Symbolic Finite Differencing - Part I

Robert Paige [1]

Computer Science Department
New York University/ Courant Institute
New York, NY 10012

ABSTRACT

Programming practice is limited by labor costs such as implementation design, program development, debugging, and maintenance (including evolution and integration). Because resource utilization is often difficult to predict precisely, the economics of software developement also depends on the risk of the implementation failing to meet its performance requirements. Consequently, complex algorithms are frequently avoided in large systems - even in optimizing compilers, where run-time performance of the compiled code is so important.

Our aim is to overcome some of these limitations by means of a transformational programming tool that facilitates implementation of complex algorithms with guaranteed worst-case asymptotic time and space. RAPTS, a working prototype of such a tool is scheduled to be demonstrated at ESOP '90.

This paper is in two parts. In part I we specify a general finite differencing framework that unifies aspects of compiler and programming methodologies, and is the basis for one of the three main transformations implemented in RAPTS. In Part II we illustrate how the transformational methodology underlying RAPTS can be used to improve the implementation of its own finite differencing transformation. Improved reduction in strength models and algorithms for conventional languages are produced as an outgrowth of this discussion.

1. Introduction.

Finite Differencing is an old and very useful mathematical idea that can speed up computation by replacing repeated costly calculations by more efficient incremental counterparts. Suppose we have a costly computation $f(x_1,...,x_n)$ in a program region R of high execution frequency. Then if we can maintain equality $E = f(x_1,...,x_n)$ as an invariant within R (at all points where $f(x_1,...,x_n)$ needs to be computed) by executing inexpensive modifications to E whenever any parameter x_i i=1,...,n is modified, then we can replace all occurrences of $f(x_1,...,x_n)$ in R by the stored value E. This idea is well established in compiler methodology, where it underlies the classical method of strength reduction (see Cocke et. al.[3, 8, 9]). It also plays a major role in programming methodology, for which see Dijkstra[12], Earley[14-16], and Gries[24].

In this paper we seek to extend and unify the finite differencing technology found in low level compilers under a single framework, which generalizes the abstract invariant found in Paige[37].

1 This work is partly based on research supported by the Office of Naval Research under Contract No. N00014-87-K-0461.

We introduce several successively more general definitions of Finite Differencing program transformations. The most general of these transformations is used to derive efficient algorithms to implement each of these. This work overcomes several conceptual and algorithmic shortcomings in current strength reduction methods for conventional optimizing compilers. We also show how certain major aspects of high level programming are susceptible to automation.

The paper is organized as follows. The remainder of this section sketches a brief history of finite differencing. This is followed by preliminary notations and assumptions. The next section presents a comprehensive language independent specification of symbolic finite differencing in the form of a conditional rewrite system. In part II of this paper we present a taxonomy of restricted finite differencing problems and algorithmic solutions. The presentation makes use of set theoretic finite differencing to derive these solutions.

1.1. History

Finite Differencing has special historical significance in Computer Science. Henry Goldstine traced the idea back to Henry Briggs, a 16th century mathematician who, faced with the formidable limitations imposed on the speed with which he could perform manual calculations, discovered the technique of finite differencing to evaluate successive polynomial values using difference polynomials[22].

Briggs was a skilled 'calculator', whose task was to compute a d-th degree polynomial $p(x)$ at points $x_0 + i \times \Delta$, i=1,... for some constant Δ. Letting $p_0(x) = p(x)$, he recognized that each polynomial $p_i(x) = p_{i-1}(x+\Delta) - p_{i-1}(x)$ has degree less than or equal to $d-i$ and that $p_d(x)$ is a constant, possibly 0. Moreover, simple rearrangement of the preceding formulas led Briggs to calculate each successive polynomial value $p(x_0+ i\times\Delta)$ using only d sums instead of the d sums and d products required by Horner's Rule. Briggs's method produced the following calculations:

$x := x_0$
compute $t_i := p_i(x)$ for i=0, 1,...,d
Repeat
  print t_0
  compute $t_i := t_i + t_{i+1}$ for i=0,...d−1
  $x := x+\Delta$

(It is believed that the special case of finite differencing for linear polynomials was known to the Phoenicians in antiquity [28].) The preceding code maintains equalities $t_i = p_i(x)$ for i=0,...,d as invariants at the point where the polynomial value $p(x)$ needs to be printed. Consequently, the costly repeated calculation of $p(x)$ is made redundant and can be avoided by simply retrieving the stored value t_0. These invariants are established by *de novo* calculations (using, say, Horner's Rule) on entry to the loop, and are reestablished by incremental updates to t_i, i=0,...,d when they are spoiled by the modification to x inside the loop.

Several hundred years later Babbage proposed to conserve manual labor further and improve reliability of polynomial tabulation by having a special purpose computer, his Analytic Difference Engine, execute the main loop of Briggs's algorithm after its registers would be supplied with initial values for t_i i=0,...,d. As Goldstine points out, finite differencing was the main idea underlying digital computers up through World War II, when such computers were used to produce gunnery tables [23].

However, Briggs's finite difference method embodies ideas extending well beyond the context in which he applied it. These ideas are of central importance to both compiler and

programming methodologies. The essence of Briggs's idea underlies the classical method of strength reduction[2, 8, 9, 30, 44], a Fortran optimization that speeds up code by recognizing and maintaining arithmetic invariants. An extensive discussion of the method appears in Cocke and Schwartz [9], where it seems to be regarded as the most important machine independent optimization, generalizing redundant code elimination, constant propagation, and code motion. However, as we shall see, the method presented in Cocke and Schwartz is impractical for polynomials of degree greater than one.

Extending finite differencing to set theoretic expressions was initiated by Jay Earley [16], and developed by Fong et. al. [18-20], and Paige, et. al. [33-35, 37]. Proposals for implementation designs as high level compiler optimizations are due to Fong [19], Paige[34, 37], Rosen[41], and Tarjan [47].

The finite differencing idea has also been developed in the implementation of programming language environments, principally in attribute evaluation and reevaluation. Since the semantic rules in an attribute grammar are invariants defined with respect to an underlying grammar, various techniques to establish and maintain these invariants relative to modifications to syntactic units in the parse tree have been developed. Of particular importance is the elegant attribute reevaluation algorithm of Reps, et. al. [39], the incremental method of handling aggregate attributes of Hoover and Teitelbaum [26], and the coupling of database relations to an attribute grammar by Horwitz and Teitelbaum [27]. Aiming for a general tool to develop incremental software, Yellin and Strom developed a stream-like language called INC with a compiler that automatically generates provably efficient incremental programs from nonincremental specifications[53]. Their language includes relational algebra augmented with transitive closure, arithmetic, and abstract data types such as fixed length tuples and bags There are also a large number of heuristic approaches to the formidable task of incremental program optimization, which, while related to differencing, in their goals, have only achieved advantages in performance for limited special cases. One of the more general of these approaches has been suggested by Marlowe and Ryder[31].

The other area of systems where differencing plays an important direct role is in database concrete view maintenance and its application to integrity control. Here we mention the work of Bernstein et. al. [5], Koenig and Paige[29, 36], and Roussopoulis[42]. Differencing has also been critical in fixed point calculations needed to process recursive database queries efficiently (see for example, Bancilhon[4] or Whang and Navathe[51]). More general methods of integrating differencing with efficient fixed point calculations are found in Cai and Paige[6, 7].

