Finite Differencing of Computable Expressions

ROBERT PAIGE and SHAYE KOENIG
Rutgers—The State University of New Jersey

Finite differencing is a program optimization method that generalizes strength reduction, and provides an efficient implementation for a host of program transformations including "iterator inversion." Finite differencing is formally specified in terms of more basic transformations shown to preserve program semantics. Estimates of the speedup that the technique yields are given. A full illustrative example of algorithm derivation is presented.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; *SETL*; D.3.4. [**Programming Languages**]: Processors—*optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Program transformation, differentiable expression

1. INTRODUCTION

Formal differentiation was developed by Paige [33] as a global program optimization method that captures a commonly occurring yet distinctive mechanism of program construction in which repeated costly calculations are replaced by inexpensive incremental counterparts. When formal differentiation is applied to algorithms expressed as high-level, lucid, but inefficient problem statements, the transformed algorithms materialize as more complex but efficient program versions. This method generalizes John Cocke's method of strength reduction, and provides a convenient framework with which to implement a host of program transformations, including Earley's "iterator inversion" [13].

However, we prefer to replace the terms "formal differentiation" and "reduction in strength" by the more accurate term "finite differencing." As we see below, the

This material is based upon work supported by the National Science Foundation under grant MCS79-05293.

Authors' present address: Department of Computer Science, Hill Center for the Mathematical Sciences, Rutgers—The State University of New Jersey, Busch Campus, New Brunswick, NJ 08903.

use of the term "finite differencing" places our method in proper historical perspective, and establishes an interesting link between modern program optimization and finite difference techniques developed by sixteenth-century mathematicians to reduce the amount of manual labor in performing calculations (e.g., Napier used finite difference techniques to construct a table of logarithms, a task that still took him a lifetime) [21].

In [33] language-independent algorithms are provided to implement finite differencing both automatically and semiautomatically, and these algorithms are adapted to FORTRAN and SETL. However, since the greatest success is achieved for very high-level languages, Paige's investigations focus on set theoretic finite differencing, which extends and formalizes Earley's transformations. Provisions are made to accommodate these transformations within a semiautomatic implementation design for a subset of SETL, and this proposed system is illustrated by considering and improving eight sample SETL programs.

In contrast to other program transformations, finite differencing is unusual in many respects; for example,

(1) finite differencing may be applied over a large spectrum of language levels and in wide-ranging contexts within these languages;
(2) finite differencing can realize swift convergence from a very high-level inefficient form of an algorithm to a much lower level, and more efficient, implementation version;
(3) finite differencing can be implemented systematically;
(4) finite differencing can be shown to yield asymptotic speedup.

In this paper, we extend and formalize the treatment of finite differencing found in [33] in order to illuminate the issues involved in an efficient implementation. Section 2 describes the historical development of our method; Section 3 presents basic definitions and notations; Section 4 develops a formal specification of finite differencing in terms of a small collection of program transformations; and Sections 5 and 6 discuss applications to algorithm development and improvement.

2. HISTORICAL PERSPECTIVE

It is interesting to note that the origins of our method may be traced back to the finite difference techniques introduced by the English mathematician Henry Briggs in the sixteenth century [21]. His method, which can be used to generate a sequence of polynomial values $p(x_0)$, $p(x_0 + h)$, $p(x_0 + 2h)$, ..., hinges on the following idea. For a given polynomial $p(x)$ of degree n and an increment h, the first difference polynomial

$$p_1(x) = p(x + h) - p(x)$$

is of degree $n - 1$ or less, the second difference polynomial

$$p_2(x) = p_1(x + h) - p_1(x)$$

is of degree $n - 2$ or less, ..., and, finally, $p_n(x)$ must be a constant. Thus, to tabulate successive values of $p(x)$ starting with $x = x_0$, we can perform

these two steps:

1. Calculate initial values for $p(x_0), p_1(x_0), \ldots, p_n(x_0)$ and store them in $t(1), t(2), \ldots, t(n+1)$.
2. Generate the desired polynomial table by iterating over the following code block:

```
print x, t(1);                    $ print x and p(x)
t(1) := t(1) + t(2);              $ place new values for
t(2) := t(2) + t(3);              $ p(x), p1(x),
        .                         $ ..., pn-1(x) into
        .
        .
t(n) := t(n) + t(n + 1);          $ t(1), t(2), ..., t(n)
x := x + h;                       $
```

Note that Briggs's method requires only n additions in step 2 to compute each new polynomial value, while Horner's rule to compute a fresh polynomial value costs n additions and n multiplications.

In an attempt to cut down further on the manual effort needed to produce accurate mathematical tables, Charles Babbage designed his analytic difference engine to perform step 2 automatically once its registers (analogous to t) were set manually as in step 1. Digital computers were designed to perform this same limited task of finite differencing through World War II when accurate gunnery tables were critically needed by the armed forces. When the von Neumann computer was finally developed just after the war, it allowed, among other things, both steps of Briggs's finite difference method to be programmed [22].

However, in the 1960s John Cocke discovered a program optimization method he called "reduction in operator strength" that revealed the greater significance of finite differencing as applied to the speedup of FORTRAN programs. His original techniques have since been generalized and implemented with various improvements (for which see [3, 8–10, 26–30]).

We illustrate Cocke's method with the following simple example. Suppose that an expression $i * c$ occurring in a strongly connected program region R cannot be moved out of R because of redefinitions to i. (We assume here that c is a region constant of R.) Suppose also that the variable i is defined before each entry to R and that all redefinitions to i within R are of the form $i = i \pm delta$ where $delta$ is a region constant of R. Then we can use the following idea to move all calculations of $i * c$ out of R. Since i is defined on entrance to R, we can insert an assignment $T = i * c$ to a unique compiler-generated variable T just prior to each entry point of R. Within R, immediately before each redefinition $i = i \pm delta$ to i, we can preserve the value of $i * c$ in T by executing the update assignment $T = T \pm delta * c$ (whose form follows from the distributive law). Note that $delta * c$ is invariant, and its calculation can be moved out of R. Finally, we see that all calculations of $i * c$ are redundant in R and can be replaced by uses of T. If the time cost of the addition operations inserted into R by strength reduction is less than the cost of the multiplications $i * c$ removed from the original text, then a constant-factor improvement in running time should be obtained.

Although Cocke's technique does not treat polynomials as special objects, strength reduction is sufficiently powerful to transform a program involving

repeated calculations of a polynomial according to Horner's rule into an equivalent program that essentially uses the more efficient finite difference method of Briggs. Indeed, this is a surprising and important result that demonstrates that the success of polynomial evaluation by differencing results from properties of the elementary operations used to form polynomials rather than from properties exclusive to polynomials. In other words, Cocke's method works because the following distributive and associative laws hold for sums and products:

$$(i \pm delta) * c \Rightarrow i * c \pm delta * c;$$

$$(i \pm delta) + c \Rightarrow (i + c) \pm delta.$$

In [10] Cocke and Schwartz extend this idea to show how reduction in strength (which we call finite differencing) applies to a wide range of arithmetic operations that exhibit appropriate distributive properties. Application of the idea of finite differencing in a set theoretic context was initiated by Earley, and has been pushed further by Fong and Ullman [15–17], who made the interesting observation that finite differencing in a set theoretic milieu could actually improve the asymptotic behavior of an algorithm, and that this fact could be used to develop a theoretical characterization of the situations in which this technique applied. In [33] finite differencing is generalized further so that the method can be applied directly to an extensive collection of expressions involving a variety of operations and data types.

3. DEFINITIONS AND NOTATIONS

3.1 Language

Although finite differencing can be applied to a variety of programming languages, we illustrate our transformations throughout this paper using SETL [41], a programming language that incorporates dictions ranging from the concrete level of FORTRAN up to the abstract level of set theory. The distinctive data types of SETL are its heterogeneous tuples, sets, and maps. Tuples are ordered from the first to the last component; sets are unordered and cannot contain repeated elements; maps are represented by sets of pairs $[x, y]$ each of which associates a domain value x with a corresponding range value y. Table I lists some of the operations that can be used to form expressions in SETL.

Like C, SETL allows assignment statements of the form

$x := x$ op exp;

to be abbreviated

x op:= exp;

SETL also has an APL reduction operation, binop/Q, that extends a binary associative operator binop to an operation over all the elements $x_1, x_2, \ldots, x_n$ of a set or tuple Q; that is,

$$\text{binop}/Q = x_1 \text{ binop } x_2 \text{ binop } \cdots \text{ binop } x_n.$$

Much of the power of SETL is due to its iterators, which provide mechanisms for constrained search through sets and tuples. These iterators can be combined

Table I. Expression Forms in SETL

Primitive operations	Remarks
$x + y$	Integer and real addition; set union; string and tuple concatenation
$x - y$	Integer and real subtraction; set difference
$x * y$	Integer and real multiplication; set intersection
$x \in y$, $x \notin y$	Membership tests on sets and tuples
$\#x$	Cardinality of sets; length of tuples and strings
arb x	An arbitrary element selected from set x; the value of **arb** { } is the undefined atom, denoted **om**
x **incs** y	Boolean-valued test whether the set x includes the set y
x **with** y	Same as $x + \{y\}$ when x is a set; same as $x + [y]$ when x is a tuple
x **less** y	Same as $x - \{y\}$
$\{x, y, \ldots\}$	Set with specified elements
$[x, y, \ldots]$	Tuple with specified components

with each other and used as arguments to various "iterative" operations. We can illustrate iterators using the following most basic example, called a **forall** loop or ∀-loop:

```
(∀x ∈ s | x mod 2 = 0)
  block(x)                                    (1)
end ∀;
```

The control structure (1) performs an execution of "block" for each even number contained in the set s. It is implemented by a search through s in which every value belonging to s is selected without repetition and stored into the bound variable x. Each time that a new value is stored in x, the predicate x **mod** $2 = 0$ is executed; if the predicate is true, then the block is executed. Since s is a set, the search through s is unordered. However, the **forall** loop (1) also permits s to be a tuple, in which case the search through s would be ordered from the first to the last component of the tuple.

forall loops can be used to implement various high-level expressions that involve iterators. One such expression is the set former, which computes the subset of a set satisfying a predicate. An example of this is

$$\{x \in s \mid x \textbf{ mod } 2 \neq 0\}, \tag{2}$$

which computes the set of all odd elements of s. To compute the set of squares of the odd elements of s, we use the following variant of (2):

$$\{x^2 : x \in s \mid x \textbf{ mod } 2 \neq 0\}.$$

SETL also allows sets and tuples to be formed using range specifications. Thus, $[2 \,..\, n - 1]$ computes a tuple whose first component is 2, second component is 3, ..., and whose last component is $n - 1$. The expression $\{1, 3 \,..\, 11\}$ computes all the odd numbers between 1 and 11 inclusive.

SETL includes bounded existential and universal quantifiers. To determine whether a natural number n is prime, we can execute the universal quantifier

$$\forall j \in [2 \,..\, n - 1] \mid n \textbf{ mod } j \neq 0,$$

which will have the value true if no value j between 2 and $n - 1$ divides evenly into n. If n is not prime, j will be assigned a value for which n **mod** $j \neq 0$ is false as a side effect.

SETL has three kinds of map retrieval operations:

1. $f(a)$ denotes function application and computes the value of f at a. If a does not belong to the domain of f or if f is not single valued at a, the value of $f(a)$ is **om** (**om** denotes the undefined atom).
2. $f\{a\}$ denotes the image set of $\{a\}$ under f. If a does not belong to the domain of f, the image set is $\{\ \}$.
3. $f[S]$ denotes the image of the set S under f and is equivalent to $+/\ \{f\{a\} : a \in S\}$.

Maps can be modified dynamically by indexed assignment. The operation $f(a) :=$ **om** removes the value a from the domain of f. The indexed assignment $f(a) := z$ is equivalent to

```
f(a) := om;
f with:= [a, z];      $ f(a) = z afterward
```

The image set of a multivalued map f at a domain point a can be modified by the operations

```
f{a} ±:= delta;       $ delta is a set                    (3)
```

or

```
f{a} := s;            $ s is a set                        (4)
```

where (4) is equivalent to

```
f(a) := om;           $ remove a from the domain of f
  (∀x ∈ s)
  f{a} with:= x;      $ add the pair [a, x] to f
end ∀;
```

Note that n-parameter maps are also represented by sets of pairs each of whose first component is an n-tuple. As a notational convenience, the map retrieval term $f(x, y, z)$ can be used to abbreviate $f([x, y, z])$.

As in mathematics, SETL uses copy value semantics.

3.2 Verification

To support verification, we extend SETL by allowing programs to be annotated with two statements,

assume cond;

and

assert cond;

where cond is any SETL predicate. During execution, **assume** and **assert** statements are no-ops. However, whenever an **assume** or **assert** statement is encountered during execution, if the condition holds, we say that the encounter

is *satisfied.* We say that an execution of a program is *valid* if

(1) whenever the execution includes an unsatisfied **assert** encounter, the first such encounter is preceded by an unsatisfied **assume** encounter;
(2) whenever all **assume** encounters are satisfied, execution terminates normally.

A program is defined as valid if all of its possible executions are valid. The *domain* of a valid program P is the set of input values that yield executions in which every assumption encounter is satisfied.

In order to develop a suitable characterization of correct program transformations, we require that every program P include special output assertions placed at the normal exit points of P. A program transformation T is said to be *validity preserving* if, whenever T maps a valid program P into a new program $P' = T(P)$, T leaves the output assertions of P intact and P' is valid. A validity-preserving program transformation T is said to be *semantics preserving* if, whenever T maps a valid program P into a new program $P' = T(P)$, the domain of P' includes the domain of P.[1]

It is frequently useful to consider transformations applied to single-entry, single-exit regions of code that we call *blocks.* Whenever the replacement of a block B within a program P by another block B' is validity (respectively, semantics) preserving, we say that B' preserves the validity (respectively, semantics) of B within P.

One of the transformations to be discussed operates on **achieve** statements

achieve $E = f(x_1, \ldots, x_n)$;

that have the same semantics as assignment statements

$E := f(x_1, \ldots, x_n)$;

3.3 Complexity

In order to demonstrate that finite differencing results in program speedup, we must utilize some measure of expected efficiency. Our heuristic complexity measure is supported by the most basic storage structures implemented within the run-time environment of SETL (cf. [18]). Sets are implemented by expandable hash tables that permit a unit-time membership test and a linear-time search through all elements of a set. Such an implementation also permits unit-time element addition and deletion whenever these operations can be performed directly on the body of a set without copying. Maps are implemented using a similar hash table for storing their domains, and range elements are accessed rapidly via their corresponding domain elements. This permits various kinds of functional application and change to be done in time proportional to a map's arity, and permits iteration through a map's domain to be done in linear time.

On the basis of the preceding measure, it is easy to verify the time estimates shown in Table II for set theoretic operations.

3.4 Miscellaneous Definitions

In order to discuss the notion of finite differencing systematically, it is convenient to introduce some definitions and notational devices that we have borrowed (with

[1] For a more comprehensive study of transformational correctness, see [4, 5].

