improved automatically or semiautomatically.

Programming methodology

The development of invariants in programming methodology can be traced back to the 16th century, when Henry Briggs discovered the method of numerical finite differencing for efficiently generating accurate tables of polynomials.[1,2] The maintenance of difference polynomials as invariants was the key idea that allowed Briggs to compute each successive value of a dth degree polynomial with only d additions instead of d additions and d multiplications. Nowadays, it is widely accepted that invariants play a major role in the design of efficient algorithms and software systems (a different use of invariants is central to program verification as shown in Floyd[3] and Hoare[4]).

By placing the essential idea underlying Brigg's algorithm in minimal form and applying the idea in the more general context of program development, Dijkstra became perhaps the most visible advocate of finite differencing.[5] Through numerous examples, Dijkstra distilled a lucid and convincing principle of finite differencing, which he called "avoiding redoing work known to have been done"[5, p.152] and "[trading] variable space for computation speed."[6, p.5] Similar to Dijkstra, Gries[7] and Reynolds[8] presented several compelling examples that illustrate finite differencing as part of a programming methodology.

Compiler methodology

The more formal idea of uncovering invariants by syntactic analysis and maintaining them by an automatic or semiautomatic implementation of finite differencing can also be traced back to Briggs. His method, though applied manually, was an algorithm for speeding up the calculation of successive polynomial values. Babbage's analytic difference engine[1] was an early attempt to mechanize Briggs's technique.

An elementary form of numerical finite differencing proposed by Bauer and Samelson[9] was eventually im-

Transformation. The essential steps of this transformation, which we regard as a kind of finite differencing, are as follows:

- On entry to the while loop, store the set q into the tuple *minheap* that implements a heap. This establishes the invariant

(4) $minheap(1) = \min/q$

on entry to the while loop.*

- Within the while loop immediately after q is diminished, insert the classical siftdown code (see Aho, Hopcroft, and Ullman[7]), which updates minheap so that the new value of q is stored as a heap. This establishes invariant (4) at both points in the selection sort where $\min/q$ occurs.
- Invariant (4), which is maintained by the preceding steps makes both occurrences of $\min/q$ in the selection sort (3) redundant, so they can be replaced by occurrences of $minheap(1)$.

The preceding transformation is based on strength reduction but extends that technique in several essential ways. Classical strength reduction was limited to equalities $E = f$, where f is a numerical expression.[8] Following Earley,[9] we consider expressions f of any data type, including user-defined types. For the $\min/q$ example, q could be any totally ordered set.

This example also illustrates how one invariant can be used to maintain another. That is, invariant (4), which is used directly to speed up the selection sort, is implied by the invariant that keeps q stored as a heap. The heap is called the *defining* invariant, and invariant (4) is called a *replacing* invariant. Also note that, because a heap representation for q is not unique, the heap invariant is actually a membership $minheap \in heaps(q)$, where $heaps(q)$ is the set of all possible heap representations of q.

The preceding transformation can be applied within the RAPTS system by defining the heap invariant, binding it to the selection sort (3), and indicating that the invariant should be maintained and exploited within the selection sort's while loop.

* Assume that both $minheap(1)$ and $\min/q$ have the unique undefined value Ω when $q = \{\}$.

plemented within an Algol compiler. A more general finite differencing transformation, which John Cocke called strength reduction, was later developed as a powerful global compiler optimization by Cocke and Schwartz,[10] Cocke and Kennedy,[11] and Allen, Cocke, and Kennedy.[12]

In the beginning of the 1970s, Jay Earley initiated an exciting investigation to extend, formalize, and automate major aspects of the more general principle of finite differencing used implicitly by algorithm designers and advocated explicitly by Dijkstra. Earley incorporated equality invariants $E = f(x_1,...,x_n)$, which he called intentional variables, as part of his proposed programming language, Vers2.[13] He also suggested a way to maintain such invariants within code regions by a method he expected to arise from his iterator inversion transformation.[14]

Fong and Ullman, and later Fong, developed an interesting method reminiscent of the calculus of variations that was the first implementation of a subset of Earley's set theoretic invariants.[15-17] Fong's implementation has since been improved by Rosen[18] and Tarjan.[19] Paige subsequently developed a general finite difference method capable of implementing invariants for numerical expressions previously handled by strength reduction and for all the expressions investigated by Earley.[20-22]

However, Fong and Paige only discussed implementations for predefined classes of invariants and did not use a notational facility for specifying and applying differencing rules within a programming language. This was initiated by Koenig and Paige,[23] who presented notations for specifying database user views as arbitrary set theoretic invariants that could be maintained across ad hoc database queries and transactions by finite differencing.

Related work

Other related work should be mentioned. Burstall and Darlington have often used their fold and unfold transformations to implement invariants.[24] However, these transformations, which Sintzoff calls transformational gotos,[25] are at a much lower semantic level of abstraction than finite differencing. Abstract data types can also be used to implement invariants, and Pepper actually specified a simple but general finite differencing scheme (but lacking a chain rule) within the CIP language.[26]

Invariant definition. An invariant definition has the following four parts:

- A header for the heap invariant is denoted by

Define Invariant $E \rightarrow heap(s)$

where *heap* identifies the defining invariant to be implemented, s is the set valued argument, and E is a tuple storing a heap data structure for s. Since there can be many heaps for a set s, E must be just one of them.

- The second part of an invariant definition is called the differencing section. This part contains the code that should be executed to restore the invariant (by updating E) when it is falsified at program points where its argument s is modified. This code is called *partial difference code.*

In the case of a heap, we can preserve the heap invariant prior to augmenting s by executing the well-known siftup operations.[7] Because this partial difference code is executed just before the parameter modification s with: = z, it is called *predifference* code, for which we write

$\partial - E < s$ with: $= z > =$ /. . siftup code. . /

The heap invariant can be restored just after the minimum value of s is deleted from s by executing the siftdown routine. Since the siftdown operation should be performed immediately after the modification s less: = min/s, it is called the *postdifference* of E with respect to s less: = min/s, for which we write

$\partial + E < s$ less: = min/s; > = /..siftdown code../

For the assignment s: = { }, which assigns the empty set to s, we can keep s stored as a heap E by setting E to the empty tuple just prior to the modificaton to s. That is,

$\partial - E < s$: = { }; > = E: = [ ];

- Code that establishes the heap invariant on entry to the program region where the difference rule is applied is contained in the initialization section. For this example, the heap can be established easily (though not in the most efficient way) by executing

$\partial - E < s$: = { } >
(for $x \in s$)
  $\partial - E < s$ with : = x >
end;

However, the semantics of abstract data types attempts to reinterpret instead of transform code. This detracts from the very specific nature of what is involved when invariants are maintained and used to improve program performance. Furthermore, abstract data types are more general than partial difference rules, and their specialization for this purpose seems awkward.

