

NOTES SUR L'OBSERVATION ET LA COMPREHENSION DE PROGRAMMES

Patrick GREUSSAY

Département d'Informatique
Université Paris VIII - Vincennes
75571 PARIS Cédex 12

&

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

Resumé :

Un important courant de recherche en Intelligence Artificielle met aujourd'hui l'accent sur le développement d'outils visant à automatiser partiellement la *compréhension* de programmes. Ce courant se situe actuellement à la croisée de trois directions de recherches très actuelles : théorie de la programmation, développement d'outils de mise au point interactive, interrogation sur les processus psychologiques mis en jeu dans l'activité des programmeurs experts. Ces recherches doivent déboucher sur l'automatisation partielle de la documentation de systèmes ainsi que sur des outils très puissants d'aide au développement de grands programmes : correction et maintenance. Il paraît indispensable, en préalable à la mise en place de systèmes de compréhension automatisés, de recueillir et de systématiser des données concernant le comportement de programmeurs experts, en situation de compréhension et d'analyse de programmes. Nous décrivons une expérience de compréhension dont l'analyse des résultats indique la difficulté, pour les programmeurs, de manipuler des représentations multiples sans outils appropriés d'aide à la programmation. Nous concluons par l'examen d'un système de compréhension automatique, utilisant un ensemble de représentations multiples pour un même programme, et fondé sur un ensemble de règles de traductions entre représentations récursives et représentations linéaires.

MOTS-CLES:

mise au point, VLIISP, programmation interactive, système RAINBOW, compréhension automatique, représentations multiples, psychologie de la programmation.

1.0 Introduction

La recherche en Intelligence Artificielle est essentiellement définie par la programmation effective de certains modèles de comportement, pour une grande part d'ordres linguistiques, conceptuels et sensori-moteurs.

La mise en oeuvre effective de ces programmes permet de tester et de falsifier très rapidement ces modèles : la construction et la vérification d'hypothèses s'apparentant très fortement, et allant même jusqu'à s'identifier avec le processus de mise au point de programmes. Des structures de données et de contrôle inédites doivent ainsi pouvoir être *très rapidement* mises en oeuvre, structures dont l'ensemble constitue ce qu'on pourrait nommer des *langages provisoires*, très dépendantes du modèle de comportement que la construction du programme d'Intelligence Artificielle permet de valider. Le modèle se révélant en première approximation inadéquat, partiellement ou totalement, la correction ou la reconsidération du modèle induisent alors l'abandon ou la modification drastique des structures évoquées plus haut. Elles conduisent dans des délais très brefs à la conception et l'implémentation de nouveaux objets pour lesquels le même processus se réitère jusqu'à stabilisation du modèle.

C'est très naturellement dans ce type de *programmation instable* dont la rapidité est encore accentuée par sa mise en oeuvre interactive que se révèlent les plus nécessaires des outils-systèmes de développement, visant à rendre automatiques la *compréhension*, *documentation* et *maintenance* de programmes. De tels systèmes doivent rendre automatiques, partiellement ou totalement, la détection et la correction des innombrables micro-erreurs inhérentes à la très grande taille des programmes développés en mode interactif.

Les systèmes de compréhension de programmes doivent permettre de faire coexister plusieurs représentations du même programme en développement : ces systèmes doivent jouer le rôle d'outils de documentation incrémentale, d'outils de *lecture* d'intentions de programmes et de production automatique de leurs propriétés.

Nous faisons l'hypothèse qu'il doit exister une continuité entre les données qui seront livrées par l'observation systématique du comportement de programmeurs experts en situation de compréhension et les nouveaux outils, dits de compréhension automatique de programmes qui commencent à apparaître: dans les deux cas, il s'agit de rendre *visibles* des comportements en rendant explicites les représentations sur lesquelles ils sont fondés. Un système de compréhension automatique original sera à cette occasion présenté, le système RAINBOW (Greussay 1979). Nous examinerons également, à partir d'un cas concret, les difficultés liées à la compréhension automatique lorsqu'on la compare à celle mise en jeu par des programmeurs.

