THE CONCEPTUAL CALCULUS FOR AUTOMATIC PROGRAM UNDERSTANDING

D. GOOSSENS
Dept. d'Informatique
Universite PARIS 8
2, Rue de la Liberte
93526 St-DENIS Cedex 02

ABSTRACT

The CAN system for automatic understanding of LISP programs is presented. Uhen applied on a LISP program, CAN needs no assertion about what the program is intended to do. CAN's general task is to associate e meaning to a program, from which it is able to diagnose semantlc properties (unused portions of code, undefinitions, underused programs, classes Of data for which evaluation of an expression never terminates), and to exhibit deep semantic properties.
CAN's underlying theory is the *conceptual calculus,* based on the clausal form of predicate logic. It involves two main ideas :

- an Induction principle specially adapted to program understanding (as opposed to program verification).

- An extension of unificetion to equation solving.

Keywords: program understanding, program meaning, concept, equation solving, unification, equilibration, Induction, clausal form of predicate logic.

1. INTRODUCTION

CAN is an implemented program understanding system. It is fully automatic and detects semantic properties :

- unused portions of code, or undefined function applicet ions
- classes of data for which a program execution never terminates
- programs which are underused in a given task

this diagnosis activity is superimposed to CAN's normal activity x assigning a meaning to a program.

CAN'S underlying formal theory, the *conceptual calculus,* is baaed on clausal form of predicate logic. It admits new types of variables (segment variables, typed element variables, index notations) and constructors (sequences, bags, sets) within terms, with associated equation solvers. It uses an Induction rule whose tesk is to *Induce* properties of data from recursive equations [6], rather then to *prove* them. The application criteria of this Induction rule is a *vicious circle* detector.

As opposed to program verification, which lies on verification conditions generation (VCG) [3,8] and theorem proving, program understanding lies on *meta-evaluation* [17,4] and *equation solving* [5,7,15,163.

Meta-eveluetion as defined in [53 is e generalisation of symbolic evaluation [9,18].

2. FROM VERIFICATION CONDITION GENERATION TO META-EVALUATION

Statements subsume implicit structures which they access and modify (tables of identifiers, I/O buffers, stacks, arrays). Symbolic evaluation pays a closer attention to these implicit structures than verification condition generetion. Instead of being hidden in logical expressions, they are Individualized as data abstractions. As an example, where assignment is reduced, within VCG, to e mere operation of substitution on a logical expression [8], symbolic evaluation uses an abstract data structure uhich models a table of identifiers. It permits to easily take in account different assignment modes, as they exist in programming languages.

But symbolic evaluation, as it is defined in most works, leaves some problems unsolved. A cell to e LISP-defined symbolic evaluator of LISP expressions (as copied on existing ones [18]), would appear as :

(EVAL-SYMB *expression identifiers-table)*

Identifiers-table would be a list of tuples :

(Identifier . value-of-1d)

Actual symbolic evaluators only allow *value~of-1d* to be meta-described. Meta-evaluation requires both *Identifier* and *expression* to be eventually meta-described too (for instance, the SET function will introduce mete-described *Identifiers* in the *Identifiers-table.* EVAL and the possibility to make programs build and use other programs, will both Introduce meta-described expressions).

The extension from symbolic evaluation to meta-evaluation is not simply a theoretical Improvement, but a practical need.

On another hand, the combined uses of EVAL and FEXPRs allow the programmer to define his own control structures; new conditionals, iterations, repetitions and miscellaneous.

We have adopted a formalism for representing program meanings which allows :

- any level of abstraction in the description of the environment of an expression to be meta-evaluated (including the expression itself).
- Incrementality : CAN's meta-evaluator is a S9t of conceptual representations of the basic LISP functions, statements, and control structures. It increases itself as it understands new user definitions. The meta-evaluator transforms each new LISP definition in a conceptual representation. This representation extends the understanding power of the meta-evaluator.

