We can compare the space used by this method and the space required by ordinary hashing. Let us suppose that the perfect hashing functions $h_i$ are all computational remainder reduction phf's and that two characters are sufficient to identify an item in its segment. In System/370 we can use:

(a) a half-word vector RCH, to record in each byte the positions of the characters identifying each item in its segment;
(b) a half-word vector RD to record the values $d_i'$;
(c) a full-word vector RSH with the following structure for its words:
  (i) the first 20 bits contain the base $b_i$ of the $i$th segment;
  (ii) the next 6 bits contain $m_i'$;
  (iii) the last 6 bits contain $q_i'$;
(d) a full-word vector RM containing the divisors $M_i$.

Let us suppose that folding leaves the segment number $i$ (doubled to allow addressing to half-word boundaries) in register 4 and that registers 1 and 2 are clear. The piece of coding in Figure 3 can be used to compute $h(w)$ in register 2. It can be used later to check equality between $w$ and the $h(w)$th element in the table and to point to other information.

Fig. 3.

```
IC    1,RCH(4)          If the two characters were consecutive,
IC    3,WORD(1)         we could improve the code considerably
SLL   3,8
IC    1,RCH+1(4)
IC    3,WORD(1)         ψ_i(w) in register 3
AH    3,RD(4)           add rotation value
AR    4,4               to address full-word boundaries
L     1, RSH(4)
SLDA  2,0(1)            only last 6 bits used in shifting
D     2,RM(4)
SRL   1,6               prepare m'
SRA   2,0(1)            prepare base
AR    2,1               base is added
```

Let us suppose that the identifiers in $I$ are eight bytes long and that each segment contains an average of nine elements. If every perfect hashing function $h_i$ is minimal, each segment occupies 72 bytes, plus 12 bytes for the relevant elements in RCH, RD, RSH, and RM. In total, we have 84 bytes, and our table corresponds to an ordinary hash table with a loading factor of 72/84 = 85.7 percent.

Received October 1975; revised September 1976

**References**


1. Greniewski, M., and Turski, W. The external language KLIPA for the URAL-2 digital computer. *Comm. ACM 6*, 6 (June 1963), 322–324.
2. Knott, G.D. Hashing functions. *Computer J. 18* (Aug. 1975), 265–278.
3. Knuth, D.E. An empirical study of FORTRAN programs. *Software–Practice and Experience 1* (April 1971), 105–133.
4. Knuth, D.E. *The Art of Computer Programming, Vol. 1–3.* Addison-Wesley, Reading Mass., 1968–1973.
5. Maurer, W.D., and Lewis, T.G. Hash table methods. *Computing Surveys 7*, 1 (March 1975), 5–20.
6. Niven, I., and Zuckerman, H.S. *An Introduction to the Theory of Numbers.* Wiley, New York, 1960.
7. Severance, D.G. Identifier search mechanisms: A survey and generalized model. *Computing Surveys 6*, 3 (Sept. 1974), 175–194.




Programming Techniques

R. L. Rivest,* S. L. Graham Editors

# An Algorithm for Reduction of Operator Strength

John Cocke
IBM Thomas J. Watson Research Center

Ken Kennedy
Rice University

**A simple algorithm which uses an indexed temporary table to perform reduction of operator strength in strongly connected regions is presented. Several extensions, including linear function test replacement, are discussed. These algorithms should fit well into an integrated package of local optimization algorithms.**

**Key words and Phrases: compilers, optimization of compiled code, program analysis, operator strength reduction, test replacement, strongly connected region**

**CR Categories: 4.12, 5.24, 5.32**



* This paper was submitted prior to the time that R. L. Rivest became editor of the department, and editorial consideration was completed under the former editor, G. K. Manacher.

Authors' addresses: J. Cocke, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598; K. Kennedy, Department of Mathematical Sciences, Rice University, Houston TX 77001. Work partially supported by the National Science Foundation Division of Computer Research Grant DCR73-03365-A01.

