SLAC PUB-1628
August 1975

AN IMPLEMENTATION OF PASCAL SETS ON THE IBM 360*

Loretta R. Guarino**

Stanford Linear Accelerator Center
Stanford, California

August 1975

ABSTRACT

An implementation of the powerset data type of the programming language PASCAL in a compiler for the IBM System 360/370, is described. Design decisions such as the choice of a bitstring representation and restrictions on the size of sets are discussed. The solutions to problems arising from the one-pass organization of the compiler and the System/360 machine architecture are presented.

(Submitted to Communications of the ACM)

*Work supported by U.S. Energy Research and Development Administration under contract E(04-3)515

**Current address is Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213

INDEX TERMS

sets
powersets
PASCAL
bitstring
data structures
compiler design

C.R. Categories: 4.12, 4.22

1. Introduction

The programming language PASCAL[6] was designed with a rich set of data and program structuring facilities. Since the design goals of PASCAL included portability, implementers are given freedom and flexibility in implementing language features within the rigorous definition of the language.[3]

This paper describes the implementation of sets, a PASCAL data type, in a compiler for the IBM 360 and 370 computers. The flexibility in the language definition allows the implementation to reflect System/360 machine characteristics, although certain implementation decisions are directed by the language definition. Specifically, the choice of a bit-string representation for sets is influenced by the set operations included in the language.

The compiler for which sets were implemented is described in Section 2, and PASCAL sets are defined in Section 3. The actual implementation decisions are discussed in Section 4: the choice of data structure, implementation restrictions on sets, and the implementation of set operators. Section 5 discusses problems resulting from these decisions and presents possible solutions to those problems. Steps taken to improve the efficiency of the implementation are described in Section 6, and other optimization suggestions for sets are included.

2. Compiler

The compiler for which sets were implemented is a modification of the January 1972 version of the PASCAL/6600 compiler. It was bootstrapped to System/360 by graduate students in the Stanford University Computer Science Department.[5] Some of the features of the full language were omitted since they were not necessary for the completion of the bootstrap.

The powerset data type is one of the features which was omitted from the bootstrapped compiler. The implementation of powersets, described in this paper, is part of a project to extend the compiler to the full language.

The compiler is written in PASCAL. It is a one-pass compiler, and uses recursive descent. These two features of the compiler caused many of the difficulties in implementing sets, but they also make the compiler easy to understand and modify. The basic structure of the compiler is described by Wirth,[7] and the modifications made during the bootstrap are described by Russell and Sue.[5]

3. PASCAL Set Definition

A PASCAL set data type is defined by Wirth[3] as

$$\underline{\text{type}}\ T = \underline{\text{set}}\ \underline{\text{of}}\ T_o$$

where T_o is a subrange of integers, a subrange of characters, or a user-defined scalar type. PASCAL allows the number of elements in the base type T_o to be limited to some implementation-dependent maximum. Sets of more complex items, such as sets of real numbers and sets of sets, are not allowed.

Sets are built from their elements by means of set constructors which enumerate the set elements. For example, [0,6, 10..20] and ['A'..'Z'] are set constructors. The empty set is denoted by [ ].

The primitive binary operations on sets are union, intersection, and difference, represented by +, *, and - respectively. The relational operators applicable to sets are equality and inequality (=, < >), set inclusion (< =, > =), and a test for membership (e.g., i in A, where i is an element of the base type of A).

As an example and for purposes of future reference, several set types

and variables are defined here.

```
type  ALPHANUM = set of CHAR;
      MONTHS   = set of (JAN,FEB,MARCH,APRIL,MAY,JUNE,JULY,
                         AUGUST,SEPT,OCT,NOV,DEC).
      SUMMER   = set of MAY .. SEPT;
      S        = set of 0 .. 100;

var   LETTERS,DIGITS : ALPHANUM;
      A,B,C : S;
      VACATION1, VACATION2 : SUMMER;
      SCHOOLYEAR : MONTHS;
      FARSET : set of 2000 .. 2003;
      NEGSET : -50 .. 50;

LETTERS : =    ['A' .. 'Z'];
DIGITS : =     ['0' .. '9'];
VACATION1 : =  [MAY .. AUGUST];
VACATION2 : =  [JUNE .. SEPT];
```

4. Implementation

4.1 Choice of Bitstring Representation

Aho, Hopcroft and Ullman[1] and Low[4] present a variety of data structures for representing sets. These include hash tables, binary search trees, 2-3 trees, linked lists and bitstrings. An appropriate choice of powerset representation depends upon the set operations to be performed and on assumptions about the size and characteristics of the underlying set. Because of the operations defined for the language and the restriction of the underlying set to scalar types and integer and character subranges, the bitstring representation was chosen for PASCAL sets.