Table II. Complexity of SETL Operations

Operation	Estimated cost
s **with**:= x;	$O(1)$[a]
s **less**:= x;	$O(1)$[a]
$x \in s$	$O(1)$
s +:= *delta*;	$O(\#delta)$[a]
$f(x) := y$;	$O(1)$[a]
$f(x_1, \ldots, x_n)$	$O(n)$
(∀$x \in s$) Block(x) **end** ∀;	$O(\#s \times \text{cost(Block)})$
$\{x \in s \mid k(x)\}$	$O(\#s \times \text{cost}(k))$
$\exists x \in s \mid k(x)$	$O(\#s \times \text{cost}(k))$
$\forall x \in s \mid k(x)$	$O(\#s \times \text{cost}(k))$
$s + t$	$O(\#s + \#t)$
$f[s]$	$O(\#\{[x, y] \in f \mid x \in s\})$

[a] These estimates hold when set copy operations are avoided.

slight modifications) from program optimization literature (for which see [1, 2, 10, 23, 25]). We sometimes use the mathematical function notation

$$C = f(x_1, \ldots, x_n)$$

to uniquely associate a text expression f involving n distinct free variables $x_1, \ldots, x_n$ with a variable C (which we call the *virtual* variable associated with f). We assume that, whenever f is executed, its value, calculated from the values of its free variables $x_1, \ldots, x_n$ and constants, is placed in C. We also assume that f and all of its subexpressions are *applicative*; that is, f behaves like a finite map.

We say that C is *available on exit* from a program point p if C is equal to the value that the expression f would have if evaluated immediately after the statement at p is executed; C is *available on entrance* to p if C is available on exit from all predecessor points of p. If C is available on entrance to p, and if C is not available on exit from p (which will happen when execution of the statement at p changes the value of a parameter x_i upon which the value of f depends), then we say that C is *spoiled* at p. If C is available on entrance to a program point p at which there is an occurrence of a retrieval expression f, we say that the occurrence of f is *redundant* at p. When this is the case, program semantics will be preserved by replacing the occurrence of f by the variable C. (Such replacement is commonly called *redundant code elimination.*)

Expression f is *well defined* at a program point p if, for every valid execution that passes through p and satisfies every assumption encounter prior to p, the values of $x_1, \ldots, x_n$ at the point p belong to the domain of f.

A *control flow path* is a sequence of program points representing a logical sequence of primitive operations that might be performed, under the assumption that, every time a predicate Q is encountered during execution, the value of Q is interpreted as being either true or false.

Within the text of a program we distinguish between two kinds of variable occurrences: *uses* and *definitions*. A use of a variable v is an occurrence at which the value of v is retrieved but not modified. A definition of v is an occurrence in

which v is modified, as, for example, at the left of an assignment statement, within a **read** statement, and as the bound variable of an iterator. We say that a definition d of a variable v *reaches* a program point p if there exists a control flow path from d to p that contains no definitions to v other than d. A use u of a variable v is *live* at a program point p if there exists a control flow path from p to u that is free of definitions to v.

On the basis of "reaches" and "live" relations, we can construct the two standard data flow maps, *usetodef* and *deftouse*, which have the following meaning. If d is a definition to a variable v, then *deftouse*$\{d\}$ is the set of uses of v reached by d; if u is a use of v, then *usetodef*$\{u\}$ is the set of definitions to v that reach u. In Section 5 we show how to use the data flow maps to perform *dead-code elimination*, a semantics-preserving transformation that eliminates code not contributing either directly or indirectly to the value of any program variables used within **print**, sequential **read**, **assume**, and **assert** statements.

4. FINITE DIFFERENCING OF APPLICATIVE EXPRESSIONS

4.1 Basic Concepts

In [10] reduction in strength (which we call finite differencing) is viewed as an extension of code motion whereby the major cost of evaluating an expression

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

is moved outside a program region R despite modifications to its parameters $x_1, \ldots, x_n$ occurring within R. The basic idea of this technique can be expressed as follows: by making E available on entry to R (by evaluating f and storing its value in E immediately prior to R), and keeping E available within R (by appropriately modifying E each time one of the parameters $x_1, \ldots, x_n$ is modified), we can avoid full calculations of f within R (by replacing redundant occurrences of f within R by the variable E). For this approach to be useful, the cost of keeping E available in R must be less than the cost of calculating f anew each time it is referenced.

We formulate finite differencing in terms of a small but powerful collection of semantics-preserving program transformations that generalize the three separate tasks implied by Cocke's schema just above.

1. The Init transformation Init$\langle P \rangle$ replaces each contiguous sequence of **achieve** statements

achieve $E = f(x_1, \ldots, x_n)$;

within a program P by a code block B that computes and stores the values of the expressions $f(x_1, \ldots, x_n)$ into their respective virtual variables E.

2. The Differential transformation, denoted $\partial J\langle R \rangle$, inserts code within a program region R in order to keep each expression $E = f(x_1, \ldots, x_n)$ belonging to a sequence of expressions J available at points in R after which redundant uses of $f(x_1, \ldots, x_n)$ are replaced by E.

3. Clean is a transformation that eliminates dead code; it is the last step of finite differencing.

Although we only illustrate finite differencing for SETL, this method can be extended to other procedural languages that have a rich supply of applicative expressions.

4.2 The Differential Operator and the Chain Rule

Since the Differential transformation is fundamental to finite differencing, a comprehensive description of this transformation is essential to an understanding of the example of algorithm derivation by finite differencing given in Section 5. However, in order to move as rapidly as possible toward this full case study, we postpone a full specification of the ancillary transformations, Init and Clean, until Section 6.

In this section we formally specify the differential with respect to a single applicative expression and a single-entry, single-exit code block and prove that it is a semantics-preserving program transformation. Next, we extend the differential so that it can be applied to sequences of expressions by means of a "chain rule." Finally, we state conditions under which the differential may be expected to improve the running time of programs.

Recall that, in calculus, the differential operator uses the value of a function $y = f(x)$ and its derivative at a point x_{old} to obtain an approximate value of f at a new domain point x_{new} that lies a "slight" distance away from x_{old}. Our differential program transformation serves a purpose similar to that of its counterpart in calculus, and is defined in terms of analogous components. These components include

(1) an applicative expression $E = f(x_1, \ldots, x_n)$ where E is a variable uniquely associated with the value of $f(x_1, \ldots, x_n)$;
(2) a single-entry, single-exit code block B that can modify the values of the variables $x_1, \ldots, x_n$ on which E depends;
(3) a computable "derivative" that allows us to determine the new value E_{new} of E from its old value E_{old} when the old value is spoiled by an assignment dx_i to a variable x_i on which E depends.

The computable *derivative* is defined formally as follows:

Definition 1. Let $E = f(x_1, \ldots, x_n)$ be an applicative expression that depends on the variables $x_1, \ldots, x_n$, and let dx_i be an assignment to the variable x_i. The code block pair $[B_1, B_2]$ is said to be a derivative of E with respect to dx_i if

(1) the only variables modified by B_1 or B_2 are E and variables local to B_1 and B_2; and
(2) the code block

achieve $E = f(x_1, \ldots, x_n)$;
B_1
dx_i
B_2
assert $E = f(x_1, \ldots, x_n)$

preserves the semantics of dx_i and contains only redundant uses of $f(x_1, \ldots, x_n)$.

Note that, when no uses of E within the derivative code (either B_1 or B_2) are live on entry to B_1, we can omit the **achieve** statement in the code block above, in which case we say that $[B_1, B_2]$ is a *strong* derivative.

Whenever $[B_1, B_2]$ is a derivative of E with respect to dx_i, we say that B_1 is a *prederivative* of E with respect to dx_i and that B_2 is a corresponding *postderivative* of E with respect to dx_i, for which we write

$$B_1 = \partial^- E\langle dx_i \rangle$$

and

$$B_2 = \partial^+ E\langle dx_i \rangle,$$

respectively. Note that occurrences of the variable x_i within B_1 refer to the old value of x_i prior to the change dx_i, while occurrences of x_i within B_2 refer to the new value.

Note that the derivative code is not unique, and it can always be defined for an applicative expression $E = f(x_1, \ldots, x_n)$ relative to parameter modifications using an empty prederivative and a postderivative that evaluates $f(x_1, \ldots, x_n)$ from "scratch" (i.e., without reference to the old value of f stored in E) and stores this value into E. Also, if the variable x_i modified by an assignment dx_i is not among the free variables $x_1, \ldots, x_n$, the derivative of E with respect to dx_i is empty.

To illustrate derivatives further, consider the set former

$$E = \{x \in A \mid x \textbf{ mod } 2 = 0\} \tag{5}$$

where A is a set of integers. A derivative of E with respect to the statement

A **with**:= i;

is given by the empty postderivative and the computationally inexpensive prederivative

if i **mod** 2 = 0 **then**
  E **with**:= i; (6)
end if;

A strong derivative of E with respect to the statement

$A := \{\}$;

is given by the empty postderivative and the prederivative

$E := \{\}$; (7)

In practice, expression (5) and derivatives (6) and (7) would be part of a finite collection of basic applicative expressions that we call *Forms* and associated derivative rules that we call *Derivs*. We represent a derivative rule as a quadruple

$$[E = f(x_1, \ldots, x_n),\ dx_i,\ \partial^- E\langle dx_i \rangle,\ \partial^+ E\langle dx_i \rangle] \tag{8}$$

associating a basic expression belonging to *Forms* and a parameter modification dx_i with a unique pre- and postderivative pair. Although *Derivs* only contains a finite number of rules, each rule (8) represents an equivalence class of rules formed from (8) by substituting n distinct variables for the parameters $x_1, \ldots,$

x_n. (We call such substitutions *d-substitutions.*) Thus, *Derivs* gives rise to derivative rules for all expressions transformed by parameter *d*-substitutions of the basic expressions belonging to *Forms*; we call these expressions *elementary* expressions. To ensure that no more than one derivative rule is applicable for a given elementary expression and parameter modification, we require that no expression belonging to *Forms* can be transformed by *d*-substitutions into a subexpression of any other expression belonging to *Forms.*

On the basis of the two sets *Forms* and *Derivs*, we can proceed to specify the differential fully.

Definition 2. Consider an applicative expression $E = f(x_1, \ldots, x_n)$ that is well defined within a single-entry, single-exit code block B occurring in a valid program P. Suppose that no uses of E in P are live within B. Suppose, also, that within our collection *Derivs* of derivative rules are rules giving derivatives for E with respect to every assignment dx_i occurring in B to a variable x_i on which E depends. Suppose, finally, that, if f is not well defined on entry to B, the first statement of B is an update statement with respect to which the derivative of E is a strong derivative. Then E is said to be *differentiable* with respect to B. The differential of E with respect to B, denoted $\partial E\langle B\rangle$, is a new code block formed from B in the following way:

1. *Derivative Code Insertion.* Replace each statement dx_i that modifies a variable x_i on which E depends by

$\partial^- E\langle dx_i\rangle$
dx_i
$\partial^+ E\langle dx_i\rangle$

2. *Redundant Code Elimination.* Replace all uses of $f(x_1, \ldots, x_n)$ within the code block that results from step 2 with uses of the variable E.

We illustrate the differential using the applicative expression (5), differentiable with respect to the following code block:

```
a := { };
(while eof = false)
  read(i);                                        (9)
  a with:= i;
end while;
print({x ∈ a | x mod 2 = 0});
```

This code reads a sequence of integers and prints the set of even integers in the sequence. The differential of E with respect to code block (9) can be expressed with our operator notation as

```
∂E⟨a := { };
    (while eof = false)
      read(i);                                    (10)
      a with:= i;
    end while;
    print({x ∈ a | x mod 2 = 0});⟩
```

If we temporarily neglect the redundant-expression elimination task of the differential operator, code block (10) can be written more concretely as

```
∂⁻E⟨a := { };⟩
a := { };
(while eof = false)
  read(i);                                          (11)
  ∂⁻E⟨a with:= i;⟩
  a with:= i;
end while;
print({x ∈ a | x mod 2 = 0});
```

Since (11) results from inserting dead code into (9), (11) is semantically equivalent to (9). By definition of the derivative, we can transform (11) into the following equivalent code:

```
E := { };
a := { };
assert E = {x ∈ a | x mod 2 = 0};
(while eof = false)
  read(i);
  achieve E = {x ∈ a | x mod 2 = 0};
  if i mod 2 = 0 then                               (12)
    E with:= i;
  end if;
  a with:= i;
  assert E = {x ∈ a | x mod 2 = 0};
end while;
print({x ∈ a | x mod 2 = 0});
```

By using a partial-correctness inference system similar to Hoare's [24] and based on Gerhart [20] and Schwartz [38], we can propagate assertions throughout (12) and eliminate redundant **achieve** statements to obtain the following code:

```
E := { };
a := { };
assert E = {x ∈ a | x mod 2 = 0};
(while eof = false)
  assert E = {x ∈ a | x mod 2 = 0};
  read(i);
  assert E = {x ∈ a | x mod 2 = 0};
  if i mod 2 = 0 then
    assert E = {x ∈ a | x mod 2 = 0};               (13)
    E with:= i;
  end if;
  a with:= i;
  assert E = {x ∈ a | x mod 2 = 0};
end while;
assert E = {x ∈ a | x mod 2 = 0};
print ({x ∈ a | x mod 2 = 0});
assert E = {x ∈ a | x mod 2 = 0};
```

Note finally that the use of $\{x \in a \mid x \bmod 2 = 0\}$ within the **print** statement of (13) is redundant and can be replaced by the variable E.

The argument used to prove that (13) preserves the semantics of (9) may be generalized to prove the following theorem, which formally justifies the correctness of the differential operator.

THEOREM 1. *Let* $E = f(x_1, \ldots, x_n)$ *be an applicative expression that is differentiable with respect to a code block* B *(occurring within a valid program* P*). Then, if any use of* E *occurring within* $\partial E\langle B\rangle$ *is live on entry to* $\partial E\langle B\rangle$*, the code block*

achieve $E = f(x_1, \ldots, x_n)$;
$\partial E\langle B\rangle$

preserves the semantics of B*; otherwise,* $\partial E\langle B\rangle$ *preserves the semantics of* B*. Furthermore,* E *is available on exit from* $\partial E\langle B\rangle$*.*

PROOF. The proof follows from the fact that the differential can be defined in terms of semantics-preserving transformations. Without loss of generality, suppose that there is a use of E within $\partial E\langle B\rangle$ that is live on entry to $\partial E\langle B\rangle$. Then it must be the case that f is well defined on entry to B. Since E is not live on entry to B, insertion of

achieve $E = f(x_1, \ldots, x_n)$;

on entry to B is a semantics preserving dead-code insertion transformation. By definition of derivatives, replacing each modification dx_i to a variable x_i on which E depends by

achieve $E = f(x_1, \ldots, x_n)$;[2]
$\partial^- E\langle dx_i\rangle$
dx_i (14)
$\partial^+ E\langle dx_i\rangle$
assert $E = f(x_1, \ldots, x_n)$;

is also semantics preserving. Since, by definition of derivatives, all uses of $f(x_1, \ldots, x_n)$ occurring within (14) are redundant, replacement of such uses (in code other than **achieve** or **assert** statements) by uses of E is semantics preserving. A simple inductive argument can be used to show that propagation of the assertion $E = f(x_1, \ldots, x_n)$ will justify elimination of all **achieve** statements inserted within the derivative blocks (14) as redundant. A similar argument can be used to show that all remaining uses of $f(x_1, \ldots, x_n)$ (outside of the **achieve** statement inserted just prior to B) are redundant and can be replaced by uses of E. Moreover, E will be available on exit from $\partial E\langle B\rangle$. □

COROLLARY 1.1 *The differential transformation is a linear operator with respect to sequential blocks; that is,* $\partial E\langle B_1\ B_2\rangle = \partial E\langle B_1\rangle\ \partial E\langle B_2\rangle$.

