CORRECTION AUTOMATIQUE DE PROGRAMMES LISP

HARALD WERTZ

Université Paris 8
U.E.R. Informatique
Route de la Tourelle
75571 PARIS CEDEX 12

RÉSUMÉ

On examinera un système de compréhension automatique de programmes LISP : PHENARETE. Ce système, écrit en VLISP est utilisé à l'Université Paris 8 par les étudiants débutants en programmation pour la mise au point de leurs programmes.

PHENARETE analyse les programmes méthodiquement à l'aide d'un processus de méta-évaluation et en utilisant une bibliothèque de règles de construction et de correction de programmes. Si elle trouve des erreurs ou des inconsistences, PHENARETE synthétise des versions corrigées du programme.

Dansce cadre nous exposerons notre méthode de méta-évaluation.

1 - INTRODUCTION

Dans n'importe quel centre de calcul universitaire se trouve un "conseiller" de programmation. Cette personne, normalement un étudiant avancé ou un "hacker" du système, en utilisant son expérience de programmation, est capable d'expliquer des messages d'erreurs cryptiques, de répondre à des questions sur la syntaxe et la sémantique des langages de programmation, de détecter des erreurs évidentes, d'indiquer des "bugs" potentiels et de fournir toute sorte d'assistance au programmeur. Puisque normalement plusieurs personnes cherchent de l'assistance, il préfère des questions directes aux questions vagues et générales.

Il existe des similarités et des différences intéressantes entre un conseiller en programmation et l'assistant programmeur décrit par Winograd (Winograd 75). Tous les deux assistent dans la recherche et dans la correction d'erreurs, et tous les deux répondent à des questions sur la programmation en général ou sur un programme spécifique. Les différences entre les deux concernent la taille des programmes considérés, le genre d'"utilisateur" attendu et les types des questions rencontrées. Le conseiller traite normalement des programmes relativement petits ou des petites sections de grands programmes, et il interagit principalement avec des programmeurs débutants qui ont des connaissances de programmation plutôt limitées. Même si un programmeur plus expérimenté utilise les services du conseiller, les questions ont tendance à être directes et peu sophistiquées. D'autre part, l'assistant aiderait des utilisateurs sophistiqués dans le développement de grands programmes et, en conséquence, il devrait traiter des demandes plus complexes qui exigent un degré de compréhension plus élevé.

Ainsi que nous l'avons déjà montré, le conseiller traite des questions plus directes, pouvant éventuellement simplifier les capacités déductives qui lui sont nécessaires. Les petits programmes se révèlent plus faciles à comprendre. Bien qu'il soit à craindre que des travaux sur des programmes relativement petits se révèlent inopérants pour de grands programmes, il faut prendre en considération qu'actuellement il n'existe pratiquement pas de consensus sur la modélisation de programmes, tels qu'ils puissent être "compris" par un système. Un facteur supplémentaire réside en l'énorme demande de conseillers (du moins dans certaines Universités). Cette situation simplifie la collecte d'informations pour la spécification et la classification des tâches d'un conseiller, et lui fournit un grand nombre d'usagers pour tester un système de "conseils à la programmation" implémenté.

2 - PHÉNARÈTE

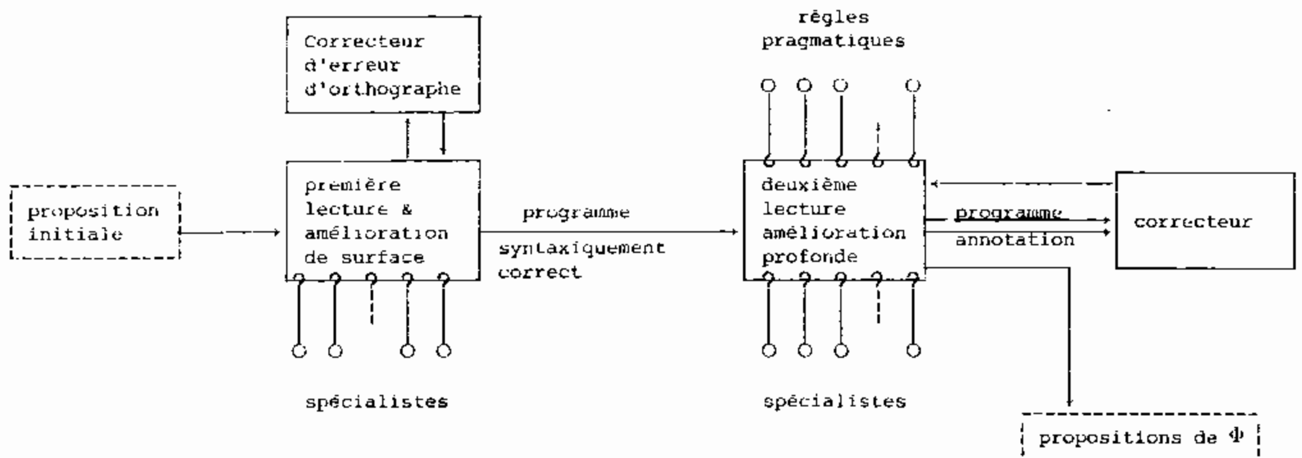
Notre recherche vers un "conseiller de programmation" nous a amené à construire un système de détection et de correction d'erreurs dans des programmes LISP écrits par des programmeurs novices. Ce système, nommé PHENARETE (Φ), est, en quelque sorte, une proposition pour un nouvel outil de mise au point, ne contenant pas les faiblesses des outils existants. Ainsi l'utilisateur de Φ ne travaille que sur une seule représentation des programmes, contrairement à la plupart des éditeurs qui demandent à l'utilisateur une familiarisation avec des représentations multiples d'un programme et une maîtrise du langage d'édition en plus du langage utilisé pour écrire son programme. Φ ne demande pas les informations supplémentaires comme des assertions, des invariances, des commentaires, etc, des informations que demandent la plupart des systèmes de vérification de programmes (Goldstein 74, Igarashi & al. 75, Rich & Shrobe 76). Le point

principal de Φ consiste à participer activement aux travaux du programmeur : contrairement aux autres outils de mise au point (traceurs, steppers,breaks,éditeurs, vérificateurs) .Φ ne se contente pas de donner des indications d'erreurs ou d'exécuter les commandes de l'utilisateur, mais lorsq'elle a détecté une erreur, elle essaye de la corriger et de livrer à l'utilisateur des propositions des programmes améliorés.

