Analysis and Optimization of Set Expressions

R. Nigel Horspool and Laurence W. Dunkelman†
School of Computer Science, McGill University, 805 Sherbrooke Street West, Montreal, Quebec H3A 2K6, Canada

The problem of minimizing the lengths of bit vectors used to implement sets in Pascal is considered. An analysis algorithm is presented that determines these minimum lengths. It is proved that two passes are both necessary and sufficient. Implementation concerns involving sets and the analysis algorithm itself are considered.

INTRODUCTION

The Pascal programming language, of all the commonly used languages, is unusual in providing sets. A program may manipulate sets of integers (or sets of any scalar type) and perform the standard set operations of union, intersection and difference. In addition, Pascal defines a set membership test, set equality tests, set inclusion tests and a function for calculating the cardinality of a set. Some less widely available languages that provide sets and set operations are SETL, MODULA, EUCLID and SUE. Of these, all but SETL owe much of their character to Pascal.

Many different data structures are suitable for the implementation of sets. Possibilities include data structures based on linked lists, on binary trees, on trie memory and on hash tables. However, the most convenient data structure from the compiler writer's point of view is a bit vector. With such a representation, a one bit in index position $i$ indicates the presence of $i$ in the set (and a zero indicates its absence). Of course, it is not mandatory for the first bit to correspond to index value 1 (or 0); nor is it necessary for set elements to be integers (we can use the internal binary representation of almost any datatype to index the bit vector).

Bit vectors provide the representation that is most economical in storage if the range of permissible element values is not great. Furthermore, set operations can be directly compiled into logical instructions that are almost always available on the target computer. For example, a set union may be compiled into an 'OR' instruction.

The Pascal Report (Ref. 9, report section 14) advocates the bit vector representation for sets and it explicitly permits implementors to limit the allowed ranges of element values for this purpose.

If bit vectors are used to implement sets, how large should these vectors be? In several Pascal implementations, including the original Zurich compiler for the CDC 6000, all sets occupy exactly one word of memory. For the Zurich compiler, this leads to the restriction that the elements of sets must have (internal) values that lie in the range 0–58 [Ref. 9, user manual section 13.C].

Giving all sets the same storage size and picking some convenient unit, such as a word or a doubleword, for this size is unnecessarily restrictive. On some computers this may have unfortunate consequences. A typical problem is caused by sets with character valued elements. According to the Pascal manual, a statement with the structure

```
if INPUT in ['A'..'Z', '0'..'9'] then ....
```

is quite legal and very useful. (In fact, statements similar to this appear in several programming primers for Pascal.) However, characters may have a six, seven or eight bit internal representation (depending on the computer and the operating system environment). This implies that a set with character elements may need up to 64, 128 or 256 bits of storage respectively. It is unlikely that the target computer's word size (or even the doubleword size) is quite that large. CDC 6000 Pascal comes quite close—six bit characters are used and the word size is large enough that nearly all sets with character elements can be represented. However, we are faced with the unfortunate fact that several Pascal implementations so restrict the allowed range of element values that sets of characters become (almost) unusable. (Explaining to a class of novice programmers why they cannot use sets of characters is a trial in itself.) As Welsh et al. points out, even an apparently implementable type definition such as 'set of 1939..1945' may be disallowed due to compiler restrictions on sets.

Ideally, we would like to totally remove any restrictions on set element values. The bit vectors used to implement set constants, set variables and intermediate set expressions should be made just as large as necessary. Code generation is certainly made a little more difficult because a single set operation such as union may have to be compiled into several logical instructions—each processing different segments of the big vectors involved. Alternatively, it may be more convenient to compile set operations into calls on run-time support routines. However the enhanced user convenience should more than outweigh the difficulties of code generation.