A similar but more general use of invariants than that discussed in this article is found in computability theory, where invariants play an important role in the notion of machine (and language) simulation. Simulation led Hoare to develop a formal basis for replacing abstract variables and operations with more efficient concrete variables and operations.[27]

Justification for this replacement is the maintenance of an invariant (called a representation function) between a set of concrete variables and each abstract variable. Gries and Prins[28] recently extended Hoare's idea so that the invariant relationship between abstract and concrete variables could be denoted by arbitrary representation relations. They propose a new language construct called a module that facilitates automatic representation changes. They also discuss some of the issues involved in an implementation.

Schwartz proposed a language feature more general than ours for specifying and applying rules that involve syntactic substitution and insertion operations within program regions.[29] He illustrated this construct with examples much like those first suggested by Earley and sketched an implementation based on a nodal span parsing method. Schwartz showed how his rules could be used to maintain what he calls relationships, which are more general than invariants. Date has elaborated extensively on a general rule system similar to Schwartz's for defining and enforcing database integrity constraints in the presense of updates.[30]

References

1. H. Goldstine, *The Computer from Pascal to Von Neumann*, Princeton University Press, Princeton, N.J., 1972.
2. H. Goldstine, *A History of Numerical Analysis*, Springer-Verlag, New York, 1977.
3. R. Floyd, "Assigning Meaning to Programs," *Proc. Symp. Applied Mathematics*, Vol. 19, American Mathematics Society, Providence, R.I., 1967.
4. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, Vol. 12, No. 10, Oct. 1969, pp. 576-581.
5. E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
6. E.W. Dijkstra, *A Personal Perspective*, Springer-Verlag, New York, 1982.
7. D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.
8. J. Reynolds, *The Craft of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

• Finally, to indicate that all occurrences of min/s should be replaced by $E(1)$ within the region where *heap* is maintained, we specify the replacing invariant as the reduction

$E(1) \rightarrow \min/s$

Note that $E(1)$ is the replacing term and min/s is the replaceable term for the heap invariant.

Heap sort. Once the heap invariant above is defined, it can be used to improve the selection sort (3) by inserting the declaration

Declare Invariant *minheap*$\rightarrow$*heap*(q);

at the top of the sort routine. This declaration binds the new program variable *minheap* to parameter E and the set valued program variable q to parameter s. The syntactic region in which the invariant is preserved is called a *maintain* block, which is delineated by the header "maintain minheap" and the trailer "end maintain."

Given the preceding *heap* definition, the following code transforms a selection sort into a heap sort.

```
Declare Invariant minheap→heap(q);
read(q);
t := [ ];
maintain minheap;
  (while q/ = { })
    t with:= min/q;
    q less:= min/q;
  end while;
end maintain;
print (t);
```

Correctness issues. In general, partial finite difference rules can be used to maintain invariants $E \rightarrow f(x_1, \ldots, x_n)$ within single-entry, single-exit program regions B. Each invariant definition has one defining invariant I and one or more replacing invariants (which could include the defining invariant) implied by I. The set of replacing and replaceable terms for an invariant I consists of the set of replacing and replaceable terms occurring in all of the replacing invariants associated with I.

To understand the differencing method, it is useful to separate the maintain block into the code that establishes a

9. K. Samelson and F. L. Bauer, "Sequential Formula Translation," *Comm. ACM*, Vol. 3, No. 2, Feb. 1960, pp. 76-83.
10. J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, lecture notes, CIMS, New York University, 1969.
11. J. Cocke and K. Kennedy, "An Algorithm for Reduction of Operator Strength," *Comm. ACM*, Vol. 20, No. 11, Nov. 1977, pp. 850-856.
12. F. E. Allen, J. Cocke, and K. Kennedy, "Reduction of Operator Strength," in *Program Flow Analysis*, S. Muchnick and N. Jones, eds., Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 79-101.
13. J. Earley, "High-Level Operations in Automatic Programming," *Proc. Symp. Very High Level Languages*, Vol. 9, No. 4, Apr. 1974.
14. J. Earley, "High-Level Iterators and a Method for Automatically Designing Data Structure Representation," *J. Computer Languages*, Vol. 1, No. 4, 1976, pp. 321-342.
15. A. Fong and J. Ullman, "Induction Variables in Very High Level Languages," *Proc. Third ACM Symp. Princ. Programming Languages*, Jan. 1976, pp. 104-112.
16. A. Fong, "Elimination of Common Subexpressions in Very High Level Languages," *Proc. Fourth ACM Symp. Princ. Programming Languages*, Jan. 1977, pp. 48-57.
17. A. Fong, "Inductively Computable Constructs in Very High Level Languages," *Proc. Sixth ACM Symp. Princ. Programming Languages*, Jan. 1979, pp. 21-28.
18. B.K. Rosen, "Degrees of Availability," in *Program Flow Analysis*, S. Muchnick and N. Jones, eds., Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 55-76.
19. R. Tarjan, "A Unified Approach to Path Problems," *J. ACM*, Vol. 28, No. 3, July 1981, pp. 577-593.
20. R. Paige, *Formal Differentiation*, UMI Research Press, Ann Arbor, Mich., 1981 (revision of PhD dissertation, New York University, June 1979).
21. R. Paige and S. Koenig, "Finite Differencing of Computable Expressions," *ACM TOPLAS*, Vol. 4, No. 3, July 1982, pp. 402-454.
22. R. Paige, "Transformational Programming—Applications to Algorithms and Systems," *Proc. 10th ACM Symp. Princ. Programming Languages*, Jan. 1983, pp. 73-87.
23. S. Koenig and R. Paige, "A Transformational Framework for the Automatic Control of Derived Data," *Proc. Seventh Int'l Conf. Very Large Databases*, Sept. 1981, pp. 306-318.
24. R. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *J. ACM*, Vol. 24, No. 1, Jan. 1977, pp. 44-67.
25. M. Sintzoff, "Understanding and Expressing Software Construction," in *Program Transformation and Programming Environments*, P. Pepper, ed., Springer-Verlag, New York, 1984, pp. 169-180.
26. P. Pepper, private communication, 1982.
27. C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, Vol. 1, No. 19, 1972, pp. 271-281.
28. D. Gries and J. Prins, "A New Notion of Encapsulation," *SIGPLAN Conf. Programming Languages and Environments*, July 1985, pp. 131-139.
29. J.T. Schwartz, "Some Syntactic Suggestions for Transformational Programming," *SETL Newsletter*, No. 205, New York University, Dept. of Computer Science, Mar. 1978.
30. C. Date, *Introduction to Database Systems, Vol. II*, Addison-Wesley, Reading, Mass., 1982.