Φ ne tient actuellement pas compte des intentions du programmeur. Ainsi elle ne peut détecter que des erreurs directement déductibles du code du programme : des inconsistences ou des informalités. Pour cela elle utilise un ensemble de modules "spécialistes", représentant une implémentation d'une sémantique opératoire de LISP, une bibliothèque de "règles pragmatiques", exprimant des connaissances générales sur la bonne formation de programmes et sur la correction d'erreurs,et d'un processus de méta-évaluation qui constituera le sujet principal de cette étude.

Φ procède en plusieurs étapes : d'abord elle effectue une lecture préliminaire de détection et correction d'informalités de surface. Pour cela elle explore les programmes, en collectant des informations pour des interprétations ultérieures. A chaque rencontre d'un nom de fonction (fonction LISP standard ou fonction utilisateur déjà traitée) un spécialiste de cette fonction se lance automatiquement par le processus de "data driven function invocation" (Sandewall 75).Ces spécialistes incarnent le savoir de Φ sur l'utilisation et les effets des fonctions auxquelles ils sont associés.

Le résulat de cette première lecture — un programme syntaxiquement correct — est ensuite interprété et méta-évalué afin de vérifier si le programme est bien formé. Lorsque Φ détecte des informalités ou des inconsistances, elle annote la partie correspondante du programme et transmet le code et l'annotation au correcteur, module de Φ chargé d'éliminer les erreurs. Ce processus s'arrête lorsque Φ ne trouve plus aucune amélioration possible



- FONCTIONNEMENT DE Φ -

3 - LA MÉTA-ÉVALUATION

La détection d'informalités dans des programmes devrait être valide quelles que soient les données sur lesquelles le programme travaille, c'est-à-dire pour *toutes* les entrées raisonnables. Une telle détection ne peut sûrement pas être faite en utilisant une collection finie d'entrées particulières, mais elle doit être faite avec des énoncés sur *toutes* les entrées. On peut utiliser la technique mathématique standard consistant à inventer des symboles pour représenter des données arbitraires, puis à essayer de démontrer la consistance en utilisant ces symboles. Si aucune propriété spéciale de ces symboles, autre que celles d'être valides pour toute entrée, n'est nécessaire ; alors la démonstration est valide pour *chaque* entrée spécifique. Si des propriétés spéciales des symboles doivent être présupposées pour construire la démonstration, alors une analyse de cas exhaustive peut être effectuée, livrant un ensemble de démonstrations, une pour chaque cas, qui donnent collectivement la démonstration complète.

Puisque Φ ne tient pas compte actuellement des données fournies lors d'appels initiaux, elle utilise ces données symboliques plutôt que des données réelles. Ainsi le programme est exécuté symboliquement sur des données symboliques.

En accord avec le vocabulaire utilisé par Balzer (Balzer 77) nous faisons une différence entre méta-évaluation et exécution symbolique.L'exécution symbolique est principalement orientée vers la vérification de programmes (King 75) — donc utilisant des assertions — ou vers la description des activités de programmes (Dershowitz & Manna 77, Goossens 78). L'exécution symbolique travaille sur un formalisme de représentation symbolique de données, qui reflète exactement les activités du programme et qui doit *toujours* être valide pour *toute* donnée réelle.

Notre travail est principalement orienté vers la correction automatique de programmes et de ce fait — bien qu'un développement ultérieur soit tout à fait souhaitable et soit prévu — nous n'avons pas implémenté la totalité du mécanisme de l'exécution symbolique. Nous en utilisons une forme simplifiée dans laquelle nous ne travaillons pas, dans la plupart des cas, avec des formules algébriques exactes pour décrire les données symboliques, mais plutôt avec des formes de représentation des *types* de valeur symbolique. Cette forme de représentation des données est moins complète et moins précise que la forme algébrique, mais tout à fait suffisante pour détecter — avec l'aide des modules "spécialistes" et des règles pragmatiques — des inconsistances dans les programmes.

3.1 - Détermination des données symboliques

Rappelons que nous n'utilisons ni commentaire ni assertion, et que nous sommes donc obligés de déduire les données-symboliques-possibles du code, éventuellement faux, du programme. Ceci crée une situation très incertaine où nous pouvons en permanence être amenés à changer, à modifier les hypothèses déjà faites.

Toutefois, pendant la première analyse, nous formons à l'aide des spécialistes les premières hypothèses sur les types de valeurs des variables. Pour ceci, les spécialistes associés aux fonctions mettent en jeu des connaissances sur le type de valeur que les différents arguments doivent prendre.

Prenons un exemple simple : si Φ rencontre quelque part la forme

(CONS A L)

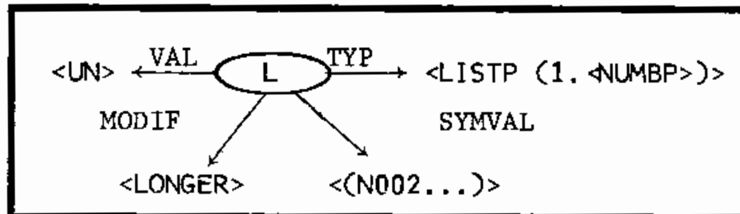
par la définition du spécialiste-CONS, elle peut déduire que la valeur de L doit être une liste, mais elle ne peut encore rien déduire sur le type de A. Si maintenant nous trouvons la forme

(CONS (ADD1 A) L)

aussi bien le type de la variable A (un nombre) que le type de L (une liste) peut être déduit, et pendant la méta-évaluation, Φ peut conclure que le résultat de cette instruction sera une liste avec au moins un élément, et que le premier élément est un nombre. C'est-à-dire, si l'on trouve la forme

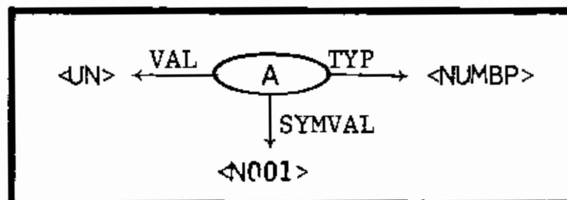
(SETQ L (CONS (ADD1 A) L))

