Compiler Analysis of the Value Ranges for Variables

WILLIAM H. HARRISON, MEMBER, IEEE

Abstract—Programs can be analyzed to determine bounds on the ranges of values assumed by variables at various points in the program. This *range information* can then be used to eliminate redundant tests, verify correct operation, choose data representations, select code to be generated, and provide diagnostic information. Sophisticated analyses involving the proofs of complex assertions are sometimes required to derive accurate range information for the purpose of proving programs correct. The performance of such algorithms may be unacceptable for the routine analysis required during the compilation process. This paper presents a discussion of mechanical range analysis employing techniques practical for use in a compiler. This analysis can also serve as a useful adjunct to the more sophisticated techniques required for program proving.

Index Terms—Constant propagation, optimizing compiler, program analysis, proof of correctness, weak interpretation.

Introduction

PROGRAMS can be analyzed to determine bounds on the ranges of values assumed by variables at various points in the program. This *range information* can then be used to eliminate redundant tests, verify correct operation, choose data representations, select code to be generated, and provide diagnostic information. Sophisticated analyses involving the proofs of complex assertions are sometimes required to derive accurate range information for the purpose of proving programs correct. The performance of such algorithms may be unacceptable for the routine analysis required during the compilation process. This paper presents a discussion of mechanical range analysis employing techniques practical for use in a compiler. This analysis can also serve as a useful adjunct to the more sophisticated techniques required for program proving.

The primary characteristic of the approach presented here is its decomposition of the problem into two mechanisms called *range propagation* and *range analysis*. Range propagation is a rather simple algorithm which uses the data and the conditional structure of a program to derive and propagate refinements in the accuracy of range information. A refinement at the point of assignment to a variable is carried to the points where it is used. This is done in the hopes of improving the accuracy of the information at those points as well. However, since the range propagation process is not inductive in nature, the presence of loops in the data flow graph severely limits the derivable results. Range analysis is an algorithm which tracks the changes applied to a variable at each point in a loop of the program, but does so in total ignorance of the conditional structure of the loop. This information is then used as a base for induction to derive a range of values for the variable. The fact that tests are ignored in this computation also leads to limited accuracy. However, the ranges produced by range analysis are *intersected* with those produced by range propagation, and the range propagation process continues with these considerably more accurate results. In the most common cases, the range information produced for loop variables is completely accurate. The accuracy of range information for other variables in a loop is refined by another technique which brings together the inductive and conditional structures of a program. This technique, called *loop counting*, is employed to derive bounds on the number of times the inductive processes of a loop need to be applied.

The resulting range information is then used to elide unnecessary tests and to produce diagnostic information. The elision of tests may alter the apparent data flow of a program, and as a result, further improvements in the range information may arise from this optimization. Diagnostic information may indicate that an error condition is possible, and may elaborate this warning with a description of the way in which the error may arise.

Assumptions

It is assumed that the program flow graph is reducible [1]. It is also assumed that the data flow for the program has been computed [2]. At each program point, therefore, we can postulate the existence of the *use* and *def* functions. The *use* function $U(p,v)$ yields the set of all program points which may use for the value of the variable v its value at program point p. For convenience, if t is the target variable at program point p, we may write $U(p)$ to mean $U(p,t)$. The *def* function $D(p,v)$ yields the set of all program points which supply definitions for the variable v at program point p. If, instead of being a variable, v is a constant, we define $D(p,v) = \{\ \}$.

Notation

The following notation is used in the manipulation of ranges:

$[l:u]$	range with lower bound l and upper bound u;
$\lfloor r$	lower bound of the range r;
$\lceil r$	upper bound of the range r;
$\bot r$	least number representable within the data type of r which is greater than $\lfloor r$; and
$\top r$	greatest number representable within the data type of r which is less than $\lceil r$.

Range Propagation

Introducing Test Points

For the purpose of range propagation, the test points in a program provide vital information. Each test point behaves,

Manuscript received August 28, 1975; revised October 4, 1976.
The author is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

p	U(p)	D(p,I)	$U_t(p)$	$D_t(p,I)$
(1)	{2,3}	{}	{2}	{}
(2)		{1,3}		{1,3}
(3)	{2,3}	{1,3}	{2}	{2If}
(2It)			{}	
(2If)			{3}	

Fig. 1. Data flow functions.

in fact, like a use of the variables involved in the test, and also like a group of definitions which supply extra information about the variables on each outbranch from the test point. The information on the outbranches are complementary, but it is necessary to know exactly which outbranches can supply information to other program points. This computation can be done exactly as in [2], except that a test point is decomposed into the use point and the several def points just mentioned. This gives rise to another pair of functions, the *use-test* and the *def-test* functions, represented as U_t and D_t, respectively.

