Optimisation of Case Statements

by

Stefan M. Freudenberger

SETL Newsletter #215
Courant Institute of Mathematical Sciences
New York University

15. April 1981

Several significant and straight-forward code optimisations are omitted in the current optimiser. This note explores one such optimisation, and outlines its implementation. This optimisation is case map optimisation, and encompasses the following:

The SETL case statement is currently implemented by creating a (necessarily constant and single-valued) map from case tags to code addresses. Since this map is undeclared, it is currently represented as an unbased map from GENERAL to GENERAL, and thus requires a hashing operation to find the branch address for a given value of the case expression. (It would in fact be easy to give this map an implicit SMAP(GENERAL) LABEL declaration, in which case the Code Generator would detect the existence of case tag duplicates, an error not picked up in the current system.)

This crude approach can be improved significantly in two important cases, namely:

(1) when the case tags form a dense range of positive integers

(2) when the case tags form a subset of a base

More formally, these optimisations amount to determining whether one of the following conditions is met, and if so, to choose the implementation suggested. These conditions are:

(1) If a case map's domain consists of positive integers only, and the size of the domain compares favorably to the maximum element of that domain, then we can represent the case map as a tuple:

```
if forall X in domain CASE_MAP |
        is_integer X and
        1 <= X and X <= C * # domain CASE_MAP then
    /* Represent the case map as a tuple */
    '* The constant C should have a plausible value,
       e.g. HL_EBM (see NL. 189B.)        */
end if;
```

(2a) If the case map is based on a constant base B, then we can first remove all case tags which are not in B since these case choices represent unreachable code:

```
CASE_MAP := { [ X, Y ] in CASE_MAP | X in B };
```

If the resulting case map is very sparse with respect to B, we might still prefer to represent it as an unbased map:

```
if # domain CASE_MAP < # B div C then
    /* Represent the case map as a unbased smap */
else
    /* Represent the case map as a local smap   */
end if;
/* Again, the constant C might be HL_EBM        */
```

(2b) If the case map is based on a base B, but B is not constant, then a REMOTE SMAP(ELMT B) LABEL representation is appropriate for the case map, for the following reasons:

Storage for locally based objects is statically allocated within each base element block, and hence these objects will require storage for elements added to the base at run-time even though we know a priori that a constant map will never be defined at a point added to the base during execution. Also, constant elements of a base are live throughout the execution of a program, and their element indices never change, and so a remotely based constant never needs to be re-allocated during run-time. (The proofs for these claims are trivial.) Thus, whereas the storage required for a locally based constant quantity depends on the dynamic size of its base, the size of a remotely based constant depends only on its initial number of elements.

The suggestions advanced above can be implemented as follows.

Assume first that no global optimisation will be performed. In this case we suggest the following implementation, beginning with the first case discussed above (dense range of integers).

When we are about to allocate the case map constant in the Semantic Pass, and before we call the routine which actually creates this map, we will have collected N pairs of the form

```
[ <case tag value>, <label> ]
```

on the semantic pass stack. Rather then calling the set former routine immediately, we first iterate over these stack entries and

check the <case tag value>'s to determine whether they are all positive integers, and at the same time determine their largest element M. If they all are positive integers, and if M is not too large, then we sort the pairs in ascending order of their first components. After this, we insert the proper number of omegas between them, giving us M entries altogether. Then we replace each pair by its <label>, allocate a tuple instead of a map, generate the form TUPLE(LABEL), and give the case 'map' this representation. When we delete the original case pairs from the stack, we also mark the corresponding symbol table entries as dead.

In the second (based) case described above, we proceed as follows:

In the Semantic Pass, when we encounter the <exp> in a case statement, we check to see if it has an ELMT B representation, and if so, declare the case map to be a LOCAL SMAP(ELMT B) LABEL (if the base B is constant and local to the current scope) or a REMOTE SMAP(ELMT B) LABEL. (In all remaining cases, we declare the map to be a SMAP(GENERAL) LABEL.)

The Code Generator can mark based case maps during the preliminary fixup phase, and then take special action when initialising the run-time representation of the case map. There are two cases where such special attention is necessary, both to produce intelligent warning messages and to allocate correct constants. The first case arises if a tag value present in the CASE statement cannot be converted to the element mode declared for the case map's base. In this situation, the case labelled by this tag can never be reached during execution (this follows from the mode conflict between the case expression and the tag value) and thus represents unreachable code. The second case arises if the base B appearing in the declaration for the CASE variable is constant, but some case tag value is not an element of B. As before, this case can never be reached. In both situations, the proper action to take is to print a warning message, and to eliminate the offending case tag from the case map. This eliminates error messages which would otherwise be issued by the run-time conversion routines, which would be confusing to inexperienced SETL users.

To implement case optimisation in our global optimiser, we can modify the pre-pass of the Automatic Data-Structure Selection phase so that we examine each Q1_CASE instruction for the conditions stated above, and introduce the corresponding declarations as appropriate.