AUTOMATIC PROGRAM SYNTHESIS IK SECOND-ORDER LOGIC

Jared L. Darlington

Gesellschaft fur Mathematik und Datenverarbeitung

Bonn, Germany

Abetract

A resolution-based theorem prover, incorporating a restricted higher-order unification algorithm, has been applied to the automatic synthesis of SN0B0L-4 programs. The set of premisses includes second-order assignment and iteration axioms derived from those of Hoare. Two examples are given of the synthesis of programs that compute elementary functions.

Descriptive Terms

Higher-order logic
Program generation
Program synthesis
Resolution
Theorem proving
Unification algorithms

The automatic synthesis of computer programs, like their automatic verification, requires a set of rules *or* axioms to account for such typical program features as assignment, iteration and branching, and a program capable of making appropriate deductions on the basis of these rules or axioms. .Following current work of Luckham and Buchanan and Manna and Vuillemin we have chosen a set of axioms based on those of Hoare[5], and we are employing a resolution-based theorem prover incorporating a reetricted higher-order unification algorithm to generate SNOBOL-4 programs from this set.

To aid the formulation of statements about programs, Hoare invented the notation

$$P \{Q\} R$$

whose interpretation 1st "If the assertion P is true before initiation of a [piece of)program Q, then the assertion R will be true on its completion'"*. ThUs, his "axiom of assignment"

DO $\quad P_n \{x,- f] P$

says that if P is true before f is assigned to *x* then P will be true after this assignment, where "x la a variable identifier", "f *is* an expression" and "P is obtained from P by substituting f for all occurrances of x"j and his "rule of iteration"

D5 $\quad$ If $\vdash P \wedge B \{S\} P$ then $\vdash P \{\text{while } B \text{ do } S\} \neg B \wedge P$

Bays that if P is "an assertion which is always true on completion of S, provided that it is also true on initiation", then "P will still be true after any number of iterations of the statement S (even no iterations)". The controlling condition B *or* the "while loop" may be assumed to be true before exeoution of S, and will be false upon termination of the loop.

The above notation is, for all its elegance, not the most convenient for use by a predicate calculus theorem prover. Accordingly, we have written our axioms for assignment and iteration i terms of the variables x_1, x_2, etc. ranging over program variables and expressions; s_1, s_2, etc. ranging over states, and f_1, f_2, etc. ranging ove predicates or functions of these objects, depending on context. Since the variables x_i and s_i are taken to be of type "individual", the use of the implicitly universally quantified variables f_i ranging over predicates or functions of these *objects means that our language is second-order.* The notation

$$IN(f_1(x_1), s_1)$$

is used to indicate that the assertion $f_1(x_1)$ is true in the state s_1, while

$$IN(f_1(x_1), s_1(s_2))$$

means that $f_1(x_1)$ is true in the state consisting of s_1 followed by s_2 (*this is the reverse of the* usual practice of reading the order of states fro right to left, *or* "inside out"). In terms of thi notation, our assignment axiom

DO′ $\quad IN(f_1(x_1), s_1) \supset IN(f_1(x_2), s_1(x_2 := x_1))$

says that if assertion $f_1(x_1)$ is true in state s_1 then assertion $f_1(x_2)$ will be true in the state resulting from assigning the program variable or expression x_1 to the program variable x_2, following s_1. DO′ may be related to DO by noting that '$f_1(x_1)$' corresponds to 'P_0', '$f_1(x_2)$' to 'P', 'x_1' to '*f*' and 'x_2' to 'x'. Its meaning may be illustrated by the instance

$$IN(IDENT(J,'STRING'), s_1)$$

$$\supset IN(IDENT(K,'STRING'), s_1(K := J))$$

derived from DO′ by instantiating 'x_1' and 'x_2', respectively, by the program variables 'J' and 'K', and 'f_1' by the lambda-function

$$\lambda u.IDENT(u,'STRING')$$

where *IDENT is a primitive SNOBOL function that* tests whether two objects of any data type have the *same* value. The statement says that if IDENT(J,'STRING') is positive in s_1 then IDENT(K,'STRING') will be positive if J is assigned to K in the state following s_1.