The principle of maintaining invariants in order to make costly operations redundant was elucidated by Dijkstra[12], Gries[24], and many others as essential to the design of efficient algorithms. However, their approaches used sweeping aesthetic principles to choose invariants instead of formal criteria based on syntactic analysis of the program being designed. Although the idea has been mainly developed for imperative languages, it seems to have generic relevance to a variety of programming styles, including logic programming (see Petorossi[38] and Sacca and Zaniolo[43]; a similiarity with 'semi-naive evaluation' is noted by Ullman[49]) and functional programming (see Meertens[32] and Smith[46]).

2. Notations and Conventions

2.1. Language

Although we will formulate finite differencing independent of any specific language, the ideas will be illustrated for imperative languages using a wide spectrum of dictions from the familiar arithmetic notations in Algol and Fortran to the finite set, map, and tuple expressions of SETL[45]. With a few exceptions to be described, we will use set theoretic expressions that conform to universally accepted mathematical notations.

One such exception is the overloaded cardinality operator #s, which denotes the number of elements in a set or tuple s. Another is the choice operation ∃s, which denotes an arbitrary element selected from the set s. If s is empty, then the value of ∃s is undefined. Of particular importance is the SETL set former {x ∈ s | k(x)}, which denotes the set of elements in a set or tuple s satisfying predicate k(x). An example of a slightly more general set former is $\{(x+y)^2: x \in s, y \in t \mid odd(x+y)\}$, which denotes the set of squares of odd sums x+y, where x ∈ s and y ∈ t. A map f is a finite set of ordered pairs with domain {x: [x,y] ∈ f} and range {y: [x,y] ∈ f}. Thus, a map can be a single-valued function or a multi-valued binary relation. The function retrieval term f(x) denotes the value of function f at domain point x. If x does not belong to the domain of f or if f contains two or more different pairs with first component value x, then f(x) is undefined. We use the image set notation f{x} to denote the set {y: [u,y] ∈ f | u = x}. If s is a set, then the extended image set f[s] denotes the set {y: [x,y] ∈ f | x ∈ s}. Tuple formers resemble set formers except that square brackets are used instead of curly brackets. For example, [sal(x): x ∈ dept] is a tuple listing the salaries (with repetitions allowed) of all members of the set dept.

Adding an element x to a set s is denoted by s with:= x. If x is undefined, then s is unchanged, and if s is initially undefined, it will be assigned {x}. To delete an element x from s, we use the notation s less:= x. Assignments of the form x op:= y abbreviate x := x op y. We use the for-loop control structure

```
(for x ∈ s)
        block(x)
end
```

to execute block (a sequence of statements) for each value x belonging to s. If s is a set, then we execute block for each value of x without repetition and in any order. If s is a tuple, then block is executed for every component value of s from the first to the last component. In either case, iteration proceeds through the initial value of s on entry to the loop (as through a copy of s, and cannot be affected by modifications to s within block).

2.2. Assumptions

The main goal, to be discussed last, is to apply finite differencing to program regions generated by transformation from high level functional specifications. In this context information about control flow, frequency of execution, and a great deal more useful information about these regions are supplied by the same transformations generating these regions. However, in order to make connections with existing compiler techniques in which control flow analysis is a required prerequisite for any code motion (see Cocke and Schwartz[9], Fong[19], Allen, Cocke, and Kennedy[3], Rosen[41], and Tarjan[47]), we can make a few simplifying assumptions. We assume that programs are single entry region of high execution frequency in which each expression is executed with roughly the same frequency and statements can be executed in any order. Essentially the same assumptions are found in the classical papers on strength reduction, e.g. Cocke and Kennedy[8].

We also assume expressions are side-affect free and behave like functions. Issues of safety and numerical precision are also regarded as orthogonal to the discussion here. Thus we assume arbitrary precision arithmetic.

3. Finite Differencing Framework

It is useful to formulate Finite Differencing in a general language-independent way that treats issues of correctness and efficiency separately. This section is entirely concerned with the correctness of a finite differencing framework that generalizes the earlier frameworks of Paige and Koenig[33] and Paige[37]. Part II of this paper presents efficient solutions.

There are three broad objectives in improving programs by Finite Differencing - (1) syntactic recognition of computational bottlenecks appearing within a program P; (2) choosing invariants whose maintenance inside P allows these bottlenecks to be removed; and (3) scheduling how collections of invariants can be maintained in P. To achieve these objectives we need to provide a syntactic characterization of bottlenecks (including the program contexts in which they may be avoided) and invariants (and the program contexts in which they can be usefully maintained). That is, we need to recognize a language of bottlenecks generated by composition and parameter substitution from elementary forms; we also need to apply a finite difference calculus to schedule code that maintains collections of possibly interacting invariants so that these bottlenecks can be removed.

Just a few informal examples of Finite Differencing indicate the wide range of applications that are possible. For a simplest example consider a product $x \times c$ appearing in the array access formula to be computed repeatedly in a program loop, where c is constant and x is a loop index variable modified only by fixed increments. Such a product can be replaced by less expensive successive additions. If the language has polynomial datatypes, then whenever a polynomial is used for tabulation, we can apply Briggs's method directly. The expense of database queries in an environment supporting a limited number of primitive update operations can be mitigated by maintaining appropriate invariants with dynamic indexes. A naive specification of Dijkstra's shortest path algorithm[13], in which the greedy computation of a local minimum is made explicit, can be implemented efficiently using the Fibonacci Heap of Fredman and Tarjan[21]. Both John Cocke and Ralph Wachter have observed the importance of Finite Differencing in implementing spreadsheets efficiently[10, 50].

3.1. Individual Invariants

In our Finite Differencing framework bottlenecks are restricted to program expressions that are *a priori* expensive, side-effect free, and functional. These expressions are single syntactic units recognizable using tree pattern matching. Examples are products (which are more expensive than sums) and set formers $\{x \in s \mid k(x)\}$ (which are costlier than computing predicate k(x) only once).

However, local analysis is not sufficient to determine which invariants, if any, are best suited to remove a bottleneck. This choice depends on a careful global analysis of the way variables are modified in the environment containing the bottleneck. The analysis should determine with reasonable assurance whether Finite Differencing would actually improve code; that is, whether the cumulative cost of maintaining invariants is less than the cost of computing the original computational bottlenecks.

To appreciate the difficulty of 'fitting' invariants to bottlenecks, let us look at a few illustrative examples. We know it is profitable to maintain an invariant $e = x \times c$ to avoid computing a

product x×c in a program region R (of high execution frequency) in which c is a region constant and x is only incremented and decremented by constants. However, if a product x × d can also be assigned to x within R, then differencing might not pay.

Sometimes it is resourceful to use the same invariant to remove different bottlenecks. For example, consider two set formers {x ∈ s | f(x) = q1} and {y ∈ s | f(y) = q2} inside a program region R where set s undergoes element addition and deletion, finite function f undergoes indexed assignments f(x) := z, and expressions q1 and q2 involve only free variables (i.e., variables not bound to their respective set formers) that can undergo arbitrary change. Then within R we can efficiently maintain the following index/invariant:

$$e = \{[f(w), w]: w \in s\}$$

and replace the expensive set formers by more efficient retrievals e{q1} and e{q2} respectively. However, if instead of the preceding two set formers, R contained set former {x ∈ s | f(x) = c}, where c is a region constant, then no index is necessary. It suffices to maintain equality

$$e = \{x \in s \mid f(x) = c\}$$

as an invariant, and replace occurrences of the set former by the stored value e.