Our formalism is based on the clausal form of predicate logic [10]. Ue have adopted a new syntax. Clauses are replaced by concepts and functional terms by conceptual representations [17,5]. One concept generally corresponds to several clauses, which share Identical subterms. This new syntax is especially adapted to the induction rule of the conceptual calculus.
Ue have also extended the classical *resolution rule* [14] to an *equilibration rule,* where unification [14] Ia extended to equation solving. This extension has been first suggested by [13). Our equation solving system includes unification under axioms of :

- associativity and commutativity [16,7]
- associativity [12,15]
- associativity, commutativity, idempotence [11]
- Index notations [6,5]

Index notations are obtained from the induction rule.

Meta-evaluation is a strategy which exclusively uses the equilibration rule. Equilibration performs in a unified way the various tasks of :

1- choosing which concept applies on the current syntactic unit, In the expression to be meta-evaluated. This confrontation process is an equation solving process, rather than a test, since the syntactic unit may be meta-described.

2- evaluating the relevant parts of the current syntactic unit, as stated by the chosen concept.

3- constraining the returned values and modified environments, as asked by the chosen concept (this Is still done by equation solving, and may cause case analysis).

4- constructing the resulting abstract environment and value returned by the syntactic unit.

CAN is not simply a meta-evaluator. It uses one to perform, with the help of induction, a program analysis task.

3. THE CONCEPTUAI, CALCULUS

SYNTAX :

The basic syntactic units of the conceptual calculus are the *concepts.* A concept is a couple :

CONCEPT = <LOGIC , CONTROL>

where CONTROL is either :

ters + term

or

term -► (BP term . cont)

the (BP term . cont) form is a *breakpoint,* tern is a procedure call and cont is a set of *subconcepts.* Subconcepts are concepts. The distinction will however be helpful for the exposition of the equilibration rule.
The LOGIC component is a conjunction of literals, terms are simple pointers to the LOGIC component.

If the **LOGIC** component were not syntactically distinguished from the control component, the LISP function **MEMBER** would be conceptually defined as :

```
(MEMBER e1 e2) → (BP e1
          x → (BP e2
                   (a1 ... an)/ai=x → NIL
                   (a1 ... an x . y)/ai=x → (x . y)))
```

This definition includes the evaluation of the arguments e1 and e2, and the purely applicative part of MEMBER. With the **LOGIC-CONTROL** distinction, it looks like (ti are element variables) :

```
<{t1=(MEMBER t2 t3)
  t2=e1
  t3=e2) , t1→(BP t2
                  <true , t4→(BP t3
                                 <{t5=(a1...an)
                                   ai=t4
                                   t6=NIL) , t5→t6>
                                 <{t5=(a1...an . t7)
                                   t7=(t4 . t8)
                                   ai=t4) , t5→t7>)>)>
```

SEMANTICS

Ue have defined a semantics for concepts, which associates e set of HORN clauses [10] to a concept. The results which hold for clausal form of predicate logic hold for the conceptual calculus.
In particular, the following equilibration rule corresponds to the resolution rule [14]. The induction rule and the vicious-circle detector, however, heve no known equivalent.

RULES OF INFERENCE

Rule of equilibration :

The equilibration rule may be interpreted as a procedure invocation mechanism, based on equation solving. It states that the **CONCEPT**, or **SUBCONCEPT** :

<L2 , t2→(BP t3 . cont)> □

may be replaced, with the help of the **CONCEPT** :
<L1 , t1→D1>
(which may be □ itself), by the set of concepts :

conto(<L1, t1→D1>o<L2, t2→t3>)

Distribution of composition (Ci are concepts) :

(C1 ... Ca)o(C1'... Cb') = (C1oC1' ... C1oCb'
...
CaoC1' ... CaoCb')

Composition :

C1o<L , t1→(BP t2 . cont)>
= <L , t1→(BP t2 (C1)ocont)>

Base case of composition :

<L1 , t1→D1>o<L2 , t2→t3>
= (<L1∧L2∧s , t2→D1>/t1=t3|-s)

t1=t3|-s means that s is a solution of equation t1=t3.
Example :
By systematic application of equilibration, and using the conceptual definitions of APPEND, CONS, MEMBER, and identifier evaluation, the LISP expression : (MEMBER X (APPEND L (CONS X M))) ① is conceptualized in 2 concepts, which correspond to the cases whether X is in L or not. The meta-evaluation of ① is detailed in [5].