Φ aurait les informations suivantes sur L

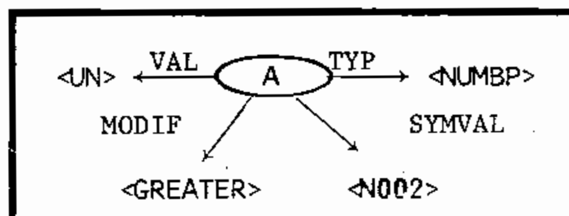


TYP : indicateur du type de la valeur
VAL : indicateur de la valeur
SYMVAL : valeur symbolique
MODIF : indicateur de modifications
UN : inconnu

et sur A les informations



La valeur symbolique de A et le CAR de la valeur symbolique de L sont deux valeurs différentes, puisque Φ considère, à juste titre, les deux valeurs comme différentes. Si, au lieu d'écrire (ADD1 A), on avait écrit (INCR A)ou (SETQ A (ADD1 A)) Les deux valeurs symboliques en question auraient été identiques et A aurait la forme :



On remarque sûrement la redondance d'informations dans ces descriptions des variables. Cette redondance est dûe au fait que nous n'utilisons des valeurs symboliques qu'à l'intérieur des répétitions (boucles) et que pour l'analyse de programmes linéaires (séquentielle sans boucle) nous n'utilisons que l'information

du TYP de la variable, celle-ci étant largement suffisante pour détecter des inconsistances dans un programme linéaire.

Considérons maintenant le cas de la suite d'instructions que voici :

```
. 1 .  (SETQ L (CONS (ADD1 A) L))
. 2 .  (CONS (ADD1 B) (CAR L))
```

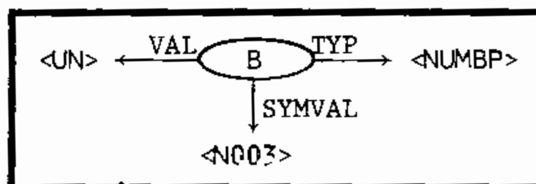
et intéressons nous maintenant à la ligne .2.. Par un procédé identique à celui appliqué à l'atome A nous déduisons facilement que B est un atome numérique. Mais, comme précedemment, la fonction CONS attend comme deuxième argument une liste. Ici, Φ reconnaît immédiatement, par l'information détectée sous l'indicateur TYP, que (CAR L) a une valeur atomique. Ne connaissant pas les "paires pointées" , Φ conclut qu'il y a une inconsistance et invoque donc le correcteur d'erreurs, qui a accès à une règle pragmatique, indiquant qu'une façon simple de transformer un argument X atomique en une liste est d'y appliquer la fonction LIST. Le résultat de cette analyse va être une proposition pour modifier la ligne .2. en la ligne:

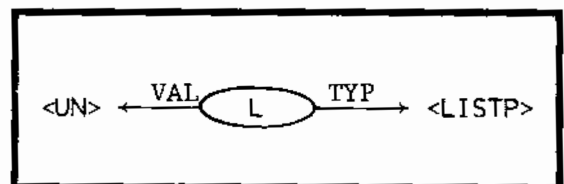
```
(CONS (ADD1 B) (LIST (CAR L)))
```

Un cas différent se présente si l'on suppose que Φ ne rencontre que la ligne .2., i.e. isolement la ligne suivante :

```
(CONS (ADD1 B) (CAR L))
```

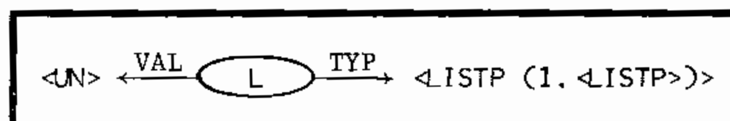
Après la première lecture, ou Φ ne travaille qu'à une profondeur 1 , elle sait que B est un atome numérique et L une liste, grâce aux spécialistes ADD1 et CAR.





Au raisonnement "en avant" appliqué précédemment, Φ substitue maintenant dans ce cas-ci le raisonnement "en arrière".

A nouveau, CONS attend un premier argument d'un type quelconque et comme deuxième argument une liste. Vérifier que (CAR L) est une liste, est indécidable . Φ peut donc supposer que le code est juste et ne contient pas d'inconsistances.Mais ainsi elle a une information supplémentaire sur L, exprimant que le premier élément de L doit être une liste. Cette information peut être,éventuellement, utile dans la suite de l'analyse du programme, donc elle peut mettre à jour la variable L :



Si le programme donne des informations sur des valeurs exactes, nous les utilisons tout naturellement. Si l'on trouve, par exemple, la sélection :

```
(COND
   ((EQ X 'A) (FOO X))
   ((MEMQ X '(A B)) (BAR X))
   (T (FIE X)))
```

nous pouvons déduire que la valeur de X est égale à l'atome A à l'appel de FOO, égale à l'atome B à l'appel de BAR et différente de A et B à l'appel de FIE.

Les énoncés directs de valeurs à l'intérieur des programmes ne posent aucun problème : ils facilitent le travail de déterminantion des valeurs et des types des variables.

Ainsi, on peut au fur et à mesure de l'analyse d'un programme, collecter de plus en plus d'informations qui à leur tour peuvent être utilisées pour une analyse plus approfondie.

Toutefois, notre forme de représentation devient très lourde si les structures de données augmentent en complexité et actuellement nous sommes en train de chercher d'autres formes de représentations symboliques. Eventuellement nous adapterons la représentation de listes développée par D. Goossens (Goossens 78).

Il convient de noter que la détermination automatique des valeurs symboliques est une des différences fondamentales avec les systèmes de vérification de programmes fondés sur l'exécution symbolique. Dans ces systèmes, les valeurs symboliques initiales sont données a priori par le biais des assertions d'entrée, qui déterminent exactement les types et les domaines des variables. Même les systèmes synthétisant les assertions d'invariants n'ont pas la possibilité de déduire les données symboliques du texte du programme et, eux aussi, utilisent l'information donnée dans les assertions pour les déterminer.

3.2 - Méta-évaluation

Dans ce paragraphe nous donnons un programme exemple et nous décrivons en détail le processus de méta-évaluation.