For different dynamic environments we may also choose different invariants to remove the same bottleneck. Consider operation min/s, which denotes the smallest element of a set s. If s is only modified by assignment to the empty set and element addition, then an oblivious implementation of invariant e = min/s in O(1) time and space is possible to avoid the linear time search through s. However, in a dynamic environment where min/s is deleted from s, we only know how to use a more expensive heap implementation.

An 'expert' system is needed to choose the 'most specific' invariant suited to a bottleneck and its dynamic environment. Recognizing the most specific invariants for handling expression min/{min/f{x}: x in domain f} (which denotes the minimum of minima) is at the heart of Floyd's single source shortest path algorithm[17]. Within an abstract form of the algorithm f{x} is either augmented by a new value or is set to empty if min/f{x} is a minimum of minima. (This is the classical deletmin operation applied to the domain of f.) By using the oblivious implementation for each of the minima E1(x) = min/f{x} and the heap implementation for the outermost minimum, we obtain an asymptotic space compression over a naive implementation.

A fully general specification of Finite Differencing requires sophisticated pattern matching, in which patterns, incorporating properties obtained from global analysis, can be partially ordered by a subsumption relation. For this purpose we use *nonlinear* patterns.

Definition: The set of nonlinear patterns consists of variables, constants, and any term $f(p_1, \cdots, p_k)$ where f is a function symbol of arity k and $p_1, \cdots, p_k$ are patterns.

Linear patterns are nonlinear ones in which no pattern variable occurs more than once. Nonlinear pattern matching is defined as follows:

Definition: Pattern p_1 is said to *subsume* (i.e., be more general than) pattern p_2, denoted by $p_1 \geq p_2$, iff either (i) p_1 is a variable, or (ii) p_1 is $f(x_1, \cdots, x_k)$, p_2 is $f(y_1, \cdots, y_k)$, $x_i \geq y_i$ for $i = 1, \cdots, k$, and every occurrence of the same pattern variable in p_1 should match equal patterns in p_2. Relation ≥ is called the subsumption relation.

Patterns can also include special variables that match two extreme cases - (1) region constant expressions, and (2) expressions whose variables undergo unrestricted modification. We call these second kind of expressions *discontinuities*; the pattern variables that match them are called

discontinuity parameters. By convention, if q is a discontinuity parameter, v matches any variable undergoing prescribed modification, c matches any region constant, and d is a specific constant, we order them q > v > c > d.

Once a collection of invariants is determined, the partial difference calculus based on Paige[37] can be used to schedule how they are maintained. Within this calculus invariants are restricted to the following general form:

(1) $\exists$ y1 $\in$ f(x1,...,xn), $\exists$ y2 $\in$ y1,...,$\exists$ yk $\in$ yk-1 | yk = E

where f is an n-variate function, which is set valued when k > 0. In its simplest form when k=0, formula (1) reduces to an equality E = f(x1,...,xn). Such an invariant is maintained in a program region R by establishing it on entry to R, and by updating E at each point in R when any of the parameters xi i=1,...,n is modified. Whenever invariant (1) can be maintained at each point in R where expression $\ni \cdots \ni$ f(x1,...,xn) (k applications of arbitrary choice from set f(x1,...,xn)) occurs, such occurrences are redundant and can be replaced by E. Invariant (1) may also imply certain other critical identities that support the correctness of other fruitful code replacements.

More formally, let E :=: I(x1,...,xn) denote a particular invariant (1) to be maintained and exploited in a program region R, where x1,...,xn are the *input* parameters, and *E* is the *output* parameter. Here, I represents the abstract form of invariant (belonging to a library of such forms) being instantiated in R; parameters x1,...,xn are program variables, region constants, or discontinuity parameters, and E represents a new program variable uniquely associated with (1). When the parameters x1,...,xn are understood or are irrelevant, it is sometimes convenient to use the output variable E to stand for invariant (1). We call (1) the *Defining Invariant* and associate with it one or more rewrite rules, ei -> ei' i=1,...k, where ei and ei' are patterns matching side-effect free expressions that behave like functions. Expressions e1,...,ek are called *replaceable terms*, and e1',...,en' are called *replacing terms*. In a program region where E is maintained each program expression that ei matches will be replaced by ei' i=1,...,k. We require that

(i) at any program point where invariant (1) holds, replacement of ei by ei' must preserve program meaning, i=1,...,k;

(ii) for i=1,...,k each replaceable term ei may depend on parameters x1,...,xn, but not E, whereas ei' must involve E and may also involve x1,...,xn;

(iii) there must be one distinguished rule e -> e', called the *binding rule*, in which e involves all of the parameters x1,...,xn; within this rule e is called the *binding term*;

(iv) no two different replaceable terms can match the same program expression;

(v) repeated reduction should terminate.

We also associate with invariant (1) code that should be executed in order to reestablish (1) (by updating E) when it is falsified at program points where any program variables among the parameters x1,...,xn is modified. This code is called partial difference code. Suppose that (1) holds just prior to a modification dx to a variable x on which (1) depends. Let B1 and B2 be two possibly empty code blocks such that

(i) B1 and B2 only modify E and variables local to these blocks;

(ii) If invariant (1) holds on entry to B1, then it holds at each point where any of its replaceable terms appear within B1, B2, and dx.

(iii) The Hoare Formula {E :=: I(x1,...,xn)} B1 dx B2 {E :=: I(x1,...,xn)} is satisfied.

Then we say that B1 and B2 are *pre-* and *post–difference* code blocks of E with respect to modification dx and are denoted by ∂-E<dx> and ∂+E<dx> respectively. Note that all occurrences of x within ∂-E<dx> refer to the old value of x and that all occurrences of x within ∂+E<dx> refer to the new value of x.

Difference rules are specified in terms of modification patterns dxi for any of the parameters xi i=1,...,n that are program variables. Pattern dxi can involve any of the input parameters x1,...,xn (but not E) and additional patterns that match program expressions. For the difference rules to be well-defined, no two different modification patterns can match the same modification within a program.

Also associated with (1) is code that establishes the defining invariant on entry to the region where it is maintained.

In the simplest application of finite differencing, where a single invariant E :=: I(x1,...,xn) is maintained in a code region R, we transform R by,

(i) inserting code to establish E on entry to R,

(ii) replacing each modification dx in R to any variable x on which E depends by ∂-E<dx> dx ∂+E<dx>, and

(ii) performing repeated *reductions* proceeding bottom-up within R, in which every occurrence of a replaceable term ei within R and within the difference code introduced in the previous step is replaced by an occurrence of ei', i=1,...,k

If we assume that E holds on entry to R, then the insertion of difference code makes all occurrences of ei inside R replaceable by ei'. The formal correctness proof found in[33] (and which follows directly from the definition of the invariant - and especially from the obvious fact that the difference code for E relative to any modification to a variable on which E does not depend is empty) extends to this more general framework. Apart from the code that establishes E on entry to R, the new program region formed from R by steps (ii) and (iii) above is called the *Differential* of E with respect to R, and is denoted by ∂E<R>. As was noted in[33] the differential operator is linear with respect to sequential code blocks; i.e., ∂E<Q;R> = ∂E<Q>;∂E<R>.