Suppose that $f(x)$ and $g(y)$ are elementary expressions, that $E_1 = f(x)$ is differentiable with respect to dx, and that $E_2 = g(E_1)$ is differentiable with respect to modifications to E_1 within the block $\partial E_1\langle dx\rangle$. The following nested application of the differential indicates that the expression $g(f(x))$ is also differentiable with respect to dx:

$$\begin{aligned}\partial E_2\langle \partial E_1\langle dx\rangle\rangle = {} & \partial E_2\langle \partial^- E_1\langle dx\rangle\rangle \\ & \partial^- E_2\langle dx\rangle \\ & dx \\ & \partial^+ E_2\langle dx\rangle \\ & \partial E_2\langle \partial^+ E_1\langle dx\rangle\rangle.\end{aligned}$$

[2] The **achieve** statement is omitted for strong derivatives.

This observation allows us to extend our class of differentiable expressions to "nonelementary" expressions formed by composition of elementary expressions and parameter d-substitutions. More generally, the following definition and theorem show how to differentiate collections of expressions using a chain rule.

Definition 3. Consider n elementary expressions $E_1 = f_1, \ldots, E_n = f_n$ and a single-entry, single-exit code block B occurring in a valid program P. Suppose that E_1 is differentiable with respect to B, that E_2 is differentiable with respect to $\partial E_1\langle B\rangle, \ldots$, and that E_n is differentiable with respect to $\partial E_{n-1}\langle\cdots\langle\partial E_1\langle B\rangle\rangle\cdots\rangle$. Suppose also that $i > j$ implies that f_j does not involve the variable E_i (i.e., $f_1, \ldots, f_n$ preserve an inner-to-outer subexpression ordering). Then the list of virtual variables $E_n, \ldots, E_1$ is said to form a *differentiable chain*, and the extended differential of this chain with respect to B is defined recursively by the following "chain rule":

$$\partial E_n, E_{n-1}, \ldots, E_1\langle B\rangle = \partial E_n, E_{n-1}, \ldots, E_2\langle\partial E_1\langle B\rangle\rangle. \tag{15}$$

It is important to further restrict the chain ordering (15) to prevent derivative code for E_i from introducing any expression $f_j, j < i$, since such an occurrence of f_j might not be eliminated as redundant within the extended differential (15).

THEOREM 2 (CHAIN RULE). *Let $E_n = f_n, \ldots, E_1 = f_1$ be a chain of n applicative expressions differentiable with respect to a code block B occurring within a valid program P. Let S be the set of indices $i = 1 \ldots n$ for which there are uses of E_i within $B' = \partial E_n, \ldots, E_1\langle B\rangle$ live on entry to B'. Then the code block*

achieve $\Lambda_{i\in S} E_i = f_i$;
$\partial E_n, \ldots, E_1\langle B\rangle$

preserves the semantics of B and keeps $E_1, \ldots, E_n$ available on exit. Furthermore, if g is any expression formed from some $f_j, j = 1 \ldots n$, by substituting f_i for E_i, $i = 1 \ldots j - 1$, then any use of g occurring within B or introduced within derivative code by the chain rule will be made redundant and replaced by E_j within $\partial E_n, \ldots, E_1\langle B\rangle$.

PROOF. Using Theorem 1 and the definition of differentiable chains (and especially taking account of the ordering of chains), the theorem follows easily from induction on the number of expressions in the differentiable chain. □

COROLLARY 2.1 *The extended differential is a linear operator with respect to sequential code blocks; that is,*

$$\partial E_n, \ldots, E_1\langle B_1\, B_2\rangle = \partial E_n, \ldots, E_1\langle B_1\rangle\ \partial E_n, \ldots, E_1\langle B_2\rangle.$$

To illustrate the chain rule, we consider two abstract elementary expressions $f(x)$ and $g(x, y)$ that are differentiable with respect to various modifications to their parameters. The following three cases demonstrate how "nonelementary" expressions formed from f and g by composition and parameter substitution are also differentiable:

$h(x, y) = g(f(x), y)$ is seen to be differentiable by applying the chain rule to $E_1 = f(x)$ and $E_2 = g(E_1, y)$;

$s(y) = g(f(y), y)$ is handled similarly by using the chain rule on $E_1 = f(y)$ and $E_2 = g(E_1, y)$;

$t(x) = g(x, x)$ is also differentiable since the identity expression $E_1 = x$ is differentiable; just apply the chain rule to E_1 and to $E_2 = g(E_1, x)$.

The chain rule leads to a calculus of computable derivatives for collections of differentiable expressions based on the following three rules:

$$\partial E_n, \ldots, E_1\langle dx\rangle = \partial^- E_n, \ldots, E_1\langle dx\rangle \; dx \; \partial^+ E_n, \ldots, E_1\langle dx\rangle;$$

$$\partial^- E_n, \ldots, E_1\langle dx\rangle = \partial E_n, \ldots, E_2\langle \partial^- E_1\langle dx\rangle\rangle \; \partial^- E_n, \ldots, E_2\langle dx\rangle;$$

$$\partial^+ E_n, \ldots, E_1\langle dx\rangle = \partial^+ E_n, \ldots, E_2\langle dx\rangle \; \partial E_n, \ldots, E_2\langle \partial^+ E_1\langle dx\rangle\rangle.$$

4.3 Speedup

In order for finite differencing to improve program performance, the overall computational cost of calculating derivative code in a differentiated program must be less than the cost of calculating differentiable expressions in an unoptimized program. Such improvement will only be possible if differentiation is restricted to those expressions f relative to code blocks B for which

(1) each derivative code block for f is computationally less expensive than the cost of a fresh recalculation of f; and
(2) within B, the number of times in which a modification to an argument of f is executed relative to each time that f is executed is reasonably small.

Ever since Cocke's original formulation of automatic strength reduction [10], the various approaches to finite differencing have satisfied condition (1) a priori for all derivative rules, and have satisfied condition (2) using control and data flow analysis to constrain the regions B where differentiation can be applied. Cocke, Allen, Kennedy, Schwartz, and Markstein [3, 8–10] have presented derivative rules for various elementary numerical expressions (e.g., for replacing costly products and quotients by less expensive additions and for replacing exponentiations by less costly products). Cocke and Schwartz used different program analysis techniques based on linear nested regions and on intervals to restrict differentiation to those expressions f occurring in loops L (i.e., single-entry, strongly connected regions) in which no proper subloop could contain any modifications to any arguments of f without also containing f [10, pp. 408–462]. We call this restriction the *boundedness requirement*, because it implies that every control flow cycle within L contains a bounded number of modifications to arguments of f. On the basis of the assumption that execution frequency of code occurring within program loops is greater than that of code occurring in regions immediately containing these loops, they were able to expect a constant-factor speedup. They recognized that differentiated code might not be "safe" but did not find a reasonable solution other than avoiding differentiation of quotients (for which division by zero is a serious problem) in favor of products (for which the problem of overflow can be accepted as a necessary evil).

Profitable successive differentiation of assorted set expressions was first discussed informally by Earley [13], who presented examples that exhibited order-

of-magnitude speedup. Earley's ingenious technique of "iterator inversion" rested on highly efficient set theoretic derivative rules. For example, he gave derivatives for set formers $\{x \in s \mid k(x)\}$ with various kinds of predicates k, and for quantifiers $\exists x \in s \mid k(x)$ and $\forall x \in s \mid k(x)$. However, he lacked a general method of combining these rules and did not elaborate on special control flow considerations for recognizing contexts where profitable differentiation was ensured.

An implementation design for the differentiation of "nonelementary" set expressions with respect to program loops that satisfied the boundedness requirement was first given serious treatment by Fong and Ullman [16, 17], who considered differentiable expressions built up from set union, intersection, and difference, as well as certain kinds of set formers. Fong and Ullman adapted several of Earley's efficient derivative rules and developed a theoretical characterization of conditions under which their method would yield asymptotic speedup. Their method differed from previous work in that derivative code to maintain the availability of expressions was deferred to the point at which the expressions were used instead of at points where arguments of these expressions were modified.

Their deferred differencing approach is based on the following idea. Let $E = f(A)$ be a set-valued expression involving a set argument A and occurring at a program point p in a program loop L. Let A_{new} and A_{old} represent the new and old values of A between any two consecutive times that control reaches p along any path contained within L. To differentiate E, two auxiliary sets $d^+ = A_{\text{new}} - A_{\text{old}}$ and $d^- = A_{\text{old}} - A_{\text{new}}$ must be maintained by updating them whenever A is modified along each cycle within L containing p. Then, at point p, d^+ and d^- are used to update E (so that $E = f(A)$) and are reassigned to the empty set.

Among many other results Fong and Ullman proved were general conditions under which differentiation of set formers $\{x \in A \mid k(x)\}$ and quantifiers $\exists x \in A \mid k(x)$ and $\forall x \in A \mid k(x)$ would result in asymptotic speedup; that is, after differentiation, these expressions could be maintained with work proportional to $\#A + \text{cost}(k(x))$ instead of the straightforward $\#A \times \text{cost}(k(x))$ [15, 17].

Using a different program analysis technique than Cocke, Allen, et al., Fong presented implementation algorithms for reducible programs that could detect differentiable expressions within loops satisfying the boundedness requirement [16]. Improvements to her algorithms have since been presented by Rosen [36] and Tarjan [46].

In contrast to the theoretical approach of Fong and Ullman, Paige and Schwartz initiated a pragmatic investigation of Earley's "iterator inversion" by generalizing Earley's transformations and stating pragmatic rules for the discovery and treatment of reasonably general cases in which their technique could be applied [35]. On the basis of this study, Paige developed a finite difference method [33, pp. 71–89] that generalized both strength reduction and iterator inversion within a unified framework. He presented an extensive collection of new, efficient set theoretic derivative rules (some of which are reformulated in Appendix B) and stressed application of these rules to algorithm derivations. Like Cocke, Allen, et al., Paige mainly treated differencing as a loop optimization, in which efficient set theoretic derivatives and the loop boundedness requirement promised speedup

under the standard assumption that code is executed more frequently inside loops than outside.

Further development of the differencing mechanism found in [33] has led to the chain rule (Theorem 2), which gives rise to a calculus of computable derivatives for handling more general classes of expressions belonging to a variety of data types (in addition to set expressions) than before. In this section, we analyze the chain rule and the attendant speedup that our finite difference method yields. We emphasize set theoretic illustrations of profitable derivative rules and state conditions under which application of these and other rules (presented in Appendix B) can yield asymptotic speedup.

However, before presenting examples of efficient derivative rules for set expressions, it is useful to state a few guiding principles. The computational cost of derivative code for a set-valued expression $E = f(x_1, \ldots, x_n)$ depends on the overall cost of executing set unions and deletions

$$E \pm:= delta; \tag{16}$$

We can minimize these costs by ensuring that the modifications (16) represent disjoint unions and subset deletions. It is also worthwhile to define derivatives for E with respect to changes in set-valued parameters x_i only when the modifications $x_i \pm:= eps$ to x_i also represent disjoint unions and subset deletions. In this way, we can prevent the chain rule from propagating unnecessary computations.

It is convenient to further regularize our treatment of set modifications by expressing these modifications in terms of element additions s **with**:= z and deletions s **less**:= z for which the respective preconditions $z \notin s$ and $z \in s$ hold. We call these specialized element operations *strict* and assume that throughout this paper all occurrences of element additions and deletions are strict operations. This poses no undue restrictions, since any program can be preprocessed by turning set additions and deletions into repeated element operations that can then be rewritten as strict operations.

Using the measure of computational cost described in Table II, we can observe a variety of set theoretic expressions for which differentiation could be profitable. The set union $E_1 = S + T$ is differentiable with respect to an element addition, S **with**:= y, and element deletion, S **less**:= y, since the prederivative code blocks

```
if y ∉ T then      $ for S with:= y
  E1 with:= y;                                   (17)
end if;
```

and

```
if y ∉ T then      $ for S less:= y
  E1 less:= y;
end if;
```

take unit time, while computing the full union can be expected to cost $O(\#S + \#T)$ units of time.

Another differentiable set theoretic expression is the set former

$$E_2 = \{x \in S \mid K(x)\}$$

in which S does not occur free within K. The prederivatives for E_2 relative to the

changes S **with**:= y and S **less**:= y are

```
if K(y) then
  E2 with:= y;
end if;
```
(18)

and

```
if K(y) then
  E2 less:= y;
end if;
```

respectively. Note that (18) requires $O(1)$ elementary steps to compute.

The set cardinality expression $E_3 = \#S$ is also differentiable with respect to S **with**:= y and S **less**:= y. The prederivatives $\partial^- E_3\langle S$ **with**:= $y;\rangle$ and $\partial^- E_3\langle S$ **less**:= $y;\rangle$ are

$$E_3 \pm:= 1; \tag{19}$$

As was observed in Section 4.2, profitable differentiation of a nonelementary expression is supported by differentiation of its subexpressions according to the chain rule. As an example of this, consider

$$E_3 = \#\{x \in (S + T) \mid K(x)\}$$

where S and T do not occur free within K. In order to differentiate E_3 with respect to element additions S **with**:= y (for arbitrary values of y), we must first decompose E_3 into its elementary subexpressions,

$$E_1 = S + T;$$

$$E_2 = \{x \in E_1 \mid K(x)\};$$

$$E_3 = \#E_2.$$

Using derivative rules (17)–(19) and the chain rule, we compute the differential $\partial E_3, E_2, E_1\langle S$ **with**:= $y;\rangle$ according to the following steps:

$\partial E_3, E_2, E_1\langle S$ **with**:= $y;\rangle$

```
→ ∂E3, E2⟨if y ∉ T then
            E1 with:= y;
          end if;
          S with:= y;⟩

→ ∂E3⟨if y ∉ T then
        if K(y) then
          E2 with:= y;
        end if;
      end if;
      S with:= y;⟩

→ if y ∉ T then
    if K(y) then
      E3 +:= 1;
      E2 with:= y;
    end if;
  end if;
  S with:= y;
```
(20)

Observe that the outermost **if** statement within (20) forms the extended prederivative $\partial^- E_3, E_2, E_1\langle S$ **with**:= $y;\rangle$, whose cost is determined by summing the constant-factor costs of the prederivative rules for E_1, E_2, and E_3. Thus, calculating E_3 differentially by (20) represents a considerable speedup over a straightforward calculation of E_3 by three separate assignments

$$\begin{aligned} &E_1 := S + T; \\ &E_2 := \{x \in E_1 \mid K(x)\}; \\ &E_3 := \#E_2; \end{aligned} \tag{21}$$

Note finally that the incremental approach (20) results in greater data and operational independence than (21). Consequently, there can be more dead code and more opportunity for parallel execution occurring in (20) than (21). In this example, it may be possible to eliminate all assignments to E_1 and E_2 within (20) as dead.

