SETL Newsletter No. 96

Pointers and "Very High Level Languages"

by N. Minsky

January 1973

The subject of this note is the question: should *pointers* (or "reference variable") be included as primitives of "very high level languages". By the phrase "very high level language" (VHLL) I mean, following Schwartz, a language which is designed to free the programmer from any consideration of efficiency and data structures. To be specific I will discuss the problem in the context of Schwartz' SETL.

As far as I understand, the main argument against the inclusion of pointer primitives in a VHLL is the following: essentially, one may argue, a pointer carries in it the physical address of an information item. And since a program written in a VHLL should be free from data structures, there is no need for *pointer* primitives.

This argument, however, is an oversimplification of the pointer concept. There is in fact much more to it than just "address of an item". To see that, consider the following sequence of instructions written in informal SETL:

```
1) A = {1,2}
2) B = {A,{3,4}}
3) A = {0}
4) Print B
```

If these instructions are interpreted by SETL, instruction (4) will print the set {{1 2} {3 4}} .

But suppose that what the programmer means by instruction (2) is: "Let the set B contain the set {3,4} and the set A whatever the set A will be (in future)". In that case instruction (4) should print the set {{0}{3,4}}.

In set theory, such a statement is meaningless, because set theory does not contain the time as one of its primitive concepts.[1] On the other hand, the "time"[2], as a sort of meta-concept, is very fundamental in programming. (In fact, the central position of time in programming may well be the most subtle difference between programming and mathematics.)

---

[1] The concept "time" is used here in a very special sense, which requires rigorous definition. But its meaning here should be obvious without such definition.

[2] That, of course, does not mean that one can not describe time dependent structures in set theory, it only means that the set theoretical formulation itself does not contain the time concept.

The simplest way of expressing, in programming, the statement "the set A is a member of the set B , whatever A may be" is by saying: "B contains a pointer (or, a reference) to the set A". So that quite apart from any consideration of storage, or implementation, the *pointer* primitive is conceptually essential in programming. Moreover, pragmatically the second interpretation of statement (2) above, and with it the pointer concept, is very useful if not indispensable in a number of problem areas. Notably, in simulation and data bases, but not only in them.

SETL, which is heavily based upon set theory does not support any natural way for expressing statements such as (2), according to its second interpretation. That is not to say that this is impossible. In fact, the blank atoms of SETL can be used as pointers, but not always very conveniently. For example, to yield the second interpretation, our programming example could have been written as following:

1) A = {1,2}
2) P = newat
3) F = {<P,A>}
4) B = {P,{3,4}}
5) f(P) = {0}
6) The printing statement is more complex because, for each item of B we have to check if it is a blank atom, and then we should print f(p), or it is non-blank.

The main disadvantage of this approach is that if we wish to refer to a set by means of a pointer, we have to include it in another set, a function, associated with this pointer. And we can not address this set by name. Apart from the direct inconvenience involved with that, it introduces a requirement for specific decision by the programmer, as to how he is going to use his information, contrary to the stated objective of the language.

I would like therefore to suggest the following extension to SETL (which should not be viewed literally, as I am not familiar enough with the language to discuss its extension in a rigorous way).

There should be a primitive "universal" function, say U , which can contain only pairs of the following type.

<q,N>

Here q is a blank atom, and N is a name of some set.

So that, if we have a set named A , and if we execute the instruction

U = U with <q,A>

for a blank atom q , then we can refer to the set A both by its name and by "U(q)".

There are several issues involved with such a construct; I will mention only one of them:

Suppose that the set B is created by

B = {q,{3,4}}

while q is a pointer to a set A . Consider the instructions

A = {1,2}

C = B .

The second assignment statement may have two different effects; it may copy to C either the set {q,{3,4}} , or {{1,2}{3,4}} .

We may resolve this problem by generalizing the assignment statement as follows:

We attach an index to the equal sign such as: $C \overset{i}{=} B$

(This, of course, is not the proposed syntax).

If i equals zero (or blank) then B is copied into C , literally. In the example above

C = B

will copy {q,{3,4}} into C .

If, however, $i \neq 0$ then the pointers will be evaluated up to the i-th depth. For example:

If A1 = {1}

A2 = {q1,2} (where q1 points to A1)

B = {q2,3} (where q2 points to A2)

Then C = B copies {q2,3} to C

$C \overset{1}{=} B$ copies {{q1,2},3} to C

$C \overset{2}{=} B$ copies {{{1},2},3} to C .

One should also have a notation for indefinite evaluation of pointers, such as $C \overset{\infty}{=} B$. In this case, however, one has to decide what to do with cycles.