defining invariant $E \rightarrow f(x_1,...,x_n)$, denoted by establish($E$), and the code that maintains and exploits E within B, denoted by $\partial E<B>$ and called the *differential* of E with respect to B. The differential code block $\partial E<B>$ is formed from B by

- inserting the difference code blocks $\partial - E<dx>$ and $\partial + E<dx>$ immediately before and after each modification dx (occurring in B) to a parameter x on which f depends, and
- for each replacing invariant $h(E) \rightarrow g(x_1,...,x_n)$, substitute $h(E)$ for all occurrences of $g(x_1,...,x_n)$ within B.*

Thus, the maintain block

```
(5) maintain E
      B
    end maintain
```

is equivalent to the code

```
establish(E)
∂E<B>
```

The actual predifference and postdifference code blocks are not unique, but they must obey the following correctness condition. For a defining invariant $E \rightarrow f(x_1,...,x_n)$, if dx_i is a modification to a parameter x_i, $i = 1,...,n$, then the predifference and postdifference code blocks that calculate the new value of E from its old value must satisfy the following general Hoare formula:

$\{E \rightarrow f(x_1,...,x_n)\}$
$\partial - E<dx_i>$
dx_i
$\partial + E<dx_i>$
$\{E \rightarrow f(x_1,...,x_n)\}$

It is also essential to the correctness of this transformation that $f(x_1,...,x_n)$ is well-defined within the region where E is maintained. In other words, the values of the variables $x_1,...,x_n$ must belong to the domain of f. Also, within these difference code blocks, only modifications to E and variables local to these blocks are allowed. As in the heap example, the predifference or postdifference code blocks can be empty.

Implementation issues. For finite differencing to actually improve code the cumulative expense of establishing a defining inveriant $E \rightarrow f$ on entry to a program region B, plus the cost of maintaining it and computing replacing terms (that result from substitutions indicated by the replacing invariants) inside B, must be less than the total cost of computing the replaceable terms within B before optimization.

As in classical strength reduction[8] and the finite differencing techniques of Fong and Ullman[10] and Paige and Koenig,[2] analysis of the control flow properties of B are important in deciding whether program improvement is likely.

*We assume that bound variables occurring within $g(x_1,...,x_n)$ can be renamed appropriately.

Our implementation of maintain blocks supports this goal of program improvement by enforcing three easy conditions:

- The partial difference rules invoked by a maintain block (5) must include difference code with respect to all modifications occurring within the program region B to variables on which the expression f depends.
- If a modification dx to a variable x on which f depends contains an occurrence of any replaceable term, then the predifference code $\partial - E<dx>$ must not modify E.
- There must not be any occurrences of replaceable terms within any of the difference code blocks for E.

The first condition above is a correctness check, while the other two conditions are for improved performance. That is, they ensure that all occurrences of replaceable terms within B are made redundant and that no new occurrences of such terms are introduced within difference code.

Note that the second condition above is satisfied for the min/q example because we use postdifference (and not predifference) code relative to deletions of min/q from q. This choice avoids the need for potentially costly copy operations and makes the occurrence of min/q within the deletion q less:= min/q redundant. Paige and Koenig provided a more theoretical treatment and formal correctness proof of finite differencing.[2]

Collections of invariants: Example 2—forming a bibliography

Systems programs that are designed independently of one another often interact according to some standard pattern

The monkey-god Hanuman served the mighty Indian conqueror, Rama.

Edward Moor,
Hindu Pantheon,
Philosophical Research Society, 1976

file is often passed to a text formatter. However, if the system is unable to anticipate this pattern and the formatter blindly processes an entire tagged text file (even though the file is only edited in a minor way relative to the last time it was processed by the formatter), then potentialy costly redundant computations will be performed.

Partly to overcome this problem, the approach taken by Janus[11] interfaces editing and text processing in the form of an incremental text formatter. Janus keeps the final text file in a form that can be easily updated at low cost whenever slight editing changes are made to the tagged text file.

In this next, more complicated example, the use of invariants and finite differencing are shown to provide a convenient way to implement an interface between editing and formatting for the simple task of bibliography formation.

Abstract code. Suppose that the text formatter constructs a bibliography from file *efile* (generated by the editor) by executing the following code:

(6) bibfrom(sort({x: $x \in$ multiref(*efile*)}))

where multiref(*efile*) yields a multiset of references occurring within *efile* to cited material; the expression

{x: $x \in$ multiref(*efile*)}

forms a set from this multiset; and bibfrom constructs the final bibliography from a sorted list of elements from this set. Note that these operations are easy to understand—but are costly to compute.

Maintaining invariants. To improve the performance of (6), we can maintain a collection of four different interdependent invariants that serve to interface editing and formatting as an efficient incremental activity. These invariants can be partly determined from an inner to outer subexpression analysis of code (6).

The first invariant, *citelist*→*multi*(*efile*), keeps the bibliographic references occurring in *efile* stored within the multiset *citelist*. When *efile* is updated by substring insertion and deletion, the *citelist* invariant can be maintained by performing inexpensive operations that avoid scanning all of *efile*. This greatly speeds up the task of bibliography formation at the expense of a small amount of additional space.

Whenever a string z is added to *efile* by an operation, insert(*efile*,z), we can locate all the citations within z and add them one by one to *citelist*. If a constant number of such insertions and similar deletions (where each insertion and deletion of a single citation takes unit time to compute) can be expected to occur between consecutive formatter runs, then an order of magnitude improvement in running time will result.