Profitable differentiation of an expression f can sometimes be supported by differentiating f together with a chain of auxiliary expressions (as in Briggs's first, second, ... difference polynomials discussed in Section 2). Thus, the prederivative $\partial^- E\langle x +:= delta;\rangle$ of the nth degree polynomial $E = P(x)$ is

$$E +:= P_1(x)$$

where $P_1(x)$ is the first difference polynomial. However, for the prederivative code above to be inexpensive, we must also differentiate the second, third, ..., nth difference polynomials, denoted $E_i = P_i(x)$, $i = 2 \,..\, n$. To realize Briggs's efficient technique, we consider the extended prederivative (of expressions ordered carefully into a "differentiable chain") $\partial^- E_{n-1}, \ldots, E_1, E\langle x +:= delta;\rangle$ that expands into

$$\begin{aligned} &E +:= E_1; \\ &E_1 +:= E_2; \\ &\quad\vdots \\ &E_{n-1} +:= E_n; \end{aligned}$$

The preceding example, with its use of auxiliary expressions, has interesting analogues among set theoretic expressions. Consider the image set

$$E = \{e(x) : x \in s\} \tag{22}$$

where s does not occur free in the subexpression e. When e behaves like a one-to-one map, the prederivatives $\partial^- E\langle s$ **with**:= $z\rangle$ and $\partial^- E\langle s$ **less**:= $z\rangle$ are given by

E **with**:= $e(z)$;

and

E **less**:= $e(z)$;

An important subcase of (22) is the set former

$$E' = \{[e(x), x] : x \in s\}, \tag{23}$$

which represents the restriction of e^{-1} on its range to s (observe that $[e(x), x]$ is one-to-one). The inverse map (23) can be used as an auxiliary expression to

differentiate (22) even when e is not one-to-one. To see this, consider the case when e is many-to-one. Then $\partial^- E\langle s \textbf{ with:=} z;\rangle$ is

```
if #{x ∈ s | e(x) = e(z)} = 0 then
  E with:= e(z);                                    (24)
end if;
```

and $\partial^- E\langle s \textbf{ less:=} z;\rangle$ is

```
if #{x ∈ s | e(x) = e(z)} = 1 then
  E less:= e(z);                                    (25)
end if;
```

However, (24) and (25) do not represent efficient derivatives, because they contain occurrences of the costly set former

$$E_1 = \{x \in s \mid e(x) = e(z)\}.$$

Moreover, any hope of differentiating E_1 together with E, as Briggs did with his difference polynomials, would seem unfeasible, because E_1 is not directly differentiable with respect to arbitrary modifications to z. However, from (23) we know that $E'\{e(z)\} = E_1$, so that differentiation of E' will keep E_1 available regardless of how z is modified. Differentiation of E' together with E will be profitable, since the extended differential $\partial^- E', E\langle s \textbf{ with:=} z;\rangle$, for example, is just

```
if #E'{e(z)} = 0 then
  E with:= e(z);
end if;
E' with:= [e(z), z];
```

which executes in $O(1)$ steps. The computational cost of $\partial^- E', E\langle s \textbf{ less:=} z;\rangle$ is also $O(1)$.

Of course, it is important to bound the number of auxiliary expressions that must be differentiated to differentiate each elementary expression. In Appendix B, we present a variety of elementary expressions that require further differentiation of costly subexpressions introduced within derivative code (these costly subexpressions are underlined). However, in the worst case only three additional auxiliary expressions must be differentiated (see Rule E2 of Appendix B).

Our method of handling E_1 by differentiation of the auxiliary expression E' illustrates a general technique (called "discontinuity removal" in [33, pp. 7, 91–107, 155–157]) based on Earley's technique of iterator inversion [13] for handling expressions that are not directly differentiable with respect to changes in some of their parameters. The basic idea is captured in the following observation about the example just presented: the set $D_{e(z)} = \{e(y) : y \in s\}$ includes all values of $e(z)$ for which the set $E_1 = \{x \in s \mid e(x) = e(z)\}$ is nonempty. Consequently, we can store all the significant values of E_1 corresponding to each value $c = e(z)$ belonging to $D_{e(z)}$ within the expression

$$E'' = \{[c, x] : c \in D_{e(z)}, x \in \{w \in s \mid e(w) = c\}\}.$$

Since E'' computes the same set as the computationally more efficient differentiable expression E', we can keep all potentially nonempty expressions $E_1 = E'\{e(z)\}$ available by profitable differentiation. More generally, whenever an expression $E_3 = f(x_1, \ldots, x_n, q)$ is not directly differentiable with respect to

modifications to a parameter q, we can often profitably differentiate a simplified variant of another expression $E_4 = \{[c, f(x_1, \ldots, x_n, c)] : c \in D_q\}$ where D_q is a set of all q values outside of which E will be equal to some constant (such as the empty set). For additional examples of this technique, see Appendix B, Rules C1, C2, D2, E2, and H2.

We have already observed that the set formers

$$E_1 = \{x \in s \mid k(x)\}$$

and

$$E_2 = \{e(x) : x \in s\}$$

are differentiable with respect to changes in s. More interestingly, these expressions are even differentiable relative to indexed assignments

$$f(y) := z; \tag{26}$$

to dynamic maps f that appear only as retrievals occurring within k and e, and when each such retrieval depends on x. Presentation of these derivative rules also illustrates the importance of postderivatives.

In the case of E_1, let all of the distinguishable f retrieval terms occurring within k be denoted

$$f(p_1(x)), \ldots, f(p_r(x))$$

where $p_i(x)$ represents the argument expression of the ith retrieval term. We note, first of all, that the set

$$E_3 = \{x \in s \mid y \in \{p_1(x), \ldots, p_r(x)\}\}$$

contains all those elements of s for which the value of the Boolean subpart $k(x)$ occurring within E_1 can change from true to false or from false to true as a result of the indexed assignment (26). We refer to E_3 as the "tunnel set" of E_1 with respect to (26) and use it in the following lemmas to derive a derivative rule for E_1.

LEMMA 1. *E_3 is not spoiled by* (26).

PROOF. Let d be the maximum depth of nesting of f terms contained within other f terms occurring within an expression e (e.g., the f depth of the term $f(g(f(x + f(0))))$ is 3). Let w belong to E_3 just prior to the change (26). Then w belongs to s, and $y = p_k(w)$ for some $k = 1 \ldots r$. If we choose this k such that $p_k(w)$ has a minimal f depth, then y must equal $p_k(w)$ after (26) is executed. For otherwise, an f term $f(p_j(w))$ occurring within $p_k(w)$ would be spoiled by the assignment (26). And this implies that, for some i different from k, $p_i(w)$ equals y prior to (26) and has smaller f depth than $p_k(w)$—a contradiction. □

LEMMA 2. *The following code block can be used to represent the differential* $\partial E_1\langle f(y) := z;\rangle$:

$$\begin{aligned}&\partial^- E_1\langle s -:= \{x \in s \mid y \in \{p_1(x), \ldots, p_r(x)\}\};\rangle\\ &f(y) := z;\\ &\partial^- E_1\langle s +:= \{x \in s \mid y \in \{p_1(x), \ldots, p_r(x)\}\};\rangle\end{aligned} \tag{27}$$

PROOF. Follows immediately from Lemma 1. □

Because the set former

$$E_3 = \{x \in s \mid y \in \{p_1(x), \ldots, p_r(x)\}\}$$

appearing in (27) requires $O(\#s \times \text{cost}(k))$ steps to compute (which is the same as the cost of a full calculation of E_1), the differential code (27) does not look promising. Furthermore, because the value of y used in (27) is not predictable, it appears at first glance that a full calculation of E_3 (which is not differentiable with respect to modifications of y) cannot be easily avoided within (27). Fortunately, however, we can make all full calculations of E_3 redundant within (27) (so that these calculations can be eliminated) by maintaining the auxiliary expression

$$A_0 = \{[w, x] : x \in s, w \in \{p_1(x), \ldots, p_r(x)\}\}$$

differentially (note that $A_0\{y\}$ equals $E_3(y)$ for all y). Lemma 3, below, gives conditions under which A_0 will be differentiable with respect to indexed assignments (26) and supports Theorem 3, which asserts that the differential $\partial E_1, A_0\langle f(y) := z;\rangle$ can be performed efficiently.

LEMMA 3. *Whenever* $\#\{x \in s \mid p_i(x) = y\} = O(1)$ *for* $i = 1..r$ *and all* y, $\exists n > 0$ *such that the set former*

$$A_0 = \{[w, x] : x \in s, w \in \{p_1(x), \ldots, p_r(x)\}\}$$

is differentiable relative to modifications of the form s **with**:= z, s **less**:= z, *and* $f(y) := z$ *for all values* $\#s > n$.

PROOF. By easy generalization of the derivative rules (18), we see that

$$\partial^- A_0\langle s \textbf{ with}:= z;\rangle = (\forall w \in \{p_1(z), \ldots, p_r(z)\})\ A_0\{w\} \textbf{ with}:= z; \textbf{ end } \forall;$$

and

$$\partial^- A_0\langle s \textbf{ less}:= z;\rangle = (\forall w \in \{p_1(z), \ldots, p_r(z)\})\ A_0\{w\} \textbf{ less}:= z; \textbf{ end } \forall;$$

in which both of the derivative code blocks above require $O(r + \text{cost}(k))$ steps to execute (where k is the Boolean-valued subpart of the expression E_1).

To handle $\partial A_0\langle f(y) := z;\rangle$, we first note that the expression A_0 is formed from E_1 in such a way that, for every retrieval term $f(p(x))$ occurring within the subexpressions $p_i(x)$, $i = 1..r$, of A_0, there exists some j, $j = 1..r$, for which the terms $p(x)$ and $p_j(x)$ are identical. Among other things, this implies that, when r equals 1, the expression

$$A_0 = \{[w, x] : x \in s, w \in \{p_1(x)\}\}$$

has an f depth of 0. Otherwise, if $f(p(x))$ were a term occurring within $p_1(x)$, the terms $p(x)$ and $p_1(x)$ would be identical; and this is clearly impossible.

Let $t(r)$ be an upper bound on the estimated cost of computing the differential $\partial A_0\langle f(y) := z;\rangle$. By preceding remarks, when r equals 1, A_0 does not involve f, so that the derivative of A_0 with respect to (26) is empty and $t(1) = 0$. Also, when A_0 has f depth 0, $t(r) = 0$ for any r.

To determine $t(r)$ for the case when $r > 0$ and A_0 has an f depth $d > 0$, we first suppose that the distinguishable f terms occurring within A_0 are $f(p_1(x)), \ldots, f(p_{r_1}(x))$. By a minor extension of Lemma 2, the differential $\partial A_0\langle f(y) := z;\rangle$ can be realized by

$$\begin{array}{l}\partial^- A_0\langle s -:= \{x \in s \mid y \in \{p_1(x), \ldots, p_{r_1}(x)\}\}\rangle \\ f(y) := z; \\ \partial^- A_0\langle s +:= \{x \in s \mid y \in \{p_1(x), \ldots, p_{r_1}(x)\}\}\rangle\end{array} \tag{28}$$

Since (by Lemma 1) the two tunnel sets $A_0\{y\}$ and

$$E_3 = \{x \in s \mid y \in \{p_1(x), \ldots, p_{r_1}(x)\}\}$$

are not spoiled by (26), and since E_3 is contained in $A_0\{y\}$, the differential code (28) can be rewritten

$$\begin{array}{l}\partial^- A_0\langle s -:= A_0\{y\};\rangle \\ f(y) := z; \\ \partial^- A_0\langle s +:= A_0\{y\};\rangle\end{array}$$

which expands into the following code:

$$\begin{array}{l}(\forall w \in A_0\{y\}, u \in \{p_1(w), \ldots, p_r(w)\} \mid u \neq y) \\ \quad A_0\{u\} \textbf{ less}:= w; \\ \textbf{end } \forall; \\ f(y) := z; \\ (\forall w \in A_0\{y\}, u \in \{p_1(w), \ldots, p_r(w)\} \mid u \neq y) \\ \quad A_0\{u\} \textbf{ with}:= w; \\ \textbf{end } \forall;\end{array} \tag{29}$$

The computational cost of (29) is

$$t(r) = O(r^2 + r \times \text{cost}(k)),$$

while a fresh calculation of A_0 costs $O(\#s \times \text{cost}(k))$. Thus, A_0 is differentiable with respect to indexed assignments to f for sufficiently large $\#s$. □

THEOREM 3. *Whenever* $\#\{x \in s \mid p_i(x) = y\} = O(1)$ *for* $i = 1..r$ *and all* y, $\exists n > 0$ *such that* E_1 *is differentiable with respect to indexed assignments* (26) *for all values* $\#s > n$.

PROOF. It follows from Lemma 2 and Lemma 3 that the differential

$$\partial E_1, A_0\langle f(y) := z;\rangle$$

can be realized by the block

$$\begin{array}{l}\partial^- E_1, A_0\langle s -:= A_0\{y\};\rangle \\ f(y) := z; \\ \partial^- E_1, A_0\langle s +:= A_0\{y\};\rangle\end{array}$$

representing the following SETL code block:

```
(∀w ∈ A0{y}, u ∈ {p1(w), ..., pr(w)} | u ≠ y)
  A0{u} less:= w;
end ∀;
(∀w ∈ A0{y})
  if k(w) then
    E1 less:= w;
  end if;
end ∀;
f(y) := z;                                                    (30)
(∀w ∈ A0{y})
  if k(w) then
    E1 with:= w;
  end if;
end ∀;
(∀w ∈ A0{y}, u ∈ {p1(w), ..., pr(w)} | u ≠ y)
  A0{u} with:= w;
end ∀;
```

Since the computational cost of (30) is $O(r^2 + r \times \text{cost}(k))$, while the cost of a fresh calculation of E_1 is $O(\#s \times \text{cost}(k))$, E_1 is differentiable relative to indexed assignments to f when $\#s$ is sufficiently large. □

The preceding arguments demonstrating that E_1 is differentiable with respect to (26) can also be applied to E_2. Generalization of Theorem 3 to the case of indexed assignment $f(y_1, \ldots, y_n) := z$ to multiparameter maps f can be found in [33, pp. 46–48].

We have now given quite a number of illustrative examples of efficient derivatives and can proceed to analyze conditions under which finite differencing can improve code. As in Fong and Ullman's earlier work [15, 17], we require set theoretic finite differencing to yield asymptotic speedup. However, our differencing technique, the required speedup, and the theoretical characterization of conditions (enforced on both preprocessing and differentiation) under which this speedup can be obtained are different from, yet also complementary to, theirs.

Without loss of generality, consider differentiation of a single nonelementary expression f occurring within a program loop L restricted by the boundedness requirement. Suppose that a chain J of expressions differentiable with respect to L are used to reduce f. Then, if we assume that f is calculated more frequently within L than in code occurring outside L, the following preprocessing conditions must hold:

1. Under the most favorable condition, the initial evaluation of all the expressions within J on entry to L should require the same asymptotic cost as a single initial calculation of f within L.
2. When initialization costs are higher than this, as can happen when J involves auxiliary expressions, asymptotic speedup can still be expected when the preprocessing costs do not exceed the asymptotic costs of code in the region just outside L.
3. Finally, when large loop iteration is expected, we can sometimes perform more costly preprocessing without sacrificing speedup; for example, the initialization for Rule J1 Method 2 in Appendix B involves sorting and was used effectively to obtain a logarithmic speedup of the bankers algorithm (see Appendix A3).

While the preceding conditions prevent preprocessing operations from retarding the asymptotic running time of a program, either of the following two conditions on the differential $\partial J\langle L\rangle$ will ensure asymptotic speedup for the work involved in computing f within L:

1. Ideally, when each derivative code block (associated with an elementary expression within J) used to form the differential of J with respect to L requires only a constant factor to compute (as is the case with most of the derivative rules found in Appendix B), the cost of computing each extended pre- and postderivative $\partial^- J\langle dx\rangle$ and $\partial^+ J\langle dx\rangle$ for every modification dx (occurring in L) to a variable x on which J depends will also be just a constant factor.
2. When the asymptotic cost of computing f is greater than this, then our assumptions about loop boundedness and relative execution frequency of f will ensure that the cost of maintaining the value of f within the differential $\partial J\langle L\rangle$ will be asymptotically less than the cost of calculating f within L.
3. When the preceding condition does not hold, asymptotic improvement can still arise if another, more powerful condition holds. Suppose that f is formed by composition and parameter substitution from the elementary expressions given in Appendix B. Suppose also that, within L, only a single variable s on which f depends is modified; suppose, finally, that within L each modification ds to s, and, hence, each derivative block pair $\partial^- J\langle ds\rangle$ and $\partial^+ J\langle ds\rangle$, is of the same form (e.g., s could be set-valued and monotonically increasing (respectively, decreasing) within L). Then the overall cost of maintaining all of the expressions of J along any path from entry to exit of L and lying entirely within L will often be of the same asymptotic order as a single full computation of all the expressions within J at either the initial or final value of s (we call this cost *worst-case-cost*(J)). In this case asymptotic improvement will occur when worst-case-cost$(J) = O$(worst-case-cost(f)) or worst-case-cost$(J) = O$(preprocessing cost for J), and such improvement will occur even without the boundedness requirement.