Programme exemple :

```
. 1 . (DE DIFFERENCE (X Y N)
. 2 .       (SETQ N 0)
. 3 .       (WHILE (GT X Y)
. 4 .           (SETQ N (ADD1 N))
. 5 .           (SETQ X (SUB1 X))
. 6 .           (SETQ Y (SUB1 Y)))
. 7 .       N)
```

C'est un programme qui, si l'on élimine l'instruction

(SETQ Y (SUB1 Y))

ou si l'on élimine l'instruction

(SETQ X (SUB1 X))

et change l'instruction

(SETQ Y (SUB1 Y))

en

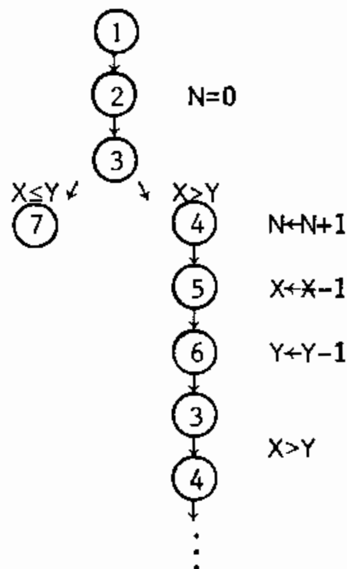
(SETQ Y (ADD1 Y))

livre comme résultat la différence entre X et Y, i.e. la valeur de N serait X-Y

si X>Y sinon 0.

Dans l'état actuel, le programme s'arrête si X≤Y en ramenant la valeur 0, et, si X>Y le programme ne s'arrête jamais et boucle indéfiniment.

L'arbre de l'exécution sera

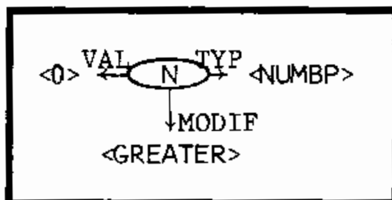


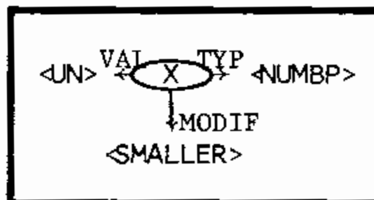
Après la première lecture nous savons que

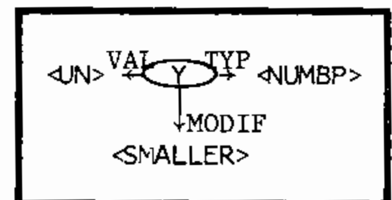
N est un nombre
est initialisé à la valeur 0
et modifié d'une façon qui augmente sa valeur

X et Y sont des nombres
modifiés d'une façon qui diminue leur valeur

Graphiquement :







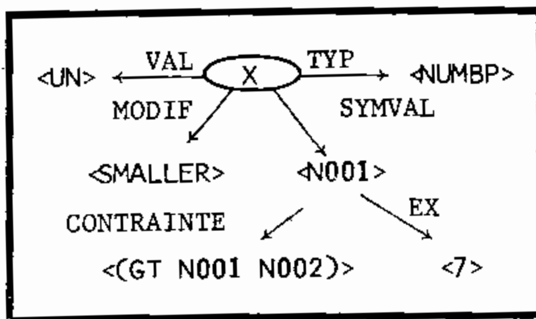
Φ va donc utiliser ces connaissances pour commencer la méta-évaluation. D'abord ligne .2. (SETQ N 0) ne lui apprend rien de nouveau et ne fait que confirmer les valeurs sur N. En ligne .3. (WHILE (GT X Y), puisque rien sur les valeurs exactes de X et Y n'est connu, nous devons procéder à une analyse de cas. Comme les prédicats LISP (ou les expressions booléennes en général) n'ont que deux valeurs possibles, T ou NIL (vrai ou faux), une étude de deux cas est suffisante. Pour cela il faut créer des valeurs symboliques appropriées X et Y.

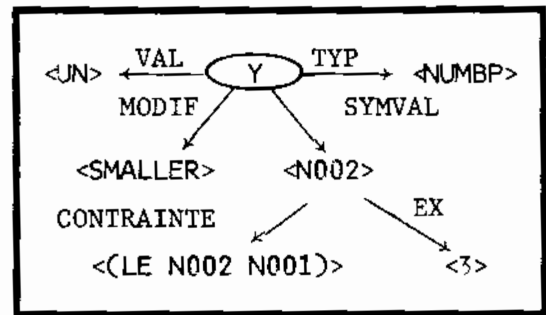
Comme nous l'avons montré dans le paragraphe précédent, nous ne créons des valeurs symboliques que pour les répétitions. Ceci n'est pas tout à fait exact, précisons: Φ crée des valeurs symboliques pour toutes les variables qu'elle trouve à l'intérieur d'une expression booléenne, i.e. d'un prédicat.

Quelle valeur faut-il créer? Notre politique est de commencer en choisissant une valeur la plus générale possible et de la raffiner au fur et à mesure que le programme exprime des contraintes supplémentaires sur ces variables.

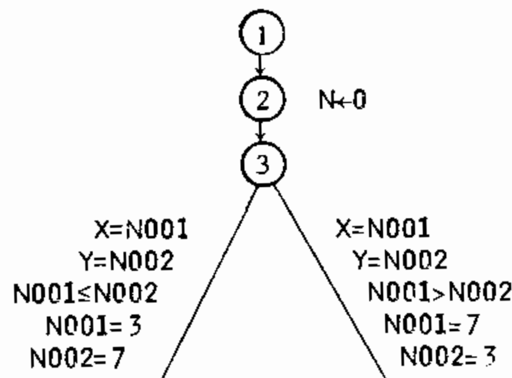
Dans le cas présent, Φ va générer deux indicateurs particuliers (par un GENSYM) N001 et N002. Les noms "gensymés" commençant par la lettre N sont obligatoirement considérés comme des entiers. Ces valeurs symboliques vont subir une restriction exprimée comme prédicat sur leur P-liste, et contenir une valeur aléatoirement générée, qui correspond au prédicat, dans leur C-valeur.