In applying DO′ to program synthesis, we hav found it necessary to supplement it with axioms expressing properties of IDENT, namely

ID1 $\quad IN(IDENT(x_1,x_2), s_1)$

$\quad\quad \supset IN(IDENT(f_1(x_1), f_1(x_2)), s_1)$

ID2 $\quad IN(IDENT(x_1,x_1), s_1)$

In practice it is simpler to use two corollaries of DO′, namely

CL1 $\quad \mathrm{IN}(\mathrm{IDENT}(x_1,x_2),\ s_1)$

$$\supset \mathrm{IN}(\mathrm{IDENT}(x_3,\ f_1(x_2)),\ s_1(x_3 := f_1(x_1)))$$

CL2 $\quad \mathrm{IN}(\mathrm{IDENT}(x_1,x_2),\ s_1(x_1 := x_2))$

derived by resolving

$$\mathrm{IN}(f_2(x_3),\ s_2) \supset \mathrm{IN}(f_2(x_4),\ s_2(x_4 := x_3))$$

(obtained from DO′ by change of variable) with ID1 to yield CL1, under the substitutions

$$f_2 = \lambda u_1.\mathrm{IDENT}(u_1,f_1(x_2))$$

$$x_3 = f_1(x_1)$$

$$s_2 = s_1$$

and with ID2 to yield CL2, under the substitutions

$$f_2 = \lambda u_1.\mathrm{IDENT}(u_1,x_1)$$

$$x_3 = x_1$$

$$s_2 = s_1$$

followed by restandardising the variables of the resolvents.

The iteration axiom used by our program is related to Hoare's D3, but is based more directly on the inductive principle for natural numbers. We start with the formula

D3′ $\quad \mathrm{IN}(f_1(x_1),\ s_1)$

$$\wedge\ \forall s_3(\mathrm{IN}(f_1(x_1),\ s_3) \supset \mathrm{IN}(f_1(x_1),\ s_3 s_2))$$

$$\supset.\ \mathrm{IN}(f_1(x_1) \wedge f_2(x_2),\ s_1(\text{while} \neg f_2(x_2)\ \text{do}\ s_2))$$

where '$f_1(x_1)$' corresponds to 'P' in D3, '$f_2(x_2)$' to '$\neg$B' and 's_2' to '{S}'. We then replace 'x_1' in D3′ by a program variable 'J', which in fact serves as a counter taking only integer values, replace 'x_2' by a program variable 'N', and partially Skolemise the result, in the process replacing 's_3' by a constant 's_o', thereby obtaining

D3″ $\quad \mathrm{IN}(f_1(J),\ s_1)$

$$\wedge\ \mathrm{IN}(f_1(J),\ s_o) \supset \underline{\mathrm{IN}(f_1(J),\ s_o s_2)}$$

$$\supset.\ \mathrm{IN}(f_1(J) \wedge f_2(N),\ s_1(\text{while}\ \neg f_2(N)\ \text{do}\ s_2))\ .$$

Resolving on the underscored literal with the following instance of DO′

$$\mathrm{IN}(f_1(J+1),\ s_1 s_2) \supset \mathrm{IN}(f_1(J),\ s_1(s_2(J := J+1)))$$

yields

D3‴ $\quad \mathrm{IN}(f_1(J),\ s_1)$

$$\wedge\ \mathrm{IN}(f_1(J),\ s_o) \supset \mathrm{IN}(f_1(J+1),\ s_o s_2)$$

$$\supset \mathrm{IN}(f_1(J) \wedge f_2(N),\ s_1(\text{while}\ \neg f_2(N)\ \text{do}\ s_2(J := J+1)))$$

thereby introducing the operation of incrementing J into the while loop. The iteration axiom used in our program,

