# INDUCTION VARIABLES IN VERY HIGH LEVEL LANGUAGES*

Amelia C. Fong and Jeffrey D. Ullman
Princeton University
Princeton, N. J. 08540

## Abstract

We explore the notion of an induction variable in the context of a set-theoretic programming language. An appropriate definition, we believe, involves both the necessity that changes in the variable around a loop be easily computable and that they be small. We attempt to justify these requirements and show why they are independent assumptions. Next the question of what operators on sets play the role of +, - and * for arithmetic languages is explored, and several theorems allowing us recursively to detect induction variables in a loop are given. It is shown that most of the usual set operations do fit nicely into the theory and help form induction variables. The reason most variables fail to be induction variables concerns the structure of control flow, more than it does the operators applied.

## 1. Background

"Reduction in strength," that is, the replacement of multiplication by addition in a loop, and its attendant detection and elimination of induction variables (those whose value assumes an arithmetic progression at a point) forms a key optimization for arithmetic languages like FORTRAN [1-4]. Recently, there has been considerable interest in algorithms for performing this kind of optimization [5-7]. However, in the FORTRAN environment, there is never more than a constant factor speedup available by these methods.

On the other hand, recent proposals such as [8-9] have dealt with reduction in strength applied to set-theoretic languages. In this context, reduction in strength becomes a method for altering algorithms to improve their asymptotic running time, and order of magnitude improvement is possible. Earley [8] proposes "iterator inversion," which is a powerful technique for improving algorithms automatically. Unfortunately, as [8] admits, it is not clear how to tell in advance whether a transformation is helping or hurting the running time.

Our answer to that problem is that a set of permissible transformations must be built up "from the bottom," starting with a few obviously safe transformations and developing additional safe transformations recursively. Schwartz [9] has a similar idea, based on the notion of "continuous" functions, where an expression $e(x_1, x_2, \ldots, x_n)$ is said to be continuous in $x_i$ if small changes in (presumably set-valued) variable $x_i$ produces a change in e which can be easily calculated from the old value of e and the old and new values of $x_i$. We propose a related idea, but one which we believe is more general in its treatment of Boolean valued operators on sets, such as the relation of set inclusion, and in its extension from expressions to programs.

## 2. The Model

We assume a set-oriented language such as SETL [10]. The operations which we assume can be done in unit time are:

(1) arithmetic on integers

(2) insertion of an element into a set

* Work partially supported by NSF grant DCR-74-15255.

(3) deletion of an element from a set

(4) selection of some member from a set

(5) testing whether an atom is in a set

These assumptions are valid, at least in an expected time sense, if one uses a hash table representation for sets, such as in [10], with elements which are sets represented by pointers to their values. We take it as a corollary to (4) that a set may be tested for emptiness in unit time.

It is assumed further that we are presented a program as a flow graph, with basic blocks consisting of three-address statements, e. g., A = B U C, but not A = B U C $\cap$ D, which would appear as:

```
T = C ∩ D
A = B U T
```

## 3. Goals

We are interested primarily in developing a theory of induction variables and reductions in strength for set-theoretic languages that is analogous to the one for FORTRAN-like ones. However, in the environment of a set-theoretic language, where large amounts of time are already given up to system overhead, it does not make sense to concern ourselves solely with constant factor speedups, as one does for FORTRAN. We orient our definitions so that induction variables are those for which an order of magnitude improvement in the running time of the program is possible by properly evaluating its induction variables.

## 4. Induction Variables

The canonical situation for a FORTRAN level induction variable is a loop in which statements like:

```
I = I + 1
J = 2 * I
```

appear. If I is not changed elsewhere in the loop, it is clearly an induction variable. Moreover, J is an induction variable, at least at the point immediately after J = 2 * I. We can arrange to maintain the value of J by additions and subtractions only, if we create a temporary T whose value is always twice that of I. Then follow I = I + 1 by T = T + 2, and where I is initialized outside the loop, initialize T to twice I. Replace J = 2 * I by J = T, yielding the sequence:

```
I = I + 1
T = T + 2
J = T
```

In many cases we can dispense with I and/or identify J with T, adding to our savings. In any event, we have eliminated the "expensive" multiplication at the cost of several copies and additions- perhaps a worthwhile change.

