A PROPOSAL FOR AN IMPERATIVE COMPLEMENT TO PROLOG

by

M.H. van Emden

CS-76-39

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF WATERLOO
WATERLOO, ONTARIO, CANADA

August 1976

A PROPOSAL FOR
AN IMPERATIVE COMPLEMENT TO PROLOG

ABSTRACT:

It is argued that it is useful to be able to interface subroutines written in an imperative program language with programs written in a descriptive language such as predicate logic. An imperative language based on flowgraphs is sketched, together with a proposed interface for the PROLOG system for logic programming. Prerequisites for this report are: principles of logic programming [2,3,5], PROLOG [4], and flowgraphs [6,7].

1. Imperative complements to descriptive program languages

A language is called *descriptive* when it can be made clear from a program written in it *what* is done by executing it. Such a program typically does not reveal the details of *how* the results of the computation are achieved; these belong to the *imperative* aspect of algorithm specification. It follows that a descriptive language allows the programmer little direct control over efficiency of execution.

A language like pure LISP, where the programmer can specify the computations to be executed by writing recursive function definitions, is an extreme example of a descriptive language. At the imperative extreme would be a language which restricts programs to assignations, tests, jumps, and the invocation of primitive functions.

It is useful for a program language to allow the composition of both descriptive programs (where understandability is critical) and imperative programs (where efficiency of execution is critical). This capability is inherent in the design of a language like Algol 60. In LISP, where the design is modeled on recursive function definitions, the imperative component has been added as the PROG feature.

The importance of LISP derives from the fact that its recursive function definitions in a mathematical style afforded a unique opportunity for running descriptive programs. Predicate logic used as a program language can be called descriptive for the same reason as LISP: programs can be written to reflect faithfully recursive definitions. An implementation such as PROLOG has a relationship to predicate logic which is similar to the one between LISP and the lambda-calculus. The recursive definitions of logic programs define relations rather than functions; the relations are of arbitrary arity defined over a data space of terms of arbitrary complexity.

I believe that it is important to be able to write both descriptive and imperative programs in a given language. The PROG feature of LISP is not just a concession to programmers' inherent perversity. The persistence of FORTRAN is not just a symptom of the power of commercial interests and of conservatism on the part of programmers. For a

certain range of applications the imperative may well be the appropriate mood for expressing useful solutions to programming problems. And it is not clear whether the distinction between imperative and descriptive is always useful: when programming with verification conditions [7] one can fluently combine the understandability of the descriptive mood with the efficiency attainable in the imperative mood.

The purpose of the present report is to propose a method of imbedding an imperative sublanguage into a logic-based program language. The utility of the method is based on the advantages of programming with verification conditions. The feasibility of the method is based on the fact that verification conditions are sentences of logic and on the simplicity of translating certain sets of verification conditions to assignations, tests, and jumps.

2. Recursion and iteration in logic programs

Consider the following example of a recursive definition of the exponentiation relation

$$exp(x,y,z) \quad \text{iff} \quad x^y = z$$

The following properties of integer exponentiation are taken as the defining ones:

$$x^0 = 1 \quad \text{i.e.} \quad \forall x \,.\, exp(x,0,1)$$

and

$$x^y = x \,.\, x^{y-1} \quad \text{i.e.} \quad \forall x,y,z \,.\, exp(x,y-1,z) \supset exp(x,y,x.z)$$

It is taken for granted that the following defining property can be added without altering the relation defined and yet making the definition more "efficient":

$$x^{2y} = (x^2)^y \quad \text{i.e.}$$

$$\forall x,y,z \,.\, (exp(x.x,y/2,z) \wedge even(y)) \supset exp(x,y,z)$$

The above relational definitions can be transcribed to the following PROLOG program, where div, mult, moins, and reste are primitive predicates for the required arithmetic operations.

```
+exp(*x,0,1).
+exp(*x,*y,*z) -even(*y) - div(*y,2,*y by 2)
               -mult(*x,*x,*xsq)
               -exp(*xsq,*y by 2,*z).
+exp(*x,*y,*z) -moins(*y,1,*ymin 1)
               -exp(*x,*ymin 1, *z1)
               -mult(*z1,*x,*z).
+even(*y) - reste(*y,2,*r) - equals(*r,0).
+equals(*x,*x).
+fin.
-exp(2,10,*z) - sorter(*z) - ligne.
```

The first procedure call causes the following sequence of calls to exp:

```
-exp(2,10,*z).
-exp(4,5,*z).
-exp(4,4,*z1) - mult(*z1,4,*z).
-exp(16,2,*z1).
-exp(256,1,*z1).
-exp(256,0,*z2) - mult(*z2,256,*z1).
```

The last call to exp succeeds without calling another procedure. It causes the substitution *z2 := 1 to be made. At this point there are 5 other calls to exp. As the stack is dismantled the partial results *z2 = 1, *z1 = 256 . *z2, and *z = 4 . *z1 are gathered together to yield *z = 1024.