It is a trivial problem for a compiler to determine the storage needed for set constants and set variables. To simplify our exposition here, we will assume that bit strings with arbitrary lengths may be accessed and manipulated fairly easily on the target computer. Later, we will discuss some implementation compromises that are desirable for simplifying code generation and for making the compiled code more efficient. A variable declared with the type 'set of 0..99' clearly requires 100 bits of storage. A set constant such as [5, 10, 12, 22] requires 18 bits because its type is implicitly 'set of 5..22' (although Welsh et al. argue that its implicit type is really 'set of INTEGER') and there are 18 values in the

†This work was supported by the Natural Sciences and Engineering Research Council of Canada under grant A4333.
ANALYSIS AND OPTIMIZATION OF SET EXPRESSIONS

var S1: set of 0..20;
S2: set of 5..15;

("PRINT OUT HOW MANY ELEMENTS ARE IN EITHER S1 OR S2 BUT NOT IN BOTH."

writeln (card ((s1 + S2) - (s1 * s2)));

Figure 1

range. It is less trivial for the compiler to decide how much storage to allocate to an intermediate set expression. Consider the sample program fragments shown in Fig. 1. The intersection S1*S2 is computed and must be saved in temporary storage. A simple one-pass compiler can easily determine that the range of values for S1*S2 is the intersection of the ranges for S1 and S2. That is, the compiler can allocate a temporary set variable whose element range is 5..15. Similarly, the range of values for the union S1 + S2 can be calculated as 0..20.

However, there are frequent situations when a one-pass compiler cannot deduce a finite range for the elements in an intermediate set. For example, consider the pair of Pascal statements given in Fig. 2. If I and J have been declared as having INTEGER type, then the possible range of element values in the newly constructed set (i.e. the term in square brackets) is unknown to the compiler. We will refer to this new set as a 'set constructor' in the discussion below. The set constructor clearly needs to be allocated some temporary storage, like any intermediate set expression. The implicit type of the set constructor's result is 'set of INTEGER', therefore we should theoretically implement the set constructor as a doubly-infinite bit vector. In practice, a one-pass compiler would have to impose some implementation restriction and assume some maximum range for the elements in a newly constructed set. But inspection of the example statements reveals that a multi-pass compiler can find a small finite range for each intermediate set in the assignment statement. There are two factors at work here. First, the final effect is to assign a set to variable S1. This variable has some declared range for its element values. There is no purpose to computing or remembering element values outside the range declared for S1. If the right-hand side of the assignment should turn out to contain element values outside the range declared for the left-hand side, this would be an error and a diagnostic compiler would be obliged to report this error. However, an optimizing compiler may be forgiven for ignoring the possibility of such errors. For efficiency, it should simply not inspect elements in operand sets that are outside the declared range for elements of the target variable. Therefore, an optimizing compiler should allocate no more storage to the set constructor than it does to S1. The same consideration applies to the two other temporary set expressions that are calculated. The second point to notice is that there is an intersection with S3. This intersection is guaranteed to eliminate all elements outside the declared range for elements in S3. Hence the set constructor and the two temporary sets need be allocated no more storage than S3. Either or both of these two effects may be important in minimizing the storage allocated to temporary sets.

The EUCLID language avoids problems with newly constructed sets because it requires an explicit type name to be supplied. For example, if INTSET has been defined as the name for the type 'set of 0..100' then the expression INTSET(I, J) would be used to denote a set constructor (for a set of type INTSET) with elements I and J. Thus, the EUCLID compiler immediately knows some reasonable bounds for a bit vector to implement the set constructor. A similar proposal has been suggested for Pascal and implemented in one version of Pascal.

In this paper, we will show that the removal of all implementation restrictions on set element ranges in Pascal is an impossible goal if we are constrained to use a pure bit vector implementation. However, for most practical situations, the goal is realizable. We will present an algorithm for analyzing set expressions and set assignments in order to determine minimum storage allocations. This may permit more efficient translations of set operations into instructions on the target computer. We will also prove that two passes through the intermediate code, one forwards and one backwards, are required by our algorithm in the worst case.

OPTIMIZATION OF SET EXPRESSIONS

We will first consider how to analyze an isolated set expression or a single assignment of a set expression to a variable. We will discuss more general analysis of set operations subsequently. We will consider only set expressions that can be represented as trees. The more general DAG (Directed Acyclic Graph) form can arise after certain optimizations have been performed will be discussed later.