Before going on to describe the remaining aspects of the partial difference calculus, it is useful to look more closely at invariants using illustrative examples. Consider once again the invariant E :=: heap(s), which maintains set s as a 2-heap [52], implemented with vector E; let min/s -> E(1) be the binding rewrite rule. The difference code for E with respect to the delete-min operation s less:= min/s is the well-known 'siftdown' code[1]. However, it is incorrect to make siftdown the pre-difference ∂-E<s less:=min/s> with an empty post-difference, because the occurrence of replaceable term min/s within the delete-min operation would not be redundant. An empty pre-difference with siftdown as the post-difference is correct.

Next, consider an invariant E = {x ∈ t | f(x) ∈ s} with binding rule {x ∈ t | f(x) ∈ s} -> E and pre-difference rule

$$\partial E\langle s \cup:= \Delta\rangle = E \cup:= \{x \in t \mid f(x) \in \Delta\}$$

where the post-difference is empty, and Δ is a pattern that matches any set valued expression. Unfortunately, the code implied by ∂E<s ∪:= {x ∈ t | f(x) ∈ s}> without performing any reductions is

$E \cup := \{x \in t \mid f(x) \in \{y \in t \mid f(y) \in s\}\}$
$s \cup := \{x \in t \mid f(x) \in s\}$

and the invariant only holds at the first of two occurrences of the replaceable term. Unlike the heap example, this time switching the pre-difference and post-difference code would not be correct. One solution is to precondition the program before finite differencing so that sets are only modified in terms of element additions and deletions. Then instead of the previous difference rule we would use

$$\partial\text{-}E\langle s \text{ with:= } x\rangle = (\text{for } y \in \{u \in t \mid f(u) = x\})$$
$$E \text{ with:= } y$$
$$\text{end}$$

where x is a pattern that matches any variable or region constant.

One last example illustrates how maintenance of one invariant may require repeated reduction to remove nested bottlenecks. Consider the invariant $e = \{[f(x),x]: x \in s\}$ to be used in removing the following costly expression:

$$\{y \in s \mid f(y) = \{w \in s \mid f(w) = q\}\}$$

where q involves only free variables (of these two set formers) that can undergo arbitrary modification. Application of the differential results in two applications of the rewrite rule $\{x \in s \mid f(x) = q'\} \rightarrow e\{q'\}$ to reduce the costly program expression to $e\{e\{q\}\}$. In this example q' is a discontinuity parameter matching any expression not containing the bound variable of the set former. Thus, if the program expression q involved y, then only only the inner set former would be reduced. Likewise if q involved w but not y, then only the outer set former would be reduced. Note that this example does not violate the rule against a replaceable term involving the output parameter e; here the replaceable term involves q', which matches e.

Of course, the condition that reductions terminate is important. Such termination is guaranteed if we can ensure that reductions simplify code by reducing the number of 'expensive' operations. Instead of providing a restricted rewrite rule syntax that guarantees termination, we prefer a more flexibile approach, which requires a separate termination proof for each finite differencing instance.

So far we have focussed on recognizing bottlenecks removable by a single invariant in isolation. However, when more than one invariant is needed, the possible interactions that arise between them complicate the process of recognizing bottlenecks, selecting invariants, and scheduling the code that maintains these invariants. The next few subsections investigate these interactions.

3.2. Sets of Invariants

The algorithms to recognize bottlenecks and to choose and maintain invariants make use of a library of invariant specifications. Within this library invariants are distinguished by the binding term together with the dynamic environment (i.e., the set of modifications contained in the difference rules). Invariants are also partially ordered by subsumption as follows. Let E1 and E2 be two invariants with dynamic environments D1 and D2, and binding terms t1 and t2 respectively. Then E1 subsumes E2 if t1 subsumes t2 and forall modifications d2 in D2, there exists a pattern d1 in D1 such that d1 subsumes d2. In order to select invariants deterministically, we require that no two incomparable invariants in the library can both subsume the same invariant (not necessarily in the

library). (This condition is adapted from Hoffmann and O'Donnell's simple pattern forest condition[25].)

Bottlenecks are recognized and invariants are chosen by matching the binding terms (which involve all the input parameters of their invariants) to the program P from innermost to outermost expression of P. As a side effect, matching binds the parameters of the selected invariant E :=: I(x1,...,xn) to program entities y1,...,yn. The Invariant is chosen only if it contains difference code with respect to every modification in P to a variable on which the invariant depends. If this invariant has not been previously selected, then output parameter E is bound to a new program variable not occurring in P. In case more than one invariant matches, we choose the most specific one. In case one bottleneck is properly enclosed in another, we process the innermost one first. Bottlenenecks occurring within assignment statements have highest priority, since simplification of the dynamic environment of P may increase opportunities for removing other bottlenecks, and may also lead to generally simpler difference code.

In order to determine all the invariants in advance (before their maintenance is scheduled), we partly simulate the reductions and difference code insertions that would be done to maintain each prospective invariant. Thus, for the recognition phase to terminate with a finite collection of invariants, we need to ensure that the rewriting system arising from the set of possibly interacting invariants terminates. By keeping track of all invariants and recognizing redundant invariants (i.e., when to use the same invariant for different bottlenecks) as the analysis proceeds bottom-up, we can keep the number of invariants down and guarantee the following crucial correctness condition: that the dependency graph, formed by drawing edges from invariants to the program variables on which they depend, is a dag.

Although termination proofs are left outside the finite differencing framework, it is worthwhile to mention several possible causes of nontermination. Nontermination can result from an invariant containing rewrite rules involving discontinuity parameters q, such as q -> e'. Also, unbridled use of identity invariants E = x can lead to runaway renaming of variables.

There is one special case, where automatic detection of a nonterminating condition is desirable. We assume that an expression f(x1,...,xn) is an unremovable bottleneck, if it appears on the right-hand-side of an assignment, whose left-hand-side can be traced in a forward data-flow direction to any of the variables x1,...,xn. The preceding condition is called an *occurs* check after a similar condition that determines that a set of term equations cannot be unified[40].

Suppose we tried to maintain invariant E = x×c to remove the product appearing in assignment x := x×c. Then the post-difference code E := E × c would just generate a new product. To remove this new product, we would generate yet another product, and so forth.

In order to make the current definitions precise and also realistic, we need to resolve three remaining issues . According to the definition of Invariant, it is possible for the replaceable terms in the nonbinding rewrite rules of two or more different invariants to match the same program expression. In this case only one of the rewrite rules will be chosen arbitrarily for reduction. This form of nondeterminism is desirable in order to exploit the global effect of several invariants. It also captures the classical technique of Linear Test Replacement in our more general framework[44].

For a simple example of Linear Test Replacement, consider two invariants e1 = x × c1 and e2 = x × c2, where c1 and c2 are distinct region constants, and x is modified by increments x +:= 1. The binding rewrite rules are x×c1 -> e1 and x×c2 -> e2, but the nonbinding rules are x > c3 -> e1 > c3×c1 and x > c3 -> e2 > c3×c2, where c3 is a region constant. Clearly, only one of these

nonbinding rules can be chosen for reduction.