CL3 $\quad \mathrm{IN}(\mathrm{IDENT}(J_n,0),\ s_1)$

$$\wedge\ \mathrm{IN}(f_1(0),\ s_1)$$

$$\wedge\ \mathrm{IN}(f_1(J_n),\ s_o) \supset \mathrm{IN}(f_1(J_n+1),\ s_c s_2)$$

$$\supset \mathrm{IN}(f_1(J_n) \wedge f_2(N),\ s_1((\mathrm{RULE}_n\ \ f_2(N)\ \ :S(\mathrm{EXIT}_n))$$

$$(s_2((\ \ J_n = J_n + 1\ \ :(\mathrm{RULE}_n))(\mathrm{EXIT}_n)))))$$

is derived from D3‴ by adding a subscript 'n' to 'J', adding the literal '$\mathrm{IN}(\mathrm{IDENT}(J_n,0),\ s_1)$' and then replacing '$J_n$' in the erstwhile first literal by '0', thereby setting the counter J_n to 0 in the entering state s_1, and finally replacing the "while" notation by the corresponding SNOBOL terminology of rule names and go-to's. The subscript 'n', in the contexts 'J_n', 'RULE_n' and 'EXIT_n', is instantiated by an integer, the serial number of the clause generated, whenever CL3 enters into a resolution.

The use of axioms like Cl1 - Cl3 in a resolution-based theorem prover requires a unification procedure capable of handling functional variables. Pietrzykowski and Jensen[5] have formulated an algorithm that generates all unifiers for any two strings in ω-order logic, and Huet[6] has an algorithm that generates all the unifiers required by his deductively complete ω-order system of "constrained resolution", but since (as previously indicated) we are dealing with a second-order language it is sufficient for our purposes to use an algorithm devised primarily for first- and second-order logic. The unification algorithm that we employ is based on that for first-order logic[7], but it introduces the higher-order operations of lambda-abstraction, in unifying for example P(a,b) and $f_1(x_1)$ under the substitutions $x_1 = a$, $f_1 = \lambda u_1.P(u_1,b)$, and lambda-normalisation, in unifying for example $\lambda u_1.P(u_1,b).a$ and $P(x_1,x_2)$ under the substitutions $x_1 = a$, $x_2 = b$. In describing the algorithm, we borrow from Pietrzykowski and Jensen the notion of a "progress triple"

$$\langle A,\ B,\ \sigma \rangle$$

where A and B are two strings to be unified and where σ is the set of unifying substitutions, if any, so far generated. In our usage, A and B need not be single well-formed-formulae but may be strings of wffs, so any progress triple $\langle A,\ B,\ \sigma \rangle$ may be written as

$$\langle a_1\ a_2,\ b_1\ b_2,\ \sigma \rangle$$

where a_1 and b_1 are respectively the leftmost wffs in A and B and where a_2 and b_2 are respectively the remaining strings of wffs in A and B. The parenthesisation of A and B follows that of Robinson[12] according to which, for example,

$$(a + (b\ .\ c))$$

is written as

$$((+a)((.b)c))$$

though, in contrast to the system described in his paper, we retain the lambda-notation. As a programming convenience we may disregard any previously unified substrings of A and B, so that at any given stage in the unification A and B represent the not-yet-unified remainders of the original A and B. The object of unification is to reduce A and B simultaneously to the null string, under a set of unifying substitutions σ. In the first-order case there will be at most one way of doing this, represented by the terminal progress triple $\langle \_, \_, \sigma \rangle$, while in the higher-order case there will normally be several, and sometimes infinitely many (as in the case where $A = f_1(F(a))$, $B = F(f_1(a))$, a is of type "individual" and f_1 and F are of type "individual $\rightarrow$ individual"), successful unifications

$$\langle \_, \_, \sigma_1 \rangle, \cdots\cdots, \langle \_, \_, \sigma_n \rangle \cdot$$

The basic way of unifying two subobjects a_1 and b_1 is to set b_1 equal to a_1 where a_1 is a variable (or vice versa if b_1 is a variable), provided that a_1 and b_1 agree in type. In our system the variables x_i agree in type with other x_i and with expressions denoting program variables or integers; the s_i agree with other s_i and with expressions denoting states, and the f_i agree with other f_i and with lambda-functions. The actual algorithm is given below; it is the current version of what we have called "f-matching" in earlier publications.