## 1. Introduction

A somewhat misunderstood fact about compiler optimization is that its purpose is not necessarily to correct coding errors by the programmer, but rather to eliminate inefficiencies which are automatically generated by the high level language compiler. An example is the code generated by a Fortran compiler to handle two-dimensional arrays. Consider the following do loop:

```
DIMENSION   A(50, 50)
      ⋮
SUM = 0.0
DO 100        I = 1, 50
      ⋮
SUM = SUM + A(3, I)
      ⋮
100 CONTINUE
```

A naive compiler will reduce this loop to something like the following:

```
        I = 1
LOOP:   ⋮
        t1 = I * 50 + 3
        t2 = indexed load (A, t1)
        SUM = SUM + t2
              ⋮
        I = I + 1
        IF (I ≤ 50) GO TO LOOP
```

The reader should note that the multiplication "$I * 50$" is automatically generated as a part of the array accessing mechanism. The primary aim of the optimization technique known as *reduction of operator strength* is to eliminate such multiplications whenever possible [1, 2].

The basic method is to define a temporary which holds the value of the multiplication throughout the loop, allowing the multiplication to be replaced by a simple load, which is faster on most digital computers. In order to maintain the correct value in such a temporary, it must be modified whenever $I$ is changed; however, this can usually be done by a simple addition. After applying this method to the above example we get the following code:

```
        I = 1
        t3 = 50
LOOP:   ⋮
        t1 = t3 + 3
        t2 = indexed load (A, t1)
        SUM = SUM + t2
              ⋮
        I = I + 1
        t3 = t3 + 50
        IF (I ≤ 50) GO TO LOOP
```

We have replaced a multiplication with an addition to temporary $t_3$.

For the sake of simplicity, we restrict ourselves to elimination of multiplications between "loop induction variables" and "region constants." A *loop induction variable* is a variable whose value is changed within the loop only by instructions which increment an induction variable by a constant amount; i.e. instructions of the form

$I = I + 1$

or

$I = J + 3$

where $J$ is another induction variable. An obvious example of an induction variable is the do-loop variable in Fortran. A *region constant* is a variable whose value is not changed within the loop. Global program constants are also included in this class. In the scheme we propose, a temporary variable will be used to hold the value of the eliminated multiplication within the loop; this temporary must be incremented every time the induction variable in the multiplication is incremented.

Once strength reduction has been performed, some induction variables will no longer be needed except for their use within a conditional branch instruction. All code involving such an induction variable can be eliminated if the test can be replaced by the test of a generated temporary. This strategy, called *linear function test replacement,* would have the following effect on our example:

```
        t3 = 50
LOOP:   ⋮
        t1 = t3 + 3
        t2 = indexed load (A, t1)
        SUM = SUM + t2
              ⋮
        t3 = t3 + 50
        IF (t3 ≤ 2500) GO TO LOOP
```

In the current paper we develop an algorithm which does a rather complete job of strength reduction in strongly connected regions of the program control-flow graph. A test replacement extension is also included.

## 2. Intermediate Code

The output of the syntactic analysis phase of a compiler is the program expressed in some intermediate text. We here describe an idealized intermediate code of "quadruples" which will be used in the subsequent discussion of algorithms. This code is simple yet flexible enough to express most of the ideas behind strength reduction.

A program will be represented by a linked list of tuples of the form

⟨*op*, *targ*, *a*1, *a*2, *next*⟩

where *op* is the operation code, *targ* is the target variable for the result, *a*1 is the first argument, *a*2 is the second argument, and *next* is a link to the next instruction. The operations available are:

| | |
|---|---|
| *nop* – no operation | *sto* – store |
| *add* – addition | *neg* – store negative |
| *sub* – subtraction | *xst* – indexed store |
| *mul* – multiplication | *br* – branch unconditionally |
| *div* – division | *brc* – branch conditionally |
| *exp* – exponentiation | *hlt* – halt |
| *xld* – indexed load | |

The "indexed load" (*xld*) operation takes an array name *a* and a simple variable *i* as its arguments and loads the value of *a(i)* into its target. "Store" (*sto*) moves the value of its first argument to the target (its second argument is ignored). "Store negative" (*neg*) negates the value of its first argument and moves the result to the target. "Indexed store" (*xst*) takes an array name *a* as its target and an index *i* and a simple variable *x* as its arguments; its effect is to move the value of *x* to *a(i)*. The target of a branch instruction is a pointer to the instruction which will be executed next if the branch is "taken." "Branch conditionally" (*brc*) takes the branch if $a1 \leq a2$ and falls through otherwise.

Note that all instructions assume integer arithmetic and that function and subroutine calls have been left out. These simplifications are made only to aid the narrative and do not substantially affect the generality of the methods.

We will often find it necessary to insert new instructions in the code list, so we assume the existence of a routine to perform this task. The methods of linked-list insertion are well known and will not be treated here.