In Linear Test Replacement the nonbinding reduction is only performed if the variable x is rendered useless; i.e., its value is not referenced in any control structure, and it does not contribute either directly or indirectly to program output. Thus, as a practical matter, it is important to consider reductions conditionally based on the global effect of maintaining several invariants. In this case, it is based on whether or not variable x would be made useless.

It is worth mentioning two other examples where useless variable analysis determines whether an invariant can be used at all. Consider a variable x and region constants c and d occurring in a program region R. It is profitable to use invariant E = x×c to remove a product x×c in R, where x is modified by x := x×d, if the difference code

$$\partial E\langle x := x \times d\rangle = E := E \times d,$$

makes x useless in R. The number of products is reduced by one. Of course, if x is not made useless, then finite differencing will degrade performance. The same argument can be made for the greedier goal of eliminating a sum x + c in a region where x can be modfied by x +:= d. This second example can be found in several papers on reduction in strength (e.g., Lowry and Medlock[30]).

For completeness we also require that any collection of invariants include an implicit rule for detection and removal of any region constant expression, both in the original code and in any difference code introduced. In the last example products c3×c1 and c3×c2, introduced by nonbinding reductions, are region constants since c1, c2, and c3 are.

Determining all of the invariants before scheduling how to maintain them allows us to integrate such global cleanup optimizations as useless code elimination, constant folding, redundant and useless code elimination with finite differencing more efficiently than if cleanup came after scheduling. By folding global analysis of useless code together with analysis of invariants we can use efficient local processing to predict which invariants do not have to be maintained. Such information can also serve as a condition for choosing other invariants. More generally, advanced knowledge about the final maintenance code contributes to efficient scheduling, especially in allowing the same invariant (and not redundant copies) to be used for preventing different occurrences of bottlenecks.

Following Paige and Koenig[33] and Paige[37], the task of scheduling the difference code for invariants can be solved by extending the definitions of difference code and the differential within a partial difference calculus. Consider n invariants to be maintained in a program region R, in which each invariant Ej, j=1,...,n can depend only on underlying program variables x1,...,xm and Ei, i=1,..,j-1. Then the difference code and differentials for single invariants can be combined algebraically to obtain differentials and difference code for all of the invariants collectively. The collective differential of E1,...,En with respect to R is denoted by $\partial\{E1,...,En\}\langle R\rangle$. The collective pre- and post-difference code blocks with respect to a modification dx to a variable x are denoted by $\partial\text{-}\{E1,...,En\}\langle dx\rangle$ and $\partial\text{+}\{E1,...,En\}\langle dx\rangle$ respectively. A chain rule is used to derive the code for collective differentials and difference blocks based on the simpler rules for single invariants.

But before stating the chain rule it is useful to illustrate three different kinds of interactions between invariants. We consider the following four basic kinds of interaction:

(i) when two distinct invariants share the same variable as an input parameter, it is called *horizontal* interaction;

(ii) when an output variable of one invariant E1 is an input variable of another invariant E2, it is called *vertical* interaction, and we say that E2 depends on E1;

(iii) When the dynamic program environment is modified by an invariant E that reduces the right-hand-side of an assignment statement, it is called *dynamic* interaction, and we say that E is *dynamic*;

(iv) when the difference code of one invariant E1 with respect to a change dx to variable x contains a replaceable term e for another invariant E2, it is called *supporting* interaction, and we say that E1 *supports* E2 with respect to dx.

For these four cases and for the general rules to be presented, we show how to carefully order the code that maintains a collection of invariants correctly and efficiently without having to perform potentially costly copy operations.

3.3. Horizontal and Vertical Interaction

When sets of invariants are limited to horizontal and vertical interaction with no dynamic or supporting interaction, then the process of recognizing bottlenecks (by matching the binding terms and dynamic environments of invariants) and scheduling difference code is greatly simplified. All such bottlenecks are detected within the program region R being analyzed (in a bottom-up pass), and none are introduced within the difference code of invariants. In the absence of dynamic interaction, the dynamic environment is stable. Also, performing reductions is straightforward with tie breaking between nonbinding rewrite rules by arbitrary selection. All of these properties make termination proofs easy.

First consider the easiest case, where the only form of interaction is horizontal. Let E1 :=: I1(x) and E2 :=: I2(x) be two invariants that depend on a common argument x but do not depend on each other. Suppose that these two invariants are falsified by a modification dx to x. Suppose also that the pre-difference code for E1 (respectively E2) with respect to dx contains no occurrences of replaceable terms of E2 (respectively E1). Then the collective pre- and post-difference code is,

$$\partial\text{-}\{E1,E2\}\langle dx\rangle = \partial\text{-}E1\langle dx\rangle\ \partial\text{-}E2\langle dx\rangle \quad \text{or} \quad \partial\text{-}E2\langle dx\rangle\ \partial\text{-}E1\langle dx\rangle$$

$$\partial+\{E1,E2\}\langle dx\rangle = \partial+E1\langle dx\rangle\ \partial+E2\langle dx\rangle \quad \text{or} \quad \partial+E2\langle dx\rangle\ \partial+E1\langle dx\rangle$$

For example, consider the defining invariants E1 = [sal(x): x ∈ s] and E2 = #s, and the modification s with:= z. In taking the differential of E1 and E2 relative to a program region R, the difference code insertion and the obvious binding rules [sal(x): x ∈ s] -> E1 and #s -> E2 would be carried out in a straightforward way.

A more unusual example is that of the two defining invariants E1 = {[f(x),x]: x ∈ s} and E2 = {[g(x),x]: x ∈ s} with binding rules {x ∈ s | f(x) = q1} -> E1{q1} and {x ∈ s | g(x) = q2} -> E1{q2}, where q1 and q2 are discontinuity parameters. Difference code insertion relative to element additions to and deletions from s and indexed assignments to f and g are straightforward. However, this example illustrates reductions being performed in different orders within the same program region. Program expression {x ∈ s | f(x) = {y ∈ s | g(y) = u+v}} is reduced to E1{E2{u+v}}, while expression {x ∈ s | g(x) = {y ∈ s | f(y) = u+v}} is reduced to E2{E1{u+v}}.

Clearly, this poses no problem for bottom-up reduction, and a generalized form of pre-difference, post-difference, and differential for collections of invariants is easy.

Let us now permit vertical as well as horizontal interaction. Suppose that one invariant E1 :=: f(E2) depends on another invariant E2 :=: g(x) and perhaps also on x. Then we must form the difference of E1 with respect to the difference of E2 with respect to changes in x, as the following collective difference rule indicates:

$$\partial\text{-}\{E1,E2\}\langle dx\rangle = \partial E1\langle \partial\text{-}E2\langle dx\rangle\rangle \; \partial\text{-}E1\langle dx\rangle$$

$$\partial+\{E1,E2\}\langle dx\rangle = \partial+E1\langle dx\rangle \; \partial E1\langle \partial+E2\langle dx\rangle\rangle$$

Exchanging E1 and E2 in the code just above would be incorrect, since E2 would not be reestablished relative to any modification to E1 occurring in the difference code for E1 with respect to dx.

The two equalities E1 = #E2 and E2 = {x ∈ s | k(s)} and the modification s with:= z exemplify this kind of interaction. Note also, that when set E2 is augmented, then E1 is incremented without making any reference to E2. Consequently, E1 is not needed to maintain E2. If it is not needed for any other purpose, it can be eliminated.

