A GENERAL CONSTRUCTION FOR EXPRESSING REPETITION

L. Lamport

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

I have became aware of a shortcoming in programming languages which makes it difficult to express some fairly simple things. As an example, consider an algorithm which processes one input at a time, obtaining that input from n different input queues. The algorithm can take as its next input the first element from any of the queues. In order to express this non-determinism, it seems natural to write the algorithm as follows, using Dijkstra's do statement [1].

```
(1)     do  queue 1 not empty ---> process first element of queue 1  □

                               . . .

            queue n not empty ---> process first element of queue n

        od
```

Unfortunately, the elipsis (...) is not part of the programming language. Moreover, (1) would not even be a convenient informal expression if "process first element of queue" were a complicated expression. This immediately suggests the following sort of notation for expressing this algorithm.

```
        do for i := 1 until n

                queue i not empty ---> process first element of queue i

        od
```

However, there are two reasons to consider a more general type of construction. First of all, the fact that the queues happened to be numbered by consecutive integers is clearly not significant; they could just as well have been indexed by the elements of any finite set. Secondly, this type of repetitive construction is also useful in contexts other than a <u>do</u> statement. I therefore propose the following general notation. The expression

(2) <u>forall</u> id <u>in</u> S separator E(id) <u>endforall</u>

is equivalent to the expression

(3) $E(x_1)$ separator $E(x_2)$ ... separator $E(x_n)$

where id is an identifier; separator is a syntactic atom; $x_1, \ldots, x_n$ is any enumeration of the distinct elements of the set S ; and $E(x_i)$ is the expression obtained by substituting x_i for id in the expression E(id) . If S is the empty set, then (3) is a null sequence, whose meaning will depend upon the separator .

As an example of this notation, the <u>do</u> statement (1) can be expressed as follows:

(4) <u>do</u> <u>forall</u> i <u>in</u> [1 .. n] □

queue i not empty ⟶ process first element of queue i

<u>endforall</u>

<u>od</u>

where [1 .. n] denotes the set of integers from 1 to n . If n = 0 , so [1 .. n] is the empty set, then (4) is equivalent to a <u>skip</u> statement.

The <u>forall</u> statement (2) does not specify any ordering of the expressions $E(x_i)$. To specify an ordering, I propose the following statement:

(5) <u>forall</u> id <u>seq</u> <u>in</u> S separator E(id) <u>endforall</u>

where id , separator and E are as in (2), and S is an <u>ordered</u> set. The meaning of (5) is given by (3), except that this time the enumeration of S must be chosen so that $x_1 < x_2 < \ldots < x_n$, where $<$ is the ordering relation on the ordered set S .

The use of the <u>forall</u> construction is further illustrated below, where several different concepts are expressed with it.

"for all" expression	equivalent expression in "ordinary" notation
Algol statement:	
forall i seq in [1 .. n] ; E(i) endforall	for i := 1 step 1 until n do E(i)
Logical expressions:	
forall x in S and E(x) endforall	$\forall x \in S : E(x)$
forall x in S or E(x) endforall	$\exists x \in S : E(x)$
Arithmetic expression:	
forall i in [1 .. n] + E(i) endforall	$\sum_{i=1}^{n} E(i)$

Note that a null sequence of ands is defined to be identically true, and a null sequence of ors is defined to be identically false.

The forall construction may be viewed either as an informal notation, or as an addition to any programming language. As a programming language construction, it has the following three possible uses, depending upon what kind of sets S may be specified.

(1) If S is a finite set which is known at compile time, then the forall simply provides "syntactic sugaring"; it can be implemented by with an ordinary "macro".

(2) If S is a set which is known to be finite, but which is not known at compile time, then the forall can provide a useful semantic extension to the language. For example, if n is an ordinary program variable, then there is no nice, simple way to write the statement (4) in Dijkstra's language. The type of semantic extension introduced in this case does not seem to raise any serious theoretical or implementation difficulties.

(3) If S is a set which may be countably infinite, then the forall provides a very strong semantic extension to the language. For example, the expression

forall i *seq* *in* [1 ...] ; *if* P *then* E *endforall*

where [1 ...] denotes the set of positive integers, P is a boolean function without side effects, and P and E do not mention i , could be interpreted to be equivalent to the expression

while P *do* E .

This type of extension seems to be difficult to interpret, and I would not recommend it. Hence, I propose that the syntax for expressing sets be restricted so S has to be finite and "easy to compute".

It is incumbent upon anyone proposing a new semantic construction to rigorously define the semantics of that construction. However, the proposed *forall* construction is a *syntactic* rather than a semantic one. One can no more speak of the semantics of the *forall* construction than of the semantics of the comma. The meaning of a statement containing a *forall* must be defined as part of the semantics of the entire language. However, the following recursive relation can be viewed as a formal "meta-definition" of the non-sequential *forall* construction. (A similar relation can be written for the sequential construction.)

forall x *in* S separator E(x) *endforall* ::=

IF S empty THEN "null separator string"

ELSE $\exists x_0 \in S :$ $E(x_0)$ separator

forall x *in* $S - \{x_0\}$ separator E(x)

endforall

where "null separator string" must be defined for the particular separator.

The *forall* construction can also aid in defining the semantics of a programming language. For example, by using the *forall* , Dijkstra's *do* and *if* statements can be defined without requiring the (informal) ellipsis employed in [1].

The *forall* seems to be a very useful informal notation, and I recommend that it be used now as part of the "natural language" of mathematical reasoning. If it succeeds in becoming popular, it will inevitably find its way into programming languages.

REFERENCES

1. E.W. Dijkstra: A Discipline of Programming, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1976).