<u>To unify two strings A and B</u>, define S as the string of progress triples so far generated, and set

$$S = \langle A, B, \sigma \rangle$$

where σ is null. GOTO START.

START: Read in a progress triple $T = \langle A, B, \sigma \rangle$ from S.
SUCCEED: GOTO S1. FAIL: GOTO EXIT.

S1: If A and B are null, T represents a successful unification.
SUCCEED: GOTO START. FAIL: GOTO S2.

S2: If A and B are headed by opposing constants, T represents an unsuccessful unification.
SUCCEED: GOTO START. FAIL: GOTO S3.

S3: $T = \langle a_1\ a_2,\ b_1\ b_2, \sigma \rangle$ where a_1 and b_1 are respectively the leftmost wffs in A and B and a_2 and b_2 are respectively the (possibly null) remaining strings of wffs in A and B. If $a_1 = b_1$, or if $a_1 = x_u$ or $b_1 = x_u$ where x_u is a dummy variable, set

$$T = \langle a_2, b_2, \sigma \rangle$$

SUCCEED: GOTO S1. FAIL: GOTO S4.

S4: If $T = \langle f_i\ a_{21}\ a_{22}, \lambda u_j . B\ \ b_2, \sigma \rangle$ form the string of progress triples

$$T' = \langle f_i\ a_{21}\ a_{21}\ a_{22}, \lambda u_j . \bar{B}(u_j/c_1)\ c_1\ b_2,$$

$$\cdots\cdots\cdots\cdots$$

$$\langle f_i\ a_{21}\ a_{21}\ a_{22}, \lambda u_j . \bar{B}(u_j/c_n)\ c_n\ b_2,$$

where $c_1 \ldots c_n$ are the distinct occurrences of wffs (other than variables bound by λ) in B and where $\bar{B}(u_j/c_i)$ results from B by replacing c_i by u_j. Add T′ to S. Switch a_1 with b_1 and a_2 with b_2 and re-execute S4.
GOTO S5.

S5: If a_1 is a variable, if a_1 and b_1 agree in type, and if a_1 does not occur in b_1, substitute b_1 for a_1 in a_2 and b_2, using lambda-normalisation where possible, and set

$$T = \langle a_2, b_2, \sigma ; a_1 = b_1 \rangle$$

SUCCEED: GOTO S1. FAIL: GOTO S6.

S6: Switch a_1 with b_1 and a_2 with b_2 and re-execute S5.
SUCCEED: GOTO S1. FAIL: GOTO S7.

S7: If $T = \langle (a_{11}\ a_{12})\ a_2,\ b_1\ b_2, \sigma \rangle$ where the head of a_{11} is f_i, form the string of progress triples

$$T' = \langle a_{11}\ a_{12}\ a_2, \\ \lambda u_{n+1} . \bar{b}_1(u_{n+1}/c_1)\ \ c_1\ b_2, \sigma \rangle$$

$$\cdots\cdots\cdots$$

$$\langle a_{11}\ a_{12}\ a_2, \\ \lambda u_{n+1} . \bar{b}_1(u_{n+1}/c_n)\ \ c_n\ b_2, \sigma \rangle$$

where n is the largest subscript on a 'u' occurring in b_1, $c_1 \ldots c_n$ are the distinct occurrences of wffs (other than variables bound by λ) in b_1, and $\bar{b}_1(u_{n+1}/c_i)$ results from b_1 by replacing c_i by u_{n+1}. Add T′ to S. Switch a_1 with b_1 and a_2 with b_2 and re-execute S7.
GOTO S8.