Now let's repeat the above in the set-theoretic context. It is generally recognized that the basic role played by statements of the form I = I + 1 and I = I - 1 in the arithmetic world belongs to S = S U {x} and S = S - {x} in the set world (see [8,9]). These statements are just insertions and deletions of elements, operations which we have taken as primitive. We might see in a loop the pair of statements:

```
A = A U {x}
C = A U B
```

where B is presumed constant within the loop for simplicity in our present informal discussion. It is natural to suppose that we could create set T, whose value will always be that of A U B. Then we could follow A = A U {x} by T = T U {x} and initialize T to A U B outside the loop. If we replace C = A U B by C = T we have:

```
A = A U {x}
T = T U {x}
C = T
```

Have we saved significantly here? The answer is that probably we haven't, since the operation of copying T and assigning the value to C takes the same order of time as computing A U B.

Of course it is possible that on examination of the entire loop we would find that C and T could be identified, thus replacing the union of arbitrary sets A and B by the adjoining of one element x. This would definitely be an order of magnitude savings. However, it is possible that the value of C is used in the loop in a way that makes its identification with T impossible. In that case, we propose the following.

Definition: A loop is a set of blocks with a header which dominates all other blocks in the loop, i. e., access to the loop is via the header only.

Definition: Define $\Delta(A, p)$, for identifier A and point p to be the pair of sets $\Delta^+(A, p)$ and $\Delta^-(A, p)$, where $\Delta^+(A, p)$ is

the set of elements added to A and not removed from A since the last time control passed point p, and $\Delta^-(A, p)$ is the set of elements removed from A and not added. That is, $\Delta^+(A, p)$ and $\Delta^-(A, p)$ are new set valued identifiers whose value it is possible to maintain. Every time control passes point p, we set $\Delta(A, p)$ to $(\emptyset, \emptyset)$ and alter $\Delta(A, p)$ as the "current" value of A changes and as long as control does not again reach p.

Definition: Call A an induction variable of loop L at point p if there is a constant upper bound on the work necessary to maintain the value of $\Delta(A, p)$ between any two consecutive times that control passes point p, staying within loop L.

Returning to our informal example of the statements:

```
A = A U {x}
C = A U B
```

we may let p be the point immediately following C = A U B. If the only assignment to A in the loop is A = A U {x}, then surely $\Delta(A, p)$ can be maintained in constant work by writing the piece of program as:

```
if x not in A
   then Δ+(A, p) = Δ+(A, p) U {x}
A = A U {x}
C = A U B
Δ+(A, p) = ∅
Δ-(A, p) = ∅
```

If B is a constant within the loop, we can use $\Delta(A, p)$ to simulate the assignment C = A U B. Technically, what happens is this. We observe that since B presumably does not change, the change in the expression A U B from point p is almost the same as $\Delta(A, p)$. In particular, $\Delta^+(AUB, p) = \Delta^+(A, p) - B$ and $\Delta^-(AUB, p) = \Delta^-(A, p) - B$. If both $\Delta^+(A, p)$ and $\Delta^-(A, p)$ are bounded, as they are in this example, then $\Delta^+(AUB, p)$ is easy to compute. In general, if A and B vary in the loop, we can compute $\Delta(AUB, p)$ from $\Delta(A, p)$ and $\Delta(B, p)$, provided both are small.

Now we assume that C is only assigned at C = A U B within the loop. Therefore, $\Delta(C, p) = \Delta(AUB, p)$, and we can replace the above program by:

```
if x not in A then
   Δ+(A, p) = Δ+(A, p) U {x}
if x not in B then
   Δ+(AUB, p) = Δ+(AUB, p) U {x}
A = A U {x}
C = C U Δ+(AUB, p) - Δ-(AUB, p)
Δ+(A, p) = ∅
Δ-(A, p) = ∅
Δ+(AUB, p) = ∅
Δ-(AUB, p) = ∅
```

Note that the $\Delta^-$'s can be ignored here but were included for form. Also, depending on what goes on elsewhere in the loop, we may drop consideration of $\Delta(A, p)$ or even of A itself.