These restrictions on the base type of a set permit a simple mapping between the elements of a set and some non-negative range of integers. For instance, in the scalar type

COLOR = (RED,BLUE,YELLOW)

RED would be represented by 0, BLUE by 1, and YELLOW by 2. Hence, the presence of an element in a powerset is represented by a 1 in the corresponding bit position of the bitstring. It is not necessary to retain any further information about the type or element at run-time. Hoare[2] discusses the advantages of this representation at length, and Wirth adopted it in the CDC 6000 PASCAL compiler.[7]

The choice of the bitstring representation seems to have influenced the choice of set operations defined for PASCAL. The PASCAL primitives union, intersection, set difference and test for membership can be implemented by one or two machine instructions with the bitstring representation. Some or all of these operations would be more awkward with the alternate data structures. With lists or trees, testing for membership involves searching. Taking the union of two sets represented as hash tables would require hashing the elements of one table into the other. However, operations such as iterating through the elements of a set, or the FIND operation of Aho, Hopcroft and Ullman,[1] which might be more appropriate for the other data structures, have been omitted from the set of PASCAL primitives.

4.2 Size Restrictions

Wirth[7] suggests an efficient implementation of powersets as a bitstring by restricting the number of elements in the underlying base set to the number of bits in a computer word. This may be

tolerable on the CDC 6000 series family of computers with its 60-bit word size. Even then, the 60 element limitation precludes powersets of the entire 64 character character set, which would be very useful for lexical scanners. The 32-bit word of the IBM System/360 series of computers is too small to make sets very useful if the set representation is limited to one word.

Using a 64-bit doubleword instead of a single word would approximate the 60-bit sets of the CDC machine. However, this inherits the limitations of small size while losing the efficiency advantages of fitting a complete set into a register. Furthermore, character sets would require a special mapping, since the EBCDIC character code ranges from 0 to 255 instead of BCD's range from 0 to 63.

Since the efficiency of single word sets is to be lost, it seems desirable to ease the restriction on set size as much as possible. The 360 storage-to-storage instructions allow operations, such as logical AND and logical OR on bitstrings in memory having lengths up to 256 8-bit bytes (2048 bits), which can be used to implement intersection and union. In accordance with Hoare's recommendation[2] that basic operations be executable by single machine instructions, 2048 elements is the upper limit on set size for this implementation.

The maximum number of elements in a set type is known at the time of type declaration. All set variables of a given type are allocated the same amount of space. Set variables of different types may have different sizes since set operations are allowed only on sets of the same type.

4.3 Implementation of Operations and Set Construction

In the two-address instruction format of a System/360 storage-to-storage instruction, the first operand address is also used as the address of the result. Hence, temporary work areas must be used to avoid destroying the operand. Since operands must be of the same type, they will always be of the same length, and no padding or alignment is needed.

The union operation A + B is implemented by moving the first operand A to a temporary location and performing a logical OR of the bitstrings. The intersection operation A * B is implemented similarly, executing a logical AND instead of a logical OR. The set difference A - B is implemented by moving the first operand A to a temporary location, performing an EXCLUSIVE-OR between the temporary bitstring and B, then performing a locigal AND with that result and A.

The System/360 Compare Logical Character instruction compares two bitstrings in memory and indicates whether or not they are identical. This is used for testing equality and inequality. Set inclusion, A < = B, tests whether A is a subset of B. This is implemented by moving A to a temporary location, performing a logical AND with B, and comparing the result with A. If they are identical, then A is a subset of B.

Inefficiencies of the PASCAL/360 code generator in the handling of logical conditions affect the implementation of the test for membership, I in A. The test is presently accomplished by inserting the byte containing the Ith bit of A into a register, shifting that bit into the low order position, and masking out the rest of the

word. When the compiler's logical condition handling is improved, it will be possible to use the Test Under Mask instruction to test for the presence of the bit in a single instruction.

A set is constructed by setting the bitstring to zero and inserting each element into the bitstring. If the value of the element is known at compile time, a single OR-IMMEDIATE instruction will insert the element into the bitstring. If the element is a variable, a byte address is calculated and then the proper bit mask is inserted into the OR-IMMEDIATE instruction at run-time by means of an EXECUTE instruction.

5. Problems of Generalization

Originally, we hoped to make sets even more general by permitting a set type to span any range containing no more than 2048 elements. This would allow sets containing negative as well as positive elements. Hence, the set type

XYZ = set of -1000 .. 1000

would be allowable, since the base type contains only 2001 elements. However, the one-pass, recursive descent organization of the compiler forced limitations on the handling of sets.

When the compiler encounters a set constructor, code is generated to construct the set in a temporary location. If all sets were of a standard size, a standard amount of space could be allocated for the temporary set. However, since different set types require different amounts of space, information about the set type is needed immediately. Unfortunately, the PASCAL definition does not always provide this information. Even the type of the elements is not sufficient to determine the size of the set, since the base type of the powerset may be a subrange of the element type.