The algorithm is based on iterative improvement of range information currently known to the compiler. If S is a set variable, intermediate set variable, set constant or a set constructor, then we will use Range(S) to denote the set of all values that the compiler believes to be useful elements of S at this point in the program. (Useless elements in S would be any elements whose presence cannot influence the computation of the complete set expression or assignment.) For example, if S is a variable with the declared type 'set of 10..100' then Range(S) will be a set containing some, but not necessarily all, values in the range 10-100. The result of our analysis will be used to influence the code generated by the compiler for the expression under consideration. One of its objectives is to reduce the amount of storage allocated to temporary set variables. A second objective is to improve the execution time of the code generated for the set expression. However, the analysis cannot immediately be used to reduce storage allocated to set variables declared by the programmer. (This would require global program analysis.)
Consider a specific set operation such as

\[ T := L \ast R \]

where \( T \) denotes a compiler generated temporary set variable. Initially, no range information would be available for the elements of \( T \) and Range \((T)\) would be initialized to the universal set. Usually a little information about \( L \) and \( R \) would be initially available to the compiler (from declarations for \( L \) and \( R \) perhaps). This information is represented by Range \((L)\) and Range \((R)\). Now, the range information for \( T \) can be improved by applying the formula:

\[ \text{Range}'(T) \leftarrow \text{Range}(T) \ast \text{Range}(L) \ast \text{Range}(R) \]

where \( \text{Range}'(T) \) represents a possibly more restricted set of possible elements in \( T \) and that should replace Range \((T)\) in future analysis.

The rule for improving the range of \( T \), above, can be applied in a conventional one-pass compiler. That is, information about element ranges is propagated both backwards in a program and sideways (from sibling to sibling) in an expression tree.

After this preliminary explanation, we can now state our analysis algorithm more formally. There are three parts to the method, of which the first two parts merely define how the data is to be presented.

**Step 1.** The set expression or set assignment to be analyzed must be decomposed into a sequence of elementary set operations. These elementary operations correspond to `code triples' that are often used as an intermediate program representation in compilers. Each operation must take one of the forms:

\[ T := L \text{ op } R \]

or

\[ V := L \]

where \( \text{op} \) represents one of the operations '+' (union), '*' (intersection) or '-' (difference); \( T \) denotes a compiler generated temporary set variable; \( V \) denotes the target variable in an assignment; \( L \) and \( R \) each denote a set constant, set variable, temporary set variable or a set constructor. Note that the form \( V := L \) is only used when we are analyzing an assignment (rather than an isolated expression) and, in this case, it is always present and appears as the last operation.

**Step 2.** The compiler must determine initial range information for every object in the expression/assignment that represents a set. When \( S \) is a constant set, Range \((S)\) is simply equated to that set. When \( S \) is a set variable, Range \((S)\) is simply initialized to a set containing all values in the declared range. If a set variable \( S \) occurs more than once in the expression or assignment, we must establish a separate range set for each occurrence of \( S \). For a set constructor or a temporary set, Range \((S)\) would normally be initialized to a set containing all values of the appropriate datatype. Of course, we need a special representation for a universal set when the element type is INTEGER. Also, it should be noted that information obtained from sources other than the program declarations (e.g. from other analysis techniques) may be used to reduce the sizes of the initial range sets.

**Step 3.** The compiler makes a forward pass and then a backward pass through all the set operations in the sequence. For each operation encountered, the compiler applies the rules of Table 1 to deduce new range sets for every operand appearing in the operation. The range sets cannot become larger as a result of these actions and would usually become smaller. Note that the analysis rule given for a set assignment should not be applied if detection of out-of-range set elements is required. After these two passes, the range sets are the smallest possible (without resorting to analysis of other components of the program) and their contents may be used to optimize set operations and storage allocations.