Since all of our derivative rules can be adapted to Fong's deferred update approach, it is useful to make a few comparative remarks. Fong handles the issue of safety better than we do, since her expressions, differentiated with respect to a loop L, are only kept available at points where they are used in L. However, both methods face the same safety problems in handling preprocessing. Maintenance of the difference sets allows Fong to deal with interesting copy optimizations that seem infeasible for us. Nevertheless, for contexts where our approach applies, the chain rule ordering serves to eliminate potentially costly copy operations on sets and tuples. Neglecting the cost of maintaining difference sets, her deferred derivatives should be expected to cost no more, and in many cases less, than our derivatives. However, maintenance of the difference sets requires additional space and time costs that could make differencing twice as expensive for her as for us. There are also situations where the maintenance of difference sets will be relatively easy, and Fong's method can work better than ours, as, for example, when loops allow branching to exit before differentiable expressions are encountered but after modifications to parameters on which they depend are encountered. However, since we intend to write our code at a high level of abstraction, a great deal of explicit branching can be avoided. Of course, application of the chain rule will introduce more complicated branching, and further

transformation can be expected to produce even greater complexity of control flow as the program becomes progressively more efficient.

5. ALGORITHM IMPROVEMENT

5.1 Generalities

Before presenting a full case study of algorithmic improvement by finite differencing, we note that there exists a whole class of transitive closure algorithms that are amenable to our transformations. The main part of such transitive closure algorithms typically consists of **while** loops that iterate a block of code until an existential quantifier becomes false, that is, that have the following general form:

```
$ initialize variables
(while ∃x ∈ s | k(x))      $ k is a Boolean expression
  block(x)
end while;
```
(31)

where "block" involves definitions of the forms s **with**:= x and s **less**:= x to sets s, indexed assignments $f(y_1, \ldots, y_n) := z$ to maps f that also have occurrences within k, and perhaps other kinds of changes to variables on which k depends (we also assume that block contains uses of x). Based on the informal measure of computational cost given in Table II, the expense of evaluating the existential quantifier

$$\exists x \in s \mid k(x)$$

within (31) is $O(\#s \times \text{cost}(k) \times n)$ where n is the number of loop iterations.

The method of differencing will often be able to transform (31) into a faster "workset" version,

```
$ initialize variables
workset := {x ∈ s | k(x)};
(while ∃x ∈ workset)
  block'(x)
end while;
```
(32)

where block′ is the differential of *workset* with respect to block. We expect that the cost of executing a single cycle of block′ will differ from the cost of executing block by only a constant factor. Moreover, the potentially costly search through s within the **while**-loop predicate of (31) can be avoided in (32) at a cost of $O(\#s \times \text{cost}(k))$ or sometimes $O(\#s \times \log \#s \times \text{cost}(k))$ in preprocessing (i.e., evaluation of *workset* on entrance to the **while** loop). The cost of keeping *workset* available in block is thus

$$O(\#s \times \text{cost}(k) + n \times \text{cost}(k))$$

or

$$O(\#s \times \log \#s \times \text{cost}(k) + n \times \text{cost}(k)),$$

which generally represents improvement. (We assume that the parameter $\#s$ in the cost estimate just above is some worst-case value.)

As is shown in [33, pp. 114–152], various sorting, parsing, graph, and general problem-solving algorithms can be written in the form (31) and transformed by our method into the form (32), in which the text of block′ will often be ten times

larger than block and much more complex. Some examples of high-level algorithms (31) that we have differentiated are found in Appendix A.

The finite difference techniques described in this paper allow algorithms to be written in a "high style" in which complicated manipulation of worksets can be avoided at no cost, since these methods can directly generate faster algorithms from these "high style" versions. The perfection of our methods will therefore enable programmers to use powerful high-level dictions to write clear high-level programs that can be transformed routinely into more efficient low-level versions. This will facilitate, among other things, correctness proofs of programs, since, for example, we can expect to prove undifferentiated programs of form (31) correct more easily than their more complicated workset counterparts (32).

5.2 Extended Example: An Algorithm to Find the Center of a Free Tree

The connection between differencing and an efficient version of an algorithm to find the center of a free tree was first observed by Sridharan [44]. In this section we use our finite differencing rules to transform an inefficient version of an algorithm to find the center of a free tree into a highly efficient variant.

A free tree T is defined as a connected, undirected, acyclic graph. A leaf of T is a node having only one adjacent node. If T consists of a single node n, then the center of T is n; if T consists of only two nodes, n_1 and n_2, then the center of T is the set $\{n_1, n_2\}$; if T consists of more than two nodes, its center is the same as the center of a tree T' formed from T by removing all the leaves of T.

Our initial algorithm specification represents T as a symmetric edge set E and a set of nodes S. E maps each node n of S into the set $E\{n\}$ of adjacent nodes. The algorithm proceeds by repeatedly searching for the leaves of T and removing these leaves from T as long as the number of nodes in T is greater than 2. Speeding up this initial algorithm entails differentiating the search for the leaves.

To begin the algorithm development, we consider SETL Program 1. In order to prepare this program for finite differencing, we apply two syntactic transformations,

$$E\{x\} * S \rightarrow \{y \in E\{x\} \mid y \in S\}$$

and

$$S -:= \{x \in S \mid \#\{y \in E\{x\} \mid y \in S\} = 1\} \rightarrow (\forall n \in \{x \in S \mid \#\{y \in E\{x\} \mid y \in S\} = 1\})$$
$$S \text{ less:= } n;$$
$$\textbf{end } \forall;$$

which places Program 1 into the canonical form shown as Program 2 (i.e., a form for which all set intersections and deletions have been turned into set formers, and set additions and deletions are implemented at a lower level in terms of set-element additions and deletions).

Analysis of Program 2 can detect three expressions differentiable within the **while** loop. These are

$enew\{x\} = \{y \in E\{x\} \mid y \in S\}$	of the form of Rule D2 in Appendix B;
$numnew(x) = \#enew\{x\}$	of the form of Rule M2 in Appendix B;
$leaves = \{x \in S \mid numnew(x) = 1\}$	of the form of Rule C1 Method 2 in Appendix B

```
Line
 no.
      repr E: sym map;                          $ declaration: E is symmetric
  1   read(E);                                  $ read the free tree
  2   S := domain E;                            $ compute the nodes of the tree
  3   (while #S > 2)                            $ remove leaves from S
  4     S -:= {x ∈ S | #(E{x} * S) = 1};
  5   end while;
  6   print(S);
```

Program 1

```
Line
 no.
      repr E: sym map;
  1   read(E);                                  $ read the free tree
  2   S := domain E;                            $ compute the nodes of the tree
  3   (while #S > 2)
  4     (∀n ∈ {x ∈ S | #{y ∈ E{x} | y ∈ S} = 1})
  5       S less:= n;
  6     end ∀;
  7   end while;
  8   print(S);
```

Program 2

where we define *enew* and *numnew* in the following way:

$$enew = \{[x, y] \in E \mid y \in S\};$$
$$numnew = \{[x, \#enew\{x\}] : x \in \textbf{domain } enew\}.$$

If we let B stand for the **while** loop forming the main body of Program 2, then our final efficient version of the algorithm should be generated by first transforming B into the code represented by

achieve $enew = \{[x, y] \in E \mid y \in S\};$
achieve $numnew = \{[x, \#enew\{x\}] : x \in \textbf{domain } enew\};$
achieve $leaves = \{x \in S \mid numnew(x) = 1\};$
$\partial leaves, numnew, enew\langle B \rangle$

and then transforming the program P which results into a final form

$\text{Clean}\langle \text{Init}\langle P \rangle \rangle$

Based on the static performance analysis for differencing presented in the preceding section, it is easy to predict at this point that the differentiated tree center algorithm will run in $O(\#E)$ steps. Preprocessing costs (determined by treating the three **achieve** statements above as three separate assignments) are clearly $O(\#E)$. Because the set S is monotonically decreasing within the **while** loop of Program 2, the cumulative costs of executing derivative code within this loop will be asymptotically the same as the preprocessing costs. The remaining statements 1, 2, 5, and 8 of Program 2 can contribute no more than $O(\#E)$ steps to the cost of its differentiated form.

```
Line
no.
     repr E: sym map;
  1  read(E);
  2  S := domain E;
  3  achieve enew = {[x, y] ∈ E | y ∈ S};
  4  achieve numnew = {[x, #enew{x}] : x ∈ domain enew};
  5  achieve leaves = {x ∈ S | numnew(x) = 1};
  6  (while #S > 2)
  7    (∀n ∈ {x ∈ S | #enew{x} = 1})
  8      (∀u ∈ E{n})
  9        enew{u} less:= n;
 10      end ∀;
 11      S less:= n;
 12    end ∀;
 13  end while;
 14  print(S);
```

Program 3

Recall that the chain rule transformation $\partial leaves, numnew, enew\langle B\rangle$ is defined recursively as $\partial leaves, numnew\langle \partial enew\langle B\rangle\rangle$. To produce $\partial enew\langle B\rangle$ we only need to insert the prederivative code, $\partial^- enew\langle S$ **less:=** $n;\rangle$, just before line 5 of Program 2. According to Rule D2 in Appendix B, this prederivative is

```
(∀u ∈ {x ∈ domain E | n ∈ E{x}})
  enew{u} less:= n;                                    (33)
end ∀;
```

Note, however, that, since E is a symmetric relation, we can replace the set former $\{x \in$ **domain** $E \mid n \in E\{x\}\}$ appearing within (33) by $E\{n\}$. This follows from the fact that E equals its inverse when it is symmetric. At this point we can also replace the occurrence of $\{y \in E\{x\} \mid y \in S\}$ within Program 2 by the retrieval $enew\{x\}$. The algorithm now has the transitional form shown as Program 3.

Next, we differentiate *numnew* relative to the **while** loop B_1 at lines 6–13 of Program 3. Then, to determine $\partial numnew\langle B_1\rangle$, we only need to introduce $\partial^- numnew\langle enew\{u\}$ **less:=** $n;\rangle$ just before line 9 of Program 3; by Rule M2, this code is just

```
numnew(u) -:= 1;
```

At this point the occurrence of $\#enew\{x\}$ at line 7 of Program 3 can be replaced by an occurrence of $numnew(x)$. The tree center algorithm that results from the preceding transformational steps is Program 4.

The final step of the chain rule involves differentiation of leaves. Note that this entails transforming the **while** loop (which we designate B_2) at lines 6–14 of Program 4 into the code implied by $\partial leaves\langle B_2\rangle$. It is easy to see that differentiation of leaves depends on determining $\partial^- leaves\langle numnew(u)$ $-:= 1;\rangle$ and $\partial^- leaves\langle S$ **less:=** $n;\rangle$. By Rule C1 Method 2, the code implied by $\partial^- leaves\langle S$ **less:=** $n;\rangle$ is

```
if numnew(n) = 1 then
  leaves less:= n;
end if;
```

```
Line
 no.
     repr E: sym map;
     repr numnew: smap( ) ↑ 0;      $ declares: x ∉ domain numnew ⇒ numnew(x) = 0
  1  read(E);
  2  S := domain E;
  3  achieve enew = {[x, y] ∈ E | y ∈ S};
  4  achieve numnew = {[x, #enew{x}] : x ∈ domain enew};
  5  achieve leaves = {x ∈ S | numnew(x) = 1};
  6  (while #S > 2)
  7    (∀n ∈ {x ∈ S | numnew(x) = 1})
  8      (∀u ∈ E{n})
  9        numnew(u) −:= 1;
 10        enew{u} less:= n;
 11      end ∀;
 12      S less:= n;
 13    end ∀;
 14  end while;
 15  print(S);
```

Program 4

```
Line
 no.
     repr E: sym map;
     repr numnew: smap( ) ↑ 0;
  1  read(E);
  2  S := domain E;
  3  achieve enew = {[x, y] ∈ E | y ∈ S};
  4  achieve numnew = {[x, #enew{x}] : x ∈ domain enew};
  5  achieve leaves = {x ∈ S | numnew(x) = 1};
  6  (while #S > 2)
  7    (∀n ∈ leaves)
               $ make a copy of leaves on entrance to loop, and iterate over the copy
  8      (∀u ∈ E{n})
  9        if u ∈ S then
 10          if numnew(u) = 1 then
 11            leaves less:= u;
 12          elseif numnew(u) = 1 + 1 then
 13            leaves with:= u;
 14          end if;
 15        end if;
 16        numnew(u) −:= 1;
 17        enew{u} less:= n;
 18      end ∀:
 19      if numnew(n) = 1 then
 20        leaves less:= n;
 21      end if;
 22      S less:= n;
 23    end ∀;
 24  end while;
 25  print(S);
```

Program 5

while the code implied by ∂^{-}*leaves*⟨*numnew*(*u*) −:= 1;⟩ is

```
if u ∈ S then
  if numnew(u) = 1 then
    leaves less:= u;
  elseif numnew(u) = 1 + 1 then
    leaves with:= u;
  end if;
end if;
```

After this, the occurrence of $\{x \in S \mid numnew(x) = 1\}$ within Program 4 is redundant and can be replaced by the occurrence of *leaves*. The result is Program 5.

Note that the **forall** loop L at lines 7–23 of Program 5 involves an iteration over the set *leaves*, and *leaves* is also modified within the body of L. In accordance with SETL semantics [41], we assume that a copy of *leaves* is made on entrance to L; iteration proceeds over this copy (which cannot be modified within the body of L). Any occurrences of the variable *leaves* within the body of L refer to the original instance of *leaves* and not to the copy.