Le langage de programmation LISP est actuellement le support massif de telles expérimentations, c'est pourquoi la plupart de nos exemples seront exprimés en langage VLISP 10 (CHAILLoux 1979) dont l'environnement de programmation très complet illustre bien la nature des outils d'aide à la programmation qu'on peut attendre d'un système LISP contemporain.

2.0 De l'observation à la compréhension

L'expérience de la programmation en Intelligence Artificielle doit nous amener à reconsidérer le point de vue actuel qui consiste à rejeter la mise au point de programmes et ses outils, au profit d'une validation préalable par preuve de la correction, partielle ou totale, de ses spécifications à différents niveaux d'abstractions.

Outre que la faillibilité n'en est certes pas absente (GERHART 1976), que les erreurs de programmation constituent en elles-mêmes une source de données considérable et encore fort mal connue sur les processus intellectuels mis en jeu en programmation (SUSSMAN 1974), la pratique de la programmation en Intelligence Artificielle ne nous permet pas de mésestimer l'importance de la mise au point (De MILLO 1979).

En programmation à long terme, le plus souvent interactive, parfois quasi-improvisée au terminal, en tout cas profondément incrémentale, la validation préalable des programmes ne semble pas être une activité prioritaire. Nous avons plus urgemment besoin d'un ensemble cohérent d'outils de mise au point de programmes, ensemble totalement intégré

- 1) aux interprètes qui interprètent ces programmes sous divers modes
- 2) aux éditeurs (CICCARELLI 1978) qui en permettent l'introduction et la modification.

Dans les cas simples, il semble qu'en programmation interactive, les étapes de mise au point, d'essais, de modifications, voire de commentaires de fragments de programmes peuvent être anticipées, et en quelque sorte comprises et résumées par de tels systèmes. Les outils classiques de mise au point en LISP permettent d'observer le comportement des programmes. Pouvons-nous les étendre à l'observations des comportements des programmeurs? Un tel système d'aide à la mise au point pourrait fournir, de surcroît à sa fonction d'aide à la programmation, cet instrument d'observation et d'enregistrement de l'activité d'un programmeur au cours de sessions interactives. Nous faisons l'hypothèse qu'il existe des régularités observables dans les cycles de mise au point.

Reste que les outils classiques d'observations en LISP (traces et intervenants de type ADVISE), s'ils ont une puissance opérationnelle considérable, laissent leur utilisateur devant un vide sémantique total. Pour passer de l'observation à la compréhension, de nombreuses questions doivent être considérées:

- 1) concernant les erreurs de programmation: comment étudier et classier les erreurs? quelles erreurs doit-on et peut-on capter? quelles erreurs sont anticipables (WERTZ 1979)?
- 2) concernant la compréhension de programmes: comment rendre explicite le fonctionnement d'un programme correct ou non? comment automatiser la compréhension de programmes simples: un programme LISP d'une certaine ampleur sera composé en majeure partie d'un très grand nombre de fonctions auxiliaires très courtes; l'expérience indique que la plupart des erreurs difficiles à déceler interviennent dans le détail de rédaction de ces fonctions auxiliaires. Comment rendre ces erreurs évidentes en livrant automatiquement l'intention de ces fonctions au fur et à mesure de leur frappe au terminal?

3.0 Représentations, Compréhension, Mise au point

Que signifie l'énoncé *concevoir un programme* ?

Voici une réponse partielle issue de la pratique de la programmation en Intelligence Artificielle : concevoir un programme, c'est en mettre au point un autre déjà connu (SUSSMAN 1975) . Ce point de vue a été repris récemment dans une perspective plus théorique par (DERSHOWITZ 1976).

Le point de vue de la mise au point implique qu'en un certain sens, un programme n'est ni juste ni faux. Ni juste, car toujours susceptible d'être modifié, amélioré ou étendu, à mesure que surgissent de nouveaux besoins ou que s'approfondit la compréhension du problème dont le programme est le modèle. Ni faux, car toujours susceptible d'être tracé et édité.