Rule of induction

If the equation t1=t2 gives a solution s which corresponds to a vicious circle, then the tail-recursive concept :

<L1 , t1→(BP t2
<true , t→t>)>

where true is the empty conjunction of literals, may be replaced by :

$$\langle L1 \wedge s^{n} , t1 \rightarrow (BP\ t2 \langle true , t \rightarrow t \rangle) \rangle$$

Non tail-recursive case : If t1=t2 gives s, and s corresponds to a vicious circle,

<L1 , t1→(BP t2
C1 ... Ck)> ②

may be replaced by the set of concepts (we do not distinguish between LOGIC and CONTROL in the definition of inductx for ease of exposition) :

(<L1∧s^{n} , t1→(BP t2
v→(BP (inductx v n)
↔↔))>,
(inductx v 0)→v,
INDUCE[C1],
...
INDUCE[Ck])

where INDUCE is defined as :

INDUCE[ti→Di] = (inductx ti nb+1)→IND[Di]
IND[term] = (BP (inductx term nb) ↔↔)
IND[(BP term
t1'→D1'
...
tm'→Dm')] = (BP term
t1'→IND[D1']
...
tm'→IND[Dm']))

Remark : ② corresponds to a set of HORN clauses. If induction were to be defined within actual syntax of clauses [10], it would need a preprocess for gathering separate clauses. This process is not needed with the syntax of concepts.

Example :

MEMBER may be defined without using index notations, using the recursive concept :

(MEMBER x (y . z))/x≠y → (BP (MEMBER x z)
v → v) ②

(MEMBER x (y . z))/x≠y is t1 and (MEMBER x z) is t2. Equation t1=t2 solves in :

s = {(y . z)=z}

which corresponds to a vicious circle. So,

$$s^{n} = \{(y1 \ldots yn . z)=z\}$$

Thus, application of the rule of induction on ② gives :

(MEMBER x (y1 ... yn . z))/x≠yi → (BP (MEMBER x z)
v → v)

Equilibration of this concept with the two stop conditions of a recursive definition of MEMBER gives the conceptually-defined and non-recursive definition of MEMBER.

Vicious circle :

A substitution list s is a vicious circle iff every of its associations is of the form variable=f(variable'), where f(v) is a term containing v, and where variable has been renamed in variable' during equation solving. In that case, s^{n} includes the association variable=f^{n}(variable').
Ways of building f^{n} from f and obtaining index notations and technics for solving equations which include index notations are developped in [5].

STRATEGIES

Evaluation and meta-eveluation are strategies which systematically use the equilibration rule. These strategies may not stop when applied to recursive concepts. CAN uses a *program analysis* strategy which controls the application of equilibration rule, and uses induction to simplify the control component of recursive concepts. To control the application of equilibration and induction on a set of concepts, CAN builds the *dependency graph* of the set of concepts, which corresponds to the connection graph [10] built from HORN clauses. This control, and the situations it is sensitive to, are developped in [5].
The strategy of program analysis used by CAN is as follows : The dependency graph of a set of concepts is developped by equilibration until all loops (self-referent concepts) are vicious circles. Then, induction is selectively applied [5] and the whole process is restarted.

4. AUTOMATIC DIAGNOSIS OF SEMANTIC PROPERTIES

During the process of meta-evaluation, several semantic properties are detected by CAN, which correspond to simple properties of the solutions of the equations encountered. All the following examples have been treated successfully by CAN.

UNUSED PORTIONS OF CODE AND UNDEFINITIONS.

An undefinition is diagnosed from the composition of two concepts :

[t1 → D1]o[t2 → t2']

when equation t1=t2' has no solution. In that case, D1 is an unused continuation. If concept t1 → D1 is part of the definition of a conditional, for instance the else part of IF :

x/x≠nil → (PROGN . expressions)

then (PROGN . expressions) is an unused portion of code.