We see that in the above simple case we have been able to replace a union of arbitrary sets A and B by unions and differences of sets that remain small, in fact they have at most one element. Thus an asymptotic order of magnitude savings has been achieved.

We would now like to formalize further the two important factors in this type of code improvement, (1) the ability to efficiently maintain $\Delta(e, p)$ for expressions e and (2) the boundedness of these sets.

Definition: We use $\Delta(e, p)$, for expression e and point p, to stand for the pair $\Delta^+(e, p)$ and $\Delta^-(e, p)$. $\Delta^+(e, p)$ represents the set of elements added to the set denoted by e and not removed from that set, since the last time control passed point p. $\Delta^-(e, p)$ represents the set of elements removed from and not added to that set since control last passed p. Note that this definition coincides with the earlier definition of $\Delta$ in the case e is a single identifier.

We say e is an induction expression of loop L at point p whenever we can maintain $\Delta(e, p)$ with a bounded amount of work between successive times through point p, as long as control does not leave L.

Definition: Let us say an expression is of limited perturbation at point p in loop L if $\Delta^+(e, p)$ and $\Delta^-(e, p)$ are of bounded size as long as control stays within loop L.

It is important to note that the notions of "induction expression" and "expression of limited perturbation" are not the same, nor does one imply the other. For example, $\Delta^+(e, p)$ might be known to be either $\emptyset$ or {a}, but we have to solve the halting problem for Turing machines to tell which. Thus an expression could be of limited perturbation yet not be an induction expression. Conversely, consider the situation of Fig. 1, where A could be

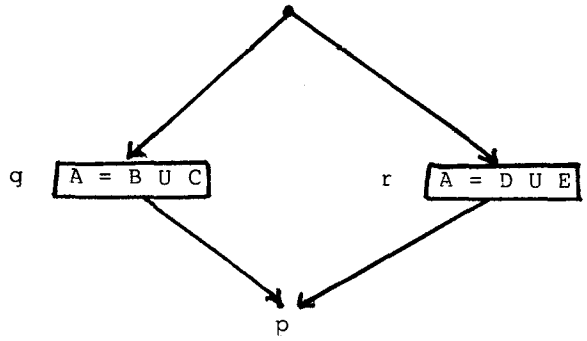


Figure 1.

assigned either B U C or D U E before control reaches point p. Then if B U C and D U E are induction expressions, at points q and r, respectively, we shall see that A is an induction variable (hence an induction expression) at point p. yet B U C and D U E can differ by arbitrary amounts, and we might travel a path such as q...p...r...p, so $\Delta(A, p)$ is surely not of limited perturbation.

## 5. Building Induction Variables and Expressions

We shall now develop the mechanism whereby induction variables can be detected in a straightfoward manner and reduction in strength performed on them where possible, retaining the assurance that what changes to the program are made will actually improve things. The theorems presented here encompass most of the standard set operators. The implication is that the reason reduction in strength cannot be performed in many cases has to do with the structure of control flow in the program rather than the properties of the operators used in calculation.

The first theorems enable us to construct new induction variables and expressions from old ones. These theorems will all be stated in a simple form that ignores the possibility that two or more variables could be mutually dependent induction variables, e. g., in a loop containing assignments A = B U {x} and B = A U {y}, both A and B might be induction variables. Once the principles are understood, this type of extension is easy.

Definition: Call an assignment incidental if it is of the form A = A U {x} or A = A - {x}.

Theorem 1: If in loop L, identifier A has only incidental assignments, then it is an induction variable and is of limited perturbation at all points in L where there is a bound on the number of incidental assignments to A encountered going from p to p in L.

Proof: The size of $\Delta^+(A, p)$ and $\Delta^-(A, p)$ changes by at most one and can be updated by a bounded amount of work each time an incidental assignment of A is encountered. Since there is only a bounded number of incidental assignments of A going from P to P within L, the size of $\Delta^+(A, p)$ and $\Delta^-(A, p)$, and the total work involved in maintaining $\Delta(A, p)$ from p to p are also bounded. Hence A is an induction variable and is of limited perturbation at p.

The example in Figure 2 shows a situation where A is not an induction variable and is not of limited perturbation at a point p in loop L even though all assignments to A in L are incidental.