Ainsi le point de vue de la mise au point met l'accent sur l'ubiquité des modifications de programmes, et donc des modifications de représentation et de compréhension.

Une expérience de compréhension

En 1977-1978, nous avons tenté de repérer concrètement quelques unes des difficultés de la compréhension de programmes, à travers une expérience menée sur un cas très simple.

On notera que des expériences de ce type ne visent à rien d'autre qu'à *apprendre à recueillir des données*, des observables des processus psychologiques mis en jeu en programmation. Ces processus sont encore trop mal connus pour fixer prématurément une systématisation de telles observations.

Nous avons soumis à plusieurs programmeurs de haut niveau la fonction suivante, avec pour mission d'en découvrir l'intention.

```

      (DE SKE (L R) (COND
        ((ATOM L) NIL)
        2 ----- ((MEMQ L R) T)
        3 ----- ((SKE (CAR L) (CONS L R)) T)
        4 ----- (T (SKE (CDR L) (CONS L R))))))

```

L'expérience consistait à présenter la fonction sans autre commentaire en laissant les sujets réfléchir quelques minutes, puis à compléter cette fonction par des *thèmes* présentés successivement, certains très intuitifs, d'autres plus formels, éléments de connaissance susceptibles d'en faciliter la compréhension.

Thème 1 : L'appel initial de SKE est

(SKE *une-liste*-L NIL)

Thème 2 : Les deux arguments de la fonction standard MEMQ peuvent être des listes.

Thème 3 : SKE décèle des *effets de bord*, antérieurs à son application

Thème 4 : L est une liste circulaire

Thème 5 : Soit x et y deux cellules de liste, et la relation

```

      x << y
      si y = (x . z) ou (z . x)
      ou
      si x << (CAR y) ou x << (CDR y)

```

Une liste est non-circulaire si, pour deux de ses cellules x et y on n'a jamais simultanément

x << y et y << x

Un seul des sujets participant à l'expérience découvrit la destination de SKE dès la présentation du thème 3. La présentation du thème 5 fut nécessaire à l'un d'entre eux. Tous les autres sujets donnèrent la réponse correcte lors de la présentation du thème 4.

Quels furent les éléments observés?

Le thème 3 indique déjà nettement que L ou une de ses sous-listes est circulaire: une liste circulaire doit être construite par la mise en jeu des primitives RPLACA et RPLACD. Le premier sujet ayant décelé l'intention de SKE programmat depuis quelques jours une routine d'impression élégante de listes circulaires.

Le thème 4 est positif, le thème 5 est négatif: il indique ce qu'une liste circulaire *n'est pas*. Le sujet ayant attendu le thème 5 pour livrer une réponse correcte admit volontiers qu'après l'avoir considéré un instant, il perdit de vue le fait que la fonction SKE est booléenne.

Tous les sujets firent cependant plusieurs observations pertinentes au cours de l'expérience. La clause 4 fut déclarée par tous itérative, opérant le balayage de L dans le sens des CDRs, et la clause 3 récursive, balayant L dans le sens des CARs. L'analogie avec le schéma de la fonction EQUAL fut unanimement constatée.

Le thème 1 semble avoir provoqué un brouillage de représentations: la variable R fut bien reconnue comme un accumulateur, la représentation statique de son contenu a fait, de l'avis général, problème. Une fois l'accent mis sur l'aspect dynamique de sa construction, la destination de R est devenue pour chacun évidente.

Qu'en conclure ?

L'expérience a mis en valeur des difficultés conceptuelles liées à des *interférences* de représentations

1) interférence entre le schéma récursif de SKE et la représentation itérative qu'entraîne la mise en jeu d'un accumulateur: la variable R retient tous les CDRs de toutes les sous-listes de L, mais R est testée dans la condition d'arrêt.

2) interférence entre construction et rétention. CONS construit une *nouvelle* structure, MEMQ teste à la clause 2 l'identité des deux listes: L la liste couramment examinée, et une sous-liste de L recueillie dans R à une étape *précédente* du balayage.