As was observed in[34], vertical interaction can make invariants and the code that maintains them useless. However, that reference applied global useless code elimination to remove useless invariants after they were maintained in a program. Here, however, we want to integrate useless invariant analysis with finite differencing to make this analysis more efficient and to make differencing more powerful. In general the output parameter E of an invariant is useless if it is the input variable to some other invariant and,

(i) no difference code for any invariant (that depends on E) with respect to E makes reference to E;

(ii) all occurrences of E introduced by reductions are eliminated by further reductions due to other invariants

(iii) no reference to E introduced within its own difference code is part of a control structure and can determine whether or not any useful statements are executed.

For example, in a program region R where n is incremented by 1 repeatedly, we can remove the nest of bottlenecks in the naive compound growth formula $P \times (1 + r)^n$ using the three invariants: t1 = 1 + r, $t2 = t1^n$, and t3 = P × t2. Invariant t2 is useless, because it is not referenced by the difference code maintaining t3.

Next consider an interesting special case when a new invariant is formed from an old one by parameter substitution in which the same parameter appears more than once. How do we maintain invariant

(4) E :=: f(x,x,...,x,xm+1,...,xn)

with respect to a modification dx to the variable x (occurring m times)? An easy answer is to replace (4) with m - 1 copies of x using invariants Ei = x, i=1,...,m-1 and E' :=: f(x,E1,...,Em-1,xm+1,...,xn). However, more space efficient methods, requiring no more than one copy (and sometimes no copies) of x, were presented in [37]. The idea is to first consider the difference rules for the related function (formed from (4) by replacing the m occurrences of x by new variables

x1,...,xm, each containing the value of x)

(4') E' = f(x1,...,xn)

with respect to successive modifications dxi i=1,...,m, each just the same as dx. Then we can sometimes distribute this difference code both before and after the modification dx in such a way that all occurrences of xi, i=1,...,m within the difference code placed before dx refers only to the old value of x (before it is changed), and all such occurrences in the difference code place after dx refers to the new value of x.

For example, consider the cross product s × s of set s, when the library of invariants only contains an invariant E = t × q with difference rules

∂-E<t with:= a> = (for y ∈ q) E with:= [a,y]
and
∂-E<q with:= a> = (for y ∈ t) E with:= [y,a]

Rearrangement of the preceding difference code around a single modification to s leads to the following correct difference code for s × s:

(for y ∈ s) E with:= [a,y]
s with:= a
(for y ∈ s) E with:= [y,a]

Various papers on Fortran strength reduction (e.g., Allen, et. al.[3]) discuss handling product i × i as a special case. The reader should refer to[37] for details of the general method.

We conclude this subsection with a simple chain rule that extends the previous definitions of pre- and post-difference code and differentials from single invariants to collections. Suppose we have a set {E1,...,En} of invariants whose dependency graph forms a dag and contains only horizontal and vertical interactions. Then the collective differential of {E1,...,En} relative to a program region R, denoted by ∂{E1,...,En}<R> is a new code block formed from R by taking the following two steps:

(i) for each modification dx occurring within R, pick some E1 that does not depend on any of the output variables E2,...,En, and replace dx by ∂{E2,...,En}<∂E1<dx>>;

(ii) perform repeated reductions associated with each of the invariants E1,..,En throughout the rest of B.

If the conditions of step (ii) above hold, then the collective pre- and post-difference code blocks are:

∂-{E1,...,En}<dx> = ∂{E2,...,En}<∂-E1<dx>>
∂-{E2,...,En}<dx>

∂+{E1,...,En}<dx> = ∂+{E2,...,En}<dx>
∂{E2,...,En}<∂+E1<dx>>

Once again we can adapt the chain rule proof from[33] to this slightly different setting. The recursive step (i) always terminates.

3.4. Dynamic and Supporting Invariants

Allowing dynamic interaction alters analysis for invariants significantly and changes the logic of the chain rule from the last subsection. Allowing maintenance of one invariant to introduce new bottlenecks whose removal would require new invariants to be maintained (as in Briggs's algorithm) complicates both the selection and scheduling of invariants. Since now invariants can be selected based on matching binding terms occurring within the difference code as well as in the original program text, proving a reasonable finite bound on the number of invariants selected can be a problem.

Let us first consider dynamic interaction. Suppose we want to remove the sum contained in the assignment j := i + c using invariant E1 = i + c in a program region R. Then because of the 'occurs check' condition, for i + c to be removable, no modification to i occurring in R can depend on j either directly or indirectly. (Recall our conservative flow assumptions that any statement can be executed after any other statement.) Further, we know that any other invariant E2 = f(j) cannot be used to remove an occurrence of expression f(j) on the right-hand-side of an assignment to any variable on which i depends either directly or indirectly.

The preceding observation leads to a general rule for analyzing invariants. Let S be the set of all expressions matched by binding terms of invariants and occurring in assignments. Then S must contain an expression f(x1,..,xn), involving variables x1,...,xn, with the following property. No other expression belonging to S can occur within any assignment to a variable on which any of the variables x1,...,xn depend, either directly or indirectly. We call f(x1,..,xn) *minimal*, and observe that S can be partially ordered. Thus, to handle dynamic interaction properly, invariant analysis must be modified to first examine assignment statements in topological order starting from minimal expressions. After that the remaining expressions can be examined bottom-up.

With dynamic interaction we modify the recursive step of the chain rule in the preceding subsection in the following way. Let dx be a modification to a variable x. To compute the differential ∂{E1,...,En}<dx> we perform the following steps:

(i) first perform repeated reductions within dx using any of the invariants that do not depend on x;

(ii) take the differential ∂{Ei,...,En}<dx'> (as in the preceding subsection) of the remaining invariants Ei,...,En with respect to the new modification dx' resulting from step (i).

Finally, consider supporting interaction. Suppose that the difference code of one invariant E1 = g(x) with respect to a change dx to variable x contains the binding term f(x) of another invariant E2 = f(x); i.e., invariant E2 supports invariant E1 with respect to dx. Suppose also that E1 does not support E2 with respect to dx or with respect to any modifications to E1 in the difference code for E1 with respect to dx. Finally, suppose that E1 does not depend on E2. In this case the differential of E2 with respect to the pre-difference block ∂-E1<dx> (respectively postdifference ∂+E1<dx>) will replace occurrences of f(x) involving the old value of x (respectively new value of x) appearing in these blocks by occurrences of E2. The correct collective difference code is,

$$\partial\text{-}\{E2,E1\}\langle dx\rangle = \partial E2\langle \partial\text{-}E1\langle dx\rangle\rangle \; \partial\text{-}E2\langle dx\rangle$$

$$\partial\text{+}\{E2,E1\}\langle dx\rangle = \partial\text{+}E2\langle dx\rangle \; \partial E2\langle \partial\text{+}E1\langle dx\rangle\rangle$$

Observe that the collection of reference counts embodied in the equalities E2(w) = #[x ∈ t | x = w] supports the equality E1 = {x: x ∈ t} with respect to the modification t with:= z. For this example, t is a tuple, t with:= z concatenates z to the end of t, E1 stores the set of values in tuple t, and expression [x ∈ t | x = w] forms a new tuple containing all the elements of t with value w.