At this point initialization can be worked out on all of Program 5. The Init transformation replaces the sequence of **achieve** statements at lines 3–5 by a valid code block that makes *enew*, *numnew*, and *leaves* available on exit. A full implementation of Init is presented in the next section; the actual block produced by Init is as follows:

```
numnew := { };
enew := { };
(∀[x, y] ∈ E)
  if y ∈ S then
    numnew(x) +:= 1;
    enew{x} with:= y;
  end if;                                          (34)
end ∀;
leaves := { };
(∀x ∈ S)
  if numnew(x) = 1 then
    leaves with:= x;
  end if;
end ∀;
```

Note that the initialization block (34) first evaluates *numnew* and *enew* together. Afterward, *leaves* is constructed by itself. Note, also, that construction of the three virtual variables is incremental and can actually be defined in terms of our differential operator; that is, (34) can be generated by

```
∂numnew⟨∂⁻enew⟨E := { };⟩
          (∀[x, y] ∈ E)
            ∂⁻enew⟨E with:= [x, y];⟩
          end ∀;⟩
∂⁻leaves⟨S := { };⟩
(∀x ∈ S)
  ∂⁻leaves⟨S with:= x;⟩
end ∀;
```

```
Line
 no.
     repr E: sym map;
     repr numnew: smap( ) ↑ 0;
  1  read(E);
  2  S := domain E;
  3  numnew := { };
  4  (∀[x, y] ∈ E)
  5    if y ∈ S then
  6      numnew(x) +:= 1;
  7    end if;
  8  end ∀;
  9. leaves := { };
 10  (∀x ∈ S)
 11    if numnew(x) = 1 then
 12      leaves with:= x;
 13    end if;
 14  end ∀;
 15  (while #S > 2)
 16    (∀n ∈ leaves)
 17      (∀u ∈ E{n})
 18        if u ∈ S then
 19          if numnew(u) = 1 then
 20            leaves less:= u;
 21          elseif numnew(u) = 1 + 1 then
 22            leaves with:= u;
 23          end if;
 24        end if;
 25        numnew(u) −:= 1;
 26      end ∀;
 27      if numnew(n) = 1 then
 28        leaves less:= n;
 29      end if;
 30      S less:= n;
 31    end ∀;
 32  end while;
 33  print(S);
```

Program 6

At this point, dead-code elimination can be performed. Our dead-code elimination procedure will regard the output statement **print**(S) at line 25 and the input statement **read**(E) at line 1 of Program 5 as "essential" statements. Any statement that can contribute to the value of S used in this **print** statement is also considered essential. Observation shows that all assignments to *enew* are unessential and thus can be eliminated as dead code. The result of all these transformational steps is Program 6, the final form of our algorithm.

It seems likely that current technology exists to mechanize finite differencing sufficiently to carry out the development of the tree center algorithm from Program 1 to Program 6 automatically. This full transformation yields a considerable speedup, since Program 1 runs in $O(\#S \times \#E)$ steps, while Program 6 runs in $O(\#E)$ steps. Moreover, the soundness of our transformations, along with a standard correctness proof of Program 1, proves the correctness of Program 6, a less perspicuous but more efficient equivalent algorithm.

The reader will note that several obvious minor improvements to Program 6 can be made. For example, we can place the statement

assert $y \in s$; (35)

just prior to line 5 in order to reduce the conditional statement at lines 5–7 to

$numnew(x)$ +:= 1;

Based on value flow analysis [40], a high-level optimizer can even perform this transformation automatically. Likewise, standard constant folding can replace the term 1 + 1 at line 21 by the constant 2.

More significant, if we provide input assumptions that E is a free tree within Program 1, it is also possible to justify the assertions

assert $numnew(u) \neq 1$;

prior to line 19 and

assert $numnew(n) = 1$;

just before line 27; and this allows us to simplify the conditional statements at lines 19 and 27. Consequently, *leaves* can be represented as a queue, and the copy of *leaves* required on entry to the **forall** loop at line 16 can be avoided. Unfortunately, the current state of correctness technology is not sufficiently advanced to make these additional improvements completely automatic.

Further automatic improvement by a large constant factor may be achieved by a combination of techniques, the most plausible of which is Schwartz's method of data-structure selection (see [12, 37]). This final transformation will produce a tree center program at about the level of PASCAL.

6. IMPLEMENTATION ALGORITHMS

In order to complete our description of finite differencing, we need to specify the transformations Init and Clean.

6.1 Init (Initialization Transformation)

If B is a code block, then $B' = \text{Init}\langle B \rangle$ is a new code block formed from B by transforming every contiguous sequence of **achieve** statements of the form

achieve $c = f(x_1, \ldots, x_n)$; (36)

into a code block that evaluates each expression f in the sequence and stores its value into c.

The Init transformation, as discussed in [33, pp. 43, 102–104], uncovers a new way of handling a particular kind of loop jamming in a programming language setting. To illustrate this idea, we first consider the problem of initializing the two expressions

$$\begin{aligned} c_1 &= \{x \in s \mid k_1(x)\}; \\ c_2 &= \{x \in c_1 \mid k_2(x)\}. \end{aligned} \tag{37}$$

A straightforward way to initialize (37) is to execute the assignments

$$\begin{array}{l} c_1 := \{x \in s \mid k_1(x)\}; \\ c_2 := \{x \in c_1 \mid k_2(x)\}; \end{array} \tag{38}$$

where the order of execution within (38) conforms to the obvious rules of dependency. However, the computation (38) requires two full iterations through possibly large sets.

A better approach arises after we consider initializing c_1 in the following incremental way:

$$\begin{array}{l} c_1 := \{\}; \\ (\forall x \in s) \\ \quad \textbf{if } k_1(x) \textbf{ then} \\ \quad\quad c_1 \textbf{ with:}= x; \\ \quad \textbf{end if}; \\ \textbf{end } \forall; \end{array} \tag{39}$$

since differentiation of c_2 with respect to code block (39) yields an efficient initialization of c_2 jammed into the loop used to initialize c_1. Using our differential operator, we can specify the collective differential initialization of c_1 and c_2 as follows:

$$\begin{array}{l} \partial c_2\langle \partial^- c_1\langle s := \{\};\rangle\rangle \\ (\forall x \in s) \\ \quad \partial c_2\langle \partial^- c_1\langle s \textbf{ with:}= x;\rangle\rangle \\ \textbf{end } \forall; \end{array} \tag{40}$$

which is the same as

$$\begin{array}{l} c_2 := \{\}; \\ c_1 := \{\}; \\ (\forall x \in s) \\ \quad \textbf{if } k_1(x) \textbf{ then} \\ \quad\quad \textbf{if } k_2(x) \textbf{ then} \\ \quad\quad\quad c_2 \textbf{ with:}= x; \\ \quad\quad \textbf{end if}; \\ \quad\quad c_1 \textbf{ with:}= x; \\ \quad \textbf{end if}; \\ \textbf{end } \forall; \end{array} \tag{41}$$

We say that (39) represents an *expansion* of c_1 about s. It is convenient to abbreviate (39) as $\partial^- c_1\langle s := s;\rangle$, from which (39) can be derived using the following identities:

$$\partial^- c_1\langle s := s;\rangle = \partial^- c_1\langle s := \{\};\rangle \; \partial^- c_1\langle s +:= s;\rangle$$

$$\partial^- c_1\langle s +:= s;\rangle = (\forall x \in s) \; \partial^- c_1\langle s \textbf{ with:}= x;\rangle \; \textbf{end } \forall;$$

We call (40) an example of *vertical* jamming. Another kind of jamming is illustrated by initialization of the two independent expressions

$$\begin{array}{l} c_3 = \{x \in c_2 \mid k_3(x)\}; \\ c_4 = \{x \in c_2 \mid k_4(x)\}, \end{array} \tag{42}$$

together with the previous two expressions (37). Suppose that c_2 does not occur free in either k_3 or k_4. Then c_3 and c_4 may be initialized in any order following or in concert with the initialization of c_2. In the case where there are also no free occurrences of c_1 in k_3 or k_4, it is worthwhile to initialize c_1, c_2, c_3, and c_4 collectively. This is achieved by executing the following differential code:

$$\partial c_3,\ c_4,\ c_2\langle \partial^- c_1\langle s := s;\rangle\rangle \tag{43}$$

However, for the case when c_1 occurs free in k_3 and k_4, the code (43), though correct, will require execution of differential code for c_3 and c_4 with respect to modifications to c_2 and c_1. In this case, (43) will not offer the same speedup as in the previous case, and it is preferable to initialize c_3 and c_4 by executing

```
∂⁻c3⟨c2 := { };⟩
∂⁻c4⟨c2 := { };⟩
(∀x ∈ c2)
  ∂⁻c3⟨c2 with:= x;⟩
  ∂⁻c4⟨c2 with:= x;⟩
end ∀;
```
(44)

(which can be abbreviated $\partial^- c_4,\ c_3\langle c_2 := c_2;\rangle$) just after executing code block (40). Note that initialization of c_3 and c_4 within (43) and (44) represents an initialization of independent expressions, a phenomenon we denote *horizontal* jamming. Further insight into our notion of jamming by differential initialization may be gained from consideration of a more complicated example that is taken from the case study of the last section: initialization of

$$\begin{aligned} enew &= \{[x, y] \in e \mid y \in s\}; \\ numnew &= \{[x, \#enew\{x\}] : x \in \textbf{domain } enew\}; \\ leaves &= \{x \in s \mid numnew(x) = 1\}. \end{aligned} \tag{45}$$

As in the previous examples, jamming succeeds after *enew* is expanded about its parameter *e*; that is,

```
enew := { };
(∀[x, y] ∈ e)
  if y ∈ s then
    enew with:= [x, y];
  end if;
end ∀;
```
(46)

which is a realization of

$$\partial^- enew\langle e := e;\rangle \tag{47}$$

Observe that it is possible to differentiate *numnew* with respect to code block (47) and obtain the following efficient initialization of *enew* vertically jammed with *numnew* in a single loop:

```
numnew := { };
enew := { };
(∀[x, y] ∈ e)
  if y ∈ s then
    numnew(x) +:= 1;
    enew with:= [x, y];
  end if;
end ∀;
```
(48)

Note that *leaves* can also be constructed differentially with respect to (48). Unfortunately, in this case, the differential code will force an element x to be added to *leaves* whenever *numnew*(x) is set to 1; but, after *numnew*(x) is incremented to 2, x will be removed from *leaves*. To avoid such extraneous operations we will initialize *leaves* (by expansion around its parameter s) by itself immediately after (48).

Aside from the constant-factor speedup that can result from jamming, there is another aspect that is equally important: vertical jamming can eliminate expression dependency. Note, for example, that in the initialization code (48) *numnew* does not depend on *enew*, although there would be dependency if *numnew* and *enew* were initialized separately by the straightforward assignments,

$$\begin{array}{l} enew := \{[x, y] \in e \mid y \in s\} \\ numnew := \{[x, \#enew\{x\}] : x \in \textbf{domain } enew\} \end{array} \tag{49}$$

Consequently, if *numnew* is essential to a program, in the case of (49) the dependency of *numnew* on *enew* will force *enew* to be essential also. However, in the case of (48) the lack of dependency permits *enew* to be removed from (48) as dead code whenever it is not essential beyond its initialization block.

The preceding examples lead to the following general rule:

Rule 1. Differential initialization costs for an expression

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

jammed together with other expressions should be no worse than those of a full separate evaluation of $f(x_1, \ldots, x_n)$.

Adherence to Rule 1 is facilitated by following two particular "rules of thumb":

Rule 2. We only initialize an expression differentially with respect to a single parameter.

Rule 3. For each elementary expression $f(x_1, \ldots, x_n)$ we only allow f to be initialized differentially (or by separate expansion) with respect to certain of its parameters, called "expandable" parameters, for which the technique is most likely to be profitable.

Having said all this, we can now go on to specify the Init transformation. Consider initialization for a chain of differentiable expressions $E_i = f_i$, $i = 1 \ldots n$. The following steps will produce a block B that makes E_i, $i = 1 \ldots n$, available on exit. The block B will consist of a sequence of subblocks each of which is used to fully construct expressions incrementally with respect to a single expandable parameter.

(1) Let B start out as an empty code block.
(2) For each $E_i = f_i$, $i = 1 \ldots n$, let b be the last subblock of B that is used to initialize a virtual variable E_k on which f_i depends (and let $k = 0$ if there is no such subblock). Then the following three cases arise:

(a) *Vertical Jamming.* Suppose that $k > 0$ and that E_k is an expandable parameter of E_i (according to Rule 3). Suppose, also, that the code subblock b that initializes E_k includes no code that initializes any other parameter on which E_i depends (according to Rule 2). Then to initialize E_i we replace b with $\partial E_i\langle b \rangle$.

(b) *Horizontal Jamming.* If case (a) does not apply, let b be the first subblock of B that occurs after all of the subexpressions of f_i are initialized, and that is formed by expansion around an expandable parameter x of f_i; that is, b is of the form

$$\ldots \partial^- J\langle x := x;\rangle \cdots\rangle$$

where J is a sequence of virtual variables all jammed horizontally around x. Then to initialize E_i we replace b with

$$\ldots \partial^- E_i, J\langle x := x;\rangle \cdots\rangle.$$

(c) *Separate Expansion.* When neither of the other cases applies, we separately initialize E_i by expansion around one of its expandable parameters x and append the subblock $\partial^- E_i\langle x := x;\rangle$ to the end of B.

We note with regard to the "approximation" procedure just above that a more general procedure that produces an initialization block with the fewest number of loops (and also obeys Rules 2 and 3) is equivalent to an NP-hard DAG covering problem where the DAG reflects expression dependency. In fact, in the simplest case in which all expressions to be initialized are independent, the problem reduces to one of finding optimal horizontal jamming. This is equivalent to the NP-complete "hitting set" problem (cf. [19, p. 222]). Although the algorithm we have presented can result in a suboptimal solution, the solution will never be worse than a straightforward unjammed solution. Also, our algorithm can be made to run in time proportional to the size of its output.

The idea of horizontal and vertical expression jamming has been studied before in the contexts of programming languages by Burstall, Darlington, and Burge [6, 7], file processing systems by Morgenstern [32], and system construction by Feather [14, pp. 5-2–5-5], although the transformations found in those references are expressed and implemented differently than here. Morgenstern's work is particularly noteworthy, since the overhead costs involved in iterating through files stored on secondary storage are much greater than those involved in cycling through program loops residing in main memory. Thus, whenever Morgenstern's jamming techniques can eliminate a full iteration through a file, the constant-factor speedup in systems performance can be considerable. Morgenstern uses a dynamic programming algorithm to obtain profitable loop jamming.

6.2 Clean$\langle P\rangle$ (Cleanup Transformation)

The Clean transformation is applied to a full program P as the final step of finite differencing. It functions to sweep up the transformational debris that the differential operator leaves in its wake. Surprisingly, the major part of this cleanup procedure can be accomplished by standard dead-code elimination. This is because, whenever the differential operator is used to keep available an expression

$$c_1 = e_1(e_2)$$

which depends on an inner expression

$$c_2 = e_2,$$

c_2 will also be kept available. However, it is often the case that the diferential code used to maintain c_1 has no uses of c_2, which allows us to perform dead-code elimination on the differential code that maintains c_2.

Our Clean transformation can be specified using a variant of an algorithm given by Kennedy [25]. This algorithm uses a negative strategy in which all "essential" statements within a program P are determined. All other statements are considered dead and can be eliminated.

The Clean procedure begins by locating (within P) a small set *crit* of program points containing essential statements of P from which all other essential statements of P can be determined. Initially, *crit* will include all of the **print** and sequential **read** statements[3] of P that are reachable from the program entry point. The algorithm proceeds by adding new essential statements to *crit* according to a standard transitive closure process. When *crit* can no longer grow, the procedure halts.

New essential statements that are added to *crit* are determined using the *usetodef* map (cf. Section 3.4) and three "local" maps, *iuses*, *instof*, and *compound*, defined as follows:

1. *iuses.* If i is a program point containing a statement q, then $iuses\{i\}$ is the set of variable uses contained within q. The value of $iuses\{i\}$ is clear when q is a simple statement. When q is a conditional statement

if c_1 **then**
  B_1
elseif c_2 **then**
  ⋮
elseif c_n **then**
  B_n
else
  B_{n+1}
end if;

$iuses\{i\}$ is the set of uses occurring within $c_1, \ldots, c_n$. When q is a **while** loop, $iuses\{i\}$ contains only those uses within the condition of the loop. When q is a **forall** loop, $iuses\{i\}$ is the set of all uses within the loop iterator.

2. *instof.* If d is a definition, then $instof(d)$ is the program point of the statement q that contains d.

3. *compound.* If a statement q is contained within an immediately enclosing compound statement r, then $compound(q) = r$.