Theorem 2: If in loop L, A and B are induction variables and of limited perturbation at point p, then A U {x}, A - {x}, A U B, A |-| B and A - B (in general, any binary Boolean operation on A and B) are induction expressions and of limited perturbation at p.

Proof: We shall show that if e is one of the expressions A U {x}, A - {x}, A U B, A |-| B, A - B, then $\Delta^+(e, p)$ and $\Delta^-(e, p)$ are bounded in size and can be obtained from $\Delta(A, p)$ and $\Delta(B, p)$ using a bounded amount of work. Hence e is an induction expression and is of limited perturbation at p.

First let e be the expression A U {x}. Then $\Delta(e, p)$ can be calculated from $\Delta(A, p)$ by the following program:

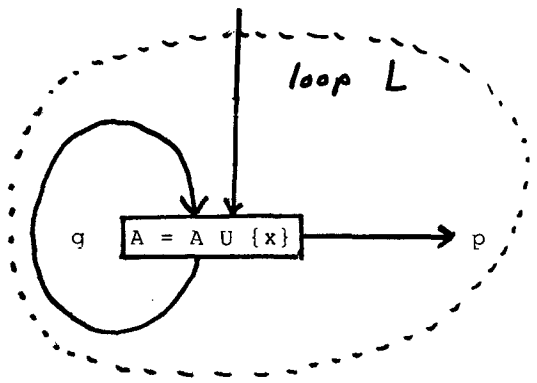


Figure 2.

$\Delta^+(e, p) = \Delta^+(A, p)$
$\Delta^-(e, p) = \Delta^-(A, p)$
if x in $\Delta^-(A, p)$
then $\Delta^-(e, p) = \Delta^-(e, p) - \{x\}$
else if x not in $\Delta^+(e, p)$
then $\Delta^+(e, p) = \Delta^+(e, p) \cup \{x\}$

Since $\Delta^+(A, p)$ and $\Delta^-(A, p)$ are bounded in size and can be maintained using a bounded amount of work, $\Delta^+(e, p)$ and $\Delta^-(e, p)$ are also bounded in size and can be obtained using a bounded amount of work.

Similarly, let e be A - {x}. $\Delta(e, p)$ can be calculated as follows:
$\Delta^+(e, p) = \Delta^+(A, p)$
$\Delta^-(e, p) = \Delta^-(A, p)$
if x in $\Delta^+(e, p)$
then $\Delta^+(e, p) = \Delta^+(e, p) - \{x\}$
else if x not in $\Delta^-(e, p)$
then $\Delta^-(e, p) = \Delta^-(e, p) \cup \{x\}$

Let e be A $\cup$ B. Then $\Delta^+(e, p) = \Delta^+(A, p) \cup \Delta^+(B, p)$ and $\Delta^-(e, p) = [\Delta^+(A, p) - v(b)] \cup [\Delta^-(B, p) - v(A)]$, where v(A) and v(B) denote the current value of A and B respectively.

Note that both $\Delta^-(A, p) - v(B)$ and $\Delta^+(B, p) - v(A)$ may be obtained in time proportional to the size of $\Delta^-(A, p)$ and $\Delta^+(B, p)$.

For example, let $C = \Delta^-(A, p) - v(B)$. C can be obtained by the following piece of code:

C = $\emptyset$
for x in $\Delta^-(A, p)$ do
if x not in v(B) then C = C $\cup$ {x}

Since we have assumed that membership testing and insertion can be performed in unit time, C can be obtained in time proportional to the size of $\Delta^-(A, p)$.

Let e be A |-| B. Then $\Delta^+(e, p) = [\Delta^+(A, p) \text{ |-| } v(B)] \cup [\Delta^+(B, p) \text{ |-| } v(A)]$ and $\Delta^-(e, p) = \Delta^-(A, p) \cup \Delta^-(B, p)$.

Let e be A - B. Then $\Delta^+(e, p) = [\Delta^+(A, p) - v(B)] \cup [\Delta^-(B, p) \text{ |-| } v(A)]$ and $\Delta^-(e, p) = [\Delta^-(A, p) \text{ |-| } \Delta^+(B, p)] \cup [\Delta^-(A, p) - v(B)] \cup [\Delta^+(B, p) \text{ |-| } v(A)]$.