The following invariant definition and declaration will achieve this goal.

```
(7) define invariant E→multi(str)
      ∂−E<insert(str,z)> =   (for x ∈ multiref(z))
                               E with:= x;
                             end for;
      . . .
      E→multiref(str)
    end
and
    declare citelist → multi(efile);
```

After *citelist* is maintained, it is important to avoid the costly computation {x: $x \in$ *citelist*}, which computes the set of references occurring in *citelist*. The value of this set expression can also be preserved using the invariant definition and declaration given below.

```
  define invariant E→setfromtuple(q)
    ∂−E<q with:= z> =   if #[x ∈ q | x = z] = 0 then
                          E with:= z;
                        end if;
    . . .
    E→{x: x ∈ q}
  end
and
  declare citeset→setfromtuple(citelist);
```

Maintaining the two different invariants, *citelist* and *citeset*, within a sequence B of updates to *efile* raises the problem of how to combine the difference code for *citelist* and *citeset* so that both invariants are maintained correctly and efficiently. It is always correct to maintain the innermost invariant *citelist* first, then to maintain *citeset*; that is,

(8) ∂ *citeset*< ∂ *citelist*<B>>

In particular, if B were just the simple update, insert(*efile*, z), then code (8) would yield

```
(9) (for x ∈ multiref(z))
      if#[y ∈ citelist | y = x] = 0 then
        citeset with:= x;
      end if;
      citelist with := x;
    end for;
    insert (efile,z);
```

The ordering above is also the only correct ordering because variable *citelist*, upon which *citeset* depends, is undefined within B. That is, the transformation

∂ *citeset*<B>

cannot be correct.

Continuing with this example, we note that the difference code for *citeset* occurring within code block (9) is inefficient, because of the costly embedded calculation

(10) #[$y \in$ *citelist* | $y = x$]

which computes the number of references to x occurring in the multiset *citelist*. This problem can be overcome, however, by maintaining counts of all occurrences of each different citation within *citelist*. A single invariant definition and a declaration for maintaining all these counts as invari-

ants is given below:

```
(11)    define invariant E(w)→count(t,w)
          ∂−E<t with:= z>=E(z) +:= 1;
          . . .
          E(w)→#[x ∈ t | x=w]
        end
and
        declare citecount(x)→count(citelist,x);
```

Note that for this example, the parameter E and the program variable *citecount* are specified in a pointwise function format. The significance of this is that x is a pattern that matches any citation. The replacing invariant within (11) indicates that occurrences of *citecount*(x) will replace occurrences of expression (10) for any value of x. Of course, for this to work, the way in which *citecount* is maintained must be correctly integrated with the other two invariants *citelist* and *citeset*.

Because *citecount* depends on *citelist*, the code to maintain invariant *citelist* must be inserted first. Because expression (10), which occurs within the predifference code for *citeset*, is involved in the old value of *citelist* before it is modified, expression (10) would be made redundant by the old value of *citecount* before the predifference code for *citecount* is executed. Consequently, in order to avoid making any copies of *efile*, *citelist*, *citeset*, and *citecount*, it is necessary to insert the code that maintains the *citecount* invariant after inserting the code for maintaining *citeset* and *citelist*; that is,

$$\partial\ citecount < \partial\ citeset < \partial\ citelist < \text{insert}(efile,z) > > >,$$

which generates the following improved code

```
(for x ∈ multiref(z))
  if citecount(x) = 0 then
    citeset with:= x;
  end if;
  citecount(x) +:= 1;
  citelist with:= x;
end for;
insert (efile,z);
```

One more improvement to the bibliography code (6) is to eliminate the expensive operation sort(*citeset*), which sorts all citations occurring in the set *citeset*. This is achieved by storing all these citations within a search tree. The following invariant definition and declaration introduces this crucial data structure

```
    define invariant E→searchtree(s)
      ∂−E<s with:= z>:  insertst(E,z);
      . . .
      inorder(E)→sort(s)
    end
and
    declare sorted→searchtree(citeset);
```

Based on the preceding rules for ordering the way invariants are maintained, it would be correct to maintain *sorted* either after *citeset* or after *citecount*. Assuming that *sorted* is maintained after *citecount*, the collective predifference code for *citelist*, *citeset*, *citecount*, and *sorted*, relative to the modification insert(*efile*,z),is

```
(12)  (for x ∈ multiref(z))
        if citecount(x) = 0 then
          insertst(sorted,x);
          citeset with:= x;
        end if;
        citecount(x) +:= 1;
        citelist with:= x;
      end for;
$the modification insert(efile,z) would appear here$
```

The collective predifference code relative to delete(*efile*,z) is similar. Maintenance of these four invariants allows us to replace the costly bibliography code (6) with the following more efficient code,

```
bibfrom(inorder(sorted));
```

Since *sorted* represents an invariant that is essential to the task of bibliography formation, it is worth examining the two possible predifference code blocks (code block (12) and the predifference code relative to deletions) to determine if we can avoid maintaining any of the other three invariants and still maintain *sorted*. A simple useless code elimination procedure determines that only *citecount*, the set of reference counts, is necessary—*citelist* and *citeset* need not be maintained, and the code to maintain them can be safely removed from the predifference code (12).

The preceding example illustrates a few basic principles of finite differencing that allow us to automate the potentially tedious task of ordering the way a collection of invariants should be maintained in a single-entry, single-exit program region B. The chain rule and method discussed below improve the treatment of finite differencing found in Paige and Koenig.[2]

Maintaining invariants.

Definition (*chain rule*). Let $E_i \rightarrow f_i$, $i=1,..,n$, be n defining invariants in which each expression f_i, $i=1,..,n$, depends only on variables $v_1,...,v_k$, $E_1,...,E_{i-1}$. The *collective differential* of $E_1,...,E_n$ with respect to a program region B, denoted by $\partial\{E_1,...,E_n\}<B>$, is a new code block formed from B by replacing each modification dx occurring in B with the differential code $\partial\{E_1,...,E_n\}<dx>$.

