

AUTOMATIC PROGRAM DEBUGGING

Harald WERTZ

C.N.R.S. LA 248 L.I.T.P.
2 place Jussieu
75005 Paris
&
Département d'Informatique
Université Paris VIII - Vincennes
75571 PARIS Cédex 12

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 some commented examples.

1.0 Introduction

Much effort is spent on the development of tools to help programmers in constructing, debugging and verifying programs. From simple editors and trace-packages, the trend is towards more and more sophisticated automatic programmers [3], automatic debuggers [7] automatic assistants [6], automatic verifiers [5] or even the construction of new - semantically more firmly based - programming languages (PASCAL, ALPHARD).

Most of these tools exhibit some weaknesses such as : they impose too many constraints on the intuitions of the programmer or they work only on a very limited subset of possible programs (cf [7]) or they work only on correct programs (cf [5]).

Our aim in the design of our program understanding system was three-fold :

- 1- we wanted to have a system which makes explicit the knowledge involved in constructing and debugging programs;
- 2- we wanted our system not to verify the correctness of programs but their consistency and
- 3- we wanted the system to provide hints for improving and correcting programs.

To this end we have built a program understanding system capable of automatically correcting and improving programs. This system, PHENARETE, assists beginning programmers during the writing and debugging of their programs.

PHENARETE analyses the text of the program and when it detects inconsistencies or informalities, it constructs and proposes possible corrections or improvements. These propositions constitute for the beginning programmer a crucial component of his apprenticeship : basing on examples the

process of improving and correcting imperfect programs, and basing on models the process of inventing future programs.

2.0 Overview of the system

The system takes as input the draft version of a LISP program and delivers as result of its treatment one or more versions of the same program, corrected and improved. PHENARETE proceeds in several steps : first it effects a preliminary analysis of the text of the program to detect and correct surface errors. Surface errors are errors detectable by local analysis : simple syntax errors such as misspellings or parenthesis errors and errors concerning the wrong number of arguments in function calls. During this first analysis PHENARETE collects information for the following steps. Every time it meets the name of a function it automatically activates a specialist, associated with this function, by the process of data driven function invocation. These specialists represent its knowledge about the use and the effects of the associated functions.

The result of the first analysis - a syntactically correct program - is then meta-evaluated to verify the programs well-formedness. We say that a program is well-formed if it doesn't contain evident infinite loops, doesn't have statements never executed and if it doesn't contradict the set of rules incorporated in PHENARETE. Every proposition our system delivers is a well-formed program.

When PHENARETE detects some informalities or inconsistencies, it annotates the corresponding part of the program and sends the code and the annotation to the repair-man, a module designed to eliminate errors. The system stops the analysis process when it doesn't find any more possible improvements.

3.0 Organisation of PHENARETE

The system is based on four main concepts :

- 1- During the analysis of a program, PHENARETE constructs an internal representation of the program in the form of cognitive atoms. These may be considered as the nodes of a network-like representation of the program. Each cognitive atom is composed of a set of facets which represent its different aspects, about which questions could be asked. The facets are filled in during the analysis process.
We distinguish between three classes of cognitive atoms, those concerning variables, those concerning labels and those concerning functions.
- 2- A set of specialists, i.e. a set of procedural specifications of the syntax and the operational semantics of the standard LISP functions.
We distinguish between two different types of specialists :
 - those describing the syntax and
 - those describing the semanticsof the associated 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 (e.g. a function call) 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 (exist (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 call the repair-man 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.

Note that the internal description constructed during the analysis of each user function is used to construct two new such specialists for each user function.

The specialists are the agents of the meta-evaluation and they represent the systems knowledge about the programming language used.