Example :

In the following unfinished pattern matcher, the indicated portion of code will never be evaluated :

```
(DE MATCH (P D SUBSTITUTION)
  (COND ((EQUAL P D)
         SUBSTITUTION)
        ((ATOM P)
         NIL)
        ((MEMBER P VARIABLES)
         (LET (VAL (ASSQ1 P SUBSTITUTION))
              (IF VAL
                  (IF (EQUAL (CDR VAL) D)
                      SUBSTITUTION
                      NIL)
                  (CONS (CONS P D) SUBSTITUTION))))
        ... ))
```

The detection of this unused portion of code would be easy if ASSQ1 was conceptually defined. But CAN is only provided its LISP definition :

```
(DE ASSQ1 (V S)
    (COND ((NULL S) NIL)
          ((EQ V (CAAR S))
           (CAR S))
          (T (ASSQ1 V (CDR S)))))
```

CAN must conceptualize this recursive definition in order to find that (ASSQ1 P SUBSTITUTION) will always return the value NIL. CAN uses the induction rule to conceptualize ASSQ1, in the ATOMS-LISTS model of LISP, where EQ is defined as :

(EQ *atom atom*) → T
(EQ x y)/x≠y → NIL

Note that CAN is <u>not</u> looking for unused portions of code. In the present example, CAN really discovers it. This is an ***understanding*** activity.

UNDERUSE OF A FUNCTION.

If, from the composition :

[t1 → D1]o[t2 → t2']

the equation t1=t2' solves in one solution, and if this solution contains an equation of the form v = constant where v is a segment variable, then t1 → D1 is underused when applied to t2 → t2'. (a variable is a segment variable when it is the argument of an associative function).

simple examples :

Let e be an expression in the else part of :

(IF (CDR L) then-part . else-part)

- e being (REVERSE L)
  leads to the equation *(a1 ... an)=L* which solves in *n=1* or *n=0*.
- e being (LAST L)
  leads to the equation *(?1 y)=L* which solves in *(?1)=()* (?1 is a segment variable).

So, REVERSE and LAST are underused in this context.

Now, CAR is not underused if e would be (CAR L). The related equation *(a . b)=L* solves in *b=()*, but *b* is not a segment variable.

NON-TERMINATION.

CAN determines the class of data for which a program never terminates from its dependency graph by the following method :

- Within the set of concepts which result from the analysis of the program, consider any recursive concept :

t1 → (BP t2 . cont) Ⓓ

where equation t1=t2 corresponds to a vicious circle. If, for any other concept t3 → D3 the equation t2=t3 has no solution, then the concept :

t1 → t2: infinite computation

replaces Ⓓ.

Example :

Here is an erroneous definition of the LISP function MEMBER :

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

CAN conceptualizes the applicative part of MEM as :

(MEM *NIL (a1 ... an)*) → *NIL*
(MEM *x (a1 ... an)/a1≠x*)
→ (MEM *x ()*): infinite-computation
(MEM *x (a1 ... an x . y)/a1≠x*) → *(x . y)*

Remark : *NIL* may not be substituted to underlined variables.
If we describe the class of data for which MEM never terminates, by the cases where L does not contain X, we miss the case where X is *NIL*. In this case, MEM stops with a correct answer. This special case corresponds to the LISP special case (CAR *NIL*)=*NIL*.

INTERESTING FEATURE.

The following POWERSET program builds the list of subsets of a set represented as a list of its elements :

```
(DE POWERSET (E)
    (IF (NULL E> (CONS NIL NIL)
        (LET (X (POWERSET (CDR E)))
             (APPEND X
                     (DCONS (CAR E) X)))))

(DE DCONS (A L>
   (IF (NULL L) NIL
       (CONS (CONS A (CAR D)
             (DCONS A (CDR L)))))
```

Though POWERSET may not be conceptualized, or simplified, more than as a recursive program, quite Identical to its LISP form, we may use CAN in a particular model, a simplified space of properties of LISP objects [2], where LISP lists are abstrected to the number of their elements. In this model, CAN conceptualizes POWERSET in :

(POWERSET n) -> 2^n

which says that if POWERSET is applied to a list of length n, its result has length 2 power n, which is the number of subsets of a set.