Before going on to Theorem 3, we need some definitions of operations on the $\Delta$'s.

Definition: Let A,B,C,D be sets. Define (A,B) [+] (C,D) to be the pair

((A-D) $\cup$ (C-B), (D-A) $\cup$ (B-C) )

Definition: Let A, B, E, F be sets. Define (E,F) [-] (A,B) to be the pair

((E $\cup$ B) - (F $\cup$ A), (F $\cup$ A) - (E $\cup$ B))

Lemma 1: Let p, q, r be points in loop L. Suppose control passes from p to q through path l1, then from q to r through path l2. Let $\Delta_1$ be the pair $\Delta(A, p)$ at q after control passes from p to q through path l1. Let $\Delta_2$ be the pair $\Delta(A, q)$ at r after control passes from q to r through path l2. Let $\Delta_3$ be $\Delta(A, p)$ at r after control passes from p to r through l1 followed by l2. Then if we write each $\Delta_i$, i = 1,2,3 as the ordered pair $(\Delta_i^+, \Delta_i^-)$, then

(1) $\Delta_3 = \Delta_1$ [+] $\Delta_2$

(2) $\Delta_2 = \Delta_3$ [-] $\Delta_1$

Proof: The proof of (1) is straightforward and is omitted here.

To prove (2), let $\Delta_1$, $\Delta_2$, $\Delta_3$ be (A, B), (C, D) and (E, F) respectively. By (1)

$$E = (A-D) \cup (C-B)$$

$$F = (B-C) \cup (D-A)$$

We claim that C = (E $\cup$ B) - (F $\cup$ A), i.e., C = [(A-D) $\cup$ (C-B) $\cup$ B] - [(B-C) $\cup$ (D-A) $\cup$ A] = [(A-D) $\cup$ C $\cup$ B] - [(B-C) $\cup$ D $\cup$ A].

Let T1 = (A-D) $\cup$ C $\cup$ B and let T2 = (B-C) $\cup$ D $\cup$ A. We shall show that C is a subset of T1 - T2, and T2 - T1 is a subset of C.

For all x in C, x is in (A-D) $\cup$ C $\cup$ B, i.e., x is in T1. Obviously x is not in (B-C). Also x is not in D because C and D are disjoint by definition of $\Delta$. Again x is not in A because A and C are disjoint due to the fact that l1 and l2 are consecutive paths. Hence x is not in (B-C) $\cup$ D $\cup$ A, i.e. not in T2. Therefore x is in T1 - T2. i.e. C is a subset of T1 - T2.

To prove that T1 - T2 is a subset of C, suppose the contrary, i.e. there exists x in T1 - T2 which is not in C. Since x is in T1 = (A-D) $\cup$ C $\cup$ B, and not in C, x must be in either (A-D)-C or in (B-C). In both cases x is in (B-C) $\cup$ D $\cup$ A, contradicting the assumption that x is not in T2. Hence T1 - T2 is a subset of C. Therefore C = T1 - T2.

That D = (F $\cup$ A) - (E $\cup$ B) can be proved in similar fashion.

Theorem 3: Suppose there is a unique assignment A = e which is always the last non-incidental assignment to A before control reaches p, and that there is a bounded number of incidental assignments to A going from assignment A = e to point p.

If e is an induction expression and of limited perturbation at the point of assignment A = e, then A is an induction variable and of limited perturbation at p.

Proof: Let q be the point of assignment A = e. We want to show that $\Delta(A, p)$ is of bounded size and can be obtained using a bounded amount of work.

Consider the path as control passes from p the i-th time to p the i+1-st time. Since q is always the last non-incidental assignment before control reaches p, we can consider the following three paths: $l_1$, followed by $l_2$, followed by $l_3$ where

$l_1$ is the path followed as control passes from q to p the ith time.

$l_2$ is the path followed as control passes from p the i-th time to q.

$l_3$ is the path followed as control passes from q to p the i+1-st time.

Let $\Delta$ denote the value of $\Delta(A, q)$ after control passes from q to q through path $l_1$ followed by $l_2$. $\Delta$ is bounded in size and can be maintained using bounded amount of work because A is an induction variable and is of limited perturbation at q.