For instance, if the set constructor [JULY] is encountered, it could be a set of type MONTH or of type SUMMER. These types require different amounts of space, but there is no indication from the set constructor itself as to the type of the set.

This lack of type information causes other problems in constructing sets. Even if a default of 256 bytes (the maximum set size possible) is allotted for the temporary set, without set type information there is no way to determine the smallest element of the set type. For example, the set [2,6] might be assigned to the variables A or NEGSET as defined in Section 3. The set would have different representations in these two instances. If the set were assigned to A, the element 2 would be the third bit in the bitstring, but if it were assigned to NEGSET, the element 2 would be the fifty-third bit.

The problem illustrated by the above example is handled by allocating space and constructing sets as if the first element were either the integer 0 or the first element of the scalar base type. For instance, the set constructor [2,6] would be constructed as if it were of type "set of 0 .. 2047", and [JULY] as if it were of type MONTH. This restricts sets to integer subrange within the range [0,2047] and to scalar types of no more than 2048 elements. It also implies that small sets require an unexpectedly large amount of space if their elements occur in the upper end of this range. Thus, the set variable FARSET, defined in Section 3, requires 2004 bits rather than the 4 bits which one might expect.

There are some advantages to this representation. Despite the potential waste of space which this implementation requires, reasonably large sets are supported while staying within the PASCAL syntax. The

unnecessary additional length of some sets will slow down some set operations slightly. However, this overhead is considerably smaller than that required for a more general scheme which minimizes the bit-string length by maintaining a header containing the set length and its smallest element with the set representation at run-time. Such a scheme would require adjustments and recalculations of these items with each set operation, since this information may not be known until run-time.

The problems resulting from lack of type information could be solved easily if the compiler were not one-pass. An initial pass could deduce the necessary type information for the set constructor from its context. It might even determine that a set had no specific type and could be constructed using as little space as necessary. One such example is the expression

```
if 5 in [2,3, 10],
```

which always evaluates to FALSE. A one-pass compiler might realize that it could evaluate such expressions at compile time and optimize other expressions involving set constructors. For instance, it might compile A + [1,10] by issuing code to insert the elements 1 and 10 into A instead of constructing the set [1,10] and taking the union of the two sets. Low[4] mentions other such optimizations. However, the one-pass constraint makes these optimizations difficult to implement.

A change to the PASCAL set syntax would also solve these problems neatly. Hoare's set examples[2] show type names preceding the set constructors. This procedure forces the user to state specifically what type of set he is using. In particular, it distinguishes between what are now lexically identical sets of two different types with overlapping ranges. Hence, the set [JULY] would become either MONTHS[JULY] or SUMMER[JULY], depending upon its use.

With this change, the necessary set type information would be available before the construction of the set is started. The size of the set and the value of its smallest element would be available from the symbol table, and a bitstring representation minimizing space use is easily implemented. This solution to the set representation problem was not adopted because it does not adhere to the PASCAL language definition. It was deemed more important to remain consistent with the standard language than to implement the most general and efficient representation of sets in a compiler which must already sacrifice some efficiency to its one-pass structure.

6. Optimization

Two global variables, SETTYPTR and SETTEMP, were introduced in the compiler to optimize space usage. SETTEMP is a Boolean variable, and SETTYPTR is a pointer into the symbol table, used to reference a symbol table type entry.

SETTYPTR retains set type information gathered from the context for a statement or expression. This information can then be used if a set constructor is encountered. Since type compatibility is required, the size of the set to be constructed can be determined from the symbol table entry referenced by SETTYPTR. This can produce considerable savings in space since most set types will be substantially smaller than the default of 256 bytes.

For example, in the statement

```
VACATION1   = [JUNE,JULY]
```

the set constructed by [JUNE,JULY] must be of the same type as VACATION1. The recursive descent technique hides this type of information until after the set has been constructed and the actual assignment is to be

processed. Whenever the left-hand side of an assignment statement is a set, SETTYPTR is assigned the symbol table entry for the type of set. This type information can then be used in compiling the expression on the right-hand side of the statement.

The type of a set constructor can often be determined from the context when the set constructor is an operand of a set operation. For instance, in the expression

```
A + [0,1]
```

the set [0,1] must be of the same type as A. However, care is needed in determining set types from expressions. In the expression

```
IF ((SCHOOLYEAR = [SEPT..DEC]) v (I in [1..100]))
```

it would be disastrous to assume that [1..100] is of the same type as SCHOOLYEAR, as might happen with a casual approach to setting the global variable SETTYPTR. SETTYPTR must always be reset to the null pointer when its range of validity terminates. Thus, in the above example, SETTYPTR must be set to NIL after the Boolean expression (SCHOOLYEAR = [SEPT..DEC]) has been compiled.