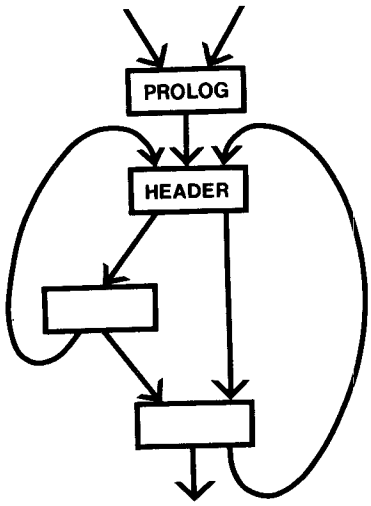
## 3. Flow Analysis

For the purposes of global program analysis, it is useful to break up the program into *basic blocks* of straight-line code in which a transfer may occur only as the last instruction. After such a block is executed, control may pass to one of a number of other blocks called *successors* of the block just executed. This leads to the representation of a program as a directed graph in which nodes stand for blocks and edges represent the successor relationship.

A *program flow graph* is a triple $(N, E, n_0)$, where $N$ is the set of basic blocks, $E$ is the set of edges (ordered pairs of blocks), and $n_0$ is the unique program entry node. Many authors [2-8] have studied the analysis of program flow graphs. Our main concern in this work will be generalized looping structures called "strongly connected regions." A *strongly connected region* $R$ is a set of basic blocks such that if $n_0$ and $n_m$ are any two blocks in the region, there exists a control flow path within the region from $n_0$ to $n_m$. That is, there exists a sequence $(n_0, n_1, \ldots, n_m)$ of blocks in $R$ such that $(n_i, n_{i+1}) \in$ for all $i$, $0 \leq i < m$. A strongly connected region is said to be *single-entry* if all edges leading into the region terminate at the same node, the *header*.

There are a number of ways to locate strongly connected regions [2, 3, 7, 8], and we will not be concerned with those here. Instead, we assume that by some method a single-entry, strongly connected region has been selected for treatment. With this region we associate a *prolog*—a basic block which is always executed prior to entry to the region. A prolog roughly corresponds to an initialization block. (See Figure 1.)

Fig. 1. Single-entry strongly connected region.




## 4. Finding Induction Variables

One of the first subtasks of the strength reduction process is to determine which variables in a region are induction variables. We assume that a previous optimization pass such as code motion has determined the set *RC* of region constants (variables whose values are not changed in the region). Induction variables are assigned values by operations of the following form:

$x \leftarrow \pm y,$
$x \leftarrow y \pm z,$

where $y$ and $z$ are either region constants or other induction variables. Let *IV* be the set of induction variables. It is somewhat simpler to define what is *not* an induction variable than it is to define what is, so we make the following observations.

(1) If $x \leftarrow op(y, z)$ and *op* is not one of the operations {*sto*, *neg*, *add*, *sub*}, then $x$ is not an induction variable.
(2) If $x \leftarrow op(y, z)$ and $y$ and $z$ are not both elements of $IV \cup RC$, then $x$ is not an induction variable.

The algorithm we present is due to Cocke and Schwartz [2] and uses a process of elimination based on these observations. Initially, *IV* is the set of targets of the operations specified in observation 1. We iterate through the code, eliminating variables from *IV* according to observation 2 until no more eliminations can be made. The remaining set contains only induction variables.

**Algorithm I.** Find Induction Variables

*Input.* (1) *SCR*, the set of instructions in the strongly connected region.
(2) *RC*, the set of region constants.

*Output.* *IV*, the set of induction variables.

*Method*

I1. Initially, let $IV = \emptyset$.
I2. Execute step I3 once for each instruction $\langle op, targ, a1, a2, next\rangle$ in *SCR*.
I3. If $op \in \{sto, neg, add, sub\}$, add *targ* to *IV*.
I4. Do step I5 for each instruction $\langle op, targ, a1, a2, next\rangle$ such that $targ \in IV$. If *IV* changes, repeat step I4; otherwise, halt.
I5. If $a1 \notin IV \cup RC$ or if $a2 \notin IV \cup RC$, remove *targ* from *IV*.

Correctness of this algorithm is clear from earlier comments. Termination follows from the fact that I4 can be executed only as many times as there are elements in the initial *IV*, computed in step I2.

## 5. Finding Candidates for Reduction

We intend to reduce all multiplications of the form $i * c$ where $i$ is an induction variable and $c$ is a region constant. The next routine merely passes through the strongly connected region looking for such multiplications.