Let us compare the computational behaviour shown above with that of the following program in logic:

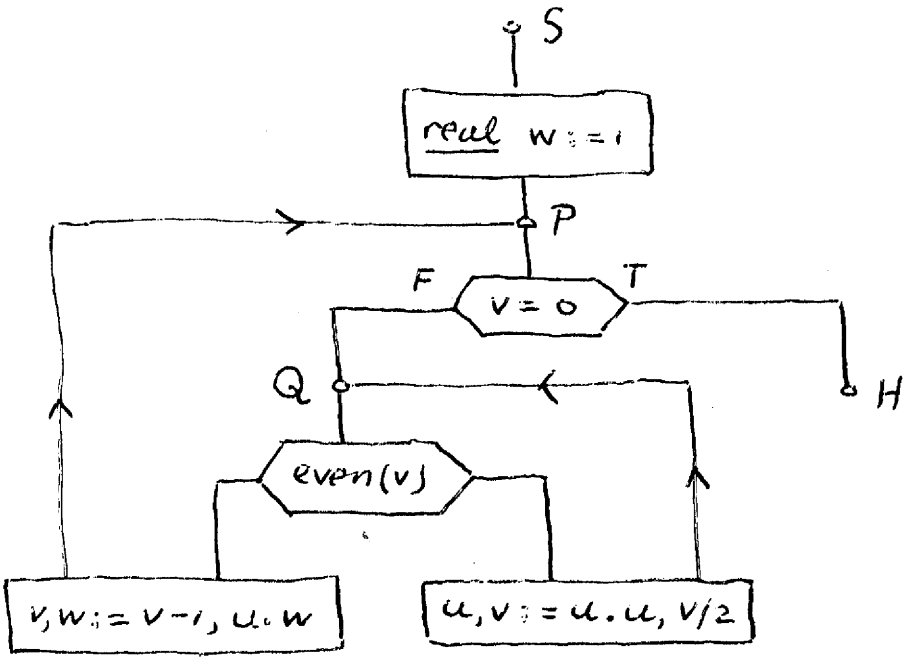
```
-S(*x) - W1(*x,*y) + P(*y).
-P(*x) - VEQ0(*x,*y) + H(*y).
-P(*x) - VNE0(*x,*y) + Q(*y).
-Q(*x) - UV(*x,*y) + Q(*y).
-Q(*x) - VW(*x,*y) + P(*y).
+H(*y) - sorter(*y) - ligne.

+W1(*u.*v,*u.*v.1).
+VEQ0(*u.0.*w,*w).
+VNE0(*x,*x) - not(VEQ0(*x,*y)).
+UV(*u.*v.*w,*u1.*v1.*w)
     -even(*v) - div(*v,2,*v1) - mult(*u,*u,*u1).
+UW(*u.*v.*w,*u.*v1.*w1)
     -moins(*v,1,*v1) - mult(*u,*w,*w1).
+even(*x) - reste(*x,2,*r) - equals(*r,0).
+equals(*x,*x).
+not(*x)-*x-/-qqq.
+not(*x).
+fin.
```

The above program computes 2^{10} by means of the following sequence of procedure calls:

```
+S(2.10).
+P(2.10.1).
+Q(2.10.1).
+Q(4.5.1).
+P(4.4.4).
+Q(4.4.4).
+Q(16.2.4).
+Q(256.1.4).
+P(256.0.1024).
+H(1024).
```

At the end of the computation the result is already completed. The procedure calls on the stack hold no information that is required later on, and they need not be retained; in fact the procedure calls can be treated as jumps: the procedures are only entered and need never be exited from. As a result the second program is more efficiently executable than the first one.

What is the secret of discovering a logic program with this desirable behaviour? Consider the following flowchart for exponentiation.