Let us compare the definitions of U_t and D_t with those of U and D by the following example:

```
(1) I = 1
(2) IF I>10 THEN EXIT
(3) I = I + 1
    loop to (2)
```

Since program point (2) is a test point, we will have occasional need to refer more explicitly to the def points which supply information on its outbranches. In the remainder of this paper, the use point in the test will be denoted simply by the number of the program point, in this case—2. When necessary, this number will be suffixed by the variable whose range information is refined by its definition in the test and by a *t* or *f* to indicate the *true* and *false* outbranches from the test. In this case, the only relevant variable is I, and so we refer to program points $2It$ and $2If$. The values of the functions of interest are displayed in Fig. 1.

Basic Range Propagation Algorithm

Information about the ranges of variables may be propagated through a program in the same way that constants may be propagated. It is only necessary to extend the compiler to apply operators to ranges of values for its operands to derive a range as the result. Thus, the compiler must be able to perform computations like "$[-\infty:13]+[-6:11]=[-\infty:24]$." Furthermore, a range is associated with every definition point in the program. The range associated with a program point p will be denoted by $\rho(p)$. The range propagation algorithm makes use of a set of program points yet to be processed, called Ψ. All ranges are initialized to $[-\infty:\infty]$, and Ψ is initialized to contain all program points whose computation involves a known range. Known ranges arise from the use of constants or from parameter information derived externally.

Ranges are propagated by iteratively processing program points chosen from Ψ. Let the chosen point be p. Program

node processed	ranges established				new Ψ
	$\rho(1)$	$\rho(2It)$	$\rho(2If)$	$\rho(3)$	
	$[-\infty:\infty]$	$[-\infty:\infty]$	$[-\infty:\infty]$	$[-\infty:\infty]$	{1,2,3}
1	[1:1]				{2,3}
2		$[11:\infty]$	$[-\infty:10]$		{3}
3				$[-\infty:11]$	{2}
2		$[11:\infty]$	$[-\infty:10]$		{}

Fig. 2. Range propagation example.

point p may correspond to a def point or to a test point, but in either case, a number of source variables s_1 through s_n must be evaluated. If the source variable s_i is actually a constant, the corresponding range π_i is just $[s_i:s_i]$. If the source variable s_i is not a constant, the corresponding range π_i is computed by taking the union of the ranges of its corresponding def points. That is,

$$\pi_i = \bigcup_{j \in D_t(p,s_i)} \rho(j).$$

If p is a def point, it applies some operator to $s_1, \cdots, s_n$ to compute a result. To propagate the range the compiler computes a new result range by applying the operator to $\pi_1, \cdots, \pi_n$. If the result is a narrower range than $\rho(p)$, it is assigned as the new value of $\rho(p)$ and Ψ is augmented by adding the elements of $U_t(p)$.

If p is a test point, it applies some test to the $s_1, \cdots, s_n$ and branches depending on whether the test is true or false. Corresponding to each test, there exists a set of rules by which the *true*-branch and *false*-branch ranges of the variables may be derived from their incoming values. For example, if the test is $s_1 > s_2$, the rules define the *true*-branch range for s_1 to be $\pi_1 \cap [\perp\pi_2:\infty]$. The *false*-branch range for s_1 is $\pi_1 \cap [-\infty:\lceil\pi_2]$. These rules are elaborated in the following section. If the resulting range for some variable on some outbranch is narrower than its previous value, the program points yielded by U_t applied to the corresponding *def* point are added to Ψ.

Whether it denotes a test or def point, p is then removed from Ψ. Fig. 2 illustrates the range propagation process as applied to the above example.

