

## PROGRAM UNDERSTANDING BY REDUCTION SETS

Patrick GREUSSAY

Departement d'Informatique  
Universite Paris VIII - Vincennes  
75571 PARIS Cedex 12

&

C.N.R.S. LA 248 L.I.T.P.  
2 place Jussieu  
75005 Paris

### Abstract :

While building or understanding large LISP systems, many small auxiliary functions are often subject to errors or misunderstanding, in the case of very involved recursions. RAINBOW is a specialized program understanding system able to reduce automatically such sets of recursive functions to a form where the goal of these sets are clearly displayed. RAINBOW can display interactively the goal-forms into two sets of new external 2-dimensional notations: recursive and linear. Program understanding is obtained by the translation of the original set of LISP functions into the open recursive notation, then by elementary symbolic evaluation yielding closed linear forms of the original functions. Those linear forms are exactly the goals wanted. RAINBOW operates efficiently on a definite class of LISP functions, and uses an extendable set of reduction rules, which constitute the symbolic interpreter. RAINBOW can be used interactively if a user want to verify that a set of functions perform its intended goal, or can be incorporated easily as a specialized component of a larger program understanding system. This paper shows how RAINBOW operates on sets of recursive functions building combinatorial objects.

### KEY-WORDS:

automatic program understanding, program debugging, VLISP, RAINBOW system, multiple representations, program transformation, symbolic interpretation.

## 1. INTRODUCTION

RAINBOW is a specialized interactive program understanding system. It asks from its user a set of recursive functions definitions, then extracts and displays graphically its *goal*, in terms of properties of lists viewed as sequences.

To display the goal of a definition set, we have introduced two classes of 2-dimensional external notations which are implemented within RAINBOW. The *open* notation is a compact notation for recursive programs or data structures, the *closed* notation expresses intrinsic properties of linear sequences in term of generic properties of their elements.

Program understanding is obtained by the translation of the original set of functions into the open notation, then by elementary symbolic evaluation yielding closed forms of the original functions. The closed forms are exactly the goals wanted.

Presently, RAINBOW can reason about classes of data-structures as lists considered as sequences, extensions to recursive LISP functions operating on other classes of data-structures are considered.

RAINBOW can be used either as a front-end of a LISP system, or as a specialized part for low-level understanding of sequences, in a larger program understanding system. An analogy with low-level machine vision is here in order: scene analysis has to rely upon intrinsic properties of pictures, as incidence, gradient, illumination or texture. We believe that a large program understanding system must also rely upon intrinsic properties of the data involved in the programs tentatively analyzed.

A programming apprentice system (RICH 1979, WERTZ 1979), if used interactively, must have the capability of focusing in a visually understandable way, to lower levels of plans. The plans diagrams described in (SHROBE 1979) seem promising either at very high level of inter-module analysis, or when the goal of a module is intermixed with other goals, or when global side-effects are involved. However these plans, displaying very well the global course of actions involved in analysed programs, do not seem appropriate to be used for lower-level modules because they do not allow to display the structure of the elementary goals for simple modules. Unfortunately, in very large systems, it seems that the simplest modules are the most error-prone.

RAINBOW allows the user to check immediately, in a visually understandable way, the goal of a set of functions definitions. The user does not have to provide any assertion to verify, about his definitions set, because in a sense RAINBOW is precisely reducing this set to its assertion.

RAINBOW is an implemented system written in VLIISP (CHAILLOUX 1978) running on DEC KI-10, and PDP 11/40. The external notations can be visualized on any kind of display or hard-copy terminal.

## 2. OVERVIEW OF THE RAINBOW SYSTEM

### *The symbolic interpreter:*

The symbolic interpreter is essentially a production system having an initial fixed set of reduction rules for the handling of LISP sequences. When a user submits a new function definition to RAINBOW, the reduced closed linear form obtained as a goal is incorporated by the system into the set of rules. The interpreter is able to expand every inner function call with the replacement part of the corresponding rule along with the renaming of variables when necessary ( $\alpha$ -conversion). As in (BOYER 1977), The interpreter operates iteratively until no more rule can apply to the reduced form.

