GENERALIZED SET COMPARISON

A. L. Furtado

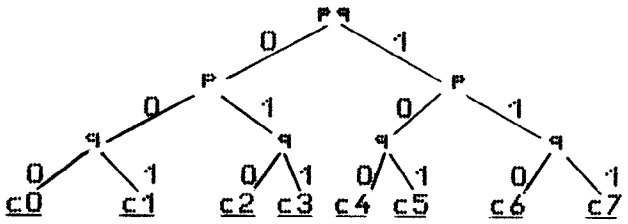
Departamento de Informática
Pontifícia Universidade Católica do R. J.
22453 Rio de Janeiro
Brasil

An extremely simple method for set comparison is proposed. The method determines what relationship holds between two sets. This contrasts with the usual set comparison operators, which merely check a given relationship.

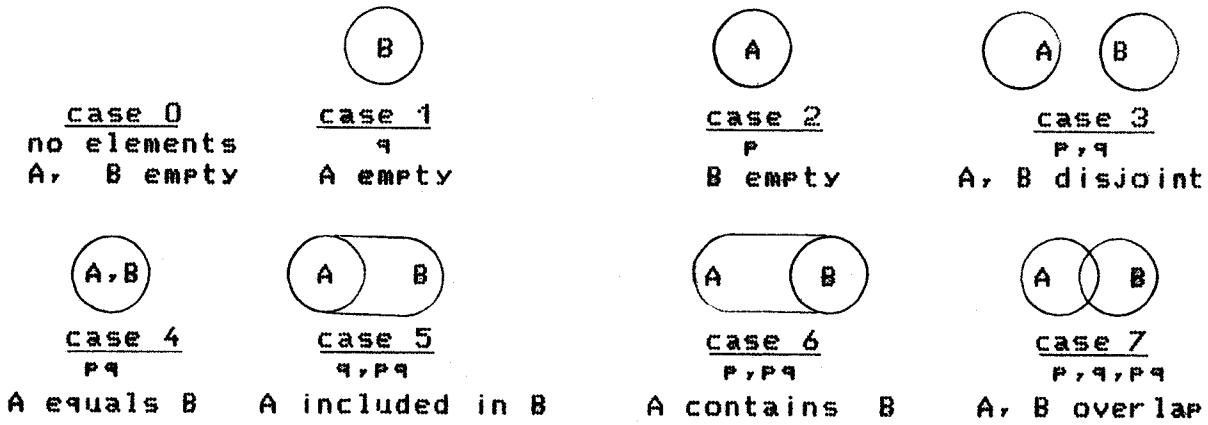
To compare two sets A and B, we consider the following classes of elements:

- class e - elements belonging only to A
- class q - elements belonging only to B
- class eq - elements belonging to both A and B

Based on these classes, we can characterize all the possible outcomes of the comparison, which correspond to the combinations of 0, 1, 2 or 3 of the classes. A decision tree [Be] displays the combinations; an edge leaving the pq, p and q nodes is labelled "1" if at least one element occurs in the respective class, and "0" otherwise.



The 8 cases c0, c1,...,c7 are further characterized, in terms of their Venn diagrams [Ko].



The 8 cases constitute an adequate choice for "primitive" set relationships. By construction, they are mutually exclusive and cover all situations. However, exactly because they have these properties, they are all strict and the "trivial" empty set cases are distinguished. So, when we write under case 5 "A included in B", we assume that both sets are non-empty and that B has at least one element that is not in A. In practice we may need such strict and non-trivial relationships; for instance, the sentence "John can do part of the jobs that Peter does" clearly implies that both sets of jobs are non-empty and that Peter's set is larger than John's.

The sequence of labels along each path of the decision tree can be interpreted as a binary number, corresponding conveniently to the case number. This suggests that we substitute a "1" or a "0" for each member of a three-digit sequence (pq,p,q), depending on the occurrence of at least one element in the respective class.