In order to prove partial correctness of this flowchart according to Floyd's method one has to prove the validity of the following verification conditions:

$\{S\}$ real w := 1 $\{P\}$

$\{P \;\&\; v=0\}\ \{H\}$

$\{P \;\&\; v\neq 0\}\ \{Q\}$

$\{Q \;\&\; even(v)\}$ u,v := u x u,v/2 $\{Q\}$

$\{Q \;\&\; \neg even(v)\}$ v,w := v - 1, u x w $\{P\}$

Each of these implications of logic has a representation in clausal form, which is expressed in the first five lines of the above PROLOG program. The remaining lines contain logic specifications of the assignations and tests of the flowchart.

Apparently, flowcharts are expressible in logic via verification conditions and the resulting logic programs are efficiently executable, but only if treated as a special case. This possibility is the basis of my proposal for obtaining an imperative complement for logic as a program language. There are two choices for realizing this possibility. According to the first, one relies on a suitable compiler to recognize that certain segments of a logic program are the verification conditions of a flowchart, which then receives special treatment. To use this possibility is counterproductive: first the programmer disguises the imperative program as logic and then the compiler, which had to be made extra clever, has to do extra work to see through the disguise.

According to the other choice, which I prefer, one writes an imperative subroutine in a different language. The change of language is signalled by an escape symbol, like the PROG of LISP.

3. A proposal for subroutines in PROLOG

The proposal is to interface PROLOG with subroutines, i.e., with subprograms written in an imperative language. In PROLOG the activation of a goal either causes it to succeed immediately, or it triggers the execution of a logic subprogram, which consists itself of goals. A goal succeeds immediately if it matches an assertion in the user's program, or if its predicate is an "evaluable" predicate. The activation of a goal formed with an evaluable predicate causes an effect (a "side" effect) which would be difficult or impossible to get by means of a logic program.

In the present form of PROLOG there is no possibility for a user to change the given repertoire of evaluable predicates. I propose to allow the user to introduce an arbitrary number of arbitrary identifiers as evaluable predicates, and to call them *imperative predicates*. Each such predicate is associated with a subroutine, which is executed as the result of activating a goal (an *imperative goal*) formed with the predicate. In the following I sketch

a) how to use the imperative goal to transmit input to the subroutine before its execution and how to retrieve the output afterwards
b) how to associate a subroutine with an imperative predicate
c) how to write the body of the subroutine

a) In BNF notation:

< imperative goal > ::= -< imperative predicate >
(< actual input vector >, < actual output vector >)

< actual input vector > ::= < actual input component > |
< actual input component >. < actual input vector >

< actual input component > ::= < logic variable > |< constant >

< actual output vector > ::= < logic variable > |
< logic variable >. < actual output vector >

Note that the input and output vectors can be regarded as terms of PROLOG written with an infix function symbol ".", and associating from right to left (at least, this alternative seems to harmonize with the above grammar). The precedence of "." will also have to be specified.

At the time of activation of an imperative goal any variables of the input vector have to be instantiated with constants, each of a type specified by a declaration in the subroutine associated with the imperative predicate. None of the variables in the output vector may be bound. It follows that the actual input vector and the actual output vector must be disjoint. On success of an imperative goal, the variables of the actual output vector are instantiated with constants of types determined by declarations in the subroutine.

b) An *imperative declaration* associates a subroutine with an imperative predicate.

```
< imperative declaration > ::=
   +PROG(< imperative predicate >:
         < formal input vector > → < formal output vector >
        ,< declarations >;
         < statements >
        ).
< imperative predicate > ::= < identifier  >
< formal input vector > ::= < formal vector >
< formal output vector > ::= < formal vector >
< formal vector > ::= < identifier > |
                      < identifier >.< formal vector >
```

Apparently, an imperative declaration has the form of an assertion (with predicate symbol PROG) of logic programming, which associates an arbitrary identifier as imperative predicate with the subroutine specified by the second argument of PROG in the assertion. The occurrence of such an assertion in a PROLOG program makes it possible to activate a subroutine via a goal containing the imperative predicate and an input vector acting as a vector of actual parameters for the subroutine.

Note that the first argument of PROG can be regarded as a term of logic, which is composed of the function symbols ":", "→", and ".". It will be seen that the second term of PROG, which specifies the subroutine, can also be regarded as a term of logic.