S8: If $a_1 = (a_{11}\ a_{12})$ and $b_1 = (b_{11}\ b_{12})$, set

$$T = \langle a_{11}\ a_{12}\ a_2,\ b_{11}\ b_{12}\ b_2, \sigma \rangle$$

If not, T represents an unsuccessful unification.
SUCCEED: GOTO S1. FAIL: GOTO START.

It may be noted that this algorithm is, except for S4 and S7, equivalent to the usual first-order unification algorithm.

To give an example, applying the algorithm to

$$\langle (f_1 x_1), a, \_ \rangle$$

yields the intermediate progress triples

$$\langle f_1\ x_1, \lambda u_1 . u_1\ \ a, \_ \rangle$$

$$\langle f_1\ x_1, \lambda u_1 . a\ \ x_u, \_ \rangle$$

and finally the unifiers

$$\langle \_, \_, f_1 = \lambda u_1 . u_1 ; x_1 = a \rangle$$

$$\langle \_, \_, f_1 = \lambda u_1 . a \rangle$$

By way of comparison with Pietrzykowski and Jensen, these are the same unifiers that are generated by their "imitation" and "projection" rules, which operate directly on the disagreeing objects to reduce the area of disagreement, but our algorithm has no equivalent of their "elimination" and "iteration" rules, which operate on variables outside the area of disagreement. Thus, it will not handle the triple

$$\langle (f_1 a), (f_1 b), \_ \rangle$$

since elimination is required to produce the unifier

$$\langle \_, \_, f_1 = \lambda u_1 . x_1 \rangle$$

In the application of second-order logic to program synthesis we have so far found no need for an equivalent of the elimination rule, nor do we need an iteration rule since by definition this applies only to languages of order higher than two (see Pietrzykowski and Jensen8 for the formal definitions of these rules). Huet9 also dispenses with these latter two rules, but in a way that preserves deductive completenees. Our algorithm is actually an incomplete one for w-order, rather than just second-order, logic, the essential restriction being that, since the unification of fj(x.,x„,..,,x) and B proceeds by trying to match the xi with well-formed pieces of B, B must actually contain well-formed pieces of the same type as the xi. Our program has in fact proved some theorems of order higher than two, such as the example in section 5 of Pietrzykowski's and Jensen's paper1 .

The generation of resolvents, like the unification algorithm, is based on the first-order prooedure, in that the substitutions generated in the course of unifying two literals are applied to the disjunctively connected "remainders" (that is, literals not being resolved on) of the two clauses being resolved, the one important difference being that lambda-normalisation is applied during the substitution prooess in order to eliminate lambda-funations from the resolvent wherever possible. The order of generating resolvente is based on the "S-L resolution" method for first-order logic of Kowalski and Kuehner11, and is essentially a "depth-first" search strategy with resolution only on first literals of clauses, but with the aid of a set of reductive rules that perform algebraic simplifications or transformations on the clauses generated. Among the problems solved by our program are the construction of SNOBOL programs for computing the faotorlal function and for iterative division! the computer printouts are given following the text. By way of explanation, axioms CL1 - CL3 are equivalent to those given in the text, but with the order of literals changed and with some implications expressed in terms of conJunction and negation, for more effective application of "first-literal resolution" and for ensuring that the resulting proofs will be linear. CL4 in each of the examples formulates the problem) H and N are input variables, and J and K are output variables. In example 1, CL4 says that if there is a state S1 in whioh K has the same value as J I where J has the same value as N (in other words, if K has the same value as Nt), then s1 is an answer. In example 2, CL4 says that if K has the same value as H - (Jn N) where the value of K is less than that of N (in order wordB, if H ■ J .N + K, K< N) in s1, then s1 is an answer. Both of these examples were taken from Manna and Waldinger13. Of the reductive rules exhibited by the examples, Rule 2 transforms (x+1)! into (x!).(x+1); Rule 3 transforms x - ((y+1).z)xinto {x - y.z) - z; Rule04 reduces 0.x, x.O or 0 to 0| Rule 5 reduces x or 01 to 1t Rule 6 reduces 0+x, x+0 or x-0 to x, and Rule 8 is a "frame" rule that reduces IN(IDERT(x.,x2), e.(s2)) to IN(IDENT(X1,X2), s1), provided that s2 "does not affeot" x. or x2. Host of these rules are purely ad hoo and are chosen with the particular examples in view; for serious program synthesis one would need a more systematically organised algebraic aimplifier. Finally, there is a routine called "ANSPRIST" that polishes up the answer and prints it out in the correct SHOBOL line-by-line format, though it may be noted that the sample programs are not as "simple" as they could be. Running times for examples of this general type are three to five minutes on the IBM 360/50 at the GMD in Bonn. The theorem-proving program, like the programs generated, is coded in SH0B0L-4* Apart from line divisions, the output is an exact transcription of the computer printouts.