Cette interférence a fait perdre de vue à nos sujets que MEMQ utilise EQ, qui teste l'identité *physique* de deux objets, indifféremment atomes ou listes, par comparaison d'adresses. Ce test d'identité physique a rendu vaine la tentative d'analyse, par nos sujets, de la structure interne des sous-listes de R.

De plus une interférence de représentations spatiales a dû jouer: entre le caractère arborescent de L, impliqué par le schéma de type EQUAL de SKE, et le caractère linéaire de R, construite par CONS successifs.

3) enfin interférence de type manipulateur: L est *décomposée* en (CAR L) et (CDR L) au cours du balayage. Mais L *n'est pas* décomposée lors de sa composition par CONS dans R.

4.0 Un système de compréhension automatique

L'étude précédente nous a indiqué quelques-unes des difficultés liées à la compréhension de programmes dont l'intention est inconnue. Nous examinerons à présent quelques-unes des possibilités offertes par un système de compréhension automatique de programmes spécialisé dans la lecture d'une classe restreinte mais très utilisée de fonctions de construction en LISP. Le système RAINBOW (GREUSSAY 1979), totalement interactif, sert de système frontal à VLISP 10. Il permet la traduction immédiate des fonctions introduites au terminal dans deux types de notations formelles, exprimant l'une le calcul récursif effectué par ces fonctions, l'autre le résultat générique livré à la suite d'appels symboliques. Nous ne décrivons pas ici le processus de transformation des appels symboliques en *visualisation d'intention* mais nous l'illustrerons par des exemples d'utilisation pratique.

Comment rendre *visible* l'intention d'une fonction de construction auxiliaire?

Nous introduirons à cet effet deux notations indicielles

1) linéaire

Elle sera utilisée pour exprimer les résultats génériques ou *intentions* des fonctions analysées: elle exprime des caractéristiques de séquences.

Cette notation est une spécialisation de celles employée dans (GOOSSENS 1979).

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

EXEMPLES:

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

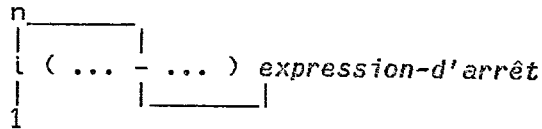
$$(CDR \begin{array}{c} n \\ | \\ [i \ E_i] \\ | \\ 1 \end{array}) \rightsquigarrow \begin{array}{c} n \\ | \\ [i \ E_i] \\ | \\ 2 \end{array}$$

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

$$(DCONS \ X \ \begin{array}{c} n \\ | \\ [i \ E_i] \\ | \\ 1 \end{array}) \rightsquigarrow \begin{array}{c} n \\ | \\ [i \ [X \ . \ E_i]] \\ | \\ 1 \end{array}$$

2) réursive

Elle sera utilisée pour exprimer le calcul récursif effectué par la fonction. Elle prendra la forme générale:



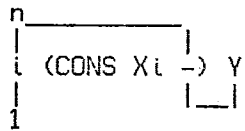
qui indique n niveaux d'imbrications d'appels s'achevant sur l'expression d'arrêt (n étant la longueur de la liste qui sera la valeur de la variable de récursion).

EXEMPLE

Le système RAINBOW traduit, en mode interactif, la définition de fonction suivante

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

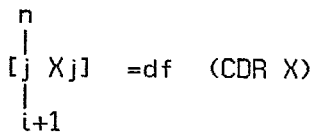
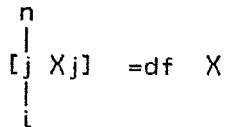
sous la forme schématique



qui résume l'expression imbriquée

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

avec $X_i =_{df} (\text{CAR } (\text{CDR}^{i-1} X))$



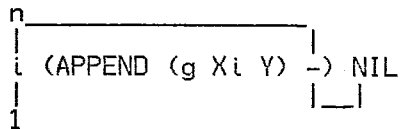
Voici deux exemples d'utilisations interactives de RAINBOW en documentation automatique de programmes, le premier assez simple, le second nettement plus complexe.