It is interesting to observe what the result would be if subscript range checks to verify that I is in the range $[1:10]$ are inserted between program points (2) and (3). Instead of merely concluding that $\rho(2If)=[-\infty:10]$, the more accurate result $[1:10]$ would be obtained, and both subscript range tests would be seen to be unnecessary. This surprising effect

data type	test	new x range	new y range
general	$x=y$	$x \cap y$	$y \cap x$
	$x \neq y$	$x \cap ([-\infty:\top y] \cup [\bot y:\infty])$	$y \cap ([-\infty:\top x] \cup [\bot x:\infty])$
arithmetic	$x<y$	$x \cap [-\infty:\top y]$	$y \cap [\bot x:\infty]$
	$x \leq y$	$x \cap [-\infty:\lceil y]$	$y \cap [\lfloor x:\infty]$
	$x \geq y$	$x \cap [\lfloor y:\infty]$	$y \cap [-\infty:\lceil x]$
	$x>y$	$x \cap [\bot y:\infty]$	$y \cap [-\infty:\top x]$
boolean	x	*true*	

Fig. 3. Result ranges on the *true* exits from a relational test.

t5	*t4,t1*	*t3,t2*
true	*true,true*	*true,**true***
		true,false
		***false**,true*
false	***true,false***	
	false,true	
	false,false	

Fig. 4. Forcing tree derived during test pushing.

results from the fact that the tests effectively introduce just the right hypotheses on which an inductive construction of the range can be based. Thus, no elaborate mechanism is needed to deal with the most common case of looping through an array to process each element. Unfortunately, if the loop contains some paths on which the subscript range checking hypotheses are not fortuitously introduced, range propagation is unable to arrive at the correct result, and the complementary mechanisms described in the section on *range analysis* are needed.

Restricting Ranges at Test Points

At any test point, information may be derived about the values of the variables being tested. This information is implied by the outbranch which is selected. In the simple program model discussed, only simple relational tests actually occur in the Boolean-expression part of the test. Depending on the data type of the operands, different tests may be permitted. The resulting implications on the ranges of the source variables along the *true* branch from the test can be summarized in Fig. 3. The results for the *false* branch can be derived by negating the relational operator.

Test Pushing

It is often the case that source programs use various Boolean combinations of relations as the basis for a conditional branch. Rather than simply deriving the range of values of the simple Boolean variable which directly appears in the test, information can often be derived about the ranges of variables mentioned in the relations which formed the Boolean value being tested. This occurs most often in languages like PL/I, where Boolean expressions are defined to be completely evaluated before a result can be produced. For example, consider the PL/I source language statement:

```
IF A>B & (C>D | E>F) THEN GO TO L;
```

This statement would be decomposed into the following sequence of six program points.

```
(1) t1 = A > B
(2) t2 = C > D
(3) t3 = E > F
(4) t4 = t2 | t3
(5) t5 = t1 & t4
(6) IF t5 THEN GO TO L;
```

To evaluate the ranges available on the *true* and *false* outbranches from program point (6), we examine its predecessor nodes. For each examined node, if its operator is a Boolean operation, and if each of its operands has only one *def*, we can determine what Boolean values the operands must have to compute the Boolean value being assumed for the target. If, on the other hand, the operator is a relational, we can use Fig. 3 to derive the constraints. The constraints employed on the exit arcs from the test are the ones derived from relationals whose truth value is completely determined by the truth value of the final test. If at any time, while pursuing the consequence of a true/false decision at the branch point, some intervening variable may be either *true* or *false*, no further conclusions about the implications of the truth or falsehood of that particular variable need be pursued. Thus, the preceding example would give rise to the forcing tree described by Fig. 4.

An examination of this table shows that on the *true* outbranch from the test, variables *t5*, *t4*, and *t1* must all be *true*. Since *t1* was derived from the relation $A>B$, Fig. 3 may be used to derive restricted ranges for A and B. The ranges for C, D, E, and F cannot be restricted, since the *true* outbranch may occur whether the relations among them are true or false. On the *false* outbranch from this test, no restrictions on the ranges of A, B, C, D, E, or F can be inferred.

Symbolic Range Propagation

It is sometimes the case that programs employ variables to parameterize various constants of program execution. For example, the size of an array parameter may be unknown at compile time, or a parameter may denote the number of elements to be processed in some array. Knowledge of the implicit or explicit relationships among these variables is occasionally crucial for successful optimization of programs.

Consider the following PL/I-like procedure:

```
    X:PROCEDURE(A,N);
    DECLARE A(*) FIXED BINARY, N FIXED BINARY;
(1) I = LBOUND(A);
(2) A(I) = 0;
(3) I = I + 1;
(4) IF I<= N THEN GO TO (2);
    END;
```

Successful removal of that implicit subscript check at program point (2) which tests if I≥HBOUND(A) requires the knowledge that the parameter constraints for this procedure are as follows:

1) N is less than or equal to HBOUND(A), and
2) HBOUND(A) is greater than or equal to LBOUND(A), i.e., A is not empty.