To appreciate the preceding conditions consider two invariants E1 = f1(x) and E2 = f2(x), which both depend on the same variable x. Let dx be an assignment to x, and let ∂+E1<x> and ∂+E2<x> be empty. If ∂-E2<x> contains an occurrence of f1(x), then it will not be redundant within the differential

$$\partial\{E2,E1\}\langle x\rangle = \partial\text{-}E1\langle x\rangle \quad \partial\text{-}E2\langle x\rangle \quad dx$$

because this occurrence involves the old value of x; but E1, being just modified, reflects the new value of x.

Briggs's method extends the preceding idea further. In order to tabulate a d-th degree polynomial p(x), he used d+1 invariants, t = p(x), t1 = p1(x), t2 = p2(x),...,td = pd, where t1 supports t with respect to a change dx in x, and ti+1 supports ti with respect to dx i= 1,...d-1. Since none of the difference code for t1,...,td contains occurrences of p(x), we can guess that the correct differential is

$$\partial\{t,t1,...,td\}\langle dx\rangle = \partial\{t1,...,td\}\langle\partial t\langle dx\rangle\rangle$$

Continuing to unwind the differential is straightforward.

It is interesting to examine some arithmetic examples that have inspired traditional strength reduction. Consider invariants for sums E1 = x + c, where c is a constant, and for products E2 = x × y. Relevant difference rules include:

$$\partial\text{-}E1\langle x +:= y\rangle = E1 +:= y$$
$$\partial\text{-}E2\langle x +:= z\rangle = E2 +:= z\times y$$

Note that the difference code for E1 makes no reference to x, so is capable of making x useless. Note also, that the difference code for E2 is worthwhile only if it can be supported by an invariant E3 = z×y.

It turns out that these invariants for sums and products can reduce all products in a d-th degree polynomial p(x) programmed using Horner's rule in a loop where x is incremented by a constant. A similar observation has been made before by Cocke and Schwartz[9] and Allen, Cocke, and Kennedy[3]. What has not been said is that the result does not reproduce Briggs's method, but produces over d! sums and temporaries. Cleanup improves matters by only small constant factors.

The reason is this. A d-th degree polynomial written out using Horner's Rule requires d invariants to remove the visible products. But each of these invariants must be supported by other products occurring in difference code but not directly in the polynomial. To count the number of invariants and also the number of operations in the collective difference code, it is useful to consider the degree of each argument of a product E = T×Q for each invariant E. Let temps(i,j) be one plus the number of invariants that either directly or indirectly support an invariant E = T×Q, where T has degree i and Q has degree j. For our example, note that one of the arguments is always x, which has degree 1, or is a constant, which has degree 0.

It is easy to see that

temps(0,0) = 1,
temps(1,0) = 2,
temps(0,i) = 1 + i temps(0,i-1), and
temps(1,i) = 1 + temps(0,i) + i temps(1,i - 1)

The total number of invariants generated is,

$$\sum_{i=0}^{d} \text{temps}(1,i) = \Theta(d!)$$

The total number of operations is comparable. In Part II of this paper we give a simple transformation integrated with the chain rule to recover Briggs's method exactly.

One last example is about detecting all region constant expressions in a parse tree, where free(x) maps each expression node x to the set of free parameters on which x depends; RC is the set of nodes that are constant denotations, and RCV is the set of variable nodes that are region constants. Then the set of nodes corresponding to region constant expressions is specified by

$$\text{lfp n. } \{x \in \text{domain free} \mid \text{free}(x) \subseteq n \cup RCV \cup RC\}$$

which is the least fixed point of the set former relative to parameter n. The fixed point can be computed by the following code:

```
n := {}
( while ∃ x ∈ {y ∈ domain free | #(free(y) - (RCV ∪ RC ∪ n)) = 0})
   n with:= x
end
```

which follows from an easy fixed point argument[7, 11, 48]. Bottom-up analysis of the costly computation at the top of the while-loop determines the following six invariants,

$e1 = RCV \cup RC$
$e2 = e1 \cup n$
$e3 = \{[y,w] \in \text{free} \mid w \notin e2\}$
$e4(y) = \#e3\{y\}$
$e5 = \text{free}^{-1}$
$e6 = \{y \in \text{domain free} \mid e4(y) = 0\}$

where e5 supports e3 relative to element additions to e2. Further analysis determines that e1 is a region constant, e2 and e3 are useless invariants, and the result of the chain rule is the expected linear time code.

Consider the general case of n invariants, Ei i=1,...,n. We say that E1 is *minimal* with respect to invariants {E1,...,En} and a modification dx if

(i) E1 does not depend on any of the variables Ei, i=2,...,n; and

(ii) E1 does not support Ei, i=2,...,n with respect to dx;

Then the collective differential ∂{E1,...,En}<R> is a new code block formed from R by,

(i) first performing reductions within dx from all those invariants that do not depend on x;

(ii) next replacing each modification dx occurring within R by ∂{E2,...,En}<∂E1<dx>> where E1 is minimal with respect to Ei, i=1,..,n and dx;

(iii) finally, perform repeated reductions associated with each of the invariants E1,..,En throughout the rest of B.

4. Postscript

Part II of this paper presents a taxonomy of finite differencing problems and solutions based on the framework discussed in Part I. In particular finite differencing is divided into linear differencing in which invariants can only depend on one variable. Within this category is one-level linear differencing in which reductions are highly simplified and only horizontal interaction of invariants are permitted. This class corresponds to early reduction in strength[44].

In multi-level linear differencing all interactions are allowed. We present a one-pass linear randomized time algorithm that improves on the classical strength reduction algorithms of[8, 9, 30]. We then consider two aspects of nonlinear differencing, where invariants can depend on more than one variable. First we present an efficient algorithm supporting a variety of arithmetic identities. Next, nonlinear finite differencing is discussed from a transformational perspective. That is, we consider a very high level, declaration free, strongly typed, functional problem specification language implemented in RAPTS. Typings and universal sets are inferred from problem specifications. Further analysis to compile efficient data structures that implement sets and maps is obtained by transformations that are both guided by earlier analysis and also propagate modified semantic information for use by later transformations. Three successive transformations - fixed point iteration, set theoretic finite differencing, and data structure selection compile the specification language into C. In this context, we discuss the finite differencing phase of the high level compiler. Set theoretic finite differencing is also used to derive major pieces of the algorithms presented.

References

1. Aho, A., Hopcroft, J., and Ullman, J., *Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. Allen, F. E., "Program Optimization," *Annual Review of Automatic Programming*, vol. 5, pp. 239-307, 1969.
3. Allen, F. E., Cocke, J., and Kennedy, K., "Reduction of Operator Strength," in *Program Flow Analysis*, ed. Muchnick, S. and Jones, N., pp. 79-101, Prentice Hall, 1981.
4. Bancilhon, F., "Naive Evaluation of Recursively defined Relations," in *On Knowledge-Base Management Systems*, ed. Mylopoulos, J, pp. 165-178, 1985.
5. Bernstein, P., Blaustein, B., and Clarke, E., "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data," in *Proceedings 6th International Conference on VLDB*, pp. 126-136, Montreal, Canada, October, 1980.
6. Cai, J. and Paige, R., "Binding Performance at Language Design Time," in *ACM POPL*, pp. 85 - 97, Jan, 1987.
7. Cai, J. and Paige, R., "Program Derivation by Fixed Point Computation," *Science of Computer Programming*, vol. 11, pp. 197-261, 1988/89.