**Algorithm F.** Find Reduction Candidates

*Input.* (1) The sets *IV* and *RC*.
(2) The set *SCR* of instructions in the strongly connected region.

*Output.* The set *CANDS* of instructions which are candidates for reduction.

*Method*

F1. Initially, let $CANDS = \emptyset$.
F2. Do step F3 once for each instruction $p = \langle op, targ, a1, a2, next\rangle$ such that $op = mul$.
F3. If $a1 \in IV$ and $a2 \in RC$, or if $a2 \in IV$ and $a1 \in RC$ then add $p$ to *CANDS*.

Termination and correctness of this algorithm are obvious.

## 6. The Temporary Table

The idea of reduction in strength is to replace $x \leftarrow i * c$ by $x \leftarrow t$ where $t$ is a temporary which holds the current value of $i * c$ over the entire region. In the package of algorithms presented in this paper, these temporaries may be accessed through a hash table which uses the names of the multiplication operands as keys. Thus $t_{i*c}$ will contain the value of $i * c$ in the region. We must do two things to assure that $t_{i*c}$ always contains the correct value.

(1) An initialization of the form $t_{i*c} \leftarrow i * c$ must be inserted at the end of the prolog.
(2) After each assignment to $i$, an instruction to modify the value of $t_{i*c}$ must be inserted.

This second step is more complicated than it first seems since instructions of the forms found in Table I must be handled. Table I shows that we must not only create temporaries for $i * c$ but also $j * c$ and $k * c$ for every induction variable $j$ and region constant $k$ that can "affect" the value of $i$ through a sequence of increment instructions. In addition, we must insert initializations and modifications for these temporaries. To do this, we associate with each induction variable $i$ the set $AFCT(i)$ of all induction variables and region constants which can affect the value of $i$ in the sense described above. The method for computing these sets is a simple transitive closure.

**Algorithm A.** Compute "Affect" Sets

*Input.* (1) The set *SCR* of instructions in the strongly connected region.
(2) The sets *IV* and *RC*.

*Output.* The set $AFCT(i)$ for each $i \in IV$.

*Method*

A1. Initially $AFCT(i) = \{i\}$ for all $i \in IV$.
A2. Do step A3 for each instruction $\langle op, targ, a1, a2, next\rangle$ such that $targ \in IV$.
A3. Add $a1$ and $a2$ to $AFCT(targ)$.
A4. Do step A5 for each $i \in IV$. If any of the $AFCT$ sets change, repeat step A4; otherwise, halt.
A5. Let $AFCT(i) = AFCT(i) \cup \bigcup_{j \in AFCT(i) \cap IV} AFCT(j)$.

Termination of this algorithm follows from the finiteness of *IV*. Correctness is obvious from the properties of the transitive closure.

Once we have these sets, we can perform strength reduction very neatly using the temporary table. The algorithm below does this. The reader should note the use of the list $C(x)$ of constants $c$ for which $t_{x*c}$ must be maintained.

**Algorithm R.** Reduction of Operator Strength.

*Input.* (1) The sets *SCR* and *PROLOG*.
(2) The set *CANDS* of reduction candidates.
(3) The sets *IV* and *RC*.
(4) The temporary table $T$, initially empty ($T(x, c)$ = the name of $t_{x*c}$).
(5) The sets $AFCT(i)$, $i \in IV$.

*Auxiliary Quantity.* For each $x \in IV \cup RC$, a list $C(x)$ of constants $c$ for which $t_{x*c}$ must be maintained.

*Output.* The modified code for *SCR* after strength reduction.

*Method*

R1. Initially let $C(x) = \emptyset$, $\forall x \in IV \cup RC$.
R2. Do step R3 once for each instruction $p \in CANDS$. Let $x$ be the induction variable and $c$ be the region constant in $p$.
R3. For each $y \in AFCT(x)$, add $c$ to $C(y)$.
R4. Do step R5 for each $x \in IV \cup RC$ such that $C(x) \neq \emptyset$.
R5. For each $c \in C(x)$ let $T(x, c)$ = new temporary name. Insert the initialization instruction $\langle mul, T(x, c), x, c, -\rangle$ at the end of *PROLOG*.
R6. Do step R7 for each instruction $p = \langle op, targ, a1, a2, next\rangle$ such that $targ \in IV$ and $C(targ) \neq \emptyset$.
R7. For each $c \in C(targ)$ insert the instruction $\langle op, T(targ, c), T(a1, c), T(a2, c), -\rangle$ after $p$. Note that the inserted instruction has the same opcode as $p$.
R8. Do step R9 for each instruction $p = \langle mul, targ, x, c, next\rangle \in CANDS$. Let $x$ be the induction variable and $c$ be the region constant.
R9. Replace $p$ by the instruction $\langle sto, targ, T(x, c), -, next\rangle$ where *targ* is the target variable for the original instruction.
R10. Halt.