This information may be expressed either in some form of interface specification, or as tests actually present in the code. For convenience of presentation, the following discussion assumes all such constraints to be expressed explicitly in the code.

Symbolic range information is propagated analogously to constant range information. The value produced at a definition point may be constrained to lie in some constant range and also to bear some relation to the values established at other definition points. For practical purposes, the form of this relationship should probably be restricted to some subset of

<relational-operator><definition-point>±<constant>.

We will hypothesize that a range of values may contain many such restrictions in conjunctive form, although this issue will be discussed more fully in the section on *Representation of Ranges*.

In addition to propagating symbolic range information, it is necessary to derive information which relates the symbolic values to one another whenever possible. This need arises in cases like that produced by the following code sequence:

```
(1) if A>B then exit;
(2) if B<=C then exit;
(3) if A>C then . . .
```

In this example, the range information for the variable A on entry to program point (3) will indicate that $A>B$, but will not carry any information relating B to C. This information must be supplied by another mechanism. For each interval in the program, there exists a set of variables whose value is not changed within that interval. These variables are often called *interval constants*. For each interval constant v, the chain of predecessors expressed by repeated application of the D_t function can be examined to derive relations between v and other region constants. The simplest search for this information is to apply the function $\lambda p.D_t(p,v)$ iteratively as long as the program points arrived at are test points. As these program points are examined, the program points which relate v to other interval constants can be examined to establish the relationship information needed to process the interval. If the same relationship can be established by such an examination of the D_t predecessors of both of two interval constants, then the relationship holds within the interval. This accumulation of region constant information can be made first for outer intervals, and later for inner intervals. Because of this order of processing, searches relating to an interval need be carried out only as far as the interval head of its containing interval. The relationship information for the containing interval can be merged indirectly since the constants for any interval are also constants for its contained intervals.

Range Analysis

Introducing Cutpoints

Range analysis complements range propagation by deriving inductive definitions of the value ranges of variables. Consequently, range analysis concerns itself primarily with the analysis of loops. Since the control flow graph is reducible, each loop must have a single node which is a cutpoint of the control flow graph. This node, the interval header, can therefore be used to cut the data flow graph as well. We can therefore postulate a new *def* function which reflects this cutting. Unlike the functions U_t and D_t, this function D_l can be computed from information gathered during the derivation of D. As described in [2], D is computed with an algorithm called the *reach* algorithm. The first phase of this algorithm derives a preliminary value for D which we may call D_p. D_p describes the definition points in an interval which can reach a program point in the interval without looping back through the header. For each variable v which has a definition point in the loop, a new program point will be introduced. This program point will be denoted $CP(v)$. For those program points which are not the introduced cutpoints, $D_l(p,v)$ includes $D_p(p,v)$. In addition, if $D(p,v)$ includes any program points not in $D_p(p,v)$, then D_l must also include $CP(v)$. For the cutpoint corresponding to the variable $v, D_l(p,v)$ is the same as $D(h,v)$ where h is the interval header.

General Range Analysis

Associated with each program point, we define a collection of information called the *derivation*. The derivation describes all that is known about how the value of each source variable at that definition point is derived from other program points. The derivation of a variable v at program point p is written $\Delta(p,v)$. In its most general form, the derivation for a source variable is a set of equations by which the value of that variable may be derived from values computed at other program points. Initially, the derivation for each source variable v at a program point p is the set of equalities with its *def* points in the cut graph, i.e., $\{v=\langle j\rangle | j\epsilon D_l(p,v)\}$. After a program point is processed while doing range propagation, a set of equations describing the value assigned to the target are established using the operator at the program point and the derivations for the source variables. These equations are used to substitute for occurrences of the definition point wherever it appears in other derivations. The program points in whose derivations such substitutions are made are added to Ψ. Note that since the cutpoints are not processed by range propagation, this substitution rule also does not apply. Instead, whenever it is the case that the derivation at a cutpoint of some loop refers only to itself and to program points outside the loop, the derivation describes a recursive set of values determined by the values on entry to the loop. This set of values is computed under some assumption about the number of iterations of the loop, and the result is established as the range at the cutpoint. The loop count is initially assumed to be infinite, but may be modified as described below in the section on *loop counting*. The derivation may thus, in some cases, be used to derive a range for the result. In fact, it is sometimes the case that mutual recursions exist among the derivations associated with several cutpoints of a loop. This subset of the derivations can be solved if:

1) the equations in each derivation refer only to program points outside the loop and to other cutpoints in the subset being considered, and

node	ranges established				derivations			new Ψ
	ρ(1)	ρ(2It)	ρ(2If)	ρ(3)	Δ(1,I)	Δ(2I,I)	Δ(3,I)	
	[-∞:∞]	[-∞:∞]	[-∞:∞]	[-∞:∞]	{}	{I=<1>, I=<3>}	{I=<2I>}	{1,2,3}
1	[1:1]					{I=1, I=<3>}		{2,3,2I}
2		[11:∞]	[-∞:10]					{3,2I}
3				[-∞:11]		{I=1, I=<2I>+1 }		{2I,2}
2I						I∈[1:∞]		{2,3}
2		[11:11]	[-∞:10]					{3}
3				[2:11]				{2}
2			[1:10]					{3}
3				[2:11]				{}

Fig. 5. Range propagation augmented by range analysis.

2) the graph showing which cutpoints are involved in the derivations for a cutpoint is a reducible graph whose order 2 induced graph [1] is acyclic.

In order to integrate the ranges thus established with the information established by range propagation, additional processing is performed. Those program points whose derivations refer to a cutpoint whose range has been established are added to Ψ and, in addition, they are flagged with an indication that derivation processing is required. Whenever a node with such an indication is processed by range propagation, an auxiliary range, denoted by δ_i, is computed by evaluating the equations in the derivation after substituting for each *def* point its range. Range propagation then uses the intersection of π_i with δ_i as the value for each operand of the instruction to compute the new result range.

Consider the example used to illustrate range propagation in an earlier section assuming that the cutpoint named $2I$ was introduced above node 2. Fig. 5 should be compared with the similar table presented to illustrate range propagation (Fig. 2). The major improvement occurs after node 3 is processed for the first time. The derivation established for node $2I$ contains sufficient information to conclude that the value of I at the cutpoint must be greater than or equal to one. But when reprocessing node 3, this constraint is intersected with the already established range of I less than or equal to 10. Hence, the input range is [1 : 10] and the result range is [2 : 11].

On occasion, it is useful to establish that a variable takes on *all* values in a range in a monotonic fashion. A recursive definition in which the value of the variable is incremented or decremented uniformly by one can be used to establish this property. The utility of this information derives from the tendency of programmers to compare for equality in the loop terminating condition, and from the fact that it may be necessary to establish that some assignment has been made to *all* elements of an array.

Restricted Range Analysis

Dealing with the most general form of derivation may be appropriate for programs intending to use the value ranges for construction of proofs about the properties of programs. In fact, for program analyses like that discussed by Sites [3] or even King [4], the use of the data flow information to direct the propagation of derived information can improve the efficiency of the process. Rather than representing the "state" of a computation completely at each point, the data flow information allows immediate propagation of information to its point of use.

For the purposes of program optimization in a compiler, however, the overhead implied by the symbolic manipulation of general expressions in a derivation is probably not worth the improvement which can be achieved over more restricted forms. From a practical point of view, range analysis is probably most useful for propagating constants of various types and for keeping track of the linear or geometric sequences encountered for the control variables of loops. In order to cope with such sequences, and to utilize certain other monotonicity properties of operators, the equations in a derivation should probably be restricted to a subset of the form:

$$\langle\text{source-variable}\rangle = \langle\text{constant}\rangle \times \langle\text{program-point}\rangle \pm \langle\text{constant}\rangle.$$

Whenever a substitution can be made which preserves the valid form of a derivation, it is made. If the substitution would violate the form it is not made. However, when the derivation at a cutpoint is being examined to determine if an induction can be made, the range of values at such nonsubstitutable *def* points may be substituted instead. For example, assume cutpoint $9J$ of some program has the derivation $\{=1, =\langle 9J\rangle+1, =\langle 15\rangle+3\}$. Furthermore, assume that although program point 15 is within the loop, $\rho(15)=[-5:4]$. The conclusion can still be drawn that the source values for J are in the range $[-2:\infty]$. Without further information, however, it is impossible to know what increments may be applied to J for each iteration of the loop.