\[ \begin{align*}
\text{Table 1} \\
\text{Operation or expression} & \quad \text{Iteration rules} \\
T := L + R & \quad \text{Range}'(T) = \text{Range}(T) \ast (\text{Range}(L) + \text{Range}(R)) \\
& \quad \text{Range}'(L) = \text{Range}(T) \ast \text{Range}(L) \\
& \quad \text{Range}'(R) = \text{Range}(T) \ast \text{Range}(R) \\
T := L \ast R & \quad \text{Range}'(T) = \text{Range}(T) \ast (\text{Range}(L) \ast \text{Range}(R)) \\
& \quad \text{Range}'(L) = \text{Range}(T) \ast \text{Range}(L) \\
& \quad \text{Range}'(R) = \text{Range}(T) \ast \text{Range}(R) \\
T := L - R & \quad \text{Range}'(T) = \text{Range}(T) \ast (\text{Range}(L) - \text{Range}(R)) \\
& \quad \text{Range}'(L) = \text{Range}(T) \ast \text{Range}(L) \\
& \quad \text{Range}'(R) = \text{Range}(T) \ast \text{Range}(R) \\
L \leq R & \quad \text{Range}'(L) = \text{Range}(L) \ast \text{Range}(R) \\
L > R & \quad \text{Range}'(L) = \text{Range}(L) \ast \text{Range}(R)
\end{align*} \]

*Note: the rule for \( T := L \) should only be applied if there is no requirement to check that the right-hand variable, \( L \), does not contain elements outside the declared range for \( V \).*

An example of the operation of this algorithm is shown in Figs 3(a), (b) and (c). The relevant fragments of a Pascal source program are shown in Fig. 3(a); the corresponding sequence of elementary set operations (code triples) is shown in Fig. 3(b); the successive contents of the range sets are shown in Fig. 3(c). (In Fig. 3(c), we have used `U' to represent the universal set containing all values of the INTEGER datatype.)

The example illustrates two things. First, it shows that the range sets do not necessarily consist of consecutive sequences of element values. This is why we chose to represent the range information by sets rather than by a pair consisting of the smallest and largest element values. The use of such a pair would be a great implementation convenience and its implications are discussed in the next section of this paper. Secondly, the example shows that a single forwards pass is not sufficient for minimizing the range sets. The reader may verify for himself that a single backwards pass on this example would not be sufficient either.
then a theorem that proves the desired result.

source code are sufficient, we will introduce a lemma and

Figure 3

cannot cause any modification to the range set for T. •

Operation being carried out.

we have the relation:

\[ \text{Range}' (X) \subseteq \text{Range} (X). \]

Applying the analysis rules in Pass 2 implies the relationship:

\[ 2 \text{Post} (T) = 2 \text{Pre} (T) \ast Z \quad (3) \]

The important observation to make is that Z denotes exactly the same set as on Pass 1. Neither Range (L) nor Range (R) can have been affected in the meantime because they simply do not appear in any analysis rules.

From Eqns 1 and 2, we infer that \( 2 \text{Pre} (T) \subseteq Z \). Therefore we have \( 2 \text{Pre} (T) \ast Z = 2 \text{Pre} (T) \). Combining with Eqn 3, we obtain the desired result, \( 2 \text{Post} (T) = 2 \text{Pre} (T) \).

Theorem. Two passes through the source code, one down followed by one up, are sufficient.

Proof. Let us assume the theorem to be untrue. Suppose that there is a sequence that requires a third pass. Let us consider the first operation that causes a range set to be modified on a third pass. (It does not matter whether we assume that this third pass goes through the code sequence up or down.) First, this operation in question cannot be an assignment operation with the form \( 'V = L' \) because it is clear that Range (L) can be changed only once by the corresponding analysis rule. Therefore, the operation must have the form \( 'T := L \op R' \).

If L represents a constant set, a set constructor or a set variable, then Range (L) is a distinct set not subject to change when any operation other than the one under consideration is analyzed. If L represents a compiler generated temporary, then L may only appear on the left-hand side of some other operation in the sequence. According to the lemma, Range (L) cannot have been changed when this other operation was processed in Pass 2. Similar arguments hold for R, that is neither Range (L) nor Range (R) can have changed since the operation was analyzed in Pass 2.