Let $\Delta'$ denote the value of $\Delta(A, q)$ at p after control passes from q to p through path $l_1$. $\Delta'$ is bounded in size and can be maintained using a bounded amount of work because there are only a bounded number of incidental assignments between q and p.

Hence by (2) of Lemma 1, $\Delta(A, P)$ at q after control passes from p to q through path $l_2$ is given by

$$\Delta \; [-] \; \Delta$$

Let $\Delta''$ denote the value of $\Delta(A, q)$ at p after control passes from q to p through path $l_3$. By (1) of lemma 1, $\Delta(A, p)$ when control passes from p to p through path $l_2$ followed by $l_3$ is given by

$$\Delta \; [-] \; \Delta' \; [+] \; \Delta''$$

which is bounded in size and can be obtained using a bounded amount of work.

Theorem 3 illustrates a situation in which A can not be identified with the expression e in the loop L, and shows how it can be handled without using a temporary the size of A or copying between A and the temporary.

We can extend part of Theorem 3 to the common case where A has a value outside loop L, and the first time through point p after entering L, the external assignment to A is the most recent non-incidental assignment. In fact, a far stronger result is possible, as far as induction variables are concerned.

Theorem 4: Suppose at point p in loop L there are k possible assignments to A, say $A = e_1$, $A = e_2, \ldots, A = e_k$ which could be the last non-incidental assignment to A, and suppose all $e_i$ for which $A = e_i$ is actually in loop L are induction expressions and of limited perturbation at the point of assignment. Let $q_i$ be the point of assignment $A = e_i$. Suppose there is a bounded number of incidental assignments of A going from $q_i$ to p. Suppose further that there is a bound on the number of assignments to A encountered going from p to p in L. Then A is an induction variable of L at p.

Proof: Intuitively, the value of A at p switches among k expressions, each of which is an induction expression. If k copies of A are kept, where the i-th copy has the value of A after its most recent assignment to $e_i$, $\Delta(A, p)$ may be represented by a switch (an integer) and the list $\Delta(e_1, p), \ldots, \Delta(e_k, p)$. The switch is set to i to indicate that the i-th copy of A is currently applicable, i.e. any reference to A should be made to the i-th copy. If the last non-incidental assignment to A before reaching p is $A = e_i$, then $\Delta(e_i, p) = \Delta(e_i, q)$, if there is no assignments to A between q and p. If there are incidental assignments to A between q and p, $\Delta(e_i, p)$ may be obtained as in THeorem 3. $\Delta(e_i, q)$ and $\Delta(e_i, p)$ are reset only if the last non-incidental assignment to A before reaching p is $A = e_i$. The work to maintain the switch and all the $\Delta$'s is bounded, since each of the $\Delta$'s is maintained in bounded time, and the switch is changed a bounded number of times by the hypothesis of the theorem.

Note that we are missing from Theorem 4 a statement about A being of limited perturbation of L at p. The reason, obviously, is that one cannot always conclude such a statement, and Fig. 1 provides the canonical example why not. Thus, while one can oscillate between Theorems 2 and 3 to find more and more induction variables and expressions within a loop, one cannot do so between Theorems 2 and 4. We can, however, extend the idea of the switch to a generalization of Theorem 4.

Theorem 5: Consider a point p in loop L, and suppose that the finite collection $e_1, \ldots, e_k$ comprises all the formulas for the value of variable A at p, in terms of the values of variables the previous time through point p. If each of the $e_i$'s are induction expressions and of limited perturbation at p, then A is an induction variable at p.

Proof: There are only a finite number of different sequences of assignments which can affect the value of A as we travel from p to p within L, or else the set of expressions for A would be infinite. Represent $\Delta(A, p)$ by a switch and the set of $\Delta(e_i, p)$'s for all $1 \leq i \leq k$. k "copies" of A have to be kept so that each $\Delta(e_i, p)$ represents the change with respect to the i-th copy of A.

## 6. Boolean Valued Expressions

While we have stated our theory of induction variables in great generality, when it comes to specific operators that help form induction expressions we have only mentioned union, intersection and similar operations. In fact, the theory does extend nicely to the usual set theoretic relations, such as inclusion.