UNDERSTANDING WHAT ft PROGRAM DOES.

As a student-examination simulation, let us write a program which tests whether two lists have the same length or not, without using the LISP function LENGTH :

```
(DE SAMELENGTH (L M)
    (IF (NULL L) (NULL M) . . .
```

Ue now try a smell subtlety: instead of testing whether M Is empty, we swap L and M at the recursive call, with only one of them reduced. Thus, we obtain :

```
(DE SAMELENGTH (L M)
    (IF (NULL L) (NULL M)
        (SAMELENGTH M (CDR L))))
```

Ue can verify if this program fits our intentions by asking CAN to understand it. CAN's conceptuelization of SAMELENGTH shows that the value is T (true) when L and M have equal lengths, but the conceptualization exhibits an undesired case : SAMELENGTH's value is T also when L has exactly one single element more than M. In the other cases, .the value is correctly NIL (false).

5. CONCEPTUAL REPRESENTATIONS

Conceptual representations are an attempt to :

- maximally simplify the CONTROL component of a program, for the benefit of the LOGIC component.
- make the most of the literals in the LOGIC component be of the form varieble=term or variable term. That is, properties of terms and relations among terms are maximally represented in the form of substitution lists, which detail their structure.

A conceptual calculus program (concept) is a couple <LOGIC , CONTROL>. The maximum simplification of the CONTROL component is as a rule : term => term. The simplification process consists in getting rid of breakpoints in the CONTROL component. Breakpoints can be suppressed by :

- Equilibration. The LOGIC component is augmented with the solutions of the solved equations.
- Induction. The LOGIC component is augmented with indefinite sequences of associations, represented with index notations.

In other words, the structure of terms, which is implicit in the CONTROL component, is explicited in the LOGIC component. The latter may then be of invaluable help in further equation solving situations.

However, the only notations (variables and constructors) which may be allowed in the LOGIC component are those for which equation solvers have been designed. For instance, equation solvers which deal with associative constructors and segment variables (prefixed here with a "?") allow to replace the classical recursive definition of APPEND :

```
(APPEND () y) -> y
(APPEND (a . x) y) - (BP (APPEND x y)          (1)
                         v -> (a . v))
```

(LOGIC and CONTROL are intermixed for more readability) by the one CAN uses in the ATOMS-LISTS model of LISP :

(APPEND (?x) (?y)> -> (?x ?y) (2)

CAN automatically obtains (2) from (1).
By developping conceptual representations, we have been able to maximally simplify the CONTROL component of the conceptual definitions of many basic LISP functions :

- Typed element variables for functions CAR, CDR, CONS, NULL, ATOM, NUMBP, EQUAL.
- Segment variables and associative constructors for functions APPEND, LAST.
- Index notations for MEMBER, NTH, ASSOC, REVERSE, LENGTH.
- ~ Associative-commutative constructors for arithmetic funct ions.
- Associative-commutative-idempotent constructors for internal functions such as the binding and unbinding of an identifier (SETQ, SET, and Identifier evaluation).

6. EQUATION SOLVERS

The equation solving methodology is an improvement on the classical procedural specification technics :
How is an automatic program understanding system going to find a redundancy in :

```
(PROGN (SETQ X ())
       (LENGTH X))
```

A typical AI solution, based on procedural specifications, would be to make LENGTH decide itself, during meta-evaluation, whether or not its abstract argument is too much restricted or not. This solution would need every LISP function, user defined or not, to do so. Our solution is to commit this decision to a general equation solving system which is independant from LISP functions (user defined or not).
In the present case, the decision is taken from the comparison of the two abstract sequences :

```
(S1 ... an) = 0
```

where a1 ... an and n are variables, which gives the constraint n=0.