Table I. Modification of tempories ($j$ = induction variable, $k$ = region constant).

| Instruction | Operation to be inserted |
|---|---|
| $i \leftarrow k$ | $t_{i*c} \leftarrow t_{k*c}$ |
| $i \leftarrow -k$ | $t_{i*c} \leftarrow -t_{k*c}$ |
| $i \leftarrow j + k$ | $t_{i*c} \leftarrow t_{j*c} + t_{k*c}$ |
| $i \leftarrow j - k$ | $t_{i*c} \leftarrow t_{j*c} - t_{k*c}$ |

Table II. Reduction in Strength Example.

| Original code | Code after reduction |
|---|---|
| prolog: $i = 1$; $j = 1$ | prolog: $i = 1$; $j = 1$; $t_{i*5} = i * 5$; $t_{j*5} = j * 5$; $t_{j*6} = j * 6$; $t_{i*6} = i * 6$; $t_{1*5} = 5$; $t_{1*6} = 6$; $t_{3*5} = 15$; $t_{3*6} = 18$ |
| region: ⋮; $i = j + 1$; ⋮; $x = j * 5$; ⋮; $j = i + 3$; ⋮; $y = i * 6$; ⋮; $j = j + 1$ | region: ⋮; $i = j + 1$; $t_{i*5} = t_{j*5} + t_{1*5}$; $t_{i*6} = t_{j*6} + t_{1*6}$; ⋮; $x = t_{j*5}$; ⋮; $j = i + 3$; $t_{j*5} = t_{i*5} + t_{3*5}$; $t_{j*6} = t_{i*6} + t_{3*6}$; ⋮; $y = t_{i*6}$; ⋮; $j = j + 1$; $t_{j*5} = t_{j*5} + t_{1*5}$; $t_{j*6} = t_{j*6} + t_{1*6}$ |

Termination of this algorithm is clear. Correctness follows from a simple argument that $T(x, c)$ contains the value of the multiplication everywhere in the region. Certainly $T(x, c)$ contains the correct value on entry to the region because of the initialization inserted in step R5. The value of $x * c$ can change only when the value of $x$ changes—but after each modification of $x$, an appropriate modification of $T(x, c)$ has been inserted.

Table III. Example from Table II after clean-up.

| | |
|---|---|
| Strongly connected region | ⋮; $t_{i*5} = t_{j*5} + t_{1*5}$; $t_{i*6} = t_{j*6} + t_{1*6}$; ⋮; $t_{j*5} = t_{i*5} + t_{3*5}$; $t_{j*6} = t_{i*6} + t_{3*6}$; ⋮; $t_{j*5} = t_{j*5} + t_{1*5}$; $t_{j*6} = t_{j*6} + t_{1*6}$ |

The effect of this strength reduction algorithm is demonstrated by the example in Table II. This example points up an obvious limitation of the algorithm as it now stands: a systematic clean-up of the code is needed if the proliferation of variables is to be reduced.

There are two aspects to such a clean-up. First a good variable-subsumption algorithm [9] should be applied to eliminate the need for stores of the form $x = t_{j*5}$. The idea is to replace uses of $x$ by uses of $t_{j*5}$, whenever possible. If all such uses can be replaced, then the instructions which define $x$ are useless. These useless instructions should then be removed by a global dead-computation elimination algorithm [10] which marks all instructions whose results are actually used and deletes the rest. The effect of these two techniques on the example in Table II can be dramatic. Suppose the subsumption is successful in removing all references to $x$ and $y$ and suppose also that the only uses of variables $i$ and $j$ are in the computation of $x$ and $y$. Then the code clean-up would remove five instructions from the strongly connected region, leaving the region in Table III. In the next section, we discuss a technique which can make the clean-up even more effective by removing induction variables from tests—thus allowing the elimination of many "increment" instructions.