We can reduce the differential $\partial\{E_1,...,E_n\}<dx>$ to the simpler differential

$$(13)\quad \partial\{E_2,...,E_n\}<\partial - E_1<dx>\ dx\ \partial + E_1<dx>>$$

if none of the difference code for $E_2,...,E_n$ inserted into (13) introduces any occurrences of replaceable terms associated with E_1. If the simplification to (13) can occur, we refer to E_1 as a *minimal invariant* for the differential $\partial\{E_1,...,E_n\}<dx>$. This simplification rule leads to the following general chain rules defining predif-

ference and postdifference code blocks for collections of invariants:

$$\partial - \{E_1,\ldots,E_n\} < dx > = \partial\{E_2,\ldots,E_n\} < \partial - E_1 < dx > > \\ \partial - \{E_2,\ldots,E_n\} < dx >$$

$$\partial + \{E_1,\ldots,E_n\} < dx > = \partial + \{E_2,\ldots,E_n\} < dx > \\ \partial\{E_2,\ldots,E_n\} < \partial - E_1 < dx > >$$

The reduction of the collective differential

$$\partial\{E_1,\ldots,E_n\} < dx >$$

depends on finding minimal invariants, which can be done by a simple graph theoretic analysis. Consider a directed acyclic graph (called a dag) whose nodes are labeled with variables $v_1,\ldots,v_k,E_1,\ldots,E_n$. We draw an edge from a node labeled E_i to a node labeled y if the variable y occurs in expression f_i. We call this dag the *data dependency dag* for $f_1,\ldots,f_n$.

Next, for each predecessor node (labeled E, say) of the node labeled x, we determine the difference code for E relative to the modification dx. We successively determine the difference code for each invariant E' relative to each of the modifications previously determined for the successor nodes of E' until the entire dag is processed.

After this, we augment the data dependency dag with an arc from a node labeled E' to a node labled E'' if any part of the difference code just determined for E'' contains an occurrence of a replaceable term g associated with E'. (When this is the case, we say that g is an auxiliary expression for E''.) Consequently, we know that E is a minimal invariant for $\partial\{E_1,\ldots,E_n\} < dx >$ if all its successor nodes in the augmented data dependency dag are either the node labeled x or nodes that have no paths to x.

To see how the preceding analysis applies to the bibliography example, consider the collective differential

(14) $\partial\{sorted, citeset, citelist, citecount\} <$ insert(*efile*,*z*)>

The augmented data dependency dag for (14) is shown in Figure 1. It is easy to see that *citelist* is a minimal invariant. In fact for this case we can simply sort the dag topologically and obtain a correct ordering of invariants; that is,

∂ *citecount* < ∂ *sorted* < ∂ *citeset* < ∂ *citelist* < insert(*efile*,*z*) > > > >

Establishing invariants. Although it is possible to establish a collection of invariants in a straightforward way using the initialization section of the invariant definition as in the heap sort example, we can produce much better code based on an improved version of the stream processing transformation found in Goldberg and Paige[1] and Paige and Koenig.[2] Stream processing is a technique that improves the time and space requirements of several iterative expressions by calculating them in a single pass and by avoiding intermediate calculations.

The method of stream processing discussed by Goldberg and Paige depends on a few preliminary notions. If we can

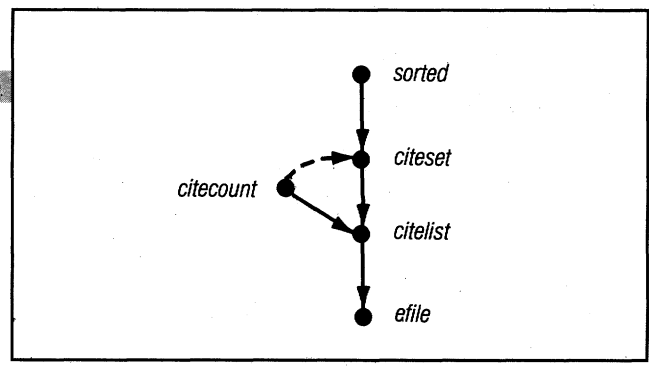


Figure 1. Augmented dependency dag (broken edge results from augmenting the data dependency dag).

efficiently evaluate an expression $E = f(x_1,\ldots,x_n)$ by searching through a set or tuple valued parameter x_i, then x_i is called a *stream parameter* for f. For example, *efile* is a stream parameter for *citelist*, because we can compute *citelist* efficiently by searching through *efile*. That is,

```
∂ – citelist < efile : = ” >
(for line = newline(efile))                $extract a new line starting
   ∂ – citelist < insert(efile,line) >     $from the left of efile
end for;
```

Also, since *citelist* is a stream parameter for *citeset*, both of the invariants *citelist* and *citeset* can be established together in the same loop using a loop combining technique we call *vertical fusion*:

```
(15) ∂ citeset < ∂ – citelist < efile : = ” > >
     (for line = newline (efile))
        ∂ < citeset < ∂ – citelist < insert(efile,line) > >
     end for;
```

As was pointed out earlier, the difference code used to construct *citeset* within loop (15) contains a costly embedded expression that does not have to be computed if *citecount* is kept invariant at the appropriate point. Since *citelist* is a stream parameter for both *citecount* and *citeset*, *citecount* can be constructed at the same time as *citeset* by another loop-combining technique called *horizontal fusion*. Although this kind of fusion technique would ordinarily allow *citecount* and *citelist* to be constructed in an arbitrary order, in this case *citecount* must come first, because *citecount* is an auxiliary expression for *citelist*. That is,

```
∂ citecount < ∂ citeset < ∂ – citelist < efile : = ” > > >
(for line = newline(efile))
   ∂ citecount < ∂ citeset < ∂ – citelist < insert(efile,line) > > >
end for;
```

Finally, since *citeset* is a stream parameter for *sorted*, all four invariants can be computed efficiently using a single search through *efile*. That is,

```
∂ sorted < ∂ citecount < ∂ citeset < ∂ –
   citelist < efile : = ” > > > >
(for line = newline(efile))
   ∂ sorted < ∂ citecount < ∂ citeset < ∂ – citelist < insert(efile,
      line) > > > >
end for;
```

We indicate the stream parameters for an invariant within the initialization part of the invariant definition. For example, initialization for the multi-invariant (7) would contain

Stream processing implementation

In general, the code to implement

establish $\{E_1,...,E_n\}$

is produced with the following logic. Given a dependency dag (directed acyclic graph) for $E_1,...,E_n$ and the designated stream edges (edges leading to stream parameters) for $E_1,...,E_n$, we want to establish these n invariants with a minimal number of loops. Let D be the set of edges in the dependency dag. First we find the smallest sequence of trees $t_1,...,t_k$ (with edges leading toward the root) satisfying the following four conditions:

- F is a subset of the stream edges in D such that for each E_i, $i=1,...,n$, only one edge leading out of E_i in the dependency dag is included in F.
- $\{t_1,...,t_k\}$ partitions F.
- There are no edges in $D-t_j$, $j=1,...,k$, leading from an internal vertex of t_i to an internal vertex of t_j where $i \le j$.
- Each edge $[E_1,E_2]$ occurring in the augmented dependency graph but not in the original dependency graph indicates that E_1 is an auxiliary expression for E_2. If the two stream edges in F leading away from E_1 and E_2 lead to the same vertex, then these stream edges must belong to the same tree; otherwise the stream edge for E_1 must belong to a tree scheduled before the tree that contains the stream edge for E_2.

