SETL Newsletter # 124

Ed Schonberg
January 30, 1974

The VERS2 Language of J. Earley Considered in relation to SETL

We comment briefly on a recent article by Jay Earley whose contents will be of interest to those familiar with the SETL project. Earley describes some of the features of a very high-level language, VERS2 (implemented on the EL1 extensible system of Wegbreit et al.) whose overall design has a number of points of contact with SETL, and which includes some very powerful primitives not provided by SETL. We will see in the course of their description that most of these can be realized in SETL without undue effort as syntactic extensions. However, the VERS2 pattern-matching operations specified by Earley are a set of semantic facilities whose realization in SETL would be non-trivial and worth studying. We discuss first the composite data-types of VERS2. We will then describe iterators and related operations, and finally examine pattern matching.

Data Types

The motivation behind VERS2 is very similar to that which gave origin to SETL: the need for a powerful language for algorithm design and description, where matters of efficiency are subordinate to expressiveness. As a result, VERS2 uses sets, mappings, and some of the notations of mathematical set theory. In addition, data-types that have proven useful in a number of programming situations, namely lists and structures with named fields, are also provided. The following list describes each type briefly:

a) sets are unordered collections of homogeneous values. Eg { 1,2,3,4 }

b) Sequences are ordered collections of homogeneous values. They correspond to the SETL tuple, and are noted [a,b...]

c) Tuples correspond to the structures of ALGOL or PL/I. They have a fixed number of named components.

d) Relations are sets of tuples describing mappings between values. They correspond to SETL tabular functions.

e) Functions like relations are mappings, but they are required to be total, so that the value of a function is defined for every point in its domain. The domain itself has to be static and cannot be redefined.

f) Ordered-sets are linked lists, constructed by presenting some ordering relation on the elements of a set. Once constructed they are accessed like sequences, but they are modified like sets, using the primitives ADD and DELETE (similar to SETL operators in and out). Conceivably the primitive ADD when applied to an ordered set, will make use of the ordering defined on this set to insert a new element in its approprate position.

g) Bags, i.e. sets with repeated elements, are also provided.

Iterators

A rich family of iterators over sets and sequences is provided. The quantifiers $\exists$ and $\forall$, have the same meaning as in SETL. An interesting semantic distinction is made between iterators and iterative operations. In the expression

$$\forall\ x \in s : C(x)$$

$x \in s : C(x)$ is an iterator, and is understood to supply a stream of values. '$\forall$' is an iterative operation, and it acts upon that stream of values to yield a result.

The iterative operations $\exists$ and $\forall$ yield boolean values. To express an iterative loop that performs a block of code, the iterative operation FOR is used (where in SETL $\forall$ would do). Iterators without iterative operations appear in the usual set former: { x∈s | C(x) } and also in a similar sequence former: [x∈s |C(x)]. Notice that the same iterator is used in both cases, bypassing the explicit mention of an index used in SETL to iterate over tuples.

For numerical iterations, the following form is provided:

A ⟵ i, n(A), f

equivalent to the SETL dictions:

A = i; until A <u>eq</u> f doing A = n(A);....

A converge iterator useful in connection with reals is provided, which iterates until the same values are generated for the iteration variable:

A ⟵ i, n(A), e

This would be expressed in SETL as follows:

new = i; while (old <u>ne</u> new) doing [old = new; new = n(new);]....
when using real variables, <u>e</u> will be an epsilon defining convergence, i.e. the argument of the *while* clause above will be ((old - new) <u>lt</u> e)
otherwise e is omitted.

Iterators appear in the definition of composite operators, as in SETL. WHILE and UNTIL operators are provided. Finally, a simple algebra of iterators is defined. The sequence

iterator; iterator

generates first one sequence of values, then the other, the sequence

iterator X iterator

corresponds to the usual nesting of iterators in SETL, eg

```
(∀ xεs, xlεx)
```

The parallel combination

```
iterator || iterator
```

generates the first value of one sequence, then of the other, and so on. The parallel combination can only be applied to sequences, and its advantages seem slight. Several additional iterative operations deserve mention: THE, FIRST, and LAST return the only value generated, the first or the last, respectively. The action of FIRST is clearly identical to that of the existential quantifier:

```
truthval = ∃ xεs | p(x);
  first = x ;
```

when iterating over sets, the action of LAST can be expressed by the code block:

```
[: ∀(xεs)  if p(x) then last = x;;; return last;]
```

the action of THE requires a slightly lengthier expression:

```
[: i = 0; ∀(xεs) if p(x) then the = x; i=i+1;;;
        return if i eq 1 then the else om; ]
```

Notice however that the power of these iterative operations is greater than that of the SETL expressions above. In particular, a different form would have to be written in SETL for each type of iterator, while in VERS2 one can write with the same ease, e.g.

```
LAST A ←1, f(A) | Q(A)
```

<u>SORT</u> is another iterative operation which produces a sorted sequence out of the stream of values generated by an iterator

DEL (delete) and REPL(replace) can be applied to iterators with obvious results. For example

REPL x∈s |(x <u>gt</u> 5) WITH(x//5)

would correspond to the following SETL code:

<u>s1</u> = s; (∀ x∈s1) if (x <u>gt</u> s) then x <u>out</u> s; (x//5) in <u>s</u>;;;

We mention finally that an iterator is provided to generate all members of the power set of a set, or all subsequences of a sequence: A ⊆ s (Notice that in the current SETL implementation, iterations over the power set of a set are carried out without explicitly constructing the power set, set rather by generating each element of it in some arbitrary order. The same intent is apparent in VERS2).

<u>Pattern Matching</u>

VERS2 is intended to have pattern-matching facilities similar to, but more powerful than those of SNOBOL. In particular, matchings can be attempted on any data-type. To support these facilities, VERS2 makes use of a primitive MATCH operation, and a new data-type, the <u>extractor variable</u>. An extractor variable is similar to an immediate assignment in SNOBOL, and it provides therefore the equivalent of the SNOBOL <u>unevaluated expression</u> facility. As a suggestive example, consider the following VERS2 statement:

s MATCH <{p,ARB}, {p, ARB}> => TRUE

(where p has been declared to be an extractor). This can be expressed in SETL as follows:

< set1, set2 > = s;

match = ∃ p∈ set1, q∈ set2 |(p <u>eq</u> q)

It is clear that as the pattern to be matched grows in complication, the SETL code necessary to describe the matching will become more obscure. What is implied by the MATCH primitive is a fairly elaborate parsing algorithm capable of building arbitrary trees.

The following details concerning pattern-matching in VERS2 deserve mention:

a) A simple set of pseudo-patterns similar to those of SNOBOL, is provided: REP(pattern, number) (equivalent to DUPL) REP(pattern) (equivalent to ARBNO), and ARB. (Notice that in the example above, ARB matches a subset of a set)

b) The names of data-types are valid patterns.

c) Patterns can be used in conjuction with iterators, thus implying any number of matching operations. For example

$$\# \{TUPLE, [REP(-, 5)]\} \in S$$

returns the number of sets in S which contain two elements: a tuple and a sequence consisting of S arbitrary elements. It is clear that an impressive economy of notation has been achieved by this combination of primitives. The advantages of such economy in describing complicated algorithms need not be emphasized to those familiar with SETL.

References

Earley, J. " Relational level data structures in programming languages", Computer Science, University of California, Berkeley, 1972

"High level operations in automatic programming Computer Science, University of California, Berkeley, October, 1973