\( T \) denotes a compiler generated temporary. It does not precede the operation under consideration in the sequence. Thus, Range (T) could not have been changed subsequently to this operation being analyzed in Pass 2. Of course, Range (T) could not have been changed in Pass 3 yet because, by supposition, we are considering the first operation in Pass 3 where a change occurs.

If none of Range (T), Range (L) and Range (R) have changed since the analysis rules were applied to the operation in Pass 2, then re-application of the rules will not produce any different results.

Thus, we have contradicted our initial assumption that some range set is altered by Pass 3 and hence a third pass cannot be required. 

As a final note, we would like to point out that no other combination of two passes is sufficient. That is, there are code sequences for which two down passes, two up passes or even an up pass followed by a down pass do not produce minimal range sets. An example of an instance where an up pass followed by a down pass is insufficient is shown in Fig. 4.

Figure 4

<table>
<thead>
<tr>
<th>Set</th>
<th>Range set, after initially Range set, after Pass 1</th>
<th>Range set, after Pass 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>{8..12}</td>
<td>{8..12}</td>
</tr>
<tr>
<td>S2</td>
<td>{17..35}</td>
<td>{17..20}</td>
</tr>
<tr>
<td>T1</td>
<td>U</td>
<td>{8..12, 17..35}</td>
</tr>
<tr>
<td>S3</td>
<td>{9..40}</td>
<td>{9..12, 17..35}</td>
</tr>
<tr>
<td>T2</td>
<td>U</td>
<td>{8..12, 17..35}</td>
</tr>
<tr>
<td>S4</td>
<td>{5..25}</td>
<td>{8..12, 17..20}</td>
</tr>
<tr>
<td>T3</td>
<td>U</td>
<td>{8..12, 17..35}</td>
</tr>
<tr>
<td>S0</td>
<td>{0..20}</td>
<td>{0..20}</td>
</tr>
</tbody>
</table>

Figure 3
IMPLEMENTATION CONCERNS

The analysis algorithm presented in the previous section is a little too powerful and too expensive in storage for most applications. The algorithm generates sets that list all relevant element values for an expression or assignment. However, the compiler probably needs to know only the values of the smallest and largest elements that need to be fetched, stored or manipulated. This information is sufficient for allocating storage to temporaries and for generating the logical instructions that implement the various set operations.

It is natural to propose that the algorithm should be modified so that the Range sets are implemented as value pairs. The pair naturally consists of the values of the smallest and largest useful elements. These value pairs may be merged and intersected in an analogous way to sets, so there should be no need to re-explain the analysis algorithm.

There is one drawback to using value-pairs as the implementation of the Range sets. It is that we can no longer guarantee two passes through the code to be sufficient. In the worst case, the number of passes may be proportional to the number of set unions in an expression. An example set assignment that needs three analysis passes is exhibited in Figs 5(a), (b) and (c). In Fig. 5(c), 'U' denotes the universal range and '# denotes an empty range. A generalization of this example is given in Fig. 6. This figure shows an example containing $2^k - 2$ unions that requires $2k$ passes.

\[
\text{Figure 5}
\]

<table>
<thead>
<tr>
<th>Set</th>
<th>Range set, initially</th>
<th>Range set, after pass 1</th>
<th>Range set, after pass 2</th>
<th>Range set, after pass 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>${1, 10}$</td>
<td>${1, 10}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>S2</td>
<td>${21, 30}$</td>
<td>${21, 30}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>T1</td>
<td>U</td>
<td>${11, 20}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>S2</td>
<td>${11, 20}$</td>
<td>${11, 20}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>S0</td>
<td>${1, 1000}$</td>
<td>${11, 20}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Inspection of Fig. 6 reveals that the sample set expression possesses a curious property. The analysis algorithm successively finds that the unions have no effect on the final result. This situation characterizes the case when additional passes through the code are required. Expressions that contain totally redundant terms are not very common. However, programmers may justifiably choose to structure their expressions for clarity rather than for efficiency. Therefore, we recommend making the following addition to our analysis algorithm. We simply add a fourth step as follows:

Step 4. If any Range sets were newly discovered to be empty in the upward pass of step 3 then return to step 3.