## 7. Linear Function Test Replacement

Consider the following loop:

```
        i = 1
START:  x = i * c
          ⋮
        i = i + 2
        IF i ≤ 100 GO TO START
```

After strength reduction, this becomes:

```
        i = 1
        t_{i*c} = i * c
        t_{2*c} = 2 * c
START:  x = t_{i*c}
          ⋮
        i = i + 2
        t_{i*c} = t_{i*c} + t_{2*c}
        IF i ≤ 100 GO TO START
```

Now if $i$ is dead on exit from the loop, as is the case with many induction variables, we can eliminate the

instruction which increments $i$ if we can eliminate the test of $i$. We do this by testing $t_{i*c}$ instead, yielding the following loop:

```
        i = 1
        t_{i*c} = i * c
        t_{2*c} = 2 * c
        t_{100*c} = 100 * c
START:  x = t_{i*c}
          :
        t_{i*c} = t_{i*c} + t_{2*c}
        IF t_{i*c} ≤ t_{100*c} GO TO START
```

In so doing, we have eliminated one instruction from the loop. This optimization is known as *linear function test replacement.*

We will now formalize the process of test-replacement, leaving the actual elimination of code to the dead-computation elimination algorithm, which will not be treated here. Suppose there is a test of the following form:

*IF* $i \leq k$ *GO TO LABEL*

Four things must be done if we are to eliminate the use of $i$.

(1) A constant $c$ such that $t_{i*c}$ is in the temporary table must be found.
(2) $t_{k*c}$ must be inserted in the temporary table.
(3) $t_{k*c}$ must be initialized in the region prolog.
(4) The test must be replaced by

*IF* $t_{i*c} \leq t_{k*c}$ *GO TO LABEL*

Note that steps 2 and 3 need not be done if $t_{k*c}$ is already in the temporary table.

From the above observations, it is clear that test replacement can be easily incorporated into the strength reduction process. The next algorithm, which can be called after strength reduction, performs the required tasks. Of course, it would be more efficient to integrate this process into the strength reduction procedure.

**Algorithm T.** Linear Function Test Replacement

*Input.* (1) The sets *SCR* and *PROLOG*.
(2) The sets *IV* and *RC*.
(3) The sets $C(x)$ for each $x \in IV$. Recall that these sets, computed by algorithm *R*, contain all constants $c$ such that $t_{x*c}$ is in the temporary table.

*Output.* The modified strongly connected region after test replacement.

*Method*

T1. Do step T2 for each conditional branch instruction $b = \langle brc, targ, x, k, next\rangle$ such that one argument is an induction variable and the other is a region constant. Let $x$ be the induction variable and $k$ the constant.

T2. Let $c$ be any element of $C(x)$. (If $C(x) = \varnothing$, this test cannot be replaced.)

T2a. If there is no temporary table entry for either $T(c, k)$ or $T(k, c)$, create one and insert the instruction

$\langle mul, T(k, c), k, c, -\rangle$

at the end of *PROLOG*; if there is such an entry use it in step T2b.

T2b. Replace the conditional branch instruction with

$\langle brc, targ, T(x, c), T(k, c), -\rangle$

where *targ* is the original target.

T3. Halt. □

Termination and correctness of this algorithm are obvious.

## 8. Extensions

Two simple extensions of the methods presented here are possible. First, all generated temporaries $t_{x*c}$ are themselves either region constants (if $x \in RC$) or induction variables (if $x \in IV$). After the initial strength reduction pass, *IV* and *RC* can be recomputed and strength reduction repeated, yielding further reductions. It is not yet clear how valuable such additional passes will be.

A second extension would generate temporaries to hold the values of additions within the region. This would pave the way for more strength reductions and, in some cases, allow the elimination of code. For example, one instruction could be eliminated from the sample loop in Section 1 if a constant were created to hold the value of $t_3 + 3$. Incorporation of this extension is straightforward.

## 9. Summary and Conclusions

We have presented a method for reduction of operator strength that can be conveniently implemented by means of a table of hashed temporaries. This algorithm can be extended to include linear function test replacement and further strength reductions. To be most effective, these algorithms should be accompanied by subsequent code "clean-up" algorithms to eliminate useless instructions.

The authors envision these algorithms as part of a general package of algorithms which can be applied to nested, strongly connected regions in an inner to outer sequence. Local algorithms for redundant subexpression elimination, code motion, variable subsumption, constant folding, and dead-code elimination can be generalized to global algorithms through this technique. For example, by applying code motion to a larger strongly connected region after strength reduction is applied to a subregion, many of the generated initializations may be moved out of the subregion prolog to less frequently executed prologs. This approach could lead to an integrated compiler optimization module.

