SETL Newsletter 56B October 10, 1971

More SETL Comments Jay Earley

A Set-Former Optimization

This describes a particular kind of optimization which can be done with some kinds of set formers. It has the effect of speeding up the program while sacrificing space. It therefore cannot be applied everywhere, but only to certain set formers where a search over the entire set is too costly.

The basic idea is to maintain explicitly the set which is formed by the set former as the program executes instead of forming it each time the set former is executed. Thus if we have one of the form

$$\{x \in S \mid P(x)\}$$

we check each time an element x is added to (or deleted from) S to see if P(x) is true. If it is we add it to (or delete it from) the set we are maintaining explicitly. The difficulty comes from the fact that the predicate P may also change. In order to take care of this we must specify more explicitly the form of P. Here we specify it for a few simple forms to illustrate the method. It can be extended, but how much further is not clear.

Figures A and B show two forms of the predicate P which we can handle. In each figure the code on the left is that which would be written by the programmer. That on the right is the equivalent code which the optimizer produces for that particular construct. "————" on the right represents the code on the left (so I didn't have to rewrite it). In Figure **A**, A is a variable and in Figure B, C is a constant. In Figure A, T is a function (a data-structure function) introduced by the optimizer which maps each different A onto a set which is explicitly maintained. In Figure B, T is the set which is explicitly maintained; we need only one in this case. Of course, these transformations only apply if the particular forms listed are the only ways in which S and F are changed by the program. If there are other ways, these must either be rewritten in terms of the forms we have given,

or new forms must be added.

We have two additional forms for the cases where we ask for only the number of elements in the set formed. In these cases (Figures C and D) we maintain only a count of the set. These turn out to have wider use than one might think, because we transform certain quantified expressions into ones of the above form. That is,

$\exists\ x \in S \mid P(x)$ becomes $\#\{x \in S \mid P(x)\} > 0$

$\nexists\ x \in S \mid P(x)$ becomes $\#\{x \in S \mid P(x)\} = 0$

We now present an example which illustrates how this method works. The algorithm is taken from Knuth [1]. It does a topological sort on a partially ordered series of inputs. The input is a sequence of pairs of objects. Each pair <B,C> defines an ordering relationship between the objects in the pair so that B < C. The goal is to output the objects in such an order that an object appears before everything that it is less than in the ordering. See [1] for a more detailed explanation. We use the same algorithm as Knuth except that we express it at a much higher level, ignoring most of his implementation details. The algorithm is to pick an object A such that there are no pairs in which it is greater than another object. A is then one of the "least" objects we have. We then output A, delete all pairs containing A, and repeat until the objects are exhausted.

In our algorithm (Figure 1), S is the set of all objects and SP is the set of all pairs. We have again taken some liberties with SETL: The first loop is over a sequence (the sequence of input pairs). The second loop is assumed to have its header recalculated each time through. This is where we pick one of the "least" objects in S.

In Figure 2 we apply a few transformations to put the program in the right form to apply the optimizations. We expand the "union" operation into two "WITH" operations. We apply the "$\nexists$" transformation. And we expand expressions of the form

{<-,A> ∈ S | P(A)} becomes {D ∈ S | P(FIRST(D))} .

In each figure we underline those parts which are changed from the previous one.

In Figures 3 through 5 we apply our set former optimizations to the three set formers in the program. In Figure 3 this introduces a function SUCC which maps each object A onto the set of all pairs <A,B>. In Figure 4, a function COUNT is introduced which maps each object A into an integer which is the number of different pairs <B,A>. Figure 5 introduces the set ZRCOUNT, which is the set of all objects for which COUNT is 0.

Figure 6 contains a cleaned-up version of Figure 5 as it might have been done by a very intelligent optimizer. It is instructive to notice what kinds of things it would have to know and what techniques it would need in order to accomplish this. First it must be able to deal with conditions on variables and domains and ranges of functions. Specifically, it must know that COUNT $\geq$ 0, that SUCC maps S into SP, that FIRST and SECOND map SP into S. If, in addition, it is specified (as we have assumed) that there are no repeats in the INPUT,then it can tell that SP is dead and eliminate it altogether. It must also know that WITH and LESS are inverses of each other so it can recognize the situation

```
IF B THEN A = A WITH C
  :
  :
IF B THEN A = A LESS C
```

where nothing relevant is changed in between.
It must also be able to recognize a situation such as

```
(∀ A ∈ S)
    :
IF A ∈ S THEN ——
    :
END ∀;
```

so that it can eliminate the test in the loop. There are also several cases of common sub-expressions which could be eliminated, but we have left them as they are for readability.