The Clean procedure is based on the following condition, under which a statement j that does not belong to *crit* can be added to *crit*:

$$\exists i \in crit,\ u \in iuses\{i\},\ d \in usetodef\{u\} \mid j \in instof\{d\} \textbf{ or } j = compound(i). \tag{50}$$

[3] The observation that sequential **read** statements must be included in the initial value of *crit* is due to Richard King. Consider this example: **read**(a); **read**(a); **print**(a);

For predicate (50) to hold, *crit* must contain a statement i that either

(1) contains a use u that is linked (via the *usetodef* map) to a definition d contained in the statement j or
(2) is immediately contained in a compound statement j.

The following high-level SETL program carries out the essential code detection phase of the Clean transformation:

```
(while ∃i ∈ (instof[usetodef[iuses[crit]]] + compound[crit]) − crit)
  crit with:= i;                                                    (51)
end while;
```

6.3 Finite Differencing Algorithm

We now sketch an algorithm that could actually automate all of the transformational steps given in our case study of Section 6.2. Certain implementation-level details (such as matching operations, macro expansion procedures, elementary expression form and derivative tables) are absent, however, and the reader is asked to refer to [33, pp. 71–114].

We assume that, before the finite differencing procedure can be applied, the code prestructure is in parse tree form, over which a control flow graph is imposed. Data flow analysis is worked out so that the *usetodef* and *deftouse* maps are defined, and type analysis is also performed (by the method of Tenenbaum [47]).

Algorithm: Automatic Finite Differencing

(1) Apply preparatory transformations (cf. [33, pp. 235–236]).
(2) Decompose the program into its loop structure $L_1, L_2, \ldots, L_n$ with the property $i < j \Rightarrow L_i$ is contained in L_j or $L_i \cap L_j = \{\}$ (cf. [2]).
(3) For $i = 1 \ldots n$ determine a chain J_i of differentiable expressions to be reduced within L_i, but not in any region enclosing L_i.
(4) For $i = n, n-1, \ldots, 1$ transform L_i into

    achieve $\Lambda_{(E=f)\in J_i}\, E = f$
    $\partial J_i\langle L_i\rangle$

(5) If P is the program that results from step (4), transform P into the final program

    Clean⟨Init⟨P⟩⟩

7. CONCLUSION

Finite differencing of applicative expressions extends an old mathematical idea to the general problem of algorithm optimization and, hence, to high-level-language implementation and design. The techniques we have discussed in this paper are likely to have an impact on a number of pragmatic issues related to both programming languages and databases. Many of these issues have yet to be fully explored.

Application of finite differencing to languages such as APL, SNOBOL, and even PL/I (whose string handling operations may be receptive to differencing techniques) should be worthwhile. Similarly, it should be useful to consider dictions for specification of derivative rules for user-defined operations and procedures. Such a capability would extend the utility of finite differencing to the

larger area of software development by facilitating the construction of large, modular, incremental programs.

Our techniques have been seen to offer new and efficient implementations of very high-level programming language dictions. In [33, pp. 157–159] it is shown how finite differencing can be used to implement fixed-point iterators that cause a code block to be executed repeatedly until there is no change of state. Finite differencing opens up new opportunities to implement exception handling. We have observed that generalized "on" conditions may be implemented efficiently by extending the differential to apply to conditional transfers as well as to applicative expressions.

Most applications of finite differencing that we have studied are based on the paradigm of differentiating costly expressions executed repeatedly within program loops L. Another paradigm discussed in [33, pp. 164–165] that has only recently been explored has to do with restructuring a program loop L containing uses of an expression e (that is not differentiable with respect to L) in order to make e differentiable with respect to L. This second paradigm suggests a useful strategy for improving the speed of a search through a power set in the following context:

$$\exists s \in \mathbf{pow}(e) \mid k(s).$$

Whenever k has subexpressions that are differentiable relative to element additions to and deletions from s, we can restructure the iteration through the power set by performing a depth-first search (through a tree in which the root represents $\{\}$, the successors of the root are all the singleton sets, etc.). This backtrack approach is still inefficient, but it avoids construction of $\mathbf{pow}(e)$ and allows subexpressions of k to be differentiated with respect to the incremental construction of s. This should represent an improvement.

Recently, the preceding idea has been pushed further by Sharir [43], who has developed a strategy based on Schwartz's *sinister assignment* [41] in which the depth-first search can be performed through a tree that has been pruned significantly. Sharir's methods provide a backtracking optimization which can speed up a nondeterministic algorithm that runs in somewhere between 2^n and n^n steps to an algorithm that runs in polynomial time.

In [31] finite differencing is applied to database view maintenance, integrity control, and exception handling. Some preliminary ideas concerning database systems that adapt their physical structures dynamically via finite differencing are discussed in [33, pp. 160–163].

Ultimately, the success of installing finite differencing as part of a conventional optimizing compiler may rest on the efficiency of the implementation. We have already implemented a semiautomatic finite differencing system (Rutgers Abstract Program Transformation System, or RAPTS) for a subset of SETL, and have used RAPTS to derive the tree center algorithm discussed here, as well as several more complicated algorithms. The results of our implementation will be reported in the near future.

Currently, we see two viable and complementary approaches to finite differencing, one that has been developed by Fong and Ullman [15, 17] and the other initiated in [33, 35] and developed further in the present paper. We believe that a unified approach to finite differencing that incorporates both methods should lead to conceptual and pragmatic improvements.

APPENDIX A. BASE FORM RUBBLE ALGORITHMS

In [39, 42] Schwartz coins the term "base form rubble" to denote the most concise form of an algorithm from which a concrete implementation-level variant may be derived without difficulty by manually selected "routine" transformations. In this appendix we present a sampling of rubble programs whose more complex optimized forms can be derived automatically by finite differencing. We provide the necessary insight into how speedup can be achieved for these algorithms, but to obtain the details involved in the actual transformational steps the reader should refer to [33, pp. 114–152].

A1. Knuth's Topological Sort

Perhaps the first example of a nontrivial algorithm transformed semimechanically by finite differencing is the topological sort case study given by Earley [13]. Many of the steps which Earley applied manually were later applied more systematically and also without manual intervention in [33, pp. 114–119] to the base form SETL algorithm given below. The input assumed by this algorithm is 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.

```
t := [ ];
(while ∃a ∈ s | (sp{a} * s) = { })     $ find a minimal element
  t with:= a;                          $ add it to the end of t
  s less:= a;                          $ obtain a new poset
end while;
```

Finite differencing will improve the algorithm above by differentiating the set of minimal elements

$$minset = \{x \in s \mid (sp\{x\} * s) = \{\ \}\}$$

so that the costly search involved in executing the existential quantifier can be avoided. Maintenance of the successor relation (which is the inverse of sp) is crucial to maintenance of *minset* and contributes to the order-of-magnitude speedup which the method yields.

A2. Transitive Closure

Another example closely related to topological sort is an algorithm to compute the image of a set s under transitive closure of a relation f. A succinct SETL version of this algorithm is

```
(while f[s] + s ≠ s)     $ while image(s) under f is not a subset of s,
  s +:= f[s];            $ augment s
end while;
```

Finite differencing will improve the running time of transitive closure by differentiation of

$$outset = \{x \in s \mid \#(f\{x\} - s) > 0\},$$

which denotes the set of elements $x \in s$ in which the image set $f\{x\}$ has values outside of s. But, in order to keep *outset* available, f^{-1} must also be kept available. The final optimized version will run in $O(\#f)$ steps.

A3. Habermann's Bankers Algorithm

While the previous two examples illustrate algorithm optimization by an order of magnitude in running times, finite differencing applied to the base form of the bankers algorithm given below will yield a logarithmic speedup in general, and an order-of-magnitude improvement if the preprocessing costs (dominated by a sort) can be neglected.

The general bankers algorithm can be used to detect deadlock among concurrent processes competing for resources in an operating system environment. This algorithm models resource allocation by bank "loans" to customers who make known demands. Different kinds of resources are represented by a set R of currency types. The concurrent processes are represented by a set *cus* of bank customers. For each kind i of currency R, $cash(i)$ represents the total amount of this currency still unallocated by the bank; $loan(i, c)$ is the loan of type i currency owed by a customer c; and $claim(i, c)$ is a customer's additional demand for currency type i. Once a customer's total demand is met, he will repay the bank his entire borrowed amount within a finite amount of time. If the bank can satisfy the demands of all of its customers, one at a time, then the initial state represented by *cash*, *cus*, *claim*, and *loan* is "safe"; that is, a deadlock can be avoided.

The algorithm follows a strategy in which the bank will try to meet the demands of any customer c whose claims can all be satisfied; that is, the condition

$$\forall i \in R \mid claim(i, c) \leq cash(i)$$

holds. The bank will then wait until c makes full repayment and is no longer a customer before scheduling any more customers. If all customers have been eliminated when the algorithm terminates, the original configuration of loans is "safe"; otherwise, it is not.

A base form version of the bankers algorithm can be written as follows:

```
(while ∃c ∈ cus | (∀i ∈ R | claim(i, c) ≤ cash(i))
  (∀i ∈ R)                          $ customer c pays back
    cash(i) +:= loan(i, c);         $ all of his loans
  end ∀;
  cus less:= c;                     $ and goes away
end while;
```

This algorithm executes in time proportional to

$$(\#cus)^2 \times \#R.$$

Differentiation of the set of good customers

$$\{c \in cus \mid \#\{i \in R \mid claim(i, u) > \text{cash}(i)\} = 0\}$$

will speed up the algorithm, so that the main **while** loop will run in time proportional to $\#R \times \#cus$; the preprocessing costs that result from sorting the claims for each research type incur a computational expense proportional to $\#R \times \#cus \times \log \#cus$.

APPENDIX B. FINITE DIFFERENCING RULES

The following is a small portion of a basic derivative table for set theoretic finite differencing. A more complete table may be found in [34], which is a longer

version of this paper. The Init code given for each basic form is an expansion around a single expandable parameter. Within the derivative code, potentially expensive subexpressions that must be reduced are underlined.

In the table below, all modification entries for set variables are expressed as "strict" operations, that is, s **with**:= z (for which we assume the precondition that z does not belong to s) and s **less**:= z (for which we assume the precondition that z belongs to s). Within derivative code entries, all set modifications are also strict.

To use the rules below for general SETL code, it is first necessary to apply preparatory transformations to this code so that set unions and differences $s \pm:= delta$ are expressed first as disjoint unions $s +:= (delta - s)$ and subset deletions $s -:= (delta * s)$, which must subsequently be rewritten in the lower level forms

```
(∀z ∈ delta | z ∉ s)
  s with:= z;
end ∀;
```

and

```
(∀z ∈ delta | z ∈ s)
  s less:= z;
end ∀;
```

Initialization	Modification Mod	Prederivative $\partial^- E\langle \text{Mod}\rangle$	Postderivative $\partial^+ E\langle \text{Mod}\rangle$

C1. Basic form: $E\{q\} = \{X \in S \mid F(X) = q\}$ where we assume that $E = \{[F(X), X] : X \in S\}$.

Method 1. To be used for the case when the free variable q undergoes arbitrary modification.

Init:
$\partial^- E\langle S := \{\};\rangle$
$(\forall x \in S)$
  $\partial^- E\langle S$ **with:=** $x;\rangle$
end $\forall$;

Mod	$\partial^- E\langle \text{Mod}\rangle$	$\partial^+ E\langle \text{Mod}\rangle$
$S := \{\}$;	$E := \{\}$;	
S **with:=** Z;	$E\{F(Z)\}$ **with:=** Z;	
S **less:=** Z;	$E\{F(Z)\}$ **less** $:= Z$;	
$F := \{\}$;	$E := \{\}$;	
$F(U) :=$ **om**;	**if** $U \in S$ **then**	**if** $U \in S$ **then**
	$\partial^- E\langle S$ **less:=** $U;\rangle$	$\partial^- E\langle S$ **with:=** $U;\rangle$
	end if;	**end if**;
$F(U) := Z$;		

Method 2. To be used for the case when T is a constant and T and F are integer valued; we assume that C is a positive integer and that $E = \{X \in S \mid F(X) = T\}$.

Init:
$\partial^- E\langle S := \{\};\rangle$
$(\forall x \in S)$
  $\partial^- E\langle S$ **with:=** $x;\rangle$
end $\forall$;

Mod	$\partial^- E\langle \text{Mod}\rangle$
$S := \{\}$;	$E := \{\}$;
S **with:=** Z;	**if** $F(Z) = T$ **then**
	E **with:=** Z;
	end if;

Initialization	Modification Mod	Prederivative $\partial^- E\langle \text{Mod} \rangle$	Postderivative $\partial^+ E\langle \text{Mod} \rangle$

	S **less:=** Z;	**if** $F(Z) = T$ **then**
		  E **less:=** Z;
		end if;	

	$F(U)$ +:= C;	**if** $U \in S$ **then**
		  if $F(U) = T$ **then**
		  E **less:=** U;
		  elseif $F(U) = T - C$ **then**
		  E **with:=** U;
		  end if;
		end if;	

	$F(U)$ −:= C;	**if** $U \in S$ **then**
		  if $F(U) = T$ **then**
		  E **less:=** U;
		  elseif $F(U) = T + C$ **then**
		  E **with:=** U;
		  end if;
		end if;	

C2. Basic form: $E\{q_1, q_2\} = \{x \in G\{q_2\} \mid F(X) = q_1\}$ where we assume that $E = \{[[F(X), Y], X] : [Y, X] \in G\}$.

Init:
$\partial^- E\langle G := \{\};\rangle$
$(\forall [y, x] \in G)$
  $\partial^- E\langle G\{y\}$ **with:=** $x;\rangle$
end $\forall$;

	$G := \{\}$;	$E := \{\}$;
	$G\{U\}$ **with:=** Z;	$E\{F(Z), U\}$ **with:=** Z;
	$G\{U\}$ **less:=** Z;	$E\{F(Z), U\}$ **less:=** Z;
	$F := \{\}$;	$E := \{\}$;
	$F(U)$:= **om**;	$(\forall Y \in \{W \in$ **domain** $G \mid U \in G\{W\}\})$ $\partial^- E\langle G\{Y\}$ **less** := $U;\rangle$ **end** $\forall$;

Initialization	Modification Mod	Prederivative $\partial^- E\langle \text{Mod} \rangle$	Postderivative $\partial^+ E\langle \text{Mod} \rangle$
	$F(U) := Z;$		$(\forall Y \in \{W \in \textbf{domain}\ G \mid U \in G\{W\}\})$ $\quad \partial^- E\langle G\{Y\}\ \textbf{with:=}\ U;\rangle$ **end** $\forall$;

D2. Basic form: $E\{q\} = \{X \in G\{q\} \mid X \in Q\}$ where we assume that $E = \{[X, Y] \in G \mid Y \in Q\}$.

Init:
$\partial^- E\langle Q := \{\};\rangle$
$(\forall x \in Q)$
$\quad \partial^- E\langle Q\ \textbf{with:=}\ x;\rangle$
end $\forall$;

$G := \{\};$	$E := \{\};$
$G\{U\}$ **with:=** $Z;$	**if** $Z \in Q$ **then** $\quad E\{U\}$ **with:=** $Z;$ **end if;**
$G\{U\}$ **less:=** $Z;$	**if** $Z \in Q$ **then** $\quad E\{U\}$ **less:=** $Z;$ **end if;**
$Q := \{\};$	$E := \{\};$
Q **with:=** $Z;$	$(\forall U \in \{X \in \textbf{domain}\ G \mid Z \in G\{X\}\})$ $\quad E\{U\}$ **with:=** $Z;$ **end** $\forall$;
Q **less:=** $Z;$	$(\forall U \in \{X \in \textbf{domain}\ G \mid Z \in G\{X\}\})$ $\quad E\{U\}$ **less:=** $Z;$ **end** $\forall$;

E2. Basic form: $E\{q\} = \{X \in G\{q\} \mid F(X) \in Q\}$ where we assume that $E = \{[Y, X] \in G \mid F(X) \in Q\}$.