8. Cocke, J. and Kennedy, K., "An Algorithm for Reduction of Operator Strength," *CACM*, vol. 20, no. 11, pp. 850-856, Nov., 1977.

9. Cocke, J. and Schwartz, J. T., *Programming Languages and Their Compilers*, Lecture Notes, CIMS, New York University, 1969.

10. Cocke, J., *private communication, 1986.*

11. Cousot, P. and Cousot, R., "Constructive versions of Tarski's fixed point theorems," *Pacific J. Math.*, vol. 82, no. 1, pp. 43-57, 1979.

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

13. Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 5, pp. 269-271, 1959.

14. Earley, J., "Toward an Understanding of Data Structures," *CACM*, vol. 14, no. 10, pp. 617-627, Oct. 1971.

15. Earley, J., "High Level Operations in Automatic Programming," in *Proc. Symp. on Very High Level Langs.*, vol. 9, Sigplan Notices, Apr, 1974.

16. Earley, J., "High Level Iterators and a Method for Automatically Designing Data Structure Representation," *J of Computer Languages*, vol. 1, no. 4, pp. 321-342, 1976.

17. Floyd, R., "Algorithm 97: shortest path," *CACM*, vol. 5, no. 6, p. 345, 1962.

18. Fong, A. and Ullman, J., "Induction Variables in Very High Level Languages," in *Proceedings Third ACM Symposium on Principles of Programming Languages*, pp. 104-112, Jan, 1976.

19. Fong, A., "Elimination of Common Subexpressions in Very High Level Languages," in *Proceedings Fourth ACM Symposium on Principles of Programming Languages*, pp. 48-57, Jan, 1977.

20. Fong, A., "Inductively Computable Constructs in Very High Level Languages," in *Proceedings Sixth ACM Symposium on Principles of Programming Languages*, pp. 21-28, Jan, 1979.

21. Fredman, M. and Tarjan, R., "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," in *25th Annual Symp. on FOCS*, pp. 338 - 346, Oct., 1984.

22. Goldstine, H., *A History of Numerical Analysis*, Springer-Verlag, New York, 1977.

23. Goldstine, H., *The Computer from Pascal to Von Neumann*, Princeton University Press, Princeton, New Jersey, 1972.

24. Gries, D., *The Science of Programming*, Springer Verlag, 1981.

25. Hoffmann, C. and O'Donnell, J., "Pattern Matching in Trees," *JACM*, vol. 29, no. 1, pp. 68-95, Jan, 1982.

26. Hoover, R. and Teitelbaum, T., "Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars," in *Proc. Symp. on Compiler Construction*, pp. 39-50, 1986.

27. Horwitz, S. and Teitelbaum, T., "Generating Editing Environments Based on Relations and Attributes," *ACM TOPLAS*, vol. 8, no. 4, pp. 577-608, Oct. 1986.

28. Knorr, W., *private communication, 1982.*

29. Koenig, S. and Paige, R., "A Transformational Framework for the Automatic Control of Derived Data," in *Proceedings 7th International Conference on VLDB*, pp. 306-318, Sep, 1981.

30. Lowry, S. and Medlock, C., "Object Code Optimization," *CACM*, vol. 12, no. 1, pp. 13-22, Jan., 1969.

31. Marlowe, T. and Ryder, B., "An efficient hybrid algorithm for incremental dataflow analysis," in *17th Symp. ACM POPL*, 1990.

32. Meertens, L., "Algorithmics," in *Mathematics and Computer Science - Proc. CWI Symp. Nov. 1983*, ed. J. W. de Bakker, M. Hazewinkel and J.K. Lenstra, CWI Monographs Vol. I, North-Holland, 1986.

33. Paige, R. and Koenig, S., "Finite Differencing of Computable Expressions," *ACM TOPLAS*, vol. 4, no. 3, pp. 402-454, July, 1982.

34. Paige, R., *Formal Differentiation*, UMI Research Press, Ann Arbor, Mich, 1981.

35. Paige, R., "Transformational Programming -- Applications to Algorithms and Systems," in *Proceedings Tenth ACM Symposium on Principles of Programming Languages*, pp. 73-87, Jan, 1983.

36. Paige, R., "Applications of Finite Differencing to Database Integrity Control and Query/Transaction Optimization," in *Advances In Database Theory*, ed. Gallaire, H., Minker, J., and Nicolas, J.-M., vol. 2, pp. 171-210, Plenum Press, New York, Mar, 1984.

37. Paige, R., "Programming With Invariants," *IEEE Software*, vol. 3, no. 1, pp. 56-69, Jan, 1986.

38. Pettorossi, A. and Proietti, M., *The Automatic Construction of Logic Programs*, IFIPS WG2.1 TR 568 CHM-15, 1989.

39. Reps, T., Teitelbaum, T., and Demers, A., "Incremental Context-Dependent Analysis for Language-Based editors," *ACM TOPLAS*, vol. 5, no. 3, pp. 449-477, July, 1983.

40. Robinson, J. A., "A Machine Oriented Logic Based on the Resolution Principle," *JACM*, pp. 23-41, January, 1965.

41. Rosen, B. K., "Degrees of Availability," in *Program Flow Analysis*, ed. Muchnick, S., Jones, N., pp. 55-76, Prentice Hall, 1981.

42. Roussopoulos, N., *The Incremental Access Method of View Cache: Concept, Algorithm, and Cost Analysis*, Computer Science Technical Report Series UMIACS-TR-89-15, CS-TR-2193, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, February, 1989.

43. Sacca, D. and Zaniolo, C., *Differential Fixpoint Methods and Stratification of Logic Programs*, MCC TR ACA-ST-032-88, 1988.

44. Samelson, K. and Bauer, F. L., "Sequential Formula Translation," *CACM*, vol. 3, no. 2, pp. 76-83, Feb, 1960.

45. Schwartz, J., Dewar, R., Dubinsky, D., and Schonberg, E., *Programming with Sets: An introduction to SETL*, Springer-Verlag, 1986.

46. Smith, D., "KIDS - A Knowledge-Based Software Development System," in *Proc. Workshop on Automating Software Design, AAAI-88*, pp. 129-136, Sept. 1988.

47. Tarjan, R., "A Unified Approach to Path Problems," *JACM*, vol. 28, no. 3, pp. 577-593, July, 1981.

48. Tarski, A., "A Lattice-Theoretical Fixpoint Theorem and its Application," *Pacific J. of Mathematics*, vol. 5, pp. 285-309, 1955.

49. Ullman, J., *Principles of Database and Knowledge-Base Systems, I,* Computer Science Press, 1988.

50. Wachter, R., *private communication, 1989.*

51. Whang, K. and Navathe, S., "An Extended Disjunctive Normal Form Approach for Processing Recursive Logic Queries in Loosely Coupled Environments," in *Proc. 13th Intl. Conf. on Very Large Data Bases*, pp. 275-287, Sept. 1987.

52. Williams, J. W. J., "Algorithm 232: Heapsort," *CACM*, vol. 7, no. 6, pp. 347-348, 1964.

53. Yellin, D. and Strom, R., *INC: A Language for Incremental Computations*, IBM Research Center/Yorktown Heights RC14375(#64375), 1989.