

META-INTERPRETATION OF RECURSIVE LIST-PROCESSING PROGRAMS

Daniel GOOSSENS
Université PARIS VIII
Route de la Tourelle
75012 PARIS

Presented is an implemented program understanding system called CAN. CAN relies on a meta-interpretation process which brings to light two new concepts concerning meta-interpretation of recursive programs: vicious circles and constructive induction. These notions express CAN's ability to extract from program code classes of data for which interpretation of the code would not stop, and in general, to relate classes of data with abstract values and environments.

keywords: Meta-interpretation, unification, conceptual representations, program understanding, propagation, vicious circle, constructive induction.

1. INTRODUCTION

Works on program verification [10] [13] have brought many useful techniques for proving properties of programs, but all the implemented systems which derived from them were to be used by expert mathematicians rather than expert programmers. That is, those systems may do nothing with a program on its own. They are intended to be interactively used by programmers who are acquainted with formal specifications manipulations and who understand their programs.

Recent works on program understanding have demonstrated the need of domain dependent knowledge [7] [15] and other types of pragmatic knowledge [19] [20]. However, systems in these areas are not equally efficient. More precisely, the systems which support these works may fail on simple tasks which are beyond the scope of their effective specificity. That is, their domain of application is far from fitting with what an expert programmer would consider as "simple", or "visually understandable".

CAN is a powerful tool for such systems since:

- It understands programs independently of traditional applications for understanding (verification, correction, improvement)
- It is particularly well suited for extracting on its own only what is "visually understandable" in a program. This prevents CAN from getting stuck on

intricate problems. This is a necessity if CAN is expected to be a tool for problem solvers. A part of CAN has been especially developed for the visual understanding of programs which manipulate flat lists.

CAN meta-interprets program bodies in abstract environments [1] [2] [4] [5] [9] [14] [19] [21]. We use conceptual representations [21] in order to represent abstract values of program variables and relations among them. Our conceptual representations are either patterns, or predicate-style assertions which may not be translated into patterns. Our patterns involve element-type and segment-type variables. segment-type variables include index mathematical notations for flat lists. They also involve auto-references. Conceptual representation of meanings is our bypass of the traditional "set of properties". Conceptual representations may be confronted and compared more easily than assertions, most often with unification procedures, which yield valuable and non-trivial information [6]. CAN involves a sophisticated unification process of conceptual representations.

2. META-INTERPRETATION

When confronted with LISP code, the meta-interpreter most often faces the case where a function is applied on its arguments. The meta-interpretation of (F A1 A2 ... An) involves the following steps:

first, CAN meta-interprets each of the Ais which should be evaluated on a simple interpretation. this yields a set of possible abstract values and environments for each Ai.

Second, CAN confronts each combination of these values and environments with F's CAN definition. After this confrontation, CAN knows:

- What conditions must hold on the Ais and the environment so that F may apply
- which features of the abstract values and the environment are relevant for the construction of F's abstract value and side-effects.

Third, CAN propagates this information over the current environment and on the "prototype" abstract value which is part of F's CAN definition.

LISP functions are conceptually defined [21] within CAN. Here are the conceptual definitions of the CAR, CDR and MEMBER functions:

("!" denotes an element variable and "?" a segment variable)

```
(CAR ()) = ()
(CAR (!a ?b)) = !a
(CDR ()) = ()
(CDR (!a ?b)) = (?b)

(MEMBER !x (!a ... !a !x ?b)) = (!x ?b)
1          n

(MEMBER !x (!a ... !a)) = ()
1          n

(when: !a = !x; i[1,n])
i
```

That is, the CAR function returns the first element of a list, the CDR function returns the list without its first element. (MEMBER X Y) tests whether the element X is contained in the list Y, at the top-level. If it is the case, MEMBER returns the part of the list which starts from the first occurrence of X.

CAN also handles the cases where F is a control structure, a functional variable, an escape label, or an unknown atom. The meta-interpretation of a LISP program is expected to yield a conceptual representation of its meaning.