EXEMPLE 1:

L'utilisateur introduit la définition suivante:

```
(DE f (X Y)
      (IF (NULL X) NIL
          (APPEND (g (CAR X) Y)
                   (f (CDR X) Y))))
```

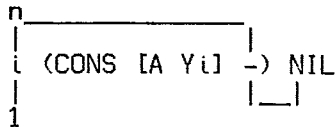
que le système traduit immédiatement en



puis la définition suivante

```
(DE g (A Y)
      (IF (NULL Y) NIL
          (CONS (LIST A (CAR Y))
                (g A (CDR Y)))))
```

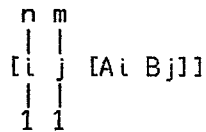
traduite alors en



puis l'utilisateur livre à RAINBOW l'appel symbolique

```
(f (*list* A) (*list* B))
```

qui se résout sous la forme linéaire

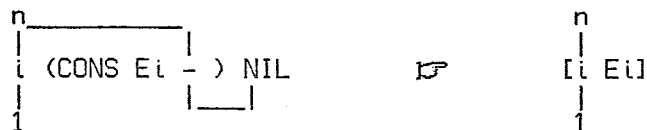


qui révèle l'intention de la fonction *f*:
f construit le *produit cartésien* de ses deux listes-arguments.

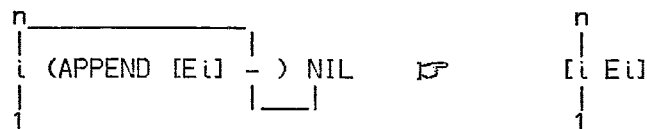
Le processus de résolution sous forme linéaire utilise un ensemble de règles de réduction de représentations récursives en des représentations linéaires.

Celles qui ont été mises en jeu dans l'exemple précédent sont:

- la définition de la fonction *g* elle-même.
- la règle RC1:



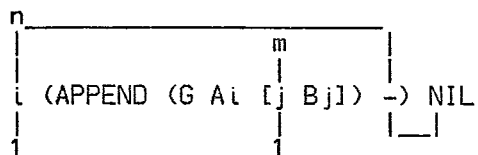
- la règle RA1:



Le système RAINBOW livre ainsi successivement :

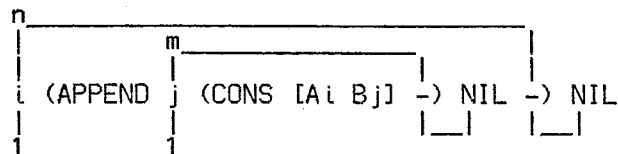
? (f (*list* A) (*list* B))

Les lignes précédées de "?" sont introduites par l'utilisateur

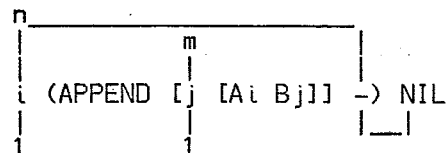


OK

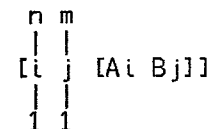
? ap all
APPLYING... G GIVING...



APPLYING... RC1 GIVING...



APPLYING... RA1 GIVING...



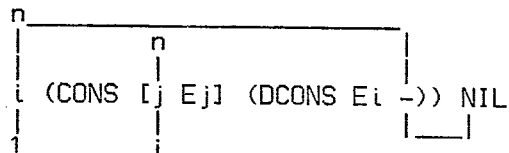
La forme linéaire obtenue

OK

EXEMPLE 2:

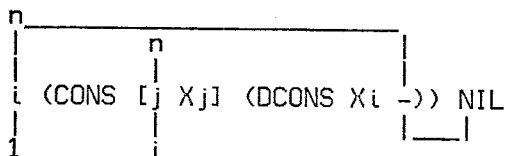
Cherchant à visualiser l'effet d'une récursion assez complexe, l'utilisateur introduit au terminal:

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



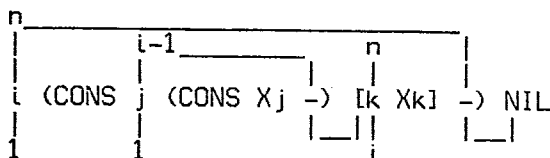
OK

```
? cl (vecmat (*list* X))           puis un appel symbolique
```