There is one additional optimization which might have been performed, but the conditions under which it can be done are difficult to determine, so we have not put it in. After

the first loop, #S is used, but S is not, except in deleting elements from it. If we had some way of knowing that all these elements were in fact in S before they were deleted (as in this case) then we could simply store #S explicitly and decrement it each time instead of actually deleting the element from S.

Now that we have presented the method, the question remains about about when it should be applied. This can't always be determined from the program, so perhaps there should be a way for the programmer to specify whether he wants it or not. On the other hand, there are cases where the optimizer can determine that the method will at least save time at the expense of extra space. Specifically, if the set S in {x ∈ S | P(x)} is only changed by adding elements to it and P is not changed at all, then the method cannot slow down the program. If elements can be deleted from S or if P can be changed, a more sophisticated analysis would be required. Notice also that the method does not necessarily cost more space. It may be that the introduction of the new sets by the optimizer causes the existing ones to be unnecessary as with SP in the example. Clearly much more needs to be done in determining when this optimization should be applied.

This optimization illustrates one distinction between what I call the "relational" and "access path" levels of data structure description. The original algorithm is at the relational level and the optimized version is at the access path level. It also illustrates another belief of mine -- that optimizers for languages like SETL which have very powerful (relational level) operations like the set former should perform their data structure optimizations in two steps. The first would,as I have shown in this note,be a transformation to the access path level, and the second would then be the transformation to the machine level. This seems to be a natural way in which to factor the decisions about data structure representation which must be made.

A history of this example program may be interesting. Knuth's original algorithm in his book is at the machine level. When I first designed VERS (an access path language) I coded the algorithm in VERS to see what kinds of implementation decisions could be

postponed in this way. When I started to think in terms of a relational level of description I coded it again, essentially as in Figure 1. Then later I developed this optimization method and applied it to my relational level program. The result (Figure 6) is essentially the same as my original VERS program. Comparing these three descriptions of one algorithm should clarify the reasons why I think these three levels of description are important. See also [2].

[1] Knuth, D. E. The Art of Computer Programming, Vol. 1. Addison-Wesley, 1968, p. 258.

[2] Earley, J. "On the Semantics of Data Structures" Courant Institute Symposium on Data Bases, 1971.

	WRITTEN BY THE PROGRAMMER	PRODUCED BY OPTIMIZER
(A)	{X ∈ S | F(X) = A}	T(A)
	S = S WITH X	————————; T(F(Y)) = T(F(Y)) WITH Y;
	S = S LESS Y	————————; T(F(Y)) = T(F(Y)) LESS Y;
	F(Y) = Z	IF Y ∈ S THEN
		T(F(Y)) = T(F(Y)) LESS Y;
		T(Z) = T(Z) WITH Y;
		END IF;
		————————;
(B)	{X ∈ S | F(X) = C}	T
	S = S WITH Y	————————; IF F(Y) = C THEN T = T WITH Y;;
	S = S LESS Y	————————; IF F(Y) = C THEN T = T LESS Y;;
	F(Y) = Z	IF Y ∈ S THEN
		IF F(Y) = C THEN T = T LESS Y;;
		IF Z = C THEN T = T WITH Y;;
		END IF;
		————————;
(C)	#{X ∈ S | F(X) = A}	K(A)
	S = S WITH Y	IF Y ∉ S THEN K(F(Y)) = K(F(Y)) + 1;————————;;
	S = S LESS Y	IF Y ∈ S THEN K(F(Y)) = K(F(Y)) - 1;————————;;
	F(Y) = Z	IF Y ∈ S THEN
		K(F(Y)) ← K(F(Y)) - 1;
		K(Z) ← K(Z) + 1;
		END IF;
		————————;
(D)	#{X ∈ S | F(X) = C}	K
	S = S WITH Y	IF F(X) = C AND Y ∉ S THEN K ← K+1; ————————;;
	S = S LESS Y	IF F(X) = C AND Y ∈ S THEN K ← K-1; ————————;;
	F(Y) = Z	IF Y ∈ S THEN
		IF F(Y) = C THEN K ← K-1;;
		IF Z = C THEN K ← K+1;;
		END IF;
		————————;

Figure 1

```
( ∀ <B,C> ∈ INPUT)
   S = S ∪ {B,C};
   SP = SP WITH <B,C>;
END ∀;
   (∀ A ∈ S | ∄ <-,A> ∈ SP)
   OUTPUT(A);
(∀D = <A,-> ∈ SP) SP = SP LESS D; END ∀;
   S = S LESS A;
   IF #S = 0 THEN RETURN;;
   END ∀;
```