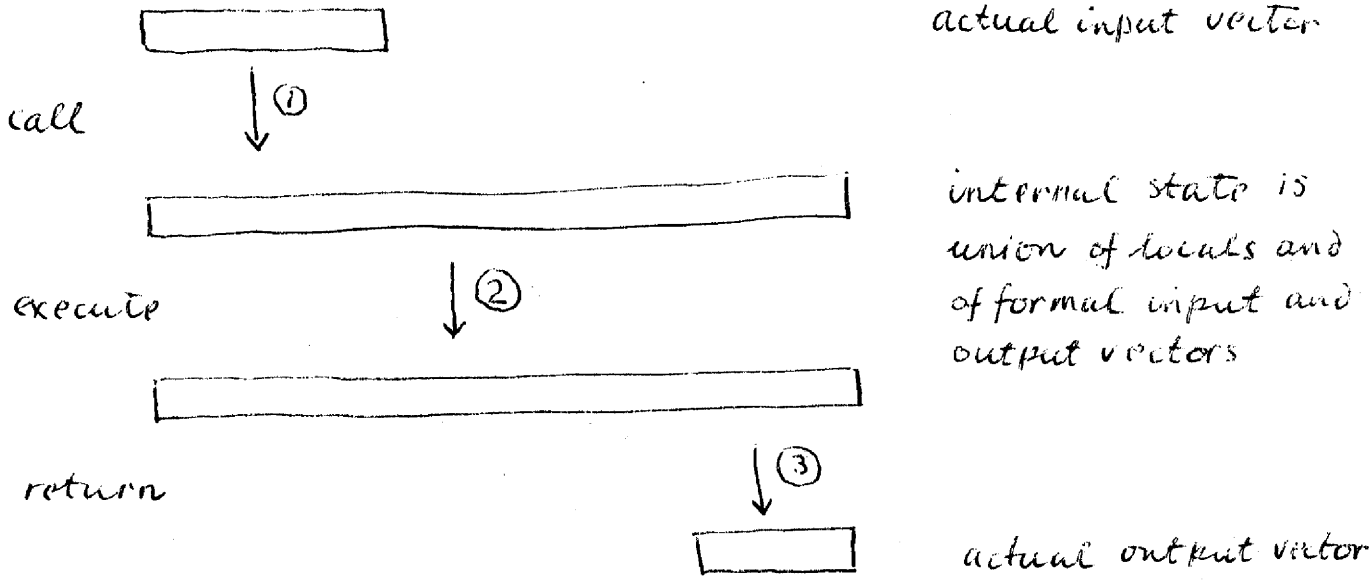
c) For the subroutines a separate language has to be designed or adopted. Anything will do that specifies binary relations from input vectors to output vectors. The literature contains much useful information on such languages, whereas my knowledge in this area is but little. This is therefore the least important part of the proposal: its defects do not cast doubt upon the proposal as a whole.

A rudimentary imperative language with only characters, integers, reals, and arrays of these as data types, some primitive functions, assignation, together with composition, test and jump as aids to sequencing, is already useful as an imperative complement to PROLOG. This is because PROLOG, among other high-level features, provides powerful facilities for calling and defining procedures and also provides fancy data structures. But it would be interesting to introduce a less rudimentary imperative language with some high-level features that overlap with features of PROLOG to see which preferences develop in the practice of programming.

The design of an imperative language is interesting in its own right. In the introduction I have contrasted descriptive languages, where sequence control is hidden, which can be easy to understand and to write in, but which allow little control over efficiency of execution, with imperative languages which, allegedly necessarily, have the opposite characteristics, because sequencing has to be controlled explicitly by the programmer. The contrast is misleading because by programming with verification conditions and then translating these to an imperative language [7], it is possible to get the best of both worlds.

One does not need to design a new imperative language to be able to experiment with this interesting method: I have shown [7] that even FORTRAN is adequate. However, when designing an imperative complement to PROLOG, one might as well make it easier to translate from verification conditions. I therefore propose to use the language of flowgraphs [6,7] because of their powerful pragmatics and their close relationship to verification conditions, as the basis for an imperative complement in PROLOG.

An imperative subroutine may be regarded as a binary relation between actual input vectors and actual output vectors. The possible presence of variables local to the subroutine slightly complicates the relation. The body of the subroutine accesses not only the locally declared variables, but also those of the input and output vectors. All these variables belong to the state manipulated by the body of the subroutine. All of them have their type declared in the subroutine and may be initialized at declaration. For the variables of the input vector it is difficult to see what use such initialization can have, because it would overwrite the actual parameter. But why explicitly forbid those acts of which *I*, now, do not see the use?