En supposant que Φ ait généré les valeurs 7 et 3 pour X et Y respectivement, nous avons graphiquement les valeurs :



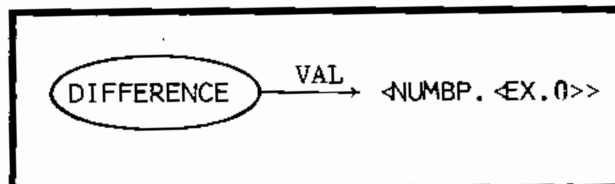


et les 2 chemins possibles



qu'il faut développer. Choisissons d'abord le chemin correspondant à la valeur booléenne "faux" (le chemin gauche de l'arbre).

Plus aucune instruction ne suit la ligne .7., donc Φ peut être sûre que dans ce cas il n'y a pas d'inconsistances, et elle peut mettre sur la P-liste de DIFFERENCE l'information indiquant qu'une valeur possible de l'application de cette fonction est un nombre (le nombre 0).



Dans le second cas, quand le prédicat est vrai, la méta-évaluation continue avec la ligne .4., à l'intérieur de la boucle. Sachant qu'elle va revenir en ligne.3. en vertu de la répétition, Φ va conserver une copie des valeurs des variables du prédicat (i.e. des atomes cognitifs) sur une pile pour les comparer ensuite aux valeurs, éventuellement modifiées, de ces variables après exécution de la boucle.

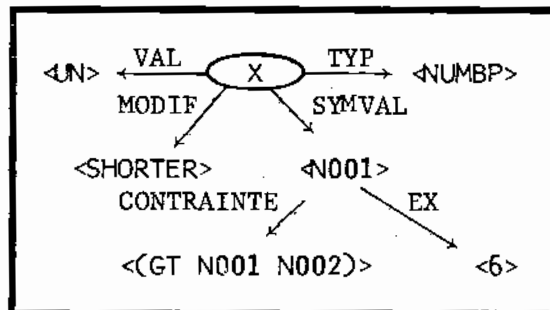
Chaque boucle n'est méta-évaluée qu'une seule fois.

ligne .4. (SETQ N (ADD1 N))

n'apporte pas de nouvelles informations et nul besoin de changer la valeur symbolique de N, puisque N n'est pas une variable sur laquelle porte le prédicat

ligne .5. (SETQ X (SUB1 X))

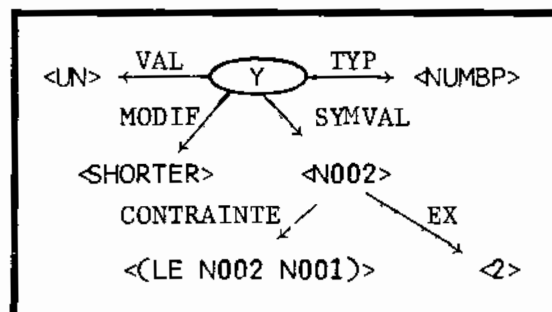
par contre nous oblige à adapter la valeur symbolique de X, ce qui donne :



et la méta-évaluation de ligne .6.

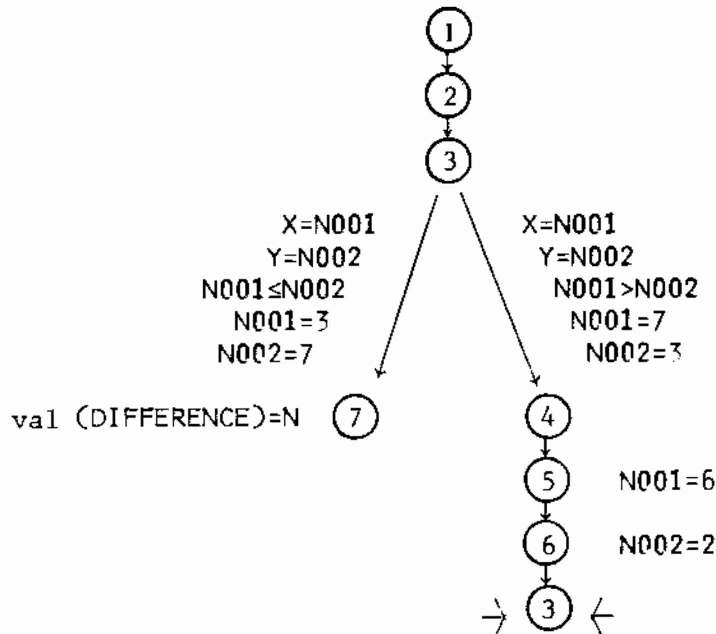
(SETQ Y (SUB1 Y))

nous donne



L'instruction suivante à méta-évaluer est la ligne .3., donc l'entrée de la boucle. Nous avons évalué une fois la boucle et nous pouvons donc interrompre l'évaluation pour analyser celle-ci.

L'arbre de la méta-évaluation, à ce stade, est :



3.3 - Détermination de la convergence

La méta-évaluation n'est qu'un outil pour constituer une description, sur des données abstraites, de l'activité de portions de programmes, et pour recueillir des informations abstraites (symboliques) suffisantes pour être traitées par des programmes sémantiques. C'est-à-dire que

a) l'implémentation de la méta-évaluation est purement algorithmique

et b) elle ne peut rien faire d'autre qu'exécuter symboliquement le programme. Tout autre travail tel que synthèse de propositions en est tout à fait indépendant.

La méta-évaluation ne permet que de transcrire le code du programme en un autre code, éventuellement plus compréhensible pour l'humain ou pour la machine.
Goossens (Goossens 78) propose un système complet d'exécution symbolique dans le but de livrer une description "visuelle" de l'activité du programme. Cette description visuelle peut alors être comparée par le programmeur à ses intentions afin de déterminer si l'activité du programme correspond à ses intentions (i.e. pour vérifier le programme).
Pour détecter des inconsistances, ou l'absence d'inconsistance, et pour les corriger, il nous faut plus. Ces outils supplémentaires sont précisément les règles pragmatiques. Elles utilisent les expressions symboliques calculées par le procédé de méta-évaluation.