Example:

Suppose that F is MEMBER, and that the meta-interpretation of its two arguments gave the two abstract values: (on !1 !2) and (?3 !1). The application of MEMBER on "(on !1 !2)" and "(?3 !1)" amounts to checking whether the list (?3 !1) contains the structure (on !1 !2). The confrontation of these values with MEMBER's prototype values is performed by a unification process of conceptual representations [6].

The unification of

$$[!x \quad (!a \dots !a \quad !x \quad ?b)]$$

1 n

with $[(on \ !1 \ !2) \quad (?3 \ !1)]$
(where !a = !x) gives only one case:

$$\begin{aligned} !x &= (on \ !1 \ !2) \\ ?3 &= !a \dots !a \quad (on \ !1 \ !2) \quad ?b1 \\ &\quad 1 \qquad n \\ ?b &= ?b1 \ !1 \end{aligned}$$

since !1 in (?3 !1) may not contain (on !1 !2).

The propagation of this information on the prototype value (!x ?b) of MEMBER gives the returned abstract value: ((on !1 !2) ?b1 !1). This value is still functionally related to the environment since ?3 is also replaced in the environment by what it is associated to. The reader is invited to perform the same task for the second rule of MEMBER's definition.

3. UNIFICATION OF CONCEPTUAL REPRESENTATIONS

The problem is, given two conceptual representations, to find the complete set of its unifiers (it's most general unifier [16] [17] [18]).

For theorem proving purposes, The task of unification may be limited to checking whether two structures can fit or not [11]. In that case, it suffices to find one unifier or show that there is none. Rather, CAN needs a conceptual representation of the complete set of unifiers of two conceptual representations, since it needs to know not only if they fit together, but under what exact constraints [5].

In the case where segment variables are restricted to ?-type variables (that is, when index notations are excluded), the problem amounts to solve an equation in a free monoid with a neutral element (the empty sequence). Our implemented unifier exhibits about the same performances as [16] and [17] (the neutral element makes very few differences). It involves too the algorithm for A-C Unification [12] [18] and a splitting algorithm [17] extended for index mathematical notations.

What is new within CAN's unification procedure concerns the unification of patterns which include index notations, auto-references, element-type variables with restricted domain, and negation.

A detailed description of its actual performances is given in [6].

4. META-INTERPRETATION OF ITERATIVE AND RECURSIVE PROGRAMS

At some stage of the process, the following program:

```
(DE MEMBER (X L)
  (COND ((NULL L) NIL)
        ((EQUAL X (CAR L)) L)
        (T (MEMBER X (CDR L))))))
```

is transformed into:

```
1 (MEMBER !x ()) = ()
2 (MEMBER !x (!x ?b)) = (!x ?b)
3 (MEMBER !x (!y ?b)) = (MEMBER !x (?b))
   (where !x ≠ !y)
```

This says among other things that if MEMBER's first argument is different from the CAR of its second argument, then the value is the recursive call: (MEMBER !x (?b)). If CAN were to go on like previously, it would confront the arguments !x and (?b) with each of the three cases. In fact, this is exactly what CAN will do, but more carefully, so as to avoid looping forever.

Application of rule 3 on !x and (?b) needs the following preconditions:

```
!x = !x'
?b = !y ?b'   (A)
```

in which case the value is (MEMBER !x' (?b')). But this is what CAN was meta-interpreting (modulo a change of the variables' name). We consider (A) as a "vicious circle". The meta-interpretation process stops at each vicious circle, and CAN "multiplies" it by a fictive number n of recursions. From (A), CAN is able to obtain abstract representations of flat lists:

```
!x = !x
?b = !y ... !y ?b
      1      n
(where !y ≠ !x and n≠0)
      i
```

This says that if the second argument of MEMBER is a list whose n first elements are different from its first argument, then they will be deleted after the nth recursive call. The meta-interpreter will then go on with the two first rules of MEMBER, and obtain the final conceptual representation of MEMBER which we used in the second paragraph.

The concept of vicious circle is important since it does not coincide with the first recursive call in a program.