Loop Counting

Given the preliminary information generated by range propagation and the recursion information established by range analysis, it is often possible to derive an upper bound on the number of iterations made when the loop is entered. This loop count can be established if three conditions hold.

Condition 1: There must exist at least one test of a numeric variable which is at a program point that will be executed on each iteration of the loop.

Condition 2: This test must compare the variable to an established range.

Condition 3: The variable involved must have a solvable set of recurrences established by range analysis.

In fact, each such test point in the loop will establish an upper bound, and the smallest of these is the least upper bound derivable with the given information. This bound may then be fed back into the range analysis to revise the ranges assigned to the cutpoints. Although more general cases may be solvable, it is useful to note that if the form of the equations in derivations is restricted to the linear form described above, the loop count may always be established in a direct manner. Furthermore, the vast majority of loops contain loop variables which take on values in some arithmetic or geometric series, and solutions to the loop counts in such programs is often possible within this restricted form.

Representation of Ranges

The choice of representation for a range depends most critically upon the data type.

Boolean and Character

These ranges are best represented by listing the values which are possible. However, the ranges of bit string data can be represented as a string of 0's, 1's, and X's which represent unknown bit positions.

Numeric

The numeric data types lend themselves to the most extensive analysis, and hence the representation of numeric ranges is the richest of the scalar types. A numeric range can be represented as a list of range descriptions. Each range description describes an arithmetic sequence with a lower bound, upper bound, and increment. The total range is the union of the ranges given by the range descriptors. An extremely useful additional piece of information is an indication for each range description of whether or not the range is densely and monotonically covered, and whether the monotonicity is increasing or decreasing. This information is useful both in the representation of array ranges and in dealing with those loops whose termination test is expressed as an equality rather than an inequality. Although machine generated code rarely has this form, humans have a tendency not to be defensive in their programming and to, in fact, write loops which terminate when the counter is equal to some set value. The lists of range descriptions should have a bounded length, and this bound provides a useful mechanism for implementing the various range operations such as union and intersection. If an algorithm is implemented which compresses, at some possible loss of accuracy, a range description list of length m to fit in length n, then the union and intersection operations can be written to merely concatenate or pairwise intersect their inputs. The resulting lists will be too long, but the choice of how to compress them can be done in a uniform way.

In addition to the range constant information, symbolic range information may also be maintained. The most general form of this information is a conjunctive (or disjunctive) normal form for propositions, where each proposition describes a bound on the range. These range bounds are most generally expressed as a relation between values in the range and an expression involving other definition points in the program. In a real implementation, these expressions would not be more complex than those manipulated by the range analysis algorithms, and might reasonably have an even simpler form. A choice consonant with the arithmetic/geometric series restriction described above is to restrict range bounds to a subset of the form:

<relational-operator><definition-point>±<constant>.

Rather than allowing the full complexity of a conjunction of disjunctions of these range bounds, it is probably sufficient to restrict the form of the symbolic range information to a conjunction of range bounds, suppressing disjunctive information entirely. The union and intersection of such ranges can then be easily computed as the intersection or union, respectively, of the sets of range bounds.

Arrays

Arrays are a continual nemesis in program analysis, but the analysis of ranges can be applied very usefully to them. Whenever an assignment is made to an array element, and the subscript range is dense and monotonic with an increment of one, then the array elements whose subscripts lie within that range are certain to have the range computed by the right-half side of the assignment. If no symbolic ranges are employed, an array range can be represented as a list of pairs. Each pair contains a range for the subscripts and a range for the values. In the presence of symbolic ranges, however, the situation is more complex since the relation between symbolic values may not be known when the range is established. A trick can be employed by which the symbolic ranges for an array at its various *def* points are chained together. An array range would then consist of a subscript range paired with an element range, and in addition, a list of previous definition point predecessors would be retained. When the range of some element is desired, the array range value is examined. If the desired element is provably in the subscript range then the value range is used. If the desired element is provably not in the subscript range, then the union of the various ranges for that element given by examining the predecessors is used. If the case is not determinable, then the union of both ranges is used.

Test Elision Using the Range Information

The data flow algorithm described in [2] maintains the *use* and *def* information associated with the edges of the control flow graph, rather than with the nodes. This complication has been avoided in the previous description because the values at a node can be derived as the union of the values on incoming (for *def*) or outgoing (for *use*) edges. Maintaining the information in edge form, however, has the useful consequence that it is possible to determine the value of a variable separately on each edge entering a test point. Whenever it is established that the range of a variable on some incoming edge is sufficiently constrained to guarantee the resulting direction of control flow, the edge may be rerouted so that it leads to that successor of the test so determined. This can result in removing a loop in the program or converting the loop into straight-line code.