The definitions entered can be also called within the LISP iterpreter, and every step of the reduction can be interactively reversed, providing an history of the reduction. Also at user-level, RAINBOW can handle symbolic function calls.

The process of resolution into linear forms uses a set of reduction rules for the translation of recursive representations into linear representations.

The reduction set of rules for a LISP function is expressed into two new classes of graphical notations for recursive programs and data, and for linear sequences.

### *Recursive and linear external notations:*

#### 1) Linear

It is used to express generic results or *goals* of the analysed functions: it expresses characteristics of sequences.

This notation is a 2-dimensional specialization of the one used in (GOOSSENS 1979).

$$\begin{array}{c} n \\ | \\ [ [ E_i ] \\ | \\ 1 \end{array} =df (E_1 E_2 \dots E_n)$$

The letter *i* is an "index variable" ranging from lower to upper indices, here from 1 to *n*. When one of the indices is 0, the notation denotes the empty sequence. In the context of RAINBOW, the range of the index variable is the LISP expression immediately following it.

EXAMPLES:

$$(\text{CAR } \begin{array}{c} n \\ | \\ [ [ E_i ] \\ | \\ 1 \end{array}) \rightsquigarrow E_1$$

$$(\text{CDR } \begin{array}{c} n \\ | \\ [ [ E_i ] \\ | \\ 1 \end{array}) \rightsquigarrow \begin{array}{c} n \\ | \\ [ [ E_i ] \\ | \\ 2 \end{array}$$

$$(\text{CONS } E_1 \begin{array}{c} n \\ | \\ [ [ E_i ] \\ | \\ 2 \end{array}) \rightsquigarrow \begin{array}{c} n \\ | \\ [ [ E_i ] \\ | \\ 1 \end{array}$$

$$(\text{REVERSE } \begin{array}{c} n \\ | \\ [ [ E_i ] \\ | \\ 1 \end{array}) \rightsquigarrow \begin{array}{c} 1 \\ | \\ [ [ E_i ] \\ | \\ n \end{array}$$

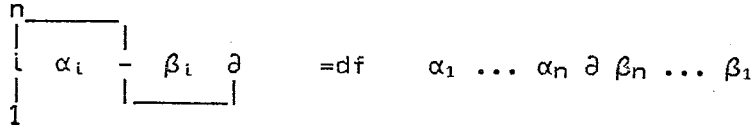
A more complex example is:

$$\begin{array}{c} n \quad i-1 \\ | \quad | \\ [ [ ] L_j \text{ (CONS A } L_i \text{ ) } k L_k ] \\ | \quad | \\ 1 \quad 1 \quad i+1 \end{array} \rightsquigarrow$$

$$[[ (\text{CONS A } L_1) \dots L_n ] [ L_1 (\text{CONS A } L_2) \dots L_n ] \dots [ L_1 \dots (\text{CONS A } L_n) ]]$$

2) recursive

It is used to express the recursive computation performed by the function. It has the general form:



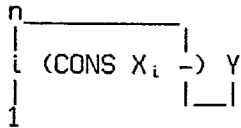
where  $\alpha_i$ ,  $\beta_i$ , and  $\partial$  are parts of LISP expressions as well as linear or recursive forms. The recursive notation is essentially an indexed context-free grammar rule. It expresses  $n$  levels of nesting of function calls terminating with the expression  $\partial$ . When the RAINBOW system uses the recursive notation,  $n$  is the length of the list which is the value of the recursion variable. The crossing of lines in the notation denotes a self-reference to the entire expression with the index variable progressing one step towards the highest index.