The last condition above is new and extends the method of Goldberg and Paige[1] to handle auxiliary expressions.

Let $t_1,...,t_k$ be a minimal sequence of trees satisfying the conditions just above. Each tree t_i represents a loop, L_i, that establishes all of the invariants associated with the nonroot vertices of t_i, $i=1,...,k$. The sequence of loops, $L_1,...,L_k$, is the code that establishes all the invariants $E_1,...,E_n$.

We generate each loop L_i, $i=1,...k$, in the following way: Suppose that t_i contains the nonroot vertices labeled $E_1,...,E_m$. Suppose also that variable S labels the root. Finally, suppose that S is initialized by the instruction start(S) (for example, *efile* := " for the bibliography example), searched through by the generator for x = next(S) (an example is (for *line* = newline(*efile*))), and augmented with the new element x by the instruction augment(S,x) (for example, insert(*efile,line*)). Then L_i is produced by the following code:

```
(1)  ∂−{E1,...Em}<start(S)>
       (for x = next(S))
         ∂−{E1,...,Em}<augment(S,x)>
       end for;
```

Note that the procedure (1), which translates trees to loops, provides a new unified treatment of finite differencing and stream processing. It is intriguing to consider Steven D. Johnson's observation[2] that the stream processing technique of Friedman and Wise[3,4] can be regarded as a restricted form of our technique (even though their technique is tailored to infinite streams while ours is fashioned around finite streams).

The reason this makes sense is that stream processing is a scheduling problem for searching through streams. Scheduling searches through infinite streams is simplified by the fact that these searches begin but never end. Finite streams are more complicated, because whenever a search through a stream completes, we need to choose another stream to be searched.

References

1. A. Goldberg and R. Paige, "Stream Processing," *ACM Symp. Lisp and Functional Programming*, Aug. 1984, pp. 53-62.
2. S.D. Johnson, private communication, 1984.
3. D. Friedman and D. Wise, "CONS Should Not Evaluate Its Arguments," in *Automata, Languages, and Programming*, S. Michaelson and R. Milner, eds., Edinburgh University Press, Edinburgh, Scotland, 1976, pp. 257-284.
4. D. Friedman and D. Wise, "Aspects of Applicative Programming for File Systems," *Proc. ACM Conf. Language Design for Reliable Software*, Mar. 1977, pp. 41-55.


```
stream: str;  ∂−E<str := "">
              (for line = newline(str))
                ∂−E<insert(str,line)>
              end for;
```

The collective establish operator, denoted by establish$\{E_1,...,E_n\}$, yields code that establishes the n invariants $E_i = f_i$, $i=1,...,n$. A general stream processing method for implementing efficient code to establish these invariants is described in the box above.

Based on the collective differential and establish operators, we can define collective maintain blocks with respect to single-entry, single-exit program regions B. The collective maintain block is denoted

```
maintain {E1,...,En}
  B
end maintain
```

and means the same as

```
establish {E1,...,En}
∂{E1,...,En}<B>
```

Parameterized invariants: Example 3—finding the largest orbit

The computable partial difference calculus just described can be used to maintain invariants of the form $E \rightarrow f(x_1,x_2,...,x_n)$, where the variables $x_1,x_2,...,x_n$ are distinct identifiers. An interesting question arises when more than one of these variables have the same identifier. In the case of classical strength reduction, where only a limited number of arithmetic operations are considered, the approach is to treat expressions with multiple occurrences of the same variable using different rules. Thus, for products, separate difference rules are defined for both $i*j$ and $i*i$.

For the more general finite differencing framework considered here, the standard strength reduction approach could require 2^n different invariants to cover all the ways in which one variable can have multiple occurrences in $f(x_1,...,x_n)$. In this section, we show that partial difference rules are complete in the sense that only n different partial

difference rules are needed to handle all of these 2^n possible cases.

Suppose we want to maintain the invariant $E \rightarrow h(x,f(x))$, which involves two occurrences of x. This is easily handled by maintaining two invariants, $E_1 \rightarrow f(x)$ and $E \rightarrow h(x,E_1)$, and by using the chain rule to combine partial difference rules for f and h in the standard way. So if dx were a modification to x, then the differential $\partial\{E,E_1\}<dx>$ would be

$\partial E<\partial - E_1<dx>>$
$\partial - E<dx>$
dx
$\partial + E<dx>$
$\partial E<\partial + E_1<dx>>$

Handling invariants such as $E \rightarrow f(x,x)$ can be reduced to the preceding case by maintaining a copy of x; that is, by maintaining the invariants $E_1 \rightarrow x$ and $E \rightarrow f(x,E_1)$. This approach wastes space by storing an extra copy of x, but it is efficient in terms of time. (A copy is maintained, but a copy operation is not performed except to establish E_1.) Unfortunately, this technique could prove too costly in space usage for the case of $E \rightarrow f(x,\ldots,x,x_{m+1},x_n)$, where $m-1$ extra copies of x are maintained.

The box at right describes a way to avoid all but one extra copy of x. Although maintaining an extra copy of an argument for expressions with multiple argument occurrences is significantly better than the first naive approach, the need to conserve space is important enough to consider an alternative technique that can sometimes avoid keeping any additional copies at all.

Improved method, an example. But before describing this approach in its full generality, it is useful to illustrate the ideas with a simple example. Consider the problem that inputs an arbitrary finite function f and a finite set q and outputs the largest subset s of q for which the restricted function $f|_s$ is a total function that maps s into itself. This problem, which we call the largest orbit problem, can be formally specified in terms of the following deterministic selection expression:

(16) the $s \subset q \cap \text{domain } f \mid f[s] \subset s$ maximizing s,

which yields the unique largest subset s of $q \cap \text{domain } f$ satisfying $f[s] \subset s$.