Dans certains cas la méta-évaluation n'est pas obligatoire, et dans Wertz (Wertz 1976) nous avons exposé la première version de Φ qui ne contenait aucun module de méta-évaluation. Cette version procédait alors par recherche de patterns et,

dans le cas où cette recherche était vaine, elle introduisait alors du code correspondant aux patterns absents. Par exemple dans une boucle de la forme:

```
(WHILE (GT X Y)
       (SETQ X (SUB1 X))
       (SETQ N (ADD1 N)))
```

elle cherchait dans le corps de la boucle, une instruction de la forme

```
      (SETQ X (SUB1 X))
      (SETQ Y (ADD1 Y))
(1)   (DECR X)
      (INCR Y)
      (SETQ Y (PLUS Y valeur-numérique-positive))
      (SETQ X (DIFFER X valeur-numérique-positive))
```

et si elle trouvait une instruction de cette suite de patterns à l'intérieur, elle supposait la boucle terminée. Ce système ne fonctionne plus dans le cas d'une boucle de la forme :

```
(WHILE (GT X Y)
       (SETQ N (ADD1 N))
       (SETQ X (SUB1 X))
       (SETQ Y (SUB1 Y)))
```

en effet il s'agit d'une boucle identique à celle de notre programme exemple. Le système ne réalisait pas que X restera toujours supérieur à Y, i.e. la distance entre X et Y ne diminue pas.

La rencontre de formes de boucles analogues dans des programmes d'étudiants nous imposait de trouver d'autres solutions.

Dans le système actuel nous interrompons la méta-évaluation après chaque parcours d'une boucle pour y appliquer les règles pragmatiques concernant les répétitions.

L'idée consiste à raffiner progressivement les règles jusqu'à ce qu'en soit obtenue une qui décèle des informalités (inconsistances), ou jusqu'à ce que toutes les règles s'appliquant à la partie code actuellement analysée aient été exécutées.

Dans le cas où cette règle de dépendance de la boucle du prédicat s'applique, un spécialiste du prédicat est lancé qui propose au correcteur un certain nombre de possibilités de correction.

Si nous avions eu la boucle

```
(WHILE (GT X Y)
       (SETQ N (ADD1 N)))
```

la règle aurait été falsifiée et le spécialiste GT aurait proposé l'introduction d'une des instructions citées en *(1)*. C'est l'ensemble des patterns de modification que la première version de Φ recherchait à l'interieur de cette boucle. Le correcteur en aurait choisit (dans le cas présent) la première instruction et l'analyse aurait continué avec la boucle modifiée suivante :

```
(WHILE (GT X Y)
       (SETQ N (ADD1 N))
       (SETQ X (SUB1 X)))
```

une boucle terminante.

La boucle de notre programme DIFFERENCE ne contredit pas cette règle, et nous pouvons donc appliquer la règle suivante : la règle de bonne formation structurelle de boucles.

Celle-ci trouve sur la P-liste du prédicat tout un programme pour déterminer si la règle est vérifiée.

Le travail consiste à tester si la distance entre X et Y diminue.

$$\mathcal{D} = (val_{avant}(X) - val_{avant}(Y)) - (val_{après}(X) - val_{après}(Y))$$

i.e. en faisant la différence entre les valeurs de X et Y avant l'exécution de la boucle, la différence entre les valeurs de X et Y après l'exécution de la boucle et en faisant la différence des deux résultats.

Trois cas peuvent alors se présenter :

.1. $\mathcal{D} = 0$

.2. $\mathcal{D} > 0$

.3. $\mathcal{D} < 0$

et suivant le cas, il faut choisir entre 3 stratégies différentes :

1) $\mathcal{D} = 0$ implique que la distance reste constante, il faut alors appliquer le même traitement que dans le cas du non changement des variables du prédicat

2) $\mathcal{D} > 0$ implique que les modifications des variables se font dans la bonne direction, il ne reste qu'à vérifier si, après un nombre quelconque d'éxécution de la boucle, le prédicat prend la valeur vraie

3) $\mathcal{D} < 0$ implique que les modifications des variables se font dans le sens inverse de celui dicté par le prédicat, en supposant que le prédicat soit vrai, il faut d'abord ramener la boucle au cas n°1, et ensuite le traiter comme en 1. Si l'on suppose que les modifications sont justes, il faut changer le test(le prédicat) en le test inverse*.

Dans le cas présent nous avons $\mathcal{D} = 0$. Alors la procédure identique à celle décrite plus haut s'applique : Φ introduit une des instructions suivantes

```
(SETQ X (SUB1 X))
(SETQ Y (ADD1 Y))
      :
```

* Actuellement le système considère que le prédicat est toujours juste, mais nous prévoyons d'introduire le code nécessaire pour que le système reconnaisse l'ambiguité et livre dans ce cas également plusieurs propositions.

et modifie la boucle en :

```
(WHILE (GT X Y)
       (SETQ N (ADD1 N))
       (SETQ X (SUB1 X))
       (SETQ Y (SUB1 Y))
       (SETQ X (SUB1 X)))
```

Une optimisation supplémentaire transforme la boucle en :

```
(WHILE (GT X Y)
       (SETQ N (ADD1 N))
       (SETQ X (DIFFER X 2))
       (SETQ Y (SUB1 Y)))
```

Deux remarques sont à faire sur cette boucle :

- le résultat proposé par Φ ne correspond pas à l'analyse du programme que nous avons menée au début de ce paragraphe, mais Φ peut par la suite "prouver" que la boucle termine

- les transformations effectuées sur cette boucle montrent très bien que Φ demeure inconsciente des intentions du programmeur. Le programme ainsi modifié va encore caluler la différence entre X et Y, mais si l'utilisateur tenait à garder la valeur de X ou de Y pour des calculs ultérieurs, il serait sûr, dès à présent, que le programme modifié ne correspond plus aux intentions du programmeur.

Insistons encore une fois : Φ n'est pas un système de vérification de programme, mais un programme d'aide à la mise au point, Φ ne livre pas des solutions, mais doit donner des idées sur les corrections possibles du programme.

Les prédicats GE, LE, LT et ZEROP sont traités de façon similaire. Les prédicats (NULL (GE ...)), etc. se transforment ainsi :

```
(NULL (GT X Y)) ⇒ (LE X Y)
(NULL (GE X Y)) ⇒ (LT X Y)
(NULL (LE X Y)) ⇒ (GT X Y)
(NULL (LT X Y)) ⇒ (GE X Y)
```

Des tests sur des listes comme

```
(WHILE L ...)
(WHILE (CDR L) ...)
...
```

sont traités naturellement par un raisonnement sur la longueur des listes.

Toutefois, à partir d'une certaine complexité de calcul, Φ ne parvient plus à vérifier la terminaison et ne donne alors qu'une indication d'erreur et une explication (sommaire) des raisons pour lesquelles elle ne peut pas encore faire des propositions de correction.