Consider the program fragment in Fig. 6. The rerouting performed in this example significantly altered the data flow of the program. Some of the original definition points for the test of J no longer reach the test, and if this alteration of the data flow goes unrecognized, the elimination of that test, and possibly of all the code at B as well, will not be made. The fact that such a significant alteration of data flow is possible requires either that a strategy for incrementally altering the data flow functions be employed, or that the alterations be "batched" in alternate cycles of analysis and flow modification. Occasionally, range propagation will compute an empty range on exit from some test. This indicates that the subsequent code cannot be reached via that test and may therefore be dead. If the "batched" approach to flow graph alteration is employed, some efficiencies can be gained by detecting the

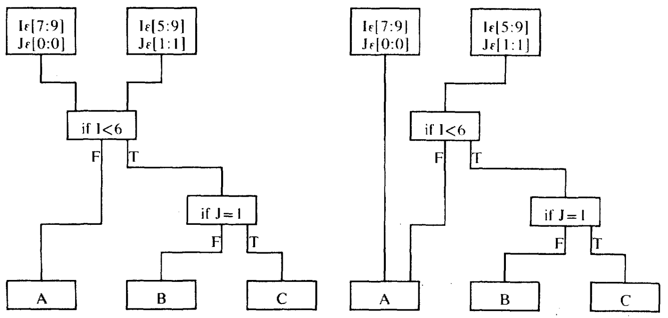


Fig. 6. Flow graph before and after eliding test of $I<6$.

presence of dead code and setting the ranges associated with all the definition points in the dead code to empty ranges. This may considerably improve the accuracy of the range propagation results on each pass of analysis, and may also reduce the number of analysis/modification cycles required to achieve a stable result.

Diagnostics Using the Range Information

The placement of *use* and *def* information on the edges of the control flow graph permits the generation of diagnostics to give the programmer insight into potential bugs in his program. After dead code is eliminated from the program, the compiler can examine the program to determine if any invocations of the error diagnostic routine remain to be invoked at run time. If so, each such point serves as a starting point for the generation of diagnostic information. Each error point is usually preceded by some test which could not be elided. The programmer should be notified of the fact that this test point can lead to an error, along with the names of the variables involved in the test and their ranges. However, better information may also be derivable. If only one *def* point leads to the point at which an error may be noticed, that *def* point should be examined. If the erroneous values resulting at that *def* point could be caused by only a single source variable, and by only a single *def* point of that source variable, then this further *def* point should be examined. This process is repeated until multiple sources for the erroneous values are detected. The compiler can then generate a diagnostic which indicates that the error indication which arises from the error point can be caused only by values at the last examined *def* point. This point can reasonably be regarded as the earliest point at which the error could be detected. The compiler should also provide range information on the value defined at that point. In addition, an option could be provided which allows the compiler to remove the error test from its original location and replace it with an equivalent test at the earliest point at which the error can be detected. If the error test is necessarily performed at run time, moving the test in such a manner may result in further optimization of the program.

Termination

One of the necessary properties of an algorithm is that it terminates. Informally, the proof of termination goes as follows.

In order to show that this process terminates, it is only necessary to show that program points may be added to the set Ψ only a finite number of times. A program point is added to Ψ for either of two reasons: a substitution is made into its derivation, or some refinement is made in a range which defines one of its source variables. These two cases are considered separately.

Derivations

The data flow graph derived from the derivations of the program points is a directed acyclic graph because it results from the introduction of cutpoints into the original data flow graph of the program. For the purpose of counting the number of additions made to Ψ, we may reflect the fact that substitutions may be made separately into each of the successors of a node by splitting the nodes in this graph to form an equivalent graph in which each node has only one successor. This equivalent graph is also directed acyclic. A program point is added to Ψ at the same time that a node is removed from this equivalent graph by a $T1$ transformation [5] which removes a node from the graph. The number of times a program point is added to Ψ by reason of derivation processing is therefore bounded by the number of nodes in the equivalent graph.