To implement the preceding problem specification, we first turn the predicate $f[s] \subset s$ into the equivalent form $s \subset f^{-1}[s]$, then into $s - f^{-1}[s] = \{\}$, and finally into

(17) $s - \{x \in s \mid f(x) \notin s\} = s$

Because the predicate $f(x) \notin s$ appearing in expression (17) is antimonotone in s (that is, the predicate cannot change from false to true when s is augmented), the left-hand-side expression in (17) is monotone. Consequently, we can use a classical fixed point argument to implement specification (16) with the following code:

(18) $s := q \cap \text{domain} f$:
(converge) \$repeat until s is unchanged
s less:= $\ni\{x \in s \mid f(x) \notin s\}$; \$pick any elemen
end converge;

Code (18) uses a naive negative workset strategy that starts with an approximation that includes the solution and then repeatedly whittles away at this and successive approximations until the solution is reached. Its running time, however, is too slow—essentially $O(\#q + \#f^2)$. It can be speeded up to run in $O(\#q + \#f)$ by maintaining the following two invariants:

A single-copy method for parameterized invariants

It is always possible to maintain invariants of the form $E \rightarrow f(x,\ldots x,x_{m+1},\ldots,x_n)$ using only one extra copy of x. Let the two copies be denoted x and x'. In deriving this technique, it is useful to consider the invariant $E' \rightarrow f(x_1,\ldots,x_m,x_{m+1},\ldots,x_n)$ formed from E by replacing the m occurrences of x by m distinct identifiers $x_1,\ldots,x_m$. Suppose that each variable $x_1,\ldots,x_m,x'$ has the same value as x, *xold*, at a program point p where x is modified by a change dx. Suppose also that just after p each of these variables is modified in the same way as x (by $dx_1,\ldots,dx_m,dx'$), so they all have the same value as the new value of x, *xnew*. Then E' can be maintained through this sequence of modifications by executing the following code:

(1)p: dx
$\partial - E'<dx_1>$
dx_1
$\partial + E'<dx_1>$
$\partial - E'<dx_2>$
dx_2
$\partial + E'<dx_2>$
...
$\partial - E'<dx_m>$
dx_m
$\partial + E'<dx_m>$
dx'

Since the values of $x,x_1,\ldots,x_m$ are equal to *xold* immediately before and to *xnew* just after code block (1) is executed, the invariant $E' \rightarrow f(x,\ldots,x,x_{m+1},\ldots x_n)$ holds both before and after code block (1) also. Observe that after the modification dx_1 in code block (1) all occurrences of variable x_1 within the predifference and postdifference code blocks represent the new value of x, *xnew*. More generally, after each modification dx_i, $i = 1,\ldots,m$, all occurrences of x_i within predifference and postdifference code blocks represent *xnew* also. Furthermore, prior to each modification dx_i in code block (1), $i = 1,\ldots,m$, all occurrences of the variable x_i within predifference and postdifference code blocks represent the old value of x, *xold*.

To transform the code (1) into difference code for E, we take the following steps: Within the difference code in code (1), replace all variable occurrences that represent *xold* by x', and replace all occurrences that represent *xnew* by x. These substitutions remove all occurrences of $x_1,\ldots,x_m$ from the difference code for E' without altering the semantics of code (1). Finally, this code can be transformed into difference code for E by replacing all occurrences of E' by E and eliminating all the modifications $dx_1,\ldots,dx_m$.

$E_1 \rightarrow \{x \in s \mid f(x) \notin s\}$
$E_2 \rightarrow \{[f(x),x] : x \in s\}$

where E_2 is an access path (or index) into the set s that helps realize efficient difference code for E_1.

Observe that invariant E_1 has two occurrences of s. Suppose that our library of invariant definitions contains E_2 and the following invariant

$E_3 \rightarrow \{x \in s_1 \mid f(x) \notin s_2\}$

but not E_1. We show how to derive a correct invariant definition of E_1 from E_3.

Let the differencing rules for E_3 and E_2 be given as follows:

```
∂−E3<s1 less:= z>=  if f(x) ∉ s2 then
                      E3 less:= z;
                    end if;
∂−E3<s2 less:= z>=  (for x ∈ {u ∈ s1 | f(u)=z})
                      E3 with:= x;
                    end for;
∂−E2<s1 less:= z>=  E2 less:= [f(z),z];
```

By the chain rule, the collective partial differencing code for E_2 and E_3 is given by

```
(19) ∂−{E2,E3}<s1 less:= z>=  if f(x) ∉ s2 then
                                E3 less:= z;
                              end if;
                              E2 less:= [f(z),z];
     ∂−{E2,E3}<s2 less:= z>=  (for x ∈ E2{z})
                                E3 with:= x;
                              end for;
```

To turn difference rules (19) into rules for E_1 and E_2 we assume that s_1 and s_2 initially have the same values and are then each modified by deletion of z. If s_1 is modified first, then $\partial - \{E_2,E_3\} < s_1$ less:= $z>$ references s_2, which has the old value of s_1. Consequently, the code for $\partial\{E_1,E_2\} < s$ less:= $z>$ can be correctly derived from rules (19) by replacing all occurrences of s_2 and E_3 within (19) by s and E_1, respectively.

Let the term that results from the parallel substitution of all free occurrences of the variables $x_1,...,x_n$ by the terms $t_1,...,t_n$ within the term t be denoted by

$[t]\, x_1,...,x_n \backslash t_1,...,t_n$

Then a correct rule for $\partial\{E_1,E_2\} < s$ less:= $z>$ is

```
[∂−{E2,E3}<s1 less:= z>] E3,s2\E1,s
  $the old value of s is referenced
s less:= z;
[∂−{E2,E3}<s2 less:= z>] E3\E1
```

By changing our assumption so that s_2 is modified first, we can derive the following, different rule:

```
[∂−{E2,E3}<s2 less:= z>] E3\E1
s less:= z;
[∂−{E2,E3}<s1 less:= z>] E3,s2\E1,s
  $the new value of s is referenced
```

Thus, to speed up code (18) we need only specify the following maintain block:

```
(20) maintain {E1,E2}
       s:= {};
       (for x ∈ domain f | x ∈ q)
         s with:= x;
       end for;
       (while ∃ z ∈ {x ∈ s | f(x) ∉ s})
         s less:= z;
       end while;
     end maintain;
```

Note that the SETL optimizer can improve code (20) still further by implementing set- and map-valued variables with conventional data structures.[12] However, by making use of invariants to specify and generate data structures (as we did in the heap example discussed earlier), we can obtain an even better orbit program that runs in $O(\#f+\#q)$ time. More generally, our notion of invariant can be used conveniently to reformulate the SETL data structure selection and aggregation transformations.