A PASCAL function [JW] using this method is given below. The function returns a literal belonging to the enumeration

```
type compcase = (emp12,emp1,emp2,disj,eq,incl,cont,overl)
```

to convey the outcome of the comparison. A brief description of how the function works is given afterwards.

```
function SCOMP(A,B: eset): compcase;
  var X,Y     : ordset;
      p,q,pq  : 0..1;
      c,i     : 0..7;
      r       : compcase;
  begin
    X := sort(A); Y := sort(B);
    p := 0; q := 0; pq := 0;
    while not null(X) and not null(Y) do
      if head(X) = head(Y) then
        begin pq := 1; X := tail(X); Y := tail(Y) end
      else if head(X) < head(Y) then
             begin p := 1; X := tail(X) end
           else begin q := 1; Y := tail(Y) end;
    if not null(X) then p := 1;
    if not null(Y) then q := 1;
    c := q + 2*p + 4*pq;
    r := emp12;
    for i := 1 to c do
      r := succ(r);
    SCOMP := r
  end;
```

The two sets are first ordered (using some O(n log(n)) method [AHU]). Variables p, q and pq are initialized to zero. Then, the sets are scanned in a merge-wise fashion (an O(n + m) process) to detect elements in the three classes.

As long as there are elements in both sets, we compare the two

leading elements. If they are equal, pq is updated. Otherwise we can conclude that the smaller element cannot be found in the other set - recall that the sets were ordered - and, accordingly, either p or q is updated; the same is done if the end of one of the sets is reached while the other still has unscanned elements.

At the end, a decimal number is obtained from (pq,p,q) and then used to find the corresponding literal in the enumeration.

The invoking program can naturally use the result of SCOMP in a case statement:

```
case SCOMP(A,B) of
  emp12: ... ;
  emp1 : ... ;
  ............
end
```

PASCAL allows to group together several cases. If we wish to consider the usual "C" comparison, thus accepting non-strict inclusion (set equality) as well as the trivial empty set situations, we can write

```
case SCOMP(A,B) of
  ........................
  emp12,emp1,eq,incl: ... ;
  ........................
end
```

or, if any other situations are irrelevant to our application,

```
if SCOMP(A,B) in [emp12,emp1,eq,incl] then ...
```

or, finally, by introducing a variable, inclusion, of a powerset type storing the above set of cases

```
if SCOMP(A,B) in inclusion then ...
```

Other set representations can lead to other implementations. For example, if A and B are represented by bit arrays and pairwise bit operations like AND and AND_NOT are available, testing occurrence of elements in the three classes can be done in a particularly efficient way. Also, other kinds of languages [GJ] and equipment suggest some interesting possibilities:

- Extensible languages, like Ada, permitting the introduction of new operators, not simply through function definition, but as a proper syntactical extension, would allow us to write something like

  A ? B

  which would stress the fact that the operator yields the relationship between A an B. Special symbols could be used instead of the literals enumerated before, both as result of

the comparison and as binary comparison operators; a possible choice is, in the same order as the literals were presented: A _ B, A / B, A \ B, A | B, A = B, A < B, A > B, A # B.

- Microcomputer software must strive for economy of space. Even if one decides to have several set comparison operators in a language, it is convenient to practically implement all of them through a single piece of code, as shown here.

- For graphical communication with the user we might display the result of set comparison as, say, pictures of Venn diagrams. In this event we would count in p, q and pq the number of occurrences in each class, thereby being able to keep the proportions in the diagrams.

- Data base languages and applicative languages largely benefit from the introduction of set processing operators. In both cases the ability to handle sets reduces procedurality by avoiding explicit iteration.

Acknowledgements - The author thanks Sergio E. R. de Carvalho and Ewerton A. P. Vieira for helpful suggestions.

References

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman - Data structures and algorithms - Addison-Wesley (1983).

[Be] A. T. Berztiss - Data structures = theory and practice - Academic Press (1971).

[GJ] C. Ghezzi and M. Jazayeri - Programming language concepts - John Wiley (1982).

[JW] K. Jensen and N. Wirth - PASCAL = user manual and report - Springer-Verlag (1975).

[Ko] R. R. Korfhage - Discrete computational structures - Academic Press (1974).