- 3- An algorithm of meta-evaluation [2] which helps the system to understand each of the possible paths of the program. This algorithm includes modules to determine the symbolic values on which the evaluation is carried on.
- 4- A set of pragmatic rules describing general program constructs and stereotyped methods to repair inconsistencies. These rules formalize and express explicitly the knowledge activated by every programmer when he is reading a program. We do not use rules concerning the task domain of the programs : our intention was to build a system which does not ask for any additional information. The system should work in any possible task domain, numeric as well as symbolic.

examples of pragmatic rules :

(1) rule of the dependence of a loop of the predicate ->
if no variable of the exit-test of a loop is modified inside the loop-body, then the loop is independent of the exit-test and its execution is non-terminating or the loop will never be executed.

A slight refinement of this rule is the following rule :

(2) rule of the structural well-formedness of loops ->
in a loop at least one of the variables used in the exit-test has to change its value inside the body of the loop, in such a way that its structure is simplified and converges towards the satisfaction of the exit-test.

In order to use this rule, we have implemented a small theorem prover which can prove the convergence by induction. For the proof of this rule, the theorem-prover takes the symbolic value of the variables at the entrance of the loop, and the modified symbolic values of the same variables after one meta-evaluation of the loop-body and tries to prove the convergence towards the stop test. Let us give one last example of a pragmatic rule :

(3) rule of the structure of recursive loops ->

in a recursive function, the recursive calls have to be inside of selection-clauses, and at least one of the clauses must not contain a recursive call.

Actually we have about a hundred rules incorporated in the system, dealing especially with iteration and recursion. For almost every rule there exists a dual one indicating how to modify the program in such a way that the first one is satisfied.

4.0 A Commented Example

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, commentary, plans etc. The system will try to understand what the user wanted to do, and, if necessary, modify the text of the program. 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))
```

After having 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 program accepted by any smart LISP interpreter or compiler):

SURFACE IMPROVEMENTS :

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

These first improvements have eliminated all the syntactic errors. However, at least two semantic errors remain:

- 1- in the recursive call of REV, the first argument L1 is not modified. This creates an infinit recursion (rule 1).
- 2- even with a modification of L1 in the recursive call, the recursion won't stop since the stop-test has as argument L2, a list which grows longer and longer on successive recursive calls (rule 2).

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

PROPOSITION 1 :

```
(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.

In this first proposition, PHENARETE supposed the stop-test given to be correct, 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.

PROPOSITION 2 :

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

AT LEAST YOUR FUNCTION SEEMS OK.

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.

PHENARETE is assured that the two corrected versions of the initial draft-program (which are not identical) will stop and deliver a result when executed.

Presently we are working on some extensions to PHENARETE 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 an explanation to the user of 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 [1,4], and is used by about 1000 students in our university. A more detailed description may be found in [8].

5.0 REFERENCES

- [1] CHAILLOUX J. "VLISP-10.3 manuel de référence." Dépt. Informatique, Université Paris 8, RT-17-78, 1978
- [2] GOOSSENS D. "A System For Visual-Like Understanding of LISP Programs." Proc. AISB/GI Conference, Hamburg, RFA, July 17-19, 1978
- [3] GREEN C. & BARSTOW D. "On Program Synthesis Knowledge." Artif. Intell. 10:3 (1978) 241-279
- [4] GREUSSAY P. "Contribution à la Définition Interprétative et à l'Implémentation des Lambda-langages." Thèse, Université Paris 7, 1977
- [5] IGARASHI S., LONDON R.L. & LUCKHAM D.C. "Automatic Program Verification 1: Logical Basis and its Implementation." Acta Informatica, vol. 4, (1975) 145-182.
- [6] RICH C. & SHROBE H.E. "Initial Report on a LISP Programmer's Apprentice." IEEE Transactions on Software Engineering SE-4:6 (1978) 456-467
- [7] RUTH G.R. "Analysis of Algorithm Implementations." MAC-TR-130, M.I.T., Cambridge, 1974
- [8] WERTZ H. "Un système de compréhension, d'amélioration et de correction de programmes incorrects." Thèse de 3ème cycle, Université Paris 6, 1978