Maintaining parameterized invariants using no copies

It is often possible to maintain invariants of the form $E \rightarrow f(x,...,x,x_{m+1}...x_n)$ using no extra copies of x. To determine the difference code for E relative to a modification dx to x, it is useful to consider the invariant $E' \rightarrow f(x_1,...,x_m,x_{m+1},...x_n)$ formed from f by rewriting the m occurrences of x by m distinct identifiers.

Suppose that the postdifference code for E' relative to the change dx_i, $i = 1,...,m$, is empty. Assume that the initial value of each of the variables $x_1,...,x_m$ is the old value of x, *xold*, and let each of these variables in turn be modified in the same way (so that their final values are all equal to the new value of x, *xnew*).

Then all occurrences of $x_i,...x_m$ within $\partial E' < dx_i >$ represent the old value of x, *xold*, and all occurrences of $x_1,...,x_{i-1}$ within $\partial E' < dx_i >$ represent *xnew*, $i = 1,...,m$. If there exists some m-permutation π and some number $L = 1,...,m$ such that $\partial E' < dx_{\pi(i)} >$ references only *xold*, $i = 1,...,L$, and $\partial E' < dx_{\pi(i)} >$ references only *xnew*, $i = L+1,...,m$, then the following code realizes $\partial E < dx >$:

```
[∂−E'<dx_π(1)>] x1,...,xn,E'\x,...,x,E   $references xold
.
.
[∂−E'<dx_π(L)>] x1,...,xn,E'\x,...,x,E   $references xold
dx
[∂−E'<dx_π(L+1)>] x1,...,xn,E'\x,...,x,E $reference xnew
.
.
[∂−E'<dx_π(m)>] x1,...,xn,E'\x,...,x,E   $references xnew
```

A more specialized variant of the preceding rule was first presented in Paige[1] and proved correct. The problem of deciding whether such a permutation exists was proved to be NP-complete by Peter Gacs.[2]

References

1. R. Paige, *Formal Differentiation*, UMI Research Press, Ann Arbor, Mich., 1981 (revision of PhD dissertation, New York University, June 1979).
2. P. Gacs, private communication, 1980.

The preceding example leads to a general method described in the box at left that generates difference code for parameterized expressions in which no extra copies of arguments are required.

This paper has developed a programming paradigm based on invariants and an implementation based on an improved finite difference calculus. Although we introduced invariants as an extension of the SETL language, the same method can be used to add invariants to any imperative programming language.

Programming with invariants involves an interesting separation of functional and imperative programming paradigms in a way that makes programming and the translation of programs into efficient code easier. This article illustrated this style of programming with three simple examples, but applications to other more significant examples should be apparent.

For future work, it would be interesting to consider whether finite differencing can be generalized to handle dynamic programming. It would also be worthwhile investigating how to implement and exploit more general invariants than the ones discussed here. Finally, the use of invariants and their implementation by finite differencing presented here is only one of several basic program transformations.

Our goal is to formalize other principles of programming methodology and to capture them as part of a small but widely applicable collection of powerful transformations. These transformations, along with dictions to combine and apply them, could form a useful supplement to a programming language. For highly restricted languages they might even be used to form a fully automatic compiler. □

Acknowledgments

Part of this work was done while I was visiting Yale University on sabbatical from Rutgers University and while I was a summer faculty member at IBM Yorktown Heights.

I am grateful to David Gries, Brent Hailpern, Lee Hoevel, and Ken Perry for their helpful comments on earlier drafts of this article. This work is partly based upon work supported by the National Science Foundation under grant MCS-8212936 and by the Office of Naval Research under grant N00014-84-K-0444.

References

1. A. Goldberg and R. Paige, "Stream Processing," *ACM Symp. Lisp and Functional Programming,* Aug. 1984, pp. 53-62.
2. R. Paige and S. Koenig, "Finite Differencing of Computatable Expressions," *ACM TOPLAS,* Vol. 4, No. 3, July 1982, pp. 402-454.
3. R. Paige, *Formal Differentiation,* UMI Research Press, Ann Arbor, Mich., 1981 (revision of PhD dissertation, New York University, June 1979).
4. J.T. Schwartz, *On Programming: An Interim Report on the SETL Project, Installments I and II,* CIMS, New York University, New York, 1974.
5. R. Paige, "Transformational Programming—Applications to Algorithms and Systems," *Proc. 10th ACM Symp. Princ. Programming Languages,* Jan. 1983, pp. 73-87.
6. R. Paige, "Supercompilers—Extended Abstract," in *Program Transformation and Programming Environments,* P. Pepper, ed., Springer-Verlag, New York, 1984, pp. 331-340.
7. A. Aho, J. Hopcroft, and J. Ullman, *Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.
8. J. Cocke and K. Kennedy, "An Algorithm for Reduction of Operator Strength," *Comm. ACM,* Vol. 20, No. 11, Nov. 1977, pp. 850-856.
9. J. Earley, "High-Level Iterators and a Method for Automatically Designing Data Structure Representation," *Journal of Computer Languages,* Vol. 1, No. 4, pp. 321-342.
10. A. Fong and J. Ullman, "Induction Variables in Very High Level Languages," *Proc. Third ACM Symp. Princ. Programming Languages,* Jan. 1976, pp. 104-112.
11. D. Chamberlin et al., "Janus: An Interactive System for Document Composition," *Proc. ACM SIGPLAN/SIGOA Symp. Text Manipulation,* June 1981.
12. R. Dewar, A. Grand, S.C. Liu, J.T. Schwartz, and E. Schonberg, "Program by Refinement, as Exemplified by the SETL Representation Sublanguage," *ACM TOPLAS,* Vol. 1, No. 1, July 1979, pp. 27-49.

Robert Paige is an associate professor in Rutgers University's Department of Computer Science. His current research interests include program development methodology. He received a BA from Occidental College and an MS and PhD from New York University's Courant Institute.

His address is Rutgers University, Department of Computer Science, New Brunswick, NJ 08903.