Ces limitations ne sont alors pas dûes à l'algorithme de méta-évaluation mais à la faiblesse de notre représentation des données symboliques. Tant que les programmes ne travaillent que sur des nombres, on pourrait facilement introduire un module de traitement algébrique et un démonstrateur de théorèmes. Mais dès que les données seront des listes de structure arbitraire (arbres, graphes, etc.) notre représentation deviendra trop lourde et incomplète. Des travaux théoriques et pratiques plus approfondis sur la représentation de données symboliques s'imposent. Ce dont nous avons besoin, ce sont des "concepts" et des "lois" qui ne sont pas seulement corrects, mais qui reflètent notre compréhension intuitive des listes, comme les concepts d'addition et de multiplication ainsi que les lois d'associativité, de commutativité et de distributivité reflètent notre compréhension intuitive des nombres. Une fois les bons concepts et lois dégagés, il sera comparativement trivial de développer une notation (une représentation) qui facilite leur application. Un premier pas dans cette direction est fait par (Elgot & Snyder 1977) qui développent une notation polynomiale pour les listes.

3.4 - Suite de la méta-évaluation de DIFFERENCE

Après chaque modification que Φ apporte au texte du programme source, Φ va re-analyser le programme entier. Elle procède par des approximations successives.

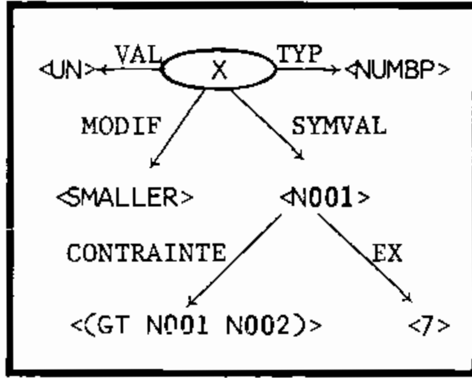
Après avoir analysé la ligne .7., Φ recommence l'analyse de la fonction DIFFERENCE. Toutes les déductions faites pendant la première analyse de la boucle sont ignorées.

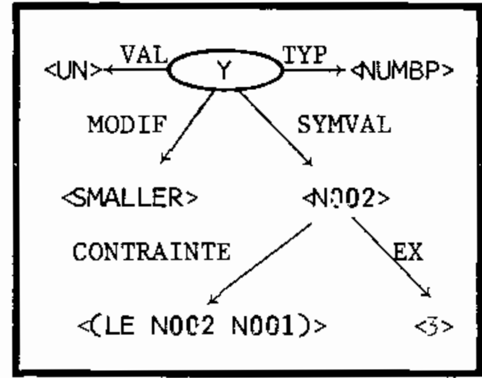
Rappelons toutefois le programme tel qu'il est à présent

```
.1.   (DE DIFFERENCE (X Y N)
.2.       (SETQ N 0)
.3.       (WHILE (GT X Y)
.4.              (SETQ N (ADD1 N))
.5.              (SETQ X (DIFFER X 2)
.6.              (SETQ Y (SUB1 Y)))
.7.       N)
```

Comme à la première lecture, Φ n'apprend rien de nouveau et ne trouve pas d'informalités dans les deux premières lignes. En ligne .3. elle n'a pas davantage de connaissances sur les valeurs de X et Y, et ne peut donc ni falsifier ni vérifier le prédicat. Elle doit procéder à une analyse de cas et analyser la suite du programme, une fois pour les valeurs de X et Y telles que X≤Y et une fois pour des valeurs de X et Y telles que X>Y.

La branche de l'arbre d'évaluation symbolique correspondant aux données de X et de Y telles que X≤Y, est tout à fait identique à celle de l'analyse précédente. Regardons donc le cas où la méta-évaluation continue à l'intérieur de la boucle. Les valeurs de X et Y sont comme auparavant, et Φ sauvegarde une copie de ces valeurs sur une pile pour les comparer ensuite avec les valeurs de X et Y après *une* méta-évaluation de la boucle.

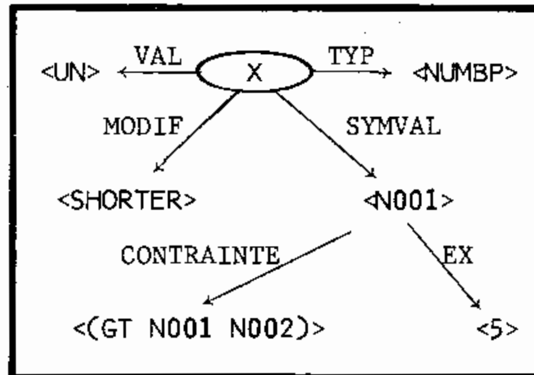




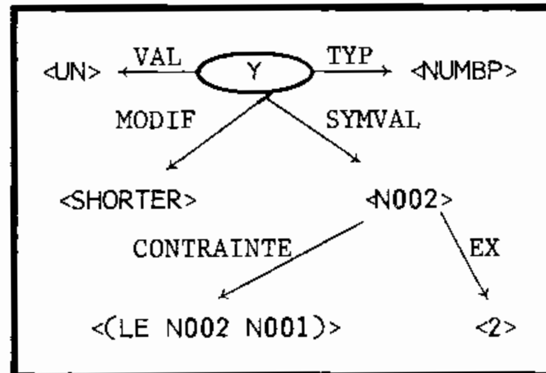
L'évaluation unique des boucles correspond à l'exécution symbolique des boucles utilisant des assertions d'invariantes. Dans ces cas on exécute la portion de programme comprise entre deux assertions, et on tente de prouver la seconde assertion à l'aide d'un démonstrateur de théorèmes. Dans notre cas on peut s'imaginer qu'il y ait des assertions implicites à l'entrée de la boucle, qui correspondent aux prédicats. Nous n'essayons pas de les prouver - c'est impossible - mais de démontrer qu'ils représentent des points de convergence ou de passage pour les valeurs successives de variables.

Poursuivons la méta-évaluation : la ligne .4. (SETQ N (ADD1 N)) ne se rapporte pas à des variables du prédicat et ne contient pas d'inconsistance, elle est donc méta-évaluée rapidement, et Φ trouve ses suppositions sur N vérifiées : N doit être un nombre et N est modifié.