When dealing with an expression like $A \subseteq B$, whose value is Boolean, the $\Delta$ notation and the notion of limited perturbation are meaningless, and in fact the actual value of the expression is no harder to maintain than it is to increment. Thus we define a Boolean valued expression to be an induction expression of L at p if its value (rather than its increment) can be maintained with a constant amount of work from p to p within L.

Theorem 6: If A and B are induction variables, then $A \subseteq B$ and $A = B$ (as well as a variety of similar relations) are induction expressions.

Proof: Consider $A \subseteq B$. By Theorem 2, $A - B$ is an induction expression and of limited perturbation. The value of $A \subseteq B$ can be maintained by computing $A - B$ and testing it for emptiness when it changes, a task we assume takes unit time per change. In the actual implementation, however, only the cardinality of $A - B$, whose value is $\emptyset$ if and only if $A \subseteq B$, need be maintained.

Theorem 7: Boolean operations on Boolean valued induction variables yield induction expressions.

## 7. Applications to Iterators

We would like to extend the notion of induction variable to interesting iterators as discussed in [8]. Let us consider the set former $\{x \in A \mid \phi(x)\}$ to be specific. Now in the intermediate code we use, this iterator is actually a loop of its own, in which x runs through every element of A. Since x is not an induction variable here, the expression $\{x \in A \mid \phi(x)\}$ cannot be an induction expression for this loop except under the most trivial of circumstances. Thus, the set former cannot be an induction expression in a loop outside its own internal loop.

There is, on the other hand, a fairly broad condition under which we can prove an order of magnitude improvement in the calculation of the set former is possible.

Theorem 8: Let L be a loop containing set former $\{x \in A \mid \phi(x)\}$ at point p. Suppose A and $\phi(x)$ for all x which could ever be members of A are all induction expressions and of limited perturbation at p in L. Then the value of $\{x \in A \mid \phi(x)\}$ can be maintained with work proportional to $|A| + \text{cost}\phi(x)$, where by $\text{cost}\phi(x)$ we mean the work necessary to compute $\phi(x)$ for any x that may be in A. (Note that the straightforward evaluation of the set former requires $|A|\ \text{cost}\phi(x)$ work.)

Proof: Between any two consecutive executions of $\{x \in A \mid \phi(x)\}$, the number of x in A such that the value of $\phi(x)$ changes is bounded by $|A|$. Since each $\phi(x)$ is an induction expression and is of limited perturbation at p, the total work involved in maintaining them is bounded by $|A|$. Between any two consecutive executions of the set former, the number of new elements added to or deleted from A is also bounded because A is an induction variable and is of limited perturbation at p. The work involved in the addition and deletion is proportional to $\text{cost}\phi(x)$. Hence the total work necessary is proportional to $|A| + \text{cost}\phi(x)$.

Theorem 9: Let L be a loop containing the predicate P(A) at p where P(A) is $\forall x \in A : \phi(x)$ or $\exists x \in A : \phi(X)$. Suppose A and $\phi(x)$, for all x which could ever be members of A, are all induction expressions and are of limited perturbation at p in L. Then the value of P(A) at p can be maintained with work proportional to $|A| + \text{cost}\phi(x)$, where $\text{cost}\phi(x)$ is the work necessary to compute $\phi(x)$ for any x that may be in A. (The straightforward evaluation of P(A) requires $|A|\ \text{cost}\phi(x)$ in the worst case.)

Proof: Let $P(A) = \forall x \in A : \phi(x)$. The value of P(A) at p can be obtained by computing t, the cardinality of the set $\{x \in A \mid \text{not } \phi(X)\}$, which can be maintained with work proportional to $|A| + \text{cost}\phi(x)$, by Theorems 7 and 8. P(A) is true iff $t = \emptyset$. Again in actual implementation, only t rather the set itself is maintained.

Similarly, if $P(A) = \exists x \in A : \phi(x)$, its value is true iff the set $\{x \in A \mid \phi(x)\}$ is nonempty.

In the actual implementation, only those x in A such that $\phi(x)$ has actually changed should be updated. If the size of

this subset of A is considerably smaller than |A| and that the mapping to obtain this subsets can be precomputed outside the loop, it may be desirable to have this mapping available. In some cases, this mapping always maps to a subset of A of constant size, then the total work involved in maintaining the set former is proportional to cost$\phi$(x). We shall illustrate this with examples.

