SETL NEWSLETTER 56 September 29, 1971

Additional Comments on Some Basic SETL Operations Jay Earley

Infinite Sets and the Set Formers

The basic point of this note is to argue against the idea of ruling out syntactically a whole class of constructs because some of them are unrealizable. The classes of constructs of this sort which are currently ruled out in SETL are infinite sets, and a more general form of the set former.

In the case of an infinite set, the only operation which cannot be performed to completion on it is the iterator, and even this can be allowed if there is a branch out of the loop at some point. Of course, many operations such as union and intersection may be little used on finite sets and difficult to implement (because one will not be able to keep around a data structure representing the entire set). But some operations such as testing for set membership will often be easy to implement and will be used frequently enough. These will come up especially in what is sometimes called type checking. In a set theoretic language, very often the declared domain of a variable or tuple will not be a type, but rather a more complicated set such as

(1) all square arrays of reals

(2) all binary one-to-one relations on blank atoms of type X

(3) all integers or sequences

One would like to be able to express these domain restrictions in exactly the same language that one uses for computation.

In order to allow infinite sets we need two things:

(1) We should allow types to specify sets. That is, "INTEGER" is a set, and therefore "INTEGER ∪ SEQUENCE" is a set, etc.

(2) We need a more general form of the set former, that is

$$\{f(x_1,\ldots,x_n) \mid P(x_1,\ldots,x_n)\}$$

This, however, would have an even more important advantage. Not only would one be able to use infinite sets, but he would in some cases be able to use more natural forms for specifying finite sets. I have frequently written set formers in the following (currently illegal) form:

$$\{g(x_1,\ldots,x_n) \mid \exists\ y \in S \mid \text{expression containing } x_i = f_i(y) \text{ and/or } x_i \in f\{y\}\ i = 1,\ldots,n\}.$$

This is straightforward to implement and produces a finite set if S is finite. It can be rewritten

$$g(E_1(x_1),\ldots,E_n(x_n))$$

where $E_i(x_i)$ is either "f_i[S]" or "[f_i{[S]}]"

However, some of these forms I have written are incredibly distorted when written under the current syntax.

Now, of course, the question arises, how do we handle this case of allowing syntactically constructions which we can't implement. I propose the following: The most general forms should be allowed. If they are used only in ways (such as membership testing) which can be implemented, then everything is fine. If one of them is used in a way which cannot be handled an appropriate message is given and the user can rewrite it. The manual for the language should specify those forms which are guaranteed to work in all situations, so that the user may stick to these if he likes. This makes him no worse off than now. In addition the manual should specify as closely as possible which additional forms the compiler currently accepts under which situations.

There are two additional advantages to this scheme:

(1) A user who is using SETL as a specification language only (that is, he is not intending to execute program) can specify certain things which are useful and finite but which no current compiler will hope to implement. For instance he can define a context free grammar and what strings are in the language defined by the grammar using a few set formers. Of course, he can't hope to execute a membership test and have

the compiler produce a parser for him, but he has at least the ability to specify (in a well defined programming language) the set he wants.

(2) New horizons are opened up for doing further work toward implementing (and optimizing) some of these more difficult set formers. The meaning is well-defined and it is a matter of devising an implementation. Ruling them out of the syntax entirely cuts off this possibility.

The following is an example SETL program which uses the general form of the set former which I propose. The forms I have used in fact fit the one specific form which I mentioned above. The program is for a context-free parser which is described in my paper [1]. This program performs the function of the recognizer in the paper without including the look-ahead feature. The data structures I have used, and the notation mirror those in the paper very closely, so I shall not describe them here. Warning: this is a somewhat different representation than that used by J. Schwartz in his description of nodal span parsing, so a familiarity with my paper is probably necessary to understanding the algorithm.

```
   DEFINEF REC(INPUT,N);
   EXTERNAL ROOT_PROD, G;
 DEFINEF ALT(N), RETURN {P ∈ PROD(G)|DEF(P)=N}; END ALT;
   SS = NL
   SS(0) = {<ROOT_PROD,0,0>};
   (0 ≤ ∀I ≤ N)
   SS(I) = SS(I) ∪* {<Q,0,I> | ∃<P,J,-> ∈ SS(I) |
                     Q ∈ ALT(P(J+1))}
                 ∪* {<Q,L+1,G> | ∃ <P,#P,F> ∈ SS(I) |
                     ∃ <Q,L,G> ∈ SS(F) | P ∈ ALT(Q(L+1))};
   SS(I+1) = {<P,J+1,F>|∃<P,J,F> ∈ SS(I) | P(J+1)=INPUT(I)};
   IF SS(I+1) = NL THEN RETURN FALSE;;
   IF SS(I+1) = {<ROOT_PROD,2,0>} THEN RETURN TRUE;;
   END ∀I;
   END REC;
```

I have taken two liberties with SETL in addition to using the more general set former. These are partly to make the algorithm clearer, and partly because I believe they should be included in SETL.

(1) One may write "{<a,-,b> ∈ S | P(X)}" where the "-" means that we don't care what the second element is, or that we mean all second elements. This is also allowed in the "∃" and "∀" forms.

(2) There is a new operator ∪*, which is essentially the transitive closure of union. In the expression "A ∪* B" , we first take the union of A and B. If this is larger than A, we take its union with B again, and repeat the process until the new union is the same as the old. Normally, of course, B will be a set former which includes A in its definition. Note that the above is a definition of ∪*, and not necessarily the way it must be implemented. Notice also that this program for the algorithm is actually more concise and contains fewer loops than that in the paper. This is because of the ∪* operator and the set former.

[1] J. Earley, "An efficient context-free parsing algorithm," Comm. ACM Jan. 1970.