For the following example:

```
(DE FOO (L)
  (IF (CAR L) (FOO (CONS (CDR L)))
      (CDR L)))
```

the meta-interpreter will never reach a vicious circle, and yield the final conceptual representation:

```
(FOO ()) = ()
(FOO (!a) ?b) = (?b)
(FOO (!a)) = ()
(FOO (!a ?b)) = ()
```

(underlined variables may not be assigned the NIL value)

5. VICIOUS CIRCLES

A vicious circle is encountered when the meta-interpretation of some form f in an environment E1 leads to the meta-interpretation of f in an environment E2 such that E1 and E2 "may be identified".

Formally, the two environments will be identifiable if their unification gives a unifier which contains no "transmitter". A transmitter is a couple of variables bindings of the form:

```
v1 = f(v2)
v2 = H
```

With the restriction that if f(v2) contains v1 and H contains v2, the couple is considered as a vicious circle.

Here are some examples of LISP programs where the first recursive call does not coincide with a vicious circle:

```
(DE FOO (X Y) (FOO Y X))
```

```
(DE FOO (L M)
  (IF (NULL L) NIL
      (FOO (CDR M) (CONS (CAR M) L))))
```

```
(DE FOO (L M P Q)
  (COND ((NULL L) T)
        ((EQUAL (CAR L) M)
         (FOO (CDR L) P Q (CDR L)))))
```

6. CONSTRUCTIVE INDUCTION (multiplication by "n")

The notion of vicious circle is motivated by the fact that if a substitution (unifier) is a vicious circle, it may be "multiplied by n" by a simple algorithm (i.e its general form after "n" recursive calls may be constructed).

From a variable binding of the form: v = f(v) the algorithm will obtain: (* v n f(v)) which may be seen as a complex variable whose name is v and f(v) is a conceptual representation with auto-references (everywhere v is). In other words, f(v) restricts v's domain. "n" names the fictive number of successive recursive calls. It relates different complex variables to each other.

In some cases, the induction algorithm may generate mathematical index notations such as those involved in MEMBER's CAN definition, instead of conceptual representations with auto-references. i.e in the case:

```
v = -A- v -B-
```

the algorithm will obtain:

```
v = -A- ... -A- v -B- ... -B-
      1          n          n          1
```

(where n≠0)

(where -A- and -B- are any sequence of conceptual representations. They may involve

indexed elements and even sequences whose length depend on the index)

The advantage is that the unification process of conceptual representations is particularly well suited for the unification of such forms.

7. RECURSIVITY

From the program:

```
(DE VECMAT (L M)
  (IF (NULL L) M
    (DCONS (CAR L) (VECMAT (CDR L)
                          (CONS NIL M))))))
```

where

$$(\text{DCONS } !x ((?a)_1 \dots (?a)_n))$$

$$= ((!x ?a)_1 \dots (!x ?a)_n)$$

The meta-interpreter will get:

```
1 (VECMAT () !m) = !m
2 (VECMAT (!1 ?2) (?3)) = ((!1 ?a)_1 \dots (!1 ?a)_k)
and ((?a)_1 \dots (?a)_k) = (VECMAT (?2) (()) ?3))
```

Recursivity will be handled by the same processes.

as the unification of $((?a)_1 \dots (?a)_k)$

$$\text{and } ((!1 ?a')_1 \dots (!1 ?a')_{k'})$$

yields a vicious circle, the induction algorithm is applied. here, from:

$$?a_i = !1 ?a_i$$

CAN gets:

$$?a_i = !1_1 \dots !1_n ?a_i$$

By propagation and with the initialisation:

$$((?a)_1 \dots (?a)_k) = ((())_1 \dots (())_n ?3)$$

(the right conceptual representation is the abstract value of M when case 1 is encountered. The initialisation is performed via the unification of the two conceptual representations and propagation of the results.)

CAN finally obtains the conceptual representation of vecmat:

$$(\text{VECMAT } (!1_1 \dots !1_n) ((?b)_1 \dots (?b)_m))$$

$$= ((!1_1 \dots !1_n) \dots (!1_1 \dots !1_n))$$

$$(!1_1 \dots !1_n ?b_1) \dots (!1_1 \dots !1_n ?b_m)$$

8. TERMINATION

The meta-interpretation of a program yields a set of rules which contain recursive calls, as a first approximation. As was shown, the same meta-interpretation process may be pursued. So, CAN may find which rules data can "go" to, when "coming out" of a recursive rule. That is, find which rules refer to which rules. If a recursive rule refers only to itself, or if a group of rules which refer to one another refers to no rule "outside", then a class of data is determined for which simple interpretation of the program never ends. Here are some simple examples:

example 1:

```
(DE FOO (X Y)
  (IF (ATOM X) (FOO Y X) X))
```

CAN yields:

```
(FOO !list !b) = !list
(FOO !atom !list) = !list
(FOO !atom1 !atom2) = infinite-loop
```

example 2:

```
(DE FOO (L M)
  (IF (NULL L) M
    (FOO (CDR M) (CONS (CAR M) L))))
```

CAN yields:

```
(FOO () !m) = !m
(FOO (?1) ()) = ((?1))
(FOO (?1) (?2)) = (!2 ?1)
(FOO (?1) (?2 ?3)) = infinite-loop
```

example 3:

```
(DE FOO (X L)
  (IF (EQUAL X (CAR L))
    L
    (FOO X (CDR L))))
```

CAN yields:

```
(FOO () (!a_1 \dots !a_n)) = ()
with !a_i \neq ()
(FOO !b (!a_1 \dots !a_n)) = infinite-loop
with !a_i \neq !b
(foo !c (!a_1 \dots !a_n !c ?d)) = (!c ?d)
with !a_i \neq !c
```

(underlined variables may not be replaced by the NIL value)

Here is an example of a different kind:

```
(DE FOO (L M)
  (IF (EQUAL L M) T
      (FOO (CDR L) (CONS (CAR L) M))))
```

CAN yields:

```
(FOO !1 !1) = T
(FOO () (??)) = infinite-computation
(FOO (!1 ... !1 !1 ... !1 ??) (??)) = T
      1      n      n      1
and
(!1 ... !1 !1 ... !1 ??) = (!1 ... !1 ??)
  i+1      n      n      1      i      1
[!1,n]
```

For this last example, CAN is not able to explicitly describe the class of data for which there is infinite computation. It is implicitly defined by the set of cases which do not fit the rules above, plus those which fit the second rule.

9. USING CAN: THE INTERACTIVE PROGRAMMER ASSISTANT.

CAN is an interactive program understanding system. When CAN is run, the user may type in LISP functions, or LISP expressions. When an expression is typed in, CAN gives its abstract value. There may be several cases. If the expression is (ATOM X) for instance, CAN will respond:

CASE 1: X=atom1 VALUE: t

CASE 2: X=atom1 VALUE: nil

The user can focus on any of the cases or keep them all.

A practical use of the CAN system, then, is the following: the user first types in a program he has just written. Once the program is understood by CAN, the user may type in any LISP expression involving calls to the program. He will get the corresponding abstract values. Note that CAN is a simplifier. That is, if it is not able to understand an expression E, E's abstract value is E itself (with an implicit meta-evaluation request). In other words, E may not be simplified further.

If the expression is an assertion about the program, it may meta-evaluate to T or NIL, but CAN's performances in this domain still raise the need for a theorem prover.

One common use is to type a call to the program on restricted classes of data. i.e if the program has name FOO and argument X, the user may define a unary LISP predicate P which is true for a restricted class of X's, then focus on the cases where (P X) may evaluate to T, and call (FOO X).

For instance, if FOO is in fact a pattern-matching function MATCH with back-referencing (two occurrences of the same variable must be assigned the same value) and the user wants to test whether

back-referencing works fine, he may first build several kinds of abstract values for MATCH's arguments by the "focus on" technique described above, and then call MATCH on them

One future work will be to develop a system for proving theorems about list structures by structural induction, which uses CAN as a simplification sub-system.

We expect that CAN will greatly help the task of proving theorems by structural induction since, for instance, many of the LISP theorems proved in [22] already meta-evaluate to the LISP truth-value T when given to CAN.