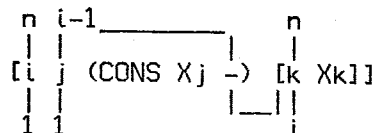


OK

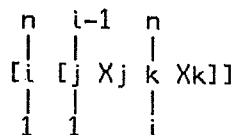
```
? ap all
  APPLYING... RDC4 GIVING...
```



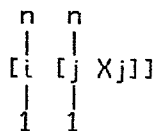
APPLYING... RC1 GIVING...



APPLYING... RC2 GIVING...



APPLYING... RIND1 GIVING...



La forme linéaire finale

OK

5.0 Conclusion

L'examen des nouveaux outils d'observation et de compréhension disponibles en LISP nous amène à infléchir notre pensée sur la programmation: comme le suggère (WINOGRAD 1979) il semble aujourd'hui assez net que l'activité principale en programmation n'est plus essentiellement la création de programmes nouveaux, mais bien plutôt l'*intégration*, la *modification* et l'*élucidation* de programmes déjà existants. Par ailleurs, de tels outils donnent à penser que la notion de langage de programmation n'est plus aussi nette qu'auparavant, en ceci que la distinction entre langage et environnement de programmation n'est plus clairement fixée. Nous attendons des développements à venir qu'ils permettent la mise en jeu, pour le même programme, de représentations multiples, autorisant la programmation directe par manipulation de représentations très proches de l'intuition du programmeur. Cette intuition peut varier au cours du développement de programmes d'une certaine ampleur. Nous attendons des nouveaux systèmes qu'ils suivent au plus près cette intuition variable, en rendant interactivement cette intuition *visible*.

Dans cette perspective, le problème de la mise au point nous paraît être de nature essentiellement *interprétative*: concernant moins le texte statique d'un programme qu'un ensemble très mal connu de processus intellectuels. Nous aurons à l'avenir besoin d'un instrument qui nous permette d'*observer* ces processus. Les données recueillies par un tel instrument peuvent nous permettre d'inférer la structure interprétative qui manipule, modifie et combine les représentations de connaissances en programmation qui sont mises en jeu implicitement par les programmeurs.

6.0 Bibliographie

- [1] CHAILLOUX J. *VLISP 10.3, Manuel de Référence* Université Paris-8-Vincennes, Août 1978
- [2] CICCARELLI E. *An Introduction to the EMACS Editor* M.I.T. Artificial Intelligence Memo 447, January 1978
- [3] De MILLO R.A., LIPTON R.J., PERLIS A.J. *Social Processes and Proofs of Theorems and Programs* Comm. ACM, Vol 22, no 5, May 1979, 271-280
- [4] DERSHOWITZ N., MANNA Z. *The Evolution of Programs : A System for Automatic Program Modification* Stanford Artificial Intelligence Laboratory, Memo AIM-294, December 1976
- [5] GERHART S.L., YELLOWITZ L. *Observations of Fallibility in Applications of Modern Programming Methodologies*, IEEE Transactions on Software Engineering, Vol.SE-2, no 3, September 1976, 195-207
- [6] GOOSSENS D. *Meta-interpretation of Recursive List-processing Programs* I.J.C.A.I 1979, August 20-23, Tokyo, s7-s11
- [7] GREUSSAY P. *How to Use the RAINBOW System* Université Paris-8-Vincennes, Département d'Informatique, RT-79-5, Mars 1979
- [8] SUSSMAN G.J. *The Virtuous Nature of Bugs* Proc. of AISB Summer Conference, Brighton, July 1974, 224-237
- [9] SUSSMAN G.J. *A computer model of skill acquisition* American Elsevier, N.Y., 1975
- [10] WERTZ H. *Automatic Program Debugging* I.J.C.A.I 1979, August 20-23, Tokyo, 951-953
- [11] WINOGRAD T., *Beyond Programming Languages*, C.A.C.M., Vol. 22, no. 7, July 1979, 391-401