The Boolean variable SETTEMP indicates whether or not a temporary work areas has been used in the set expression being compiled. Because of the two-address instruction format of System/360 storage-to-storage instructions, a temporary area must be allocated for set expressions to avoid destroying the value of the operand variables. However, once an intermediate result has been generated in this temporary area, additional set operations can use the same work area.

For example, consider the statement

```
C: = A * B + C ;
```

the set A is moved to a temporary work area so that the intersection of

A and B can be taken without destroying A or B. The result is left in the temporary area. The standard implementation of the union operation would then move this intermediate result to another work area before taking its union with C. However, the intermediate result will not be needed again so there is no need to move it to avoid destroying it. SETTEMP would indicate that the first operand need not be saved after the set operation. The union of the intermediate result and C would be left in the same temporary area, and this result is assigned by moving the contents of the temporary area to the storage area assigned to C.

SETTEMP is set to TRUE whenever a temporary set variable is generated, and is reset to FALSE when an expression has been completed. As with SETTYPTR, care must be taken that this global variable is rese after leaving its range of validity. When large sets are used in complex expressions, however, substantial savings can be made in time and space through the use of SETTEMP.

Since the compiler is one-pass, global optimizations cannot even be attempted. This lack of global optimization leads to some extremely inefficient situations. For example, consider the loop

```
WHILE  CH  IN  ['A' .. 'Z']  DO
    BEGIN  S1 ; S2 ; .. SN  END;
```

The compiler produces code to construct the set ['A' .. 'Z'] in the middle of the loop exit test. Hence each time through the loop, the set is reconstructed, requiring approximately 40 instructions. In this instance, the set should be constructed outside the loop and stored in a variable.

Another desirable global optimization would be to recognize constant sets and construct them at compile time. PASCAL presently has a facility

for declaring constant scalar types. Extending this facility to sets (and possibly other structured types) would assist the compiler in this optimization.

7. Future Work

Unit sets [2,p.124] containing only one element should be recognized and handled separately in most cases. For instance, in the expression

A + [13]

great savings of time and space could be accomplished by simply inserting the element 13 into the set A instead of constructing a set containing only the element 13 and forming the union of this set and the set A. The sets could be recognized and handled separately without too much difficulty, even within the one-pass recursive descent format of the compiler.

If a set type contains no more than 32 elements, the efficient word-oriented implementation suggested by Wirth can be used. These sets will fit into a System/360 machine register, and the faster register-to-register instructions would be used.

For sparse sets with exceedingly large underlying base types, the bitstring representation is impractical. An alternate representation for such sets could be provided if PASCAL allowed a SPARSE attribute for set variables, similar to the PACKED attribute for records. Explicit conversion routines similar to PACK and UNPACK or implicit conversion would be necessary for set operations between SPARSE sets and regular sets of the same type.

8. Conclusion

The PASCAL language definition is flexible enough to allow the implementor to determine an efficient implementation of a language feature such as sets. However, Wirth may have compromised the portability of

PASCAL by reflecting the CDC 6000 series machine structure in the PASCAL language definition. The modifications to the language definition suggested in this paper were not included in the original definition because the CDC implementation did not require them, although they are necessary for a more general set implementation.

This implementation of sets also points out the limitations of a one-pass compiler. Most of the problems encountered are aggravated by the need to generate code before sufficient information about a set is available. If efficiency is an extremely important design goal, an initial pass is necessary to gather the necessary type information and attempt some global optimization.

ACKNOWLEDGMENTS

I would like to thank Professor Forest Baskett, V. Bruce Hunt, and Charles T. Zahn, Jr., for their stimulating discussions and valuable suggestions. In addition, I would like to thank Jon Bentley and John Ehrman for their encouragement and helpful comments on this manuscript.

REFERENCES

(1) Aho, A.V., Hopcroft, J.E. and Ullman, J.D. The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass. 1974.

(2) Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. Structured Programming, Academic Press, London and New York, 1972.

(3) Hoare, C.A.R., and Wirth, N. An asiomatic definition of the programming language PASCAL, Acta Informatica, 2 (1973), 335-355.

(4) Low, J.R. Automatic coding: choice of data structures, Stanford Computer Science Report CS-452, Stanford (August 1974).

(5) Russell, D.L, and Sue, J.Y. Stanford PASCAL 1360 Implementation Guide, Technical Memo 89, Stanford Linear Accelerator Center, Stanford (November 1974).

(6) Wirth, N. The programming language PASCAL, Acta Informatica, 1, (1971), 35-63.

(7) Wirth, N. The design of a PASCAL compiler, Software-Practice and Experience, 1, (1971), 309-333.

(8) Wirth, N. The programming language PASCAL (revised report). Berichte der Fachgruppe Computer-Wissenschaften, Eidgenossische Technische Tlochschule, Zurich (December 1973).