The formal input vector need not be disjoint from the formal output vector. The statements of the subroutine specify a binary relation on "state vectors". The relation is referred to by the arrow labelled 2; the state vector is the union of local variables and the union of formal input and output vectors.

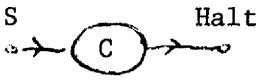
Programs in the form of flowgraphs are described in [6,7]. The statements of a subroutine may be regarded as a flowgraph. Here I only describe a certain text representation of flowgraphs.

The commands, which label the arcs of a flowgraph, can be (parallel) assignations, which are of the form

$$x_1,\ldots,x_k := f_1,\ldots,f_k$$

where $x_1,\ldots,x_k$ are variables and $f_1,\ldots f_k$ are primitive functions of sets of variables. A command can also be a relational expression, as in Algol 60. The meaning of such a command in the semantics of flowgraphs is a subset of the identity relation: the set of states where the relational expression yields true. It is tempting to consider the generalization of relational expressions to the "goals" of logic programming, thus returning in full circle to PROLOG.

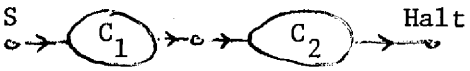
For the flowgraph



I will write

$$S : C .$$

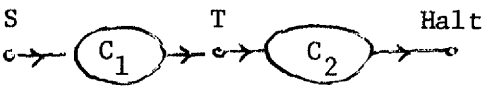
For the (composition) flowgraph



I will write

$$S : C_1 \rightarrow C_2.$$

Read "go on with" for "→". Here C_1 and C_2 are commands. In place of a command a node may also appear: for the graph



I write

$$S : C_1 \rightarrow T.$$

$$T : C_2.$$

If during execution a label is encountered instead of a command, execution continues at a defining occurrence of the label. In the above example "go on with" acquires the meaning of "goto".

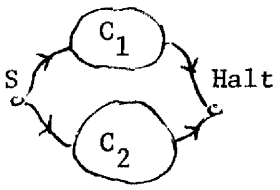
"Go on with" may also acquire the meaning of a procedure call. For instance, consider

$$S : C_1 \rightarrow T \rightarrow C_3.$$

$$T : C_2.$$

where S and T are nodes, and C_1, C_2, and C_3 are commands. Because now → T is followed by "go on with", execution resumes there after the subcomputation, which started at T:, terminates. Note that the above two statements have no graph representation. This is because one corresponds to a production rule of type 2 in the grammar model of programs [6].

For the possibly indeterminate branch:



I will write

$$S : C_1 \mid C_2.$$

where the bar is borrowed from BNF grammars. The usual

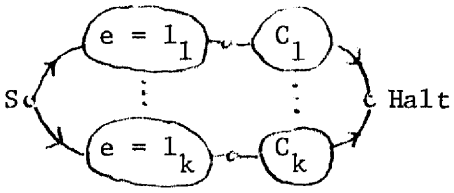
$$S : \underline{\text{if}}\ c\ \underline{\text{then}}\ C_1\ \underline{\text{else}}\ C_2$$

appears as

$$S : c \rightarrow C_1 \mid \neg c \rightarrow C_2.$$

which is a special (the determinate) case of the indeterminate branch. It may not be desirable to let the programmer disguise as an indeterminate branch something never intended as such and then to make the compiler do extra work to recognize it as an if ... then ... else ... . It seems useful to have a determinate branch also available, as well as a branch such as the *cases* of PASCAL. For the time being I lump them together.

For the flowgraph



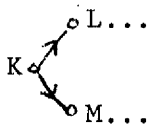
where $l_1,\ldots,l_k$ are disjoint lists of values (hence the branch is determinate) of the same type (which must also be the type of the expression e).