(1) Consider P(X) = $\forall y \in B : f(y) = X$.

If B and f (consider a function as a set of ordered pairs with distinct first elements) are induction variables and are of limited perturbation at p, then the value of P(X) can be maintained by maintaining the value t = cardinality of the set $\{y \in B \mid f(y) \neq X\}$ as follows:

```
for y in Δ+(B, p) do
    if f(y) ≠ X then t = t + 1
for y in Δ-(B, p) do
    if f(y) ≠ X then t = t - 1
```

A change of f, say f(y) = z is rewritten as:

```
if y is in B then
   begin
   if f(y) = x then t = t - 1
   if z = x then t = t + 1
   end
f(y) = z
```

(2) Consider $e = \{x \in A \mid x \subseteq B\}$.

If B is an induction variable and is of limited perturbation at p, then $\phi(x) = x \subseteq B$ is an induction expression and is of limited perturbation at p for all x in A.

If A is also an induction variable and is of limited perturbation at p, then e is an induction variable and is of limited perturbation at p.

Let t(x) = |x-B|

Let FIND(y) = $\{x \in A \mid y \in x\}$

$\Delta$(e, p) can be computed as follows:

```
Δ+(e, p) = ∅
Δ-(e, p) = ∅
for y in Δ-(A, p) do
   Δ-(e, p) = Δ-(e, p) U {y}

for y in Δ+(A, p) do
   begin
   compute t(y) = |y-B|
   if t(y) = 0 then
      Δ+(e, p) = Δ+(e, p) U {y}
   end

for y in Δ-(B, p) do
   begin
   for z in FIND(y) do
       begin
       if t(z) = 0 then
          Δ-(e, p) = Δ-(e, p) U {z}
       t(z) = t(z) + 1
       end
   end

for y in Δ+(B, p) do
   begin
   for z in FIND(y) do
       begin
       t(z) = t(z) - 1
       if t(z) = 0 then
          Δ+(e, p) = Δ+(e, p) U {z}
       end
   end
```

If FIND is computed outside the loop, then the work necessary to obtain $\Delta$(e, p) is proportional to c + cost$\phi$(x), where c is an upper bound on the size of FIND(y) for any y in A, and $\phi$(x) is |x-B|. In the worst case c is |A|. However, if c is a constant independent of |A|, then $\Delta$(e, p) can be computed using work proportional to cost$\phi$(x).

## Bibliography


[1] F. E. Allen, "Program Optimization," in Annual Review in Automatic Programming, Vol. 5, Pergamon, 1969, pp. 239-307.

[2] F. E. Allen and J. Cocke, "A Catalogue of Optimizing transformations," in Design and Optimization of Compilers (R. Rustin, ed.), Prentice Hall, 1972, pp. 1-30.

[3] J. Cocke and J.T. Schwartz, Programming Languages and Their Compilers, Courant Institute, New York, 1971.

[4] A. V. Ano and J. D. Ullman, The Theory of Parsing, Translation and Compiling, Vol. II, Compiling, Prentice Hall, 1973.

[5] F. E. Allen, J. Cocke and K. Kennedy, "Reduction of Operator Strength," TR 476-093-6, Dept. of Math. Sciences, Rice Univ., Houston, Aug., 1974.

[6] J. Cocke and K. Kennedy, "An Algorithm for Reduction of Operator Strength," TR 476-093-2, Dept. of Math. Sciences, Rice Univ., Houston, March, 1974.

[7] A. C. Fong, J. B. Kam and J. D. Ullman, "Application of Lattice Algebra to Loop Optimization," Proc. 2nd ACM Symp. on Principles of Programming Languages, Jan., 1975.

[8] J. Earley, "High Level Iterators and a Method of Automatically Designing Data Structure representation," ERL-M416, Computer Science Division, Univ. of Calif., Berkeley, Feb., 1974.

[9] J. T. Schwartz, "On Earley's Method of `Iterator Inversion'," SETL Newsletter, No. 138, Courant Institute, 1974.

[10] J. T. Schwartz, On Programming, Vols. I and II, Courant Institute, 1971 and 1973.