As an example RAINBOW translates interactively, the following function definition

```

(DE append (X Y)
  (IF (NULL X) Y
      (CONS (CAR X)
            (append (CDR X) Y))))
  
```

into the recursive form



which schematizes the nested expression

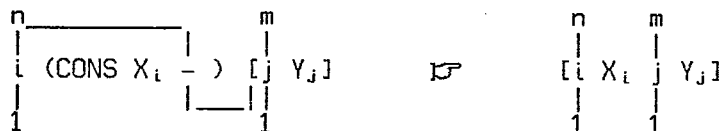
```

(CONS X1 (CONS X2 ... (CONS Xn Y) ... ))
  
```

where  $X_i$  translates  $(\text{CAR } (\text{CDR}^{i-1} X))$



It happens that the goal of the append function is itself expressed into the reduction rule RC2:



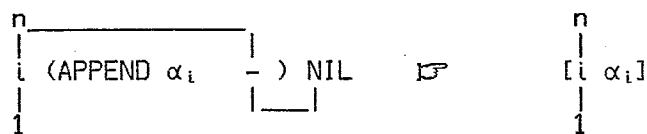


The reduction set of rules which has been used in the previous example is:

- the definition of the function  $g$  itself.
- the rule RC1:



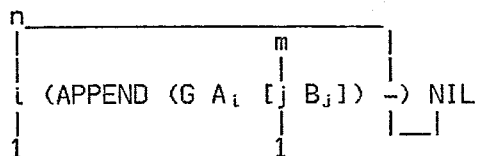
- the rule RA1:



So the RAINBOW system yields in succession :

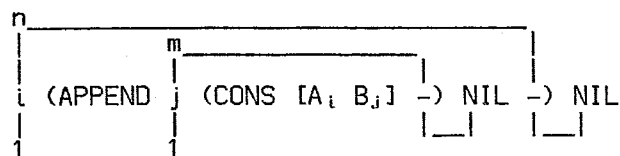
? (f (\*list\* A) (\*list\* B))

*Lines beginning with "?" are typed directly by the user*

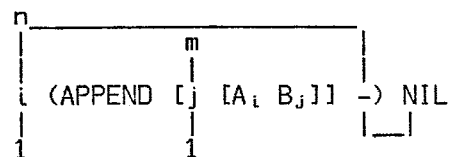


OK

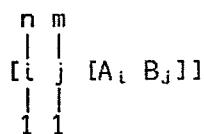
? ap all  
APPLYING... G GIVING...



APPLYING... RC1 GIVING...



APPLYING... RA1 GIVING...



*The final linear form*

4. UNDERSTANDING COMPLEX RECURSIONS

Sometimes recursion can be quite involved, as in the following square matrix generator:

```
(DE vecmat (E)
  (IF (NULL E) NIL
      (CONS E
            (DCONS (CAR E) (vecmat (CDR E))))))
```

where

$$(\text{DCONS } \alpha \begin{bmatrix} n \\ | \\ E_i \\ | \\ 1 \end{bmatrix}) \quad \rightsquigarrow \quad \begin{bmatrix} n \\ | \\ E_i \\ | \\ 1 \end{bmatrix} (\text{CONS } \alpha E_i)$$

The goal of VECMAT extracted by RAINBOW is:

$$(\text{VECMAT } \begin{bmatrix} n \\ | \\ E_i \\ | \\ 1 \end{bmatrix}) \quad \rightsquigarrow \quad \begin{bmatrix} n \\ | \\ E_i \\ | \\ 1 \end{bmatrix} \begin{bmatrix} n \\ | \\ E_i \\ | \\ 1 \end{bmatrix}$$

