SETL Newsletter # 179

S. C. Liu

Oct. 9, 1976

Dynamic Multiple Member Basings

A declared membership basing x :εs improves the efficiency of locating x in s by keeping within x a pointer to the element which has the same value as x in s. This pointer however does not help in locating x in any other set. To ascertain whether (x in s') requires a standard search or multi-level referencing. Quite often an object appears as an element of several sets. If we insist that member basings be static as domain basings are (e.g., based maps), then the full advantage of basings is not achieved. Consider the following example.

```
define          example:
declare         s1 : base set,
                s2 : base set,
                f1 : smap (εs1) int,
                f2 : smap (εs2) int;
.................
L1 : x = ∋s1;
L2 : s2 with x;
L3 : f2(x) = f1(x);
.........
end;
```

Whether x is declared as ' εs1' or ' εs2', a redundant locating operation will be required at instruction L3 if no extra code is inserted; however since L1 and L2 provide pointers to s1 and s2 respectively, no locating should actually be required to reach f1(x) and f2(x) at L3.

It is important to observe that member basing pointers can be obtained as 'fringe benefits' from several operations (∋, with, mapping application, etc.) at little or even no cost. If we can keep these pointers and retrieve them efficiently then the efficiency of a program may be considerably improved. In other words, dynamic multiple member basings are probably more appropriate than static single member basings; although users can force static single basings by explicit declarations. In the following we aim to show how this scheme can be applied to undeclared variables.

Derived Pointers

Usually one or more pointers are involved in the execution of a SETL operation. For example, in the case of 's with x', a pointer pointing to the element x in s is always available after the insertion in s is performed; another pointer to the corresponding element in the base of s is also available if s is a based set. These pointers can be useful subsequently if they are kept. The pointer which points to the element of s (if s itself is a base) or the element of the base of s (if s is based) is most

useful. This pointer will be called the derived pointer of the operation. SETL operations such as with, ε, from, and map application all yield a unique derived pointer which can be determined statically during compilation, if the set which is the argument of the operation is a base or based. Operations with this property will be called pointer-producing operations.

Incarnations

Whenever a pointer-producing operation is performed, the produced pointer will be kept with x. If the original x already has some other basing pointer, a new version of x, which is called an 'incarnation of x', is then created. All incarnations of a variable have the same value, but have different basing pointers. On subsequent uses of x, an appropriate incarnation will be selected to replace x thus avoiding a locating operation; if no appropriate incarnation exists, then locating is still required and a new incarnation will be created as the byproduct of the instruction. All the incarnations of a variable become dead whenever a new value is assigned to the variable; of course, an assignment may also create or recover an incarnation.

If we apply this idea to the preceeding example, two incarnations of x, say x1 and x2, which have the basing 'ε s1' and 'ε s2' respectively, will be created at L1 and L2. The x at L3 is then

replaced by x1 on the right hand side of the assignment, and x2 on the left. The code sequence becomes

```
L1 : x =∋ s1 is x1 ;
L2 : s2 with (x is x2) ;
L3 : f2(x2) = f1(x1) ;
```

Algorithm

Our scheme applies this idea globally by the assistance of standard interval analysis techniques. It is assumed that domain basing information be available. Just before each use of variable x which requires locating operation we assign x the appropriate incarnation. The original use of the variable is then replaced by the newly created incarnation. Finally, the redundant expression elimination algorithm described on O.P II pp.273 can be performed to delete redundant incarnation creations or assignments. Note that all the incarnation assignments associated with a variable become unavailable when the variable is assigned a new value. An incarnation becomes unavailable whenever the corresponding assignment becomes unavailable.

Refinements

In the redundant expression elimination algorithm an expression is available at a given point in a program only if the expression is available along every path leading to this point. However, quite often an incarnation assignment is available along one path but not along another path. In such a case, it might be worth inserting an extra assignment instruction

to create an incarnation which is already available along a path of higher frequency (such as backward branch from the exit of a loop), to make it available also on the paths of lower frequency (such as the initial entry to a loop). If so, the 'meet' function which the elimination algorithm uses can be defined as 'the intersection of all incarnation assignments which are available along the main paths (of highest frequency) to the interval' instead of 'the intersection of all incarnation assignments which are available along every path'. Furthermore, we only need to consider the assignments whose target incarnations are live, if a live-dead analysis is performed beforehand. An incarnation need not be created if it is used only once. Finally, a post-process is required to insert necessary incarnation assignments on every minor path.

Efficiency of incarnation creation

Another possible refinement is to improve the efficiency of incarnation creation by using the pointers which link elements of based sets to the corresponding elements in their base. An incarnation can be created by series of referencings instead of by search if any other equivalent incarnations with proper base exist. (Two incarnations are called equivalent if both are associated with the same variable). This can be achieved statically as follows. A tree is constructed to reflect the basing relation between base sets. Each base set is represented by a node

in this tree. A node n1 appears as an immediate descendant of another node n2 if the base set, say s1, represented by n1 is based on the base set, say s2, represented by n2, i.e., s1 : base set {εs2} . This implies that each element of a node (set) contains a pointer pointing to an element of the immediate ancestor of the node. Then, at an incarnation creation, other available equivalent incarnations are checked to see whether their bases appear in the tree as descendants of the set s which the new incarnation is going to reference; if such exist, the incarnation whose base is a nearest descendant of s can be used to replace the original variable. Then a series of referencings is sufficient to obtain the desired incarnation.