I write

$$\underline{\text{if}}\ e = l_1 \rightarrow C_1$$
$$= l_2 \rightarrow C_2$$
$$\vdots$$
$$= l_k \rightarrow C_k.$$

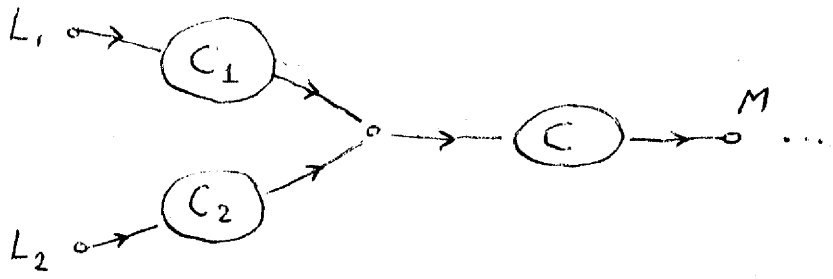
This contruct has in common with the *cases* of PASCAL that one does not need to repeat the expression e. This economy is not the only advantage if the remaining "=" signs act as unambiguous terminators for $C_1,\ldots,C_{k-1}$.

Some further examples:



is written as

K : → L | → M.
L : ...
M : ...

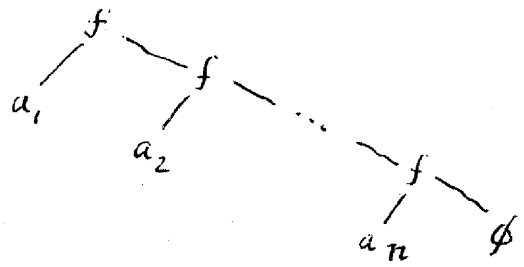


is written as

$L_1 : C_1 \rightarrow L.$
$L_2 : C_2 \rightarrow L.$
$L \ : C \ \rightarrow M.$
$M \ : \ldots$

4. Concluding remarks

There are but few interesting programs using only unstructured data. On the other hand, the kind of program for which one would use an imperative subroutine, typically uses only arrays. It is therefore desirable to have arrays available as data structure for imperative subroutines, either as a local or as a parameter. Actual parameters exists in the logic part of the program, and therefore have to be terms of logic. Formal parameters have to be data types of the imperative language. To a formal parameter specified as a *type* array $a_1,\ldots,a_n$ there will have to correspond as actual parameter a term of logic. An obvious choice seems to be a term of the form



where f is a distinguished 2-place function symbol, $\emptyset$ is a distinguished constant, and $a_1,\ldots,a_n$ are *type* constants.

It should be noted that the entire declaration of an imperative subroutine is to be given as the two arguments of PROG, that is, as two terms of logic. It could just as well have been done as only one term, or as more terms. The point is here that programs in a language, with a syntax defined like the one of Algol 60, are quite naturally represented as terms of logic. Hence, a language with terms as data type (or recursive data structures in the sense of Hoare [1]) can treat as data programs, in the same language or in another. The use of terms

for this purpose would give an improvement with respect to LISP, where LISP programs can be data. LISP has a very restricted repertoire of terms: there is only one function symbol (the dot) and very few constants. As a result LISP programs, in order to be a LISP data structure, have to be represented as such restricted terms, albeit embellished by the list notation for certain "dotted pairs". This situation accounts for the monotonous aspect of LISP programs.

The role of verification conditions in this proposal has two aspects. A mathematical semantics for the flowgraph language can be obtained by a simple translation to a sentence of logic which is a conjunction of implications, each of which is a verification condition; the minimal model of this sentence represents the computations of the program [6]. The other aspect concerns the method of programming with verification conditions [7], which can be regarded as a pragmatics for flowgraphs.

5. References

1. C.A.R. Hoare: Recursive data structures, Research Report STAN-CS-73-400, Dept. of Computer Science, Stanford University.

2. R.A. Kowalski: Predicate logic as programming language, Proc. IFIP 74, North Holland, 1974, pp. 569-574.

3. R.A. Kowalski: Logic for problem-solving, Memo no. 75, Dept. of Computational Logic, University of Edinburgh, 1974.

4. P. Roussel: PROLOG manuel d'utilisation, Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille, 1975.

5. M.H. van Emden: Programming in resolution logic. To appear in 'Machine Representations of Knowledge' published as Machine Intelligence 8 (eds. E.W. Elcock and D. Michie) by Ellis Horwood Ltd. and John Wylie.

6. M.H. van Emden: Verification Conditions as Programs, in 'Automata, Languages, and Programming' (eds. S. Michaelson and R. Milner), Edinburgh University Press, 1976.

7. M.H. van Emden: Unstructured Systematic Programming, Research Report CS-76-09, Dept. of Computer Science, University of Waterloo, 1976.