References

1. Luckham, D. C. and Buchanan, J. R. Automatic Generation of Simple Programs; a Logical Basis and Implementation. Artificial Intelligence Projeot Report, Stanford University, 1973.

2. Manna, Z. and Vuillemin, J. Fixpoint approach to the theory of computation. Comm. ACM 15. 526-536, 1972.

3. Hoare, C. A. R. An axiomatio basis for computer programming. Comm. ACM 12, 576-580, 583, 1969.

4. Reference 3, p. 577-

5. Pietrzykowski, T. and Jensen, D. A Complete Mechanization of w-order Logic. Report CSRR-2060, University of Waterloo, 1972.

6. Huet, G. P. A Unification Algorithm for Type Theory. IRIA Laboria, 1973.

7. Robinson, J. A. A machine-oriented logic based on the resolution principle. Jour. ACM 12, 23-41, 1965.

6. Reference 5.

9. Huet, G. P. Constrained Resolutioni A Complete Method for Higher Order Logic. Report 1117, Case WeBtern Reserve University, 1972.

10. Reference 5* PP. 30-32.

11. Kowalski, R. and Kuehner, D. Linear resolution with selection function. Artificial Intelligence 2. 227-260, 1971 -

12. Robinson, J. A. A note on mechanizing higher-order logic Machine Intelligence 5. ed. Meltzer, B. and Michle, D., Edinburgh University Press, 123-153, 1969.

13. Manna, Z. and Waldinger, R. J. Toward automatic program synthesis. Comm. ACM 14« 151-165, 1971.

__Example 1__. Construct a program to compute K such that K = N!

```
CL1 = (¬((AND((IN((IDENTX1)X2))S1))(¬((IN((IDENTX3)(F1X2)))(S1(   X3 = (F1X1)))))   AXIOM

CL2 = ((IN((IDENTX1)X2))(S1(   X1 = X2)))   AXIOM

CL3 = ((IN((AND(F1JN))(F2N)))(S1((RULEN   (F2N)   :S(EXITN))(S2((   JN = JN + 1   :(RULEN))(EXITN)))))
      ((AND((IN(F1JN))SC))(¬((IN(F1((+JN)1)))(SCS2))))
      (¬((IN(F10))S1))
      (¬((IN((IDENTJN)0))S1))   AXIOM

CL4 = (¬((IN((AND((IDENTK)(FACTORIALJN)))((IDENTJN)N)))S1))
      (ANSS1)   AXIOM

CL5 = ((AND((IN((IDENTK)(FACTORIALJ5)))SC))(¬((IN((IDENTK)(FACTORIAL((+J5)1))))(SCS1))))
      (¬((IN((IDENTK)(FACTORIAL0)))S2))
      (¬((IN((IDENTJ5)0))S2))
      (ANS(S2((RULE5   ((IDENTJ5)N)   :S(EXIT5))(S1((   J5 = J5 + 1   :(RULE5))(EXIT5)))))) FROM CL3 CL4

          F1 = (*LU1((IDENTK)(FACTORIALU1)))
          F2 = (*LU1((IDENTJN)U1))
          S5 = (S1((RULEN   ((IDENTJN)N)   :S(EXITN))(S2((   JN = JN + 1   :(RULEN))(EXITN)))))

CL6 = ((AND((IN((IDENTK)(FACTORIALJ5)))SC))(¬((IN((IDENTK)((.(FACTORIALJ5))((+J5)1))))(SCS1))))
      (¬((IN((IDENTK)(FACTORIAL0)))S2))
      (-((IN((IDENTJ5)0))S2))
      (ANS(S2((RULE5   ((IDENTJ5)N)   :S(EXIT5))(S1((   J5 = J5 + 1   :(RULE5))(EXIT5)))))) FROM RULE2
         CL5

CL7 = ((AND((IN((IDENTK)(FACTORIALJ5)))SC))(¬((IN((IDENTK)((.(FACTORIALJ5))((+J5)1))))(SCS1))))
      (¬((IN((IDENTK)1))S2))
      (¬((IN((IDENTJ5)0))S2))
      (ANS(S2((RULE5   ((IDENTJ5)N)   :S(EXIT5))(S1((   J5 = J5 + 1   :(RULE5))(EXIT5))))))  FROM RULE5
         CL6

CL8 = (¬((IN((IDENTK)1))S1))
      (¬((IN((IDENTJ5)0))S1))
      (ANS(S1((RULE5   ((IDENTJ5)N)   :S(EXIT5))((   K = ((.K)((+J5)1)))((   J5 = J5 + 1
         :(RULE5))(EXIT5))))))   FROM CL1 CL7

          X1 = K
          X2 = (FACTORIALJ5)
          S1 = SC
          X3 = K
          F1 = (*LU1((.U1)((+J5)1)))
          S5 = (   K = ((.K)((+J5)1)))

CL9 = (¬((IN((IDENTJ5)0))(S1(   K = 1)))
      (ANS((S1(   K = 1))((RULE5   ((IDENTJ5)N)   :S(EXIT5))((   K = ((.K)((+J5)1)))
         ((   J5 = J5 + 1   :(RULE5))(EXIT5))))))   FROM CL2 CL8

          X1 = K
          X2 = 1
          S5 = (S1(   K = 1))

CL10 = (¬((IN((IDENTJ5)0))S1))
       (ANS((S1(   K = 1))((RULE5   ((IDENTJ5)N)   :S(EXIT5))((   K = ((.K)((+J5)1)))
          ((   J5 = J5 + 1   :(RULE5))(EXIT5))))))   FROM RULE8 CL9

CL11 = (ANS(((S1(   J5 = 0))(   K = 1))((RULE5   ((IDENTJ5)N)   :S(EXIT5))((   K = ((.K)((+J5)1)))
          ((   J5 = J5 + 1   :(RULE5))(EXIT5))))))   FROM CL2 CL10

          X1 = J5
          X2 = 0
          S5 = (S1(   J5 = 0))

ANS
S1
   J5 = 0
   K = 1
RULE5   IDENT(J5,N)   :S(EXIT5)
   K = K * (J5 + 1)
   J5 = J5 + 1   :(RULE5)
EXIT5
END
```