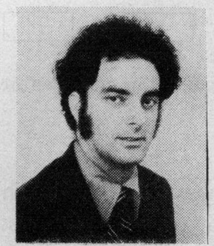
Range Refinements

Additions are made to Ψ only when the range of a variable at some *def* point is narrowed. But since computer representations allow for only a finite number of values, this narrowing can be performed only a finite number of times at each program point. But there are only a finite number of program points and hence, only a finite number of additions may be made to Ψ.

This proof of the termination of the range propagation process is, however, unsettling since the guarantee that a loop will iterate only several trillion times is not comforting. To help provide a more realistic estimate of the processing requirements two observations are in order. First, most ranges are completely narrowed when each program point has been processed at most three times. The first time establishes a bound derived from the termination test of some loop containing the statement. The second time forwards the derivation information to construct the induction characteristics of the loop. The third time establishes a bound derived from the initial conditions and the induction information of the loop. Second, some ranges cannot be completely represented. For example, a variable which ranges over the primes can have no expression other than as a list of specific values. Furthermore, even a geometric series of values is not representable with the restricted range representation suggested above. In these cases, the range propagation algorithm will attempt to completely enumerate the list of acceptable values whenever the range is not an arithmetic series. This complete enumeration is prevented by the choice of a finite representation for ranges. Because the representation is constrained in size, the number of iterations is limited to the number of separately listable values in the representation. At some point in the processing, an overly broad range is derived which is representable within the size constraint. At that point, no further refinement is possible.

References

[1] F. E. Allen, "Control flow analysis," in *Proc. ACM SIGPLAN Symp. Compiler Optimization, SIGPLAN Notices*, vol. 5, pp. 1-19, July 1970.
[2] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 137-147, Mar. 1976.
[3] R. L. Sites, "Proving that programs terminate cleanly," Doctoral dissertation STAN-CS-74-418, Dep. Comput. Sci., Stanford Univ., Stanford, CA, May 1974.
[4] J. C. King, "A new approach to program testing," in *Proc. Int. Conf. on Reliable Software*, Los Angeles, CA, Apr. 1975.
[5] M. S. Hecht and J. D. Ullman, "Characterizations of reducible flow graphs," *J. Ass. Comput. Mach.*, vol. 21, pp. 367-375, July 1974.



William H. Harrison (M'72) received the B.S. degree in electrical engineering from Massachusetts Institute of Technology, Cambridge, in 1968 and the M.S. degree in systems and information science from Syracuse University, Syracuse, NY, in 1973.

While associated with the system development divisions of IBM from 1966 to 1970, he worked on FORMAC, a system for algebraic manipulation, on the design and implementation of an experimental time-sharing system for building large programming systems, and on the design of the OS/360 Time Sharing Option. Since 1970, he has been associated with the IBM Thomas J. Watson Research Laboratory, Yorktown Heights, NY. During this time 4e has participated in the design of a language with extensible and highly structured data types, and has consulted with the systems development divisions on the formal definition of advanced systems. He is currently investigating experimental techniques for program analysis and optimization and for compiler organization.

Mr. Harrison is a member of Sigma Xi and the Association for Computing Machinery.

A Study of the Physical Structure of Algorithms

STUART H. ZWEBEN

Abstract—A theory of the structural composition of an algorithm is presented which allows the frequencies of occurrence of the individual operators and operands to be estimated. It provides justification for some recent hypotheses which suggest certain functional relationships between properties of algorithms.

The theory for operands is based in part on models of program construction due to Bayer, while that of operators is based on the work of Zipf and Mandelbrot in natural language. A further relationship between the construction of algorithms and natural language text is indicated by demonstrating that the size of an algorithm as predicted by one of Bayer's programming models and the size of a piece of text as predicted by Zipf's natural language model are identical.

The theory is tested experimentally on a variety of algorithms written in several programming languages with good statistical results.

Index Terms—Algorithm structure, frequency analysis, programming models, programming study, software physics.

Manuscript received June 23, 1976; revised August 30, 1976.

The author is with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210.

I. Introduction and Background

FOR THE past several years, many experiments have been undertaken which have proposed some interesting functional relationships among properties of algorithms. In particular, there have been a number of papers (see, for example, [2]-[6]) written in support of a hypothesis due to Halstead [1] that the length N of a well-written algorithm (defined to be the total number of occurrences of all of its operators and operands) is related to the cardinalities $\eta 1$ and $\eta 2$ of its operator and operand sets, respectively, by the equation

$$N = \eta 1 \log_2 \eta 1 + \eta 2 \log_2 \eta 2. \quad (1)$$