Init:
$\partial^- E\langle G := \{\};\rangle$
$(\forall [y, x] \in G)$
$\quad \partial^- E\langle G\{y\}\ \textbf{with:=}\ x;\rangle$
end $\forall$;

Initialization	Modification Mod	Prederivative $\partial^- E\langle \text{Mod} \rangle$	Postderivative $\partial^+ E\langle \text{Mod} \rangle$
	$G := \{\};$	$E := \{\};$	
	$G\{U\}$ **with:=** $Z;$	**if** $F(Z) \in Q$ **then** $E\{U\}$ **with:=** $Z;$ **end if;**	
	$G\{U\}$ **less:=** $Z;$	**if** $F(Z) \in Q$ **then** $E\{U\}$ **less:=** $Z;$ **end if;**	
	$Q := \{\};$	$E := \{\};$	
	Q **with:=** $Z;$	$(\forall [U, X] \in \underline{\{[Y, W] \in G \mid F(W) = Z\}})$ $E\{U\}$ **with:=** $X;$ **end** $\forall;$	
	Q **less:=** $Z;$	$(\forall [U, X] \in \underline{\{[Y, W] \in G \mid F(W) = Z\}})$ $E\{U\}$ **less:=** $X;$ **end** $\forall;$	
	$F := \{\};$	$E := \{\};$	
	$F(U) :=$ **om;**	$(\forall Y \in \underline{\{W \in \textbf{domain } G \mid U \in G\{W\}\}})$ $\partial^- E\langle G\{Y\}$ **less:=** $U;\rangle$ **end** $\forall;$	
	$F(U) := Z;$		$(\forall Y \in \underline{\{W \in \textbf{domain } G \mid U \in G\{W\}\}})$ $\partial^- E\langle G\{Y\}$ **with:=** $U;\rangle$ **end** $\forall;$

H2. Basic form: $E\{q_1, q_2\} = \{X \in G\{q_2\} \mid q_1 \in F\{X\}\}$ where we assume that $E = \{[[U, X], W] : [W, X] \in G, U \in F\{X\}\}$.

Init:

$\partial^- E\langle G := \{\};\rangle$
$(\forall [y, x] \in G)$
  $\partial^- E\langle G\{y\}$ **with:=** $x;\rangle$
end $\forall;$

Initialization	Modification Mod	Prederivative $\partial^- E\langle \text{Mod} \rangle$	Postderivative $\partial^+ E\langle \text{Mod} \rangle$
	$G := \{\}$;	$E := \{\}$;	
	$G\{U\}$ **with**:= Z;	$(\forall X \in F\{Z\})$ $E\{X, U\}$ **with**:= Z; **end** $\forall$;	
	$G\{U\}$ **less**:= Z;	$(\forall X \in F\{X\})$ $E\{X, U\}$ **less**:= Z; **end** $\forall$;	
	$F := \{\}$;	$E := \{\}$;	
	$F\{U\}$ **with**:= Z;	$(\forall Y \in \{X \in \textbf{domain}\ G \mid U \in G\{X\}\})$ $E\{Z, Y\}$ **with**:= U; **end** $\forall$;	
	$F\{U\}$ **less**:= Z;	$(\forall Y \in \{X \in \textbf{domain}\ G \mid U \in G\{X\}\})$ $E\{Z, Y\}$ **less**:= U; **end** $\forall$;	

J1. Basic form: $E = \{X \in S \mid F(X) < R\}$.

Method 1. To be used when $F[S]$ is dense on the interval of integers containing the range of R values; F must be integer valued.

Init:
$\partial^- E\langle S := \{\};\rangle$
$(\forall x \in S)$
  $\partial^- E\langle S$ **with**:= $x;\rangle$
end $\forall$;

	$S := \{\}$;	$E := \{\}$;

	S **with**:= Z;	**if** $F(Z) < R$ **then**
		E **with**:= Z;
		end if;

	S **less**:= Z;	**if** $F(Z) < R$ **then**
		E **less**:= Z;
		end if;

Initialization	Modification Mod	Prederivative $\partial^- E\langle \text{Mod} \rangle$	Postderivative $\partial^+ E\langle \text{Mod} \rangle$

	R +:= T;	($\forall U \in [R\,..\,R+T-1]$)
		  ($\forall W \in \{X \in S \mid F(X) = U\}$)
		  E **with:=** W;
		  end $\forall$;
		end $\forall$;	

	R −:= T;	($\forall U \in [R-1, R-2\,..\,R-T]$)
		  ($\forall W \in \{X \in S \mid F(X) = U\}$)
		  E **less:=** W;
		  end $\forall$;
		end $\forall$;	

	$F(U)$ +:= T;	**if** $U \in S$ **and** $F(U) < R$ **and** $F(U) + T \geq R$ **then**
		  E **less:=** U;
		end if;	

	$F(U)$ −:= T;	**if** $U \in S$ **and** $F(U) \geq R$ **and** $F(U) - T < R$ **then**
		  E **with:=** U;
		end if;	

Method 2. To be used when $F[S]$ is sparse on the interval of integers containing the range of R values, or when F and R can be real; assume that $V = \mathit{Sorted}(F[S])$ and that $K = \mathbf{min}/\ \{I \in [1\,..\,\#V + 1] \mid \mathbf{not}(V(I) < R)\}$.

Init:
V := *Sorted*($F[S]$);
E := { };
K := 1;
(while $K < \#V$ **and** $V(K) < R$)
  ($\forall y \in \{X \in S \mid F(x) = V(K)\}$)
    E **with:=** y;
  end $\forall$;
  K +:= 1;
end while;

Initialization	Modification Mod	Prederivative $\partial^- E\langle \text{Mod} \rangle$	Postderivative $\partial^+ E\langle \text{Mod} \rangle$
	$R +:= T;$	**(while** $K < \#V$ **and** $V(K) < R + T$**)** $(\forall y \in \{X \in S \mid F(X) = V(K)\})$ E **with:=** Y; **end** $\forall$; $K +:= 1;$ **end while;**	
	$R -:= T;$	**(while** $K > 1$ **and** $V(K-1) \geq R - T$**)** $K -:= 1;$ $(\forall Y \in \{X \in S \mid F(X) = V(K)\})$ E **less:=** Y; **end** $\forall$; **end while;**	

M2. Basic form: $E(q) = \#F\{q\}$ where we assume that any use of $E(q)$ for which q is outside the domain of E has the value 0 and that any assignment of 0 to $E(q)$ removes q from the domain of E. We assume that $E = \{[y, \#F\{y\}] : y \in \textbf{domain } F\}$.

Init:
$\partial^- E\langle F := \{\};\rangle$
$(\forall [u, z] \in F)$
  $\partial^- E\langle F\{u\}$ **with:=** $z;\rangle$
end $\forall$;

$F := \{\};$	$E := \{\};$
$F\{U\}$ **with:=** $Z;$	$E(U) +:= 1;$
$F\{U\}$ **less:=** $Z;$	$E(U) -:= 1;$

ACKNOWLEDGMENTS

We have benefited from conversations with Martin Dowd, Jerome Feldman, Sheldon Finkelstein, Patrick Fischer, Amelia Fong, Matthew Hecht, Chuck Hedrick, Nieba Jones, Richard King, and Barry Rosen.

REFERENCES

(Note. References [11, 45] are not cited in the text.)

1. AHO, A., AND ULLMAN, J. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1978.
2. ALLEN, F.E. Program optimization. *Annu. Rev. Autom. Program. 5* (1969), 239–307.
3. ALLEN, F.E., COCKE, J., AND KENNEDY, K. Reduction of operator strength. In *Program Flow Analysis*, S. Muchnick and N. Jones (Eds.). Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 79–101.
4. BAUER, F.L., AND THE C.I.P. LANGUAGE GROUP. Report on a wide spectrum language for program specification and development. Tech. Rep. TUM-I8104, Institut für Informatik, Technische Universität München, Munich, W. Germany, May 1981.
5. BROY, M., PARTSCH, H., PEPPER, P., AND WIRSING, M. Semantic relations in programming languages. In *Information Processing 80* (1980), 101–106.
6. BURGE, W. An optimizing technique for high level programming languages. Computer Science Tech. Rep. RC 5834, #25271, IBM Research Center, Yorktown Heights, N.Y., 1976.
7. BURSTALL, R.M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44–67.
8. COCKE, J., AND KENNEDY, K. An algorithm for reduction of operator strength. *Commun. ACM 20*, 11 (Nov. 1977), 850–856.
9. COCKE, J., AND MARKSTEIN, P. Strength reduction for division and modulo with application to accessing a multilevel store. Computer Science Tech. Rep. RC 7013, #30059, IBM Research Center, Yorktown Heights, N.Y., 1978.
10. COCKE, J., AND SCHWARTZ, J.T. Programming languages and their compilers. Lecture notes, Courant Institute of Mathematical Sciences, New York Univ., New York, N.Y., 1969.
11. DEWAR, R.B.K. The SETL programming language. Unpublished manuscript.
12. DEWAR, R.B.K., GRAND, A., LIU, S.-C., SCHWARTZ, J.T., AND SCHONBERG, E. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Trans. Program. Lang. Syst. 1*, 1 (July 1979), 27–49.
13. EARLEY, J. High level iterators and a method for automatically designing data structure representation. *Comput. Lang. 1*, 4 (1975), 321–342.
14. FEATHER, M.S. A System for Developing Programs by Transformation. Ph.D. dissertation, Dep. of Artificial Intelligence, Univ. of Edinburgh, Edinburgh, Scotland, 1979.
15. FONG, A.C. Inductively computable constructs in very high level languages. In Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Tex., Jan. 29–31, 1979, pp. 21–28.
16. FONG, A.C. Generalized common subexpressions in very high level languages. In Conference Record of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, Calif., Jan. 17–19, 1977, pp. 48–57.
17. FONG, A.C., AND ULLMAN, J.D. Induction variables in very high level languages. In Conference Record of the 3d ACM Symposium on Principles of Programming Languages, Atlanta, Ga., Jan. 19–21, 1976, pp. 104–112.
18. FREUDENBERGER, S.M. SETL data structures. *SETL Newsl.* 189B, Dep. of Computer Science, New York Univ., New York, N.Y., May 1980.
19. GAREY, M.R., AND JOHNSON, D.S. *Computers and Intractability.* W.H. Freeman, San Francisco, 1979.
20. GERHART, S.L. Correctness-preserving program transformations. In Conference Record of the 2d ACM Symposium on Principles of Programming Languages, Palo Alto, Calif., Jan. 20–22, 1975, pp. 54–66.
21. GOLDSTINE, H.H. *A History of Numerical Analysis.* Springer-Verlag, New York, 1977.
22. GOLDSTINE, H.H. *The Computer from Pascal to Von Neumann.* Princeton Univ. Press, Princeton, N.J., 1972.

23. HECHT, M. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, 1977.
24. HOARE, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12,* 10 (Oct. 1969), 576–580, 583.
25. KENNEDY, K. A survey of compiler optimization techniques. In *Program Flow Analysis,* S. Muchnick and N. Jones (Eds.). Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 5–54.
26. KENNEDY, K. Variable subsumption with constant folding. *SETL Newsl.* 123, Dep. of Computer Science, New York Univ., New York, N.Y., Feb. 1974.
27. KENNEDY, K. Global dead computation elimination. *SETL Newsl.* 111, Dep. of Computer Science, New York Univ., New York, N.Y., Aug. 1973.
28. KENNEDY, K. An algorithm to compute compacted use definition chains. *SETL Newsl.* 112, Dep. of Computer Science, New York Univ., New York, N.Y., Aug. 1973.
29. KENNEDY, K. Linear function test replacement. *SETL Newsl.* 107, Dep. of Computer Science, New York Univ., New York, N.Y., May 1973.
30. KENNEDY, K. Reduction in strength using hashed temporaries. *SETL Newsl.* 102, Dep. of Computer Science, New York Univ., New York, N.Y., Mar. 1973.
31. KOENIG, S., AND PAIGE, R. A transformational framework for the automatic control of derived data. In Proceedings, 7th International Conference on Very Large Data Bases, Cannes, France, Sept. 9–11, 1981, pp. 306–318.
32. MORGENSTERN, M. Automated Design and Optimization of Management Information System Software. Ph.D. dissertation, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., Sept. 1976.
33. PAIGE, R. *Formal Differentiation.* UMI Research Press, Ann Arbor, Mich., 1981. Revision of Ph.D. dissertation, Dep. of Computer Science, New York Univ., New York, N.Y., June 1979.
34. PAIGE, R., AND KOENIG, S. Finite differencing of computable expressions. Tech. Rep. LCSR-TR-8, Dep. of Computer Science, Rutgers Univ., New Brunswick, N.J., Aug. 1980.
35. PAIGE, R., AND SCHWARTZ, J.T. Expression continuity and the formal differentiation of algorithms. In Conference Record of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, Calif., Jan. 17–19, 1977, pp. 58–71.
36. ROSEN, B.K. Degrees of availability. In *Program Flow Analysis,* S. Muchnick and N. Jones (Eds.). Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 55–76.
37. SCHONBERG, E., SCHWARTZ, J.T., AND SHARIR, M. An automatic technique for selection of data representations in SETL programs. *ACM Trans. Program. Lang. Syst. 3,* 2 (Apr. 1981), 126–143.
38. SCHWARTZ, J.T. Correct program technology. Courant Computer Science Rep. 12, Dep. of Computer Science, New York Univ., New York, N.Y., Sept. 1977.
39. SCHWARTZ, J.T. On the "base form" of algorithms. *SETL Newsl.* 159, Dep. of Computer Science, New York Univ., New York, N.Y., Nov. 1975.
40. SCHWARTZ, J.T. Optimization of very high level languages, parts I and II. *Comput. Lang. 1,* 2–3 (1975), 161–218.
41. SCHWARTZ, J.T. *On Programming: An Interim Report on the SETL Project, Installments I and II.* Courant Institute of Mathematical Sciences, New York Univ., New York, N.Y., 1974.
42. SCHWARTZ, J.T. Structureless programming, or the notion of "rubble," and the reduction of programs to rubble. *SETL Newsl.* 135A, Dep. of Computer Science, New York Univ., New York, N.Y., July 1974.
43. SHARIR, M. Some observations concerning formal differentiation of set theoretic expressions. *ACM Trans. Program. Lang. Syst. 4,* 2 (Apr. 1982), 196–225.
44. SRIDHARAN, N. Private communication, 1980.
45. STANDISH, T., HARRIMAN, D., KIBLER, D., AND NEIGHBORS, J. The Irvine program transformation catalogue. Tech. Rep., Dep. of Information and Computer Science, Univ. of California, Irvine, Irvine, Calif., Jan. 1976.
46. TARJAN, R.E. A unified approach to path problems. *J. ACM 28,* 3 (July 1981), 577–593.
47. TENENBAUM, A. Type Determination for Very High Level Languages. Ph.D. dissertation, Dep. of Computer Science, New York Univ., New York, N.Y., Oct. 1974; also in Courant Computer Science Rep. 3, Dep. of Computer Science, New York Univ., New York, N.Y., Oct. 1974.

Received August 1980; revised November 1980 and January and March 1982; accepted March 1982