We shall omit a formal proof that the revised algorithm is correct. Instead, we will suggest some modifications to the proof of the original algorithm that render it applicable to the new version of the algorithm. First, the lemma must be restated to say that the upward pass through the code cannot modify the range set for $T$ unless one of the range sets for $L$ or $R$ has been found to be empty in this pass. (The proof of the lemma needs to be restructured so that the properties of range unions and range intersections are used rather than the properties of set unions and set intersections.) Second, the theorem must be restated to say that two passes are sufficient except when a range set is newly discovered to be empty in the second, upward, pass. The proof of the theorem needs no substantive change. Finally, we argue that step 4 of the revised algorithm automatically forces two more passes through the code whenever the theorem does not apply. Hence, we can conclude that step 4 guarantees that a sufficient number of passes through the code are made.

A minor issue of concern to implementors is bit vector alignment in the generated code. Most computers group bits together into basic units such as bytes or words. The logical instructions of the computer would normally operate on an entire storage unit at a time. It would be quite inconvenient to generate code for a set union if the first operand were not aligned with respect to word boundaries in the same way as the second operand.

To avoid alignment problems, it is desirable to pad sets with filler bits so as to maintain consistency between all sets in the program. For example, with a 16-bit word size, we would pad the sets so that the first bit in every word used for set storage corresponds to the same element value modulo 16.

DISCUSSION

There are a few contexts in Pascal where strict typing restrictions are not imposed on sets. One obvious example occurs with the built-in function CARD, which counts the number of elements in a set. CARD is a generic function that may be applied to a set of any size or with any element type. An implication is that we may be unable to derive a suitable size for a bit vector that implements the argument to the CARD function. For example, the statement

\[ I := \text{CARD}([1, 4, 9, J*K]); \]
ANALYSIS AND OPTIMIZATION OF SET EXPRESSIONS

defeats our analysis algorithm. Unless there are some reasonably tight bounds available for the values of J and K in the example, we cannot determine any bounds on elements of the set constructor.

Other examples of contexts where generic sets are permitted occur with comparisons between sets and in passing sets to formal procedures (i.e. procedures that are parameters of the current procedure).

A Pascal compiler implementor has only two choices, unless he changes the language by, perhaps, following the lead of EUCLID. Either the implementor must retain some default size of bit vector that will be used for set constructors (in cases where the compiler cannot determine the element range) or else he must use a different implementation for sets. Possibly a sophisticated compiler might support two different implementations for sets, using a general implementation such as linked-lists for only those situations where the compiler cannot derive the limits.

Most analysis and optimization algorithms can operate on several different program levels. The four levels often cited in the literature (e.g. see Refs 12 and 13) are: (1) the expression level; (2) the basic block level; (3) the procedure level and (4) the program level.

We have presented our analysis algorithm as one that operates on the lowest level only, a single expression (or assignment). Our choice was made so as to simplify the explanation and also because the most worthwhile optimizations should involve the use of temporary set variables. Such temporary sets can only occur within the code generated for a set expression.

It is possible to extend the algorithm to higher program levels. We will not give the necessary extensions in great detail because we are not convinced of the usefulness of analysis performed at the higher levels. We will simply sketch out the algorithm as adapted to basic blocks (and therefore also to DAG-structured expressions). We leave it to the reader to fill in any missing details.

A basic block is simply a sequence of program statements which has no internal control flow (other than the implicit sequencing from one statement to the next). At the code triple level, a basic block is a sequence of triples where only the last triple is permitted to effect a control transfer (such as a conditional branch). Our analysis algorithm will process all the code triples twice, once in a downwards pass and then once in an upwards pass.

There are two preliminaries. We should first perform a use-definition analysis\(^{13}\) of the basic block. This analysis determines all the places within the basic block where a value is used after it has been created (or defined). We also must associate separate range sets with every occurrence of each set variable or temporary in the basic block. These range sets should be initialized in the same way as before.