Figure 2

```
( ∀ <B,C> ∈ INPUT)
   S = S WITH B;
   S = S WITH C;
   SP = SP WITH <B,C>;
   END ∀;
   (∀ A ∈ S | #{P ∈ SP|SECOND(P)=A} = 0)
   OUTPUT(A);
   (∀D ∈ SP|FIRST(P)=A) SP = SP LESS D;  END ∀;
   S = S LESS A;
   IF #S = 0 THEN RETURN;;
   END ∀;
```

Figure 3

```
( ∀ <B,C> ∈ INPUT)
   S = S WITH B;
   S = S WITH C;
   SP = SP WITH <B,C>; SUCC(B) = SUCC(B) WITH <B,C>;
   END ∀;
   (∀A ∈ S | #{P ∈ SP|SECOND(P)=A} = 0)
   OUTPUT(A);
   (∀D ∈ SUCC(A)) SP = SP LESS D;
   SUCC(FIRST(A)) = SUCC(FIRST(A)) LESS D;
   END ∀;
   S = S LESS A;
   IF #S = 0 THEN RETURN;;
   END ∀;
```

Figure 4

```
(∀ <B,C> ∈ INPUT)
S = S WITH B;
S = S WITH C;
IF <B,C> ∉ SP THEN COUNT(C) = COUNT(C) + 1;
SP = SP WITH <B,C>; SUCC(B) = SUCC(B) WITH <B,C>;
END IF;
END ∀;
(∀ A ∈ S | COUNT(A) = 0)
OUTPUT(A);
(∀ D ∈ SUCC(A))
IF D ∈ SP THEN COUNT(SECOND(D)) = COUNT(SECOND(D)) -1;
SP = SP LESS D; SUCC(FIRST(A)) = SUCC(FIRST(A)) LESS D;
END IF;
END ∀;
S = S LESS A;
IF #S = 0 THEN RETURN;;
END ∀;
```

Figure 5

```
( ∀ <B,C> ∈ INPUT)
S = S WITH B; IF COUNT(B)=0 THEN ZRCOUNT = ZRCOUNT WITH B;;
S = S WITH C; IF COUNT(C)=0 THEN ZRCOUNT = ZRCOUNT WITH C;;
IF <B,C> ∉ SP THEN
   IF C ∈ S THEN
   IF COUNT(C)=0 THEN ZRCOUNT = ZRCOUNT LESS C;;
   IF COUNT(C)+1=0 THEN ZRCOUNT = ZRCOUNT WITH C;;
   END IF;
   COUNT(C) = COUNT(C)+1;
SP = SP WITH <B,C>; SUCC(B) = SUCC(B) WITH <B,C>;
END IF;
END ∀;
(∀ A ∈ ZRCOUNT)
OUTPUT(A);
(∀ D ∈ SUCC(A))
IF D ∈ SP THEN
IF SECOND(D) ∈ S THEN
   IF COUNT(SECOND(D)) = 0 THEN ZRCOUNT = ZRCOUNT LESS SECOND(D);;
   IF COUNT(SECOND(D))-1=0 THEN ZRCOUNT = ZRCOUNT WITH SECOND(D);;
END IF;
COUNT(SECOND(D)) = COUNT(SECOND(D))-1;
SP = SP LESS D; SUCC(FIRST(A)) = SUCC(FIRST(A)) LESS D;
END IF;
END ∀;
S = S LESS A; IF COUNT(A)=0 THEN ZRCOUNT = ZRCOUNT LESS A;;
IF #S = 0 THEN RETURN;;
END ∀;
```

Figure 6

```
( ∀ <B,C> ϵ INPUT)
S = S WITH B; IF COUNT(B)=0 THEN ZRCOUNT=ZRCOUNT WITH B;;
S = S WITH C;
COUNT(C) = COUNT(C)+1;
SUCC(B) = SUCC(B) WITH <B,C>;
END ∀;
(∀ A ϵ ZRCOUNT)
OUTPUT(A);
S = S LESS A; ZRCOUNT = ZRCOUNT LESS A;
(∀ D ϵ SUCC(A))
IF COUNT(SECOND(D))=1 THEN ZRCOUNT=ZRCOUNT WITH SECOND(D);;
COUNT(SECOND(D)) = COUNT(SECOND(D)) -1;
SUCC(FIRST(A)) = SUCC(FIRST(A)) LESS D;
END ∀;
IF #S = 0 THEN RETURN;
END ∀;
```