The equation solving methodology consists in grouping the procedural capacities which are redundantly scattered into independant *reasoning systems.*
The implemented equation solvers that CAN uses, including the index notation cases, are developped in [5]

7. CONCLUSION

lie have presented the CAN system for automatic understanding of LISP programs. CAN is programmed in the conceptual calculus framework and shows capabilities widely beyond the scope of other existing program understanding systems. The reasons behind these achievements will not be found in a special effort to devise ad-hoc solutions, but in the directions we focussed our attention to, which indivisibly concern the representation and deductive power of the conceptual calculus.
While traditional logical expressions and theorem proving were adapted to program verifying, new variables, constructors and equation solving have raised the program understanding task to previously unreached summits.
We have recently studied a promising new direction : the automatic choice of a level of representation. This possibility would allow CAN to conceptualize, using for instance set notations, programs which are not built from set operations, but whose operation may advantageously be abstracted to a set operation.

In front of the multiplicity of new directions, and with regard to the promising achievements we expect from them, we can consider the program understanding problem as highly deserving entirely devoted works.

CAN and the conceptual calculus are implemented on a POP KL-10 in the VLISP language [1].

REFERENCES

[1] CHAILLOUX J. 1980 *Le modele VLISP: description, Implementation, evaluation.* These de 3items cycle, Univ P. et M. CURIE.

[2] COUSOT P. *Examples d'analyse semantique automatique de programmes* Laboratoire IMAG, BP 53, 38041 Grenoble Cedex.

[31 FLOYD R.W. 1967 *Assigning meanings to programs* Math. Aspects of Comp. Science Proc. Symp. in Applied Math. 19, Providence (RI>, Amer. Math. Soc. 1967, Schwartz ed.

[43 GOOSSENS D. 1979 *Meta-Interpretation of recursive list-processing programs* 6th. IJCAI. Tokyo. Aug 79.

[5] GOOSSENS D. 1980 *La meta-4valuat1on au service de la comprehension automatique de programmes* These de 3eme cycle, Univ. Pierre et Marie Curie - PARIS 6, Dec. 80

[6] GREUSSAY P. 1980 *Program understanding by reduction sets* A.I.S.B.-80 Conference 1980, Amsterdam, July 1980, G-l G-7.

[7] HUET G. 1978 *An algorithm to generate the basis of solutions to homogeneous linear diophantlne equations* INRIA. Rapport de recherche n.274. Jan. 1978.

[8] IGARASHI LONDON LUCKHAM 1975 *Automatic program verification I: A logical basis and its implementation* Acta Informatics, Vol. 4, pp. 145-182.

[9] KING J. 1976 *Symbolic execution and program testing* Comm. of ACM. Vol 19. n.7 July 76. *pp* 385-394

[103 KOUALSKI R. 1979 *Logic for problem solving* A.I. series. Nils J. NILSSON ed. North Holland, New York Oxford.

[II] KUHNER S. MATHIS C. RAULEFS P. SIEKMANN J. *Unification of Idempotent functions* 5th. IJCAI. MIT. Cembrige. Aug 77. p 528.

[12] MAKANIN G.S. 1977 *The problem of solvability of equations in a rree semi-group* Soviet Math. Dokl. Vol 18 n.2 1977.

[13] PLOTKIN G.D. 1972 *Building-1n equational theories* Mach. Int. 7, Meltzer Michie eds., 1972.

[14] ROBINSON J. 1965 *A machine-oriented logic based on the resolution principle* Journal of Assoc. for Comp. Machinery. Vol 12. n.l. Jan 1965. pp 23-41.

[15] SIEKMANN J. 1975 *String unification* Essex university, Memo CSM-7.

[163 STICKEL 1975 *A complete unification algorithm for associative-commutative functions* 4th Ijcai, Tbilisi Georgia USSR. sept. 1975.

[173 YONEZAWA A. HEWITT C. 1976 *Symbolic evaluation using conceptual representations ror programs with side-errects* M.I.T., A.I. Lab., AI-Memo 399. Dec. 76.

[183 WERTZ H. 1979 *A System to Improve Incorrect Programs,* Proc. 4th International Conference on Software Engineering, Munich, R.F.A., pp 286-293, sept. 1979