When the downward pass encounters a triple performing a set operation, with the form \(\langle T := L \ op \ R \rangle\), the actions are as follows. If L represents a variable or temporary that has been previously defined (i.e. assigned to) in this basic block, then there is a range set associated with that definition. Let us call this set Range \((L_o)\). We use Range \((L_o)\) to restrict the range set associated with the current use of L by performing the intersection

\[
\text{Range} (L) \rightarrow \text{Range} (L) \times \text{Range} (L_o).
\]

A similar action should be performed for R, if required. Then, the rules given in Table 1 may be applied.

The actions needed for the upward pass are different. When a set triple of the form \(\langle T := L \ op \ R \rangle\) is processed, we must propagate range information backwards into Range \((T)\). The value of T that is computed by this triple is used in one or more subsequent triples. Each of these uses has an associated range set; let us denote these sets by Range \((\langle T_i \rangle)\), Range \((\langle T_j \rangle)\), etc. The propagation is performed by the operation

\[
\text{Range} (T) \leftarrow \text{Range} (T) \times (\text{Range} (\langle T_i \rangle) + \text{Range} (\langle T_j \rangle) + \ldots)
\]

After this, the rules of Table 1 may be applied again to this triple.

We contend that set analysis is unlikely to produce very interesting or useful results when applied at program levels higher than the expression level. Assuming that the programmer is not writing redundant code, the kind of information that is generated is limited. All that this information can show is that some set variables do not get used to their full potential. That is, at some points in a program a set variable may not hold the full range of elements that are implied by the variable’s declaration. Sometimes, this information may be used to optimize a subsequent set operation, but we feel that this will occur rarely.

The set analysis algorithm is quite unusual in that it requires both forward and backward propagation of information. The standard algorithms of the literature perform their information propagation in one direction only. Some algorithms, such as available expression analysis, propagate forwards and other algorithms, such as live variable analysis, propagate backwards.\(^{12, 13}\) There is a simple intuitive reason why our algorithm must propagate in both directions. It is because we deal with two kinds of information which, when combined, yield tight ranges on set elements. In the forwards direction, we propagate information about which element values can possibly occur in the result of a set operation. (Since this information is derived from similar information about the operands of the operation, the propagation is forward.) In the backwards direction, we propagate information about which element values are useful. Knowledge that elements outside a certain range can have no effect on the final result of an expression can be propagated from an operation’s result to its operands.

As a final topic for discussion, we consider interactions between set analysis and other compiler analysis and optimization algorithms. The order that different optimizations are applied can be quite important. We recommend that the set analysis algorithm be applied before common subexpression elimination.\(^{12}\) One trivial reason is that common subexpression elimination can destroy the normal tree structure of expressions. If two occurrences of the same subexpression are found in the same expression, combining them transforms the expression tree into a DAG. Our original algorithm does not operate correctly on a DAG and requires the extensions outlined earlier. A second, and more important, reason is that common subexpression elimination may actually worsen the code generated for set expressions. An example of this is illustrated in Fig. 7. However, if the set analysis is
\[ S := ((A + B) \cdot C + D) - (A + B) \cdot E; \]

Figure 7. (Note: The subexpression \( A + B \) should not be recognized as being common. If it is, it will require a logical OR of two 100-bit vectors. If it is not, both occurrences separately require a logical OR of only 10 bits each.)

performed first and if operations that manipulate different ranges of elements are not considered to be identical, the two subexpressions are no longer found to be common.

## CONCLUSIONS

The problem of determining minimum (finite) storage allocations for Pascal sets implemented as bit vectors is known to be unsolvable. For most practical situations, however, the goal can be realized without imposing restrictions on set element ranges. An algorithm for performing the necessary analysis was presented. The first version of the algorithm required two passes through the intermediate code of the Pascal program. A second version of the algorithm that is more easily implementable in a compiler was subsequently presented; however, this version of the algorithm sometimes requires more than two passes.

## REFERENCES


Received May 1981

© Heyden & Son Ltd, 1982