10. CONCLUSION

We have designed a system whose job is to understand what an expert programmer could consider as "visually understandable". CAN does no problem solving. CAN's meta-interpretation process is straightforward, and simple interpretation is a simple case of meta-interpretation [5]. We then think that CAN should be put in the service of expert programmers, or program understanding systems which would be specialised in narrow areas, since conceptual representations are much more easily manipulated than LISP code, for instance. It may effectively be expected that the size of program understanding systems is directly related to the number of ways of representing a relation within a programming language.

The CAN system is written in VLISP-10 [3] [8] and occupies about 20K words.

REFERENCES

- [1] BALZER, GOLDMAN & WILE (1977) "Meta-evaluation as a tool for program understanding" 5th IJCAI, MIT Cambridge, August 1977
- [2] BOYER, ELSPASS & LEVITT (1975) "SELECT--A formal system for testing and debugging programs by symbolic execution" Int. Conf. on Reliable Software, April 1975, pp. 234-245.
- [3] CHAILLOUX (1976) "VLISP-10 manuel de référence" Dept. d'informatique, Univ. PARIS 8, RT 17-76.
- [4] GOOSSENS (1978) "A system for visual-like understanding of LISP programs" A.I.S.B. Conf. Hamburg, July 1978.
- [5] GOOSSENS (1978) "Compréhension visuelle de programmes contrôlée par méta-filtrage" Groplan: Bulletin de l'AFCEt, Groupe Programmation et langages, 1978

- [6] GOOSSENS (1978) "L'unification au service de la compréhension" RT-14. Dept. d'informatique, Université de Vincennes.
- [7] GREEN & BARSTOW (1975) "A hypothetical dialogue exhibiting a knowledge base for a program understanding system" Stanford A.I. Lab. MA 258, Cpt Science Dept. Report n. Stancs-75-476.
- [8] GREUSSAY (1977) "Contribution a la définition interprétative et à l'implémentation des λ -langages" Thèse, Université PARIS 8.
- [9] HEWITT & SMITH (1975) "Towards a programming apprentice" IEEE Trans. on soft. engineering, Vol. SE-1. pp. 26-45.
- [10] HOARE & LAUER (1973) "Consistent and complementary formal theories of the semantics of programming languages" Ibid 3, pp. 145-182.
- [11] HUET (1975) "A unification algorithm for typed lambda-calculus" Theoretical computer science 1. pp 27-57
- [12] HULLOT J.M. (1979) "Associative-commutative pattern-matching" 6th IJCAI. Aug 1979. Tokyo.
- [13] IGARASHI, LONDON, LUCKHAM (1975) "Automatic program verification I: A logical basis and its implementation" Acta Informatica, Vol. 4, pp. 145-182.
- [14] KING (1975) "A new approach to program testing" Int. Conf. on reliable software, April 1975, pp. 228-233.
- [15] RICH & SCHROBE (1975) "Understanding LISP programs: Towards a programming apprentice" Master's Thesis, EECS M.I.T.
- [16] SIEKMANN (1975) "String unification" Essex university, Memo CSM-7.
- [17] SIEKMANN LIVESEY (1975) "Termination and decidability results for string unification" Essex university, CSM-12.
- [18] STICKEL (1975) "A complete unification algorithm for associative-commutative functions" 4th Ijcai, Tbilisi Georgia USSR. sept. 1975.
- [19] WERTZ (1978) "Un système de compréhension, de programmes incorrects" Proc. 3ème colloque sur la programmation, Paris B. Robinet éd. pp 31-49.
- [20] WERTZ (1978) "Un système de compréhension, d'amélioration, et de correction de programmes incorrects" Thèse de 3ème cycle, Univ. P. et M. CURIE.
- [21] YONEZAWA & HEWITT (1976) "Symbolic evaluation using conceptual representations for programs with side-effects" M.I.T., A.I. Lab., AI-Memo 399.
- [22] BOYER & MOORE (1973) "Proving theorems about LISP functions", Stanford University, Stanford CA. 486-493.