The body of VECMAT is slightly generalized to obtain the main reduction rule:

$$\begin{bmatrix} n \\ | \\ (\text{CONS } \alpha_i (\text{DCONS } E_i \text{ ---})) \text{ NIL} \\ | \\ 1 \end{bmatrix} \quad \rightsquigarrow \quad \begin{bmatrix} n \\ | \\ (\text{CONS } \begin{bmatrix} i-1 \\ | \\ (\text{CONS } E_j \text{ ---}) \alpha_i \text{ ---} \end{bmatrix} \text{ NIL} \\ | \\ 1 \end{bmatrix}$$

which by the rule RC1 is reduced itself to:

$$\rightsquigarrow \begin{bmatrix} n \\ | \\ (\text{CONS } \begin{bmatrix} i-1 \\ | \\ (\text{CONS } E_j \text{ ---}) \alpha_i \end{bmatrix} \\ | \\ 1 \end{bmatrix} \quad (\text{rule RDC4})$$

Now if we sets  $\alpha_i$  to  $\begin{bmatrix} n \\ | \\ E_k \\ | \\ 1 \end{bmatrix}$ , RAINBOW obtains the linear form for VECMAT by:

$$\begin{bmatrix} n \\ | \\ (\text{CONS } E_j \text{ ---}) \begin{bmatrix} n \\ | \\ E_k \\ | \\ 1 \end{bmatrix} \\ | \\ 1 \end{bmatrix} \quad \rightsquigarrow \quad \begin{bmatrix} n \\ | \\ \begin{bmatrix} i-1 \\ | \\ E_j \end{bmatrix} \begin{bmatrix} n \\ | \\ E_k \\ | \\ 1 \end{bmatrix} \\ | \\ 1 \end{bmatrix} \quad \rightsquigarrow \quad \begin{bmatrix} n \\ | \\ \begin{bmatrix} i \\ | \\ E_j \end{bmatrix} \\ | \\ 1 \end{bmatrix}$$

The final result being obtained by the reduction rule RIND2:

$$\begin{bmatrix} i-1 \\ | \\ \begin{bmatrix} n \\ | \\ X_i \end{bmatrix} \begin{bmatrix} n \\ | \\ X_j \end{bmatrix} \\ | \\ 1 \end{bmatrix} \quad \rightsquigarrow \quad \begin{bmatrix} n \\ | \\ X_i \\ | \\ 1 \end{bmatrix}$$



### 5. CONCLUSION

The 2-dimensional display for formula within RAINBOW is claimed to be intuitively understandable by using intrinsic properties of sequences: the use of indices relates the length and order of the sequence to the general form of the generic elements.

Along with the powerful reductions from recursive to linear forms, we believe that the external notations presented here reflect our intuitive understanding of lists and their properties.

In a fast checking situation, this graphical style of verification, displaying the generic structure of data, gives the user excellent control over the goal obtained from a functions sets.

The extendability provided by the incorporation of every new function definition as a new reduction rule in the spirit of (BOYER 1977) gives a useful tool to system design and programming methodology: as advocated by (GERHART 1975), the resulting schema and transformations that are saved will have to be ultimately organized into "handbooks of knowledge" about programming.

Presently, RAINBOW is able to reason about classes of data structures as sequences, we are considering its extension to arrays using the rules given in (REYNOLDS 1979). Program understanding with RAINBOW can be viewed as a kind of simplification, and its extension to several classes of data-structures may involve combination of decision procedures for several theories described in (NELSON 1978), (OPPEN 1978).

Internally, RAINBOW is mainly driven, out of the fixed initial set, by user-provided rules obtained by reduction of previous definitions to their goals. Thus, the power of RAINBOW is strictly limited by the class of expressions that the external notations are able to denote. Most of the rules in the fixed initial set are properties of the function CONS, extended to APPEND and REVERSE.

In the present state of the rules, RAINBOW is restricted to primitive recursive functions. A single recursion variable is handled within each rule: an obvious extension to an arbitrary number of recursion variables can be incorporated, each having the same pattern of sequence as an argument.

Another extension is currently implemented to handle iterative function schema as in:

```
(DE itrev (X Y)
  (IF (NULL X) Y
      (itrev (CDR X) (CONS (CAR X) Y))))
```

where Y is acting as an accumulator. With such an extension, the translation of *itrev* into the recursive notation should be:

