A SYSTEM TO UNDERSTAND INCORRECT PROGRAMS

Harald Wertz

Universite de Paris VIII (Vincennes)
route de la tourelle
75571 Paris

Abstract :
This paper presents a system (PHENARETE) which understands and improves incompletely defined LISP programs, such as those written by students beginning to program in LISP. This system takes, as input, the program without any additional information. In order to understand the program, the system meta-evaluates it, using a library of "pragmatic rules", describing the construction and correction of general program constructs, and a set of "specialists", describing the syntax and semantics of the standard LISP functions. The system can use its understanding of the program to detect errors in it, to debug them and, eventually, to justify its proposed modifications. This paper gives a brief survey of the working of the system, emphasizing on some commented examples.

A lot of effort is actually spent on the developement of tools to help programmers in constructing, debugging and verifying programs. Unfortunately most of these tools
- impose too much constraints on the intuitions of the programmer [cf DIJKSTRA 1976],
- are working only on a very limited subset of possible programs [cf RUTH 1974],
- are only working on correct programs [cf ARSAC 1977, IGARASHI et al 1975].

Our aim is two-fold :

-1- to make explicit the knowledge involved in constructing and debugging programs and

-2- to verify - not the correctness of programs - but their "consistency" and to provide "hints" for improving and correcting their programs to the programmer.

To this end we have build our system on four main concepts :
1 an algorithm of meta-evaluation [cf GOOSSENS 1978] to help the system to understand each of the possible paths of the program,
2 a set of "specialists", i.e. a set of procedural specifications of the syntax and the operational semantics of the standard LISP functions,

example : specialist CAR for the syntax

```
[CAR-1 (X) =>
        v (& atom (CAR X)
             & type (X) = LISTP)
        v (& S-expression (CAR X)
             & type (val (X)) = LISTP)
  else :
             modify X until CAR-1 (X) = T]
```

paraphrasing :

CAR expects that its argument is
- an atom
  and the type of the value of the argument is a list
- a S-expression
  and the type of the value of that S-expression is a list

else
CAR has to modify the argument until one of these two conditions is true

and the specialist CAR for the semantics

```
[CAR-N =>
      arg : (X (meta-eval X))
     test : (type (val (X)) = LISTP) ->
            (type (val (X)) = ?)
                   -> hypothesize (X, type LISTP)
              T -> complain (X, type : LISTP)
   action : if (existe (CAR X)) --> (CAR X)
            else (create (CAR, X)) --> (CAR X)]
```

or in paraphrasing :

CAR-N

has an argument named X, which must be evaluated
one must verify
if
the type of value of the argument is a list, all is ok
else
if the type of value of the argument isn't known, one has to create a hypothetical value of type LIST for X
else
one has to ask the debugger to change the text of the program in such a way that the value of X becomes a list

the value of CAR is
if
there exists already a CAR of X, this CAR
else
one has to create a symbolic value for X, the CAR of which will be the desired value

These specialists are the agents of the meta-evaluation and they represent the system's knowledge about the programming language used.

3 a set of "pragmatic rules" describing general program constructs and methods to repair inconsistencies

example :
rule of the dependence of a loop of the predicate =>

if no variable of the exit-test is modified inside the loop, then the loop is independent of the exit-test and, its execution is non-terminating or the loop will never be executed.

The set of these rules expresses the system's general knowledge about the well-formedness of programs and about the correction of errors;

4 during the analysis of a program, PHENARETE contructs some description - an internal representation - of the program under the form of "cognitive atoms". These may be considered as the nodes of a network-like representation of the program actually analysed.

The system accepts every LISP program conforming to the following restrictions :

- partitioning of the names of variables, funtions and labels;
- all funtion calls must be "call by name";
- the unique functional arguments admitted are explicit lambda-expressions.

We call this subset of LISP : extended first order LISP.

To use PHENARETE, the user has to give to the system only the text of the draft version of the program he wants to write, without any additional information like input/output assertions, commentaries, plans etc. The system will try to understand what the user wanted to do, and, if necessary, modify the text of the programm.

To give some feeling of the working of the system, let us examine some examples in detail :
Our first example is a (very) erroneous version of the well known REVERSE function. Here is the actual input to the system :

```
?  (P '(DE REV L1 L2 COND ULL L22 A1 T RVE A1 ONS CRA A1 A2))
```

PHENARETE will first correct the spelling errors :

```
ERREUR:
            NOM --> (? ULL --> NULL)
ERREUR:
            NOM --> (? L22 --> L2)
ERREUR:
            NOM --> (? A1 --> L1)
ERREUR:
            NOM --> (? A1 --> L1)
ERREUR:
            NOM --> (? RVE --> REV)
ERREUR:
            NOM --> (? RVE --> REV)
ERREUR:
            NOM --> (? A1 --> L1)
ERREUR:
            NOM --> (? ONS --> CONS)
```

```
ERREUR:
            NOM --> (? CRA --> CAR)
ERREUR:
            NOM --> (? A1 --> L1)
ERREUR:
            NOM --> (? A2 --> L2)
```

After having very well corrected the spelling errors, PHENARETE proceeds to a first analysis where she uses only her syntactic knowledge. The result of this first analysis is a "syntactically correct" LISP program (i.e. a programm accepted by any smart LISP interpreter or compiler) :

PROPOSITION 1 :

```
(DE REV (L1 L2)
   (COND
      ((NULL L2) L1)
      (T (REV L1 (CONS (CAR L1) L2)))))
```

These first improvements have eliminated all the syntactic errors. Anyway, there subsist two semantic errors :

-1 in the recursive call of rev, the first argument L1 is not modified. This creates an infinit recursion.

-2 even with a modification of L1 in the recursive call, the recursion won't stop either since the stop-test has as argument L2, a list which grows longer and longer in the run of the succesive recursive calls.

PHENARETE can not disambiguate this function - she does not know anything of the intentions of the programmer - so she gives two different propositions :

PROPOSITION 2 :

```
(DE REV (L1 L2)
   (COND
      ((NULL L2) L1)
      ((NULL L1) L2)
      (T (REV (CDR L1) (CONS (CAR L1) L2)))))
```

AT LEAST YOUR FUNCTION SEEMS OK TO ME.

In this first proposition, PHENARETE supposed the stop-test given to be true, but that the user omitted a second stop-test for the case where the second argument is not NULL at the initial call of REV.

P R O P O S I T I O N :

```
(DE REV (L1 L2)
   (COND
      ((NULL L1) L2)
      (T (REV (CDR L1) (CONS (CAR L1) L2)))))
```

AT LEAST YOUR FUNCTION SEEMS OK TO ME.

In this second proposition, PHENARETE supposed that the user inadvertently inverted the arguments of the stop-test, so she inverts the two arguments L1 and L2.
Of the two corrected versions of the initial draft-program PHENARETE is assured that they will stop and deliver a result when running.

Our second example is an extremly "simplified" version of the equally well known function FACTORIAL. Here she is :

```
?  (DE FACT N TIMES N FACT N)
```

As in the previous example, PHENARETE will first translate this unparentesized expression into an well parentesized one :

PROPOSITION 1 :

```
(DE FACT (N) (TIMES N (FACT N)))
```

This first proposition is a syntactically correct program, but semantically it is not very correct :

-1 at the recursive call N is not modified. This is the same kind of error as in the previous example, exept the argument here is of numeric type.

-2 there is no stop-test at all, so there are two (!) reasons to make the recursion infinit.

Remember that PHENARETE doesn't know the intentions of the programmer, so she must detect these errors without any additional information : all she can use in the further analysis are the semantic specialists and the pragmatic rules.
So let us look at her proposition :

P R O P O S I T I O N :

```
(DE FACT (N) (COND
   ((LE N 0) 1)
   (T (TIMES N (FACT (SUB1 N))))))
```

AT LEAST YOUR FUNCTION SEEMS OK TO ME.

This corrected version is actually a correct version of the factorial-program. The performance is really astonishing knowing that the system works completely automatically whithout asking any question to the user and without any information about the supposed intention.

One last (uncommented) example :

```
?  (DE ADDIT M N  ((ZEROP N) M)
                  (T (ADDIT SUB1 M ADD1 N)))

   PROPOSITION 1 :

  (DE ADDIT (M N)
      (COND
         ((ZEROP N) M)
         (T (ADDIT (SUB1 M) (ADD1 N)))))


   P R O P O S I T I O N :
  (DE ADDIT (M N)
      (COND
         ((ZEROP N) M)
         ((LE M 0) N)
         (T (ADDIT (SUB1 M) (ADD1 N)))))

              AT LEAST YOUR FUNCTION SEEMS OK TO ME.
```

Presently we are working on some extensions as to find automatically the intentions and the goals of given pieces of code. We would also like to adjoin to PHENARETE a module permitting to explain the reasoning of the system. This would be a great help to the user.
The system is running on PDP-10, uses about 25k word memory, is implemented in VLISP [CHAILLOUX 1976, GREUSSAY 1977], and is used by about 1000 students in our university.
A more detailed description may be found in [WERTZ 1978].

references

ARSAC J., (1977), La construction de programmes structures, Dunod-Informatique, Paris

CHAILLOUX J., (1976), VLISP-10 manuel de reference, Dept. Informatique, Universite Paris 8, RT-17-76

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

GOOSSENS D., (1978), A System For Visual-Like Understanding of LISP Programs, Proc. AISB/GI Conference, Hamburg, RFA, July 17-19, 1978

GREUSSAY P., (1977), Contribution a la Definition Interpretative et a l'Implementation des Lambda-langages, These, Universite Paris 7.

IGARASHI S., LONDON R.L. & LUCKHAM D.C., (1975), Automatic Program Verification 1 : Logical Basis and its Implementation, Acta Informatica, vol. 4, pp. 145-182.

RUTH G.R., (1974), Analysis of Algorithm Implementations, M.I.T., MAC-TR-130.

WERTZ H., (1978), Un systeme de comprehension, d'amelioration et de correction de programmes incorrects, These de 3eme cycle, Universite Paris 6.