Example 2. Construct a program to compute K such that M = J.N + K, K< N

```
CL1 = (¬((AND((IN((IDENTX1)X2))S1))(¬((IN((IDENTX3)(F1X2)))(S1(   X3 = (F1X1))))))   AXIOM

CL2 = ((IN((IDENTX1)X2))(S1(   X1 = X2)))   AXIOM

CL3 = ((IN((AND(F1JN))(F2N)))(S1((RULEN   (F2N)   :S(EXITN))(S2((   JN = JN + 1   :(RULEN))(EXITN)))))
      ((AND((IN(F1JN))SC))(¬((IN(F1((+JN)1)))(SCS2))))
      (¬((IN(F10))S1))
      (¬((IN((IDENTJN)0))S1))   AXIOM

CL4 = (¬((IN((AND((IDENTK)((-M)((.JN)N))))((LTK)N)))S1))
      (ANSS1)   AXIOM

CL5 = ((AND((IN((IDENTK)((-M)((.J5)N))))SC))(¬((IN((IDENTK)((-M)((.((+J5)1))N))))(SCS1))))
      (¬((IN((IDENTK)((-M)((.0)N))))S2))
      (¬((IN((IDENTJ5)0))S2))
      (ANS(S2((RULE5   ((LTK)N)   :S(EXIT5))(S1((   J5 = J5 + 1   :(RULE5))(EXIT5))))))   FROM CL3 CL4

         F1 = (*LU1((IDENTK)((-M)((.U1)N))))
         F2 = (*LU1((LTK)U1))
         S5 = (S1((RULEN   ((LTK)N)   :S(EXITN))(S2((   JN = JN + 1   :(RULEN))(EXITN)))))

CL6 = ((AND((IN((IDENTK)((-M)((.J5)N))))SC))(¬((IN((IDENTK)((-((-M)((.J5)N)))N)))(SCS1))))
      (¬((IN((IDENTK)((-M)((.0)N))))S2))
      (¬((IN((IDENTJ5)0))S2))
      (ANS(S2((RULE5   ((LTK)N)   :S(EXIT5))(S1((   J5 = J5 + 1   :(RULE5))(EXIT5))))))  FROM RULE3 CL5

CL7 = ((AND((IN((IDENTK)((-M)((.J5)N))))SC))(¬((IN((IDENTK)((-((-M)((.J5)N)))N)))(SCS1))))
      (¬((IN((IDENTK)((-M)0)))S2))
      (¬((IN((IDENTJ5)0))S2))
      (ANS(S2((RULE5   ((LTK)N)   :S(EXIT5))(S1((   J5 = J5 + 1   :(RULE5))(EXIT5))))))  FROM RULE4 CL6

CL8 = ((AND((IN((IDENTK)((-M)((.J5)N))))SC))(¬((IN((IDENTK)((-((-M)((.J5)N)))N)))(SCS1))))
      (¬((IN((IDENTK)M))S2))
      (¬((IN((IDENTJ5)0))S2))
      (ANS(S2((RULE5   ((LTK)N)   :S(EXIT5))(S1((   J5 = J5 + 1   :(RULE5))(EXIT5))))))  FROM RULE6 CL7

CL9 = (¬((IN((IDENTK)M))S1))
      (¬((IN((IDENTJ5)0))S1))
      (ANS(S1((RULE5   ((LTK)N)   :S(EXIT5))((   K = ((-K)N))((   J5 = J5 + 1   :(RULE5))(EXIT5))))))
         FROM CL1 CL8

         X1 = K
         X2 = ((-M)((.J5)N))
         S1 = SC
         X3 = K
         F1 = (*LU1((-U1)N))
         S5 = (   K = ((-K)N))

CL10 = (¬((IN((IDENTJ5)0))(S1(   K = M))))
       (ANS((S1(   K = M))((RULE5   ((LTK)N)   :S(EXIT5))((   K = ((-K)N))((   J5 = J5 + 1
          :(RULE5))(EXIT5))))))   FROM CL2 CL9

         X1 = K
         X2 = M
         S5 = (S1(   K = M))

CL11 = (¬((IN((IDENTJ5)0))S1))
       (ANS((S1(   K = M))((RULE5   ((LTK)N)   :S(EXIT5))((   K = ((-K)N))((   J5 = J5 + 1
          :(RULE5))(EXIT5))))))   FROM RULE8 CL10

CL12 = (ANS(((S1(   J5 = 0))(   K = M))((RULE5   ((LTK)N)   :S(EXIT5))((   K = ((-K)N))
          ((   J5 = J5 + 1   :(RULE5))(EXIT5))))))   FROM CL2 CL11

         X1 = J5;  X2 = 0;  S5 = (S1(   J5 = 0))

ANS
S1
   J5 = 0
   K = M
RULE5   LT(K,N)   :S(EXIT5)
   K = K - N
   J5 = J5 + 1   :(RULE5)
EXIT5
END
```