Before concluding, we should point out a few of the pitfalls of these optimization techniques. We have made several important assumptions in developing these methods. First, we assume that the variables under consideration have no hidden value changes, i.e. all possible changes in their values are explicit at com-

pile time. Second, we assume no problems of finite precision. Both of these assumptions could lead to difficulties in an actual implementation. Whenever the effective position of a computation is moved, as it is in these strength reduction methods, proper attention should be paid to such "safety" considerations, lest an unexpected error interrupt occur. It does no good to produce a faster version of the program if the faster version is incorrect. The interested reader can find an expanded treatment of safety in [11].

A final assumption made in using such a strength reduction technique is that it is profitable. The profitability assumption is based on the observations that code within a loop is executed frequently and multiplications are significantly more expensive than additions on most machines. These assumptions are not valid in all cases, however. For example, if we were to replace a multiplication on a little-used branch in the loop by additions on more frequent paths, we might significantly deoptimize the compiled code. This situation can be avoided by a more careful analysis of profitability such as the one described in [12].

*Acknowledgments.* The authors wish to thank Frances Allen of IBM and Jack Schwartz of New York University for their many helpful suggestions. The referee provided several valuable suggestions for improvement of the final manuscript. We are particularly grateful for his comments on the question of safety, some of which we adapted in the summary and conclusions.

Received October 1974, revised August 1976

**References**

1. Allen, F.E. Program Optimization. *Annual Review of Automatic Programming, Vol. 5,* Pergamon Press, New York, 1969.
2. Cocke, J., and Schwartz, J. *Programming Languages and Their Compilers.* Courant Institute of Mathematical Sciences, New York U., New York, 1970.
3. Kennedy, K. A global flow analysis algorithm. *Int. J. Comptr. Math., Sect. A, 3* (Dec. 1971), 5-15.
4. Hecht, M.S., and Ullman, J.D. Flow graph reducibility. *SIAM J. Compting. 1*, 2 (June 1972), 188-202.
5. Kildall, G.A. A unified approach to global program optimization. Conf. Rec. ACM Conf. on Principles of Programming Languages, Boston, Mass., Oct. 1973, pp. 194-206.
6. Ullman, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica 2* (1973), 191-213.
7. Earnest, C., Balke, K.G., and Anderson, J. Analysis of graphs by ordering of nodes. *JACM 19*, 1 (Jan 1972), 23-42.
8. Allen, F.E. Control flow analysis, SIGPLAN Notices (ACM) *5*, 7 (July 1970), 1-19.
9. Kennedy, K. Variable subsumption with constant folding. SETL
York, Feb. 1974.
10. Kennedy, K. Use-definition chains with applications. Tech. Rep. 476-093-9, Dept. Math. Sci., Rice U., Houston, Tex., April 1975.
11. Kennedy, K. Safety of code motion. *Int. J. Comptr. Math., Sect. A,* 3 (1972), 117-130.
12. Cocke, J., and Kennedy, K. Profitability computations on program flow graphs. Tech. Rep. 476-093-3, Dept. Math. Sci., Rice U., Houston, Tex., May 1974.



Programming Techniques

R.L. Rivest,* S.L. Graham Editors

# Improving Programs by the Introduction of Recursion

R.S. Bird
University of Reading

**A new technique of program transformation, called "recursion introduction," is described and applied to two algorithms which solve pattern matching problems. By using recursion introduction, algorithms which manipulate a stack are first translated into recursive algorithms in which no stack operations occur. These algorithms are then subjected to a second transformation, a method of recursion elimination called "tabulation," to produce programs with a very efficient running time. In particular, it is shown how the fast linear pattern matching algorithm of Knuth, Morris, and Pratt can be derived in a few steps from a simple nonlinear stack algorithm.**

**Key Words and Phrases: program transformation, optimization of programs, recursion elimination, pattern matching algorithms, stacks, computational induction**

**CR Categories: 4.0, 4.2, 5.20, 5.24, 5.25**



* This paper was submitted prior to the time that Rivest became editor of the department, and editorial consideration was completed under the former editor, G. K. Manacher.

Author's address: Department of Computer Science, University of Reading, Whiteknights Park, Reading, Berkshire, England.