SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 12, 879–882 (1982)

# Compilation of the Pascal Case Statement

JOHN L. HENNESSY AND NOAH MENDELSOHN

*Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 U.S.A.*

**SUMMARY**

**Pascal case statements can be compiled using a variety of methods, including comparison trees and branch tables. The scheme discussed here combines the two techniques to allow comparison trees with entries that are branch tables. The use of a combination of the two techniques is shown to adapt well to certain instances of case statements. Extensions to the standard case statement also require such a scheme to obtain an efficient implementation.**

KEY WORDS Case statements Compiler design Pascal

## INTRODUCTION

In an excellent paper,[1] Sale discusses the implementation of Pascal case statements. Our work on the Pascal★ compiler[2] led to similar analysis of the problem, and we agree that some combination of comparison trees and branch tables is a most effective implementation of the case statement. We have carried this solution somewhat further: when appropriate, our compiler uses a combination of the two techniques within a single case statement. Handling of large case statements whose labels exhibit clustering is facilitated, while simpler forms are a natural special case.

Consider the following examples, which the Tasmania and Pascal★ compilers handle in similar fashion.

```
case I of
  3,5,4: stmt1;
      6: stmt2;
      7: stmt3
end;
```

This code compiles into a single branch table for selector values 3 to 7.

The following code has a much less dense distribution of case labels.

```
case J of
  3,5,4: stmt1;
   100: stmt2;
   200: stmt3
end;
```

A branch table for the case statement on $J$ would require 198 entries, an unnecessary waste, so for the reasons discussed in Sale's article, we adopt a tree of comparisons. Recognizing the continuous subrange 3,4,5 in the first case-list, our compiler

0038–0644/82/090879–04$01.00

*Received 30 December 1981*
*Revised 26 May 1982*

produces code of the form:

```
        if j≤100 goto L1;
        if j = 200 then stmt3
        else error;
L1:     if j>5 then goto L2;
        if j≥3 then stmt1
        else error
L2:     if j = 100 then stmt2
        else error
```

The primary distinction of our algorithm is its ability to gracefully handle case statements with clustered labels:

```
case K of
        1: Stmt1;
        2: Stmt2;
        3: Stmt3;
        4: Stmt4;
        5: Stmt5;
        6: Stmt6;
        7: Stmt7;
        8: Stmt8;
     1001: Stmt9;
     1002: Stmt10;
     1003: Stmt11;
     1004: Stmt12;
     2001: Stmt13;
     2002: Stmt14;
     2003: Stmt15;
     2004: Stmt16
end;
```

A case statement of this sort is quite reasonable from a programmer's perspective, but presents a difficult tradeoff to an algorithm choosing between branch tables and comparison trees. The tree approach involves approximately 23 emitted value comparisons, with a typical path length of at least 4 comparisons for any given value of the selector $K$. The case-labels span the range 1..2004, and cannot be efficiently handled by an algorithm restricted to use of a single branch table.

Additionally, Pascal★, like many other Pascal extensions, allows an extended form of the case statement. Constant subranges are allowed for case labels; an **otherwise** clause, executed if the selector does not match any case label, may be appended to the case statement. This extended form of the case statement offers the programmer significant additional flexibility; however, it also increases the likelihood that sparse case statements may arise.

## IMPLEMENTATION TECHNIQUE

We use a simple heuristic, described below, to detect clustering of the case-labels and to split up large branch tables when appropriate. In the above example, three such

clusters are identified: 1..8, 1001..1004 and 2001..2004. Using a comparison tree to choose only among the three clusters, we emit 2 comparisons, with an average pathlength of only 1·5. Three separate branch tables, with a total of 16 entries, are used for the clusters. The object code produced is similar to the following:

```
      if K≤1004 then goto L1
      else Branchtable(2001..2004);
L1:   if K≤8 then Branchtable(1..8)
      else Branchtable(1001..1004)
```

(*Branchtable* implies use of a simple computed jump with bounds checking, a feature of our intermediate code.)

The function of the heuristic algorithm is to split a sorted list of case-labels (containing single constants and constant subranges) into a series of clustered label sets, each of which should be implemented as a branch table. The algorithm presumes that a branch table is the fastest implementation for any case statement, and should be used unless prohibitively large.

The algorithm starts with a series of sorted case-labels, initially grouped together for inclusion in a single branch table. The algorithm then searches for cost-effective places at which to split the table into pieces; each such split adds a node to the comparison tree. In the example above, splits are located above case-labels 8 and 1004.

Choosing appropriate places to split a cluster is a space–time tradeoff. The incremental cost in pathlength of a split is approximately proportional to $1/n$, where $n$ is the number of nodes in the tree. Thus, the first split of a branch table into two nodes implies a runtime test that is executed each time through the case statement. Addition of a third node adds a test that is only executed half of the time (assuming even distribution among legal selector values), and so on. Additional splits become increasingly attractive as the tree grows large.

By introducing a split, the amount of branch table space required is decreased, but the amount of code needed to implement the branch tree is increased. Initially, the cluster requires branch table space equal to the entire range of values in the cluster. After the split the branch table space is equal to the sum of the sizes for each new cluster plus some overhead to create the branch tree consisting of the two tables. Thus, in the previous example, when considering the split after case-label 8, we see that the branch table initially contains 2004 entries (1..2004); if a split occurs after case-label 8, the two branch tables will contain 8 (1..8) and 1004 (1001..2004) elements. This gives a total size of 1112 elements plus some overhead. To compute the size of a split entry, the compiler sums the resulting branch tables and adds an estimate of the code size associated with the new branch tree element.

A compiler constant specifies the relative importance of space and efficiency in the object code. Expressed as the desired ratio of instructions to be executed vs. instructions and branch locations to be emitted, this parameter determines the viability of each possible split. For a sparse case statement, the actual ratio before any splitting will be very low, since the number of branch table elements is high. Thus, a split will be advantageous if it increases the ratio. The algorithm terminates when there are no further splits that can increase the actual ratio without exceeding the desired ratio.

Once the splitting has been done, it is straightforward to generate a balanced comparison tree. We use an array to represent the list of clusters; once the splitting is

complete, the root node for the comparison tree is the median element in the sorted list of clusters. Roots for the left and right subtrees correspond to the median elements of the left and right halves of the array. Using this representation, the tree may be traversed in preorder format to generate the appropriate tests and branches as described in Reference 1.

## CONCLUDING REMARKS

The algorithm described herein was added to the Pascal★ compiler during the summer of 1980, and has seen regular use since then. As expected, we have found it to be effective in compiling a wide variety of case statements both in standard Pascal and using the Pascal★ extensions. In comparison to the Tasmania compiler, we offer slightly better treatment of certain case statements, at the expense of some compile time complexity. Programmers typically avoid constructs that are poorly handled by compilers, so the elaborate case statements that benefit from our approach are rare in existing Pascal programs. We hope that the enhanced facilities of the Pascal★ and Tasmania compilers will provide programmers greater freedom in the future.

### ACKNOWLEDGEMENT

This research was partially supported by the U.S. Office of Naval Research, under a contract to the University of California Lawrence Livermore Laboratory, LLL Contract #9628303.

### REFERENCES

1. A. Sale, 'The implementation of case statements in Pascal', *Software—Practice and Experience*, **11**, 929–942 (1981).
2. J. L. Hennessy, 'Pascal★: a Pascal based systems programming language', *Tech. report 174*, Computer Systems Laboratory, Stanford University, June 1980.