La ligne .5. (SETQ X (DIFFER X 2)) porte sur une variable du prédicat et nous oblige donc à adapter la valeur symbolique de X, ce qui donne :



et la méta-évaluation de la ligne .6. (SETQ Y (SUB1 Y)) livre l'atome cognitif Y modifié que voici :



La ligne .6. étant la dernière instruction de la boucle, nous devons donc interrompre l'évaluation et analyser la boucle modifiée.

La première règle à appliquer, la règle de dépendance de la boucle du prédicat, est naturellement vérifiée : aussi bien X que Y sont modifiées à l'intérieur de la boucle. L'application de la règle de bonne formation structurelle implique le calcul de

$$\mathbb{D} = (val_{avant}(X) - val_{avant}(Y)) - (val_{après}(X) - val_{après}(Y))$$

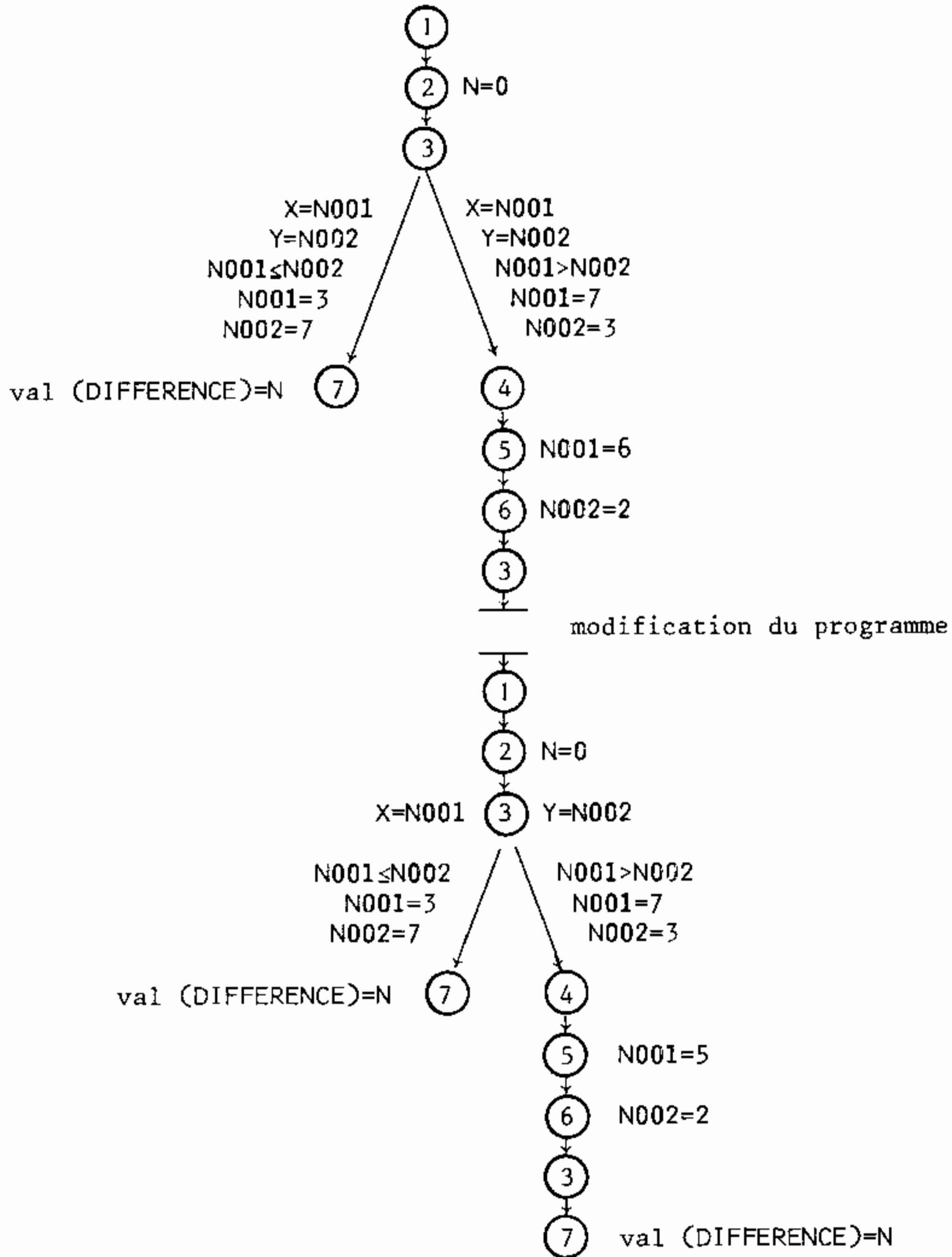
dans notre cas, avec pour X et Y, avant la boucle, les valeurs 7 et 3 respectivement, et pour la suite de la méta-évaluation de la boucle les valeurs 5 et 2 respectivement pour X et Y. Ce qui donne

$$\mathbb{D} = (7 - 3) - (5 - 2) = 4 - 3 = 1$$

Ceci implique que les modifications de X et Y se font dans la bonne direction. La vérification que la boucle termine, donc qu'après un nombre fini d'exécutions de la boucle le prédicat prend la valeur vraie, est faite puisque Φ possède à présent des connaissances suffisantes pour le déduire du fait que la distance entre X et Y diminue.

Donc Φ n'a plus à détecter des inconsistances dans le programme modifié et il ne reste qu'à méta-évaluer la ligne .7. qui se ramène à une méta-évaluation d'une variable.

- ARBRE COMPLET DE LA META-EVALUATION DE DIFFERENCE -



Conclusion

Dans cet article nous n'avons pleinement exposé que le processus de méta-évaluation. Ce n'est qu'un aspect de Φ, mais la performance et la puissance dusystème dépend essentiellement de ce processus. Les développements de la méta-évaluation pourront exercer une profonde influence sur les directions qu'un système de conseil à la programmation pourrait prendre.

L'intérêt du processus réside en sa similarité à la démarche humaine d'exécution conceptuelle de programmes ainsi qu'aux possibilités de formalisation qu'il offre, enfin dans l'introduction de *types* inédits dans un langage a priori sans type.

Le système Φ actuel est implémenté sur PDP-10 en VLISP-10 (Chailloux 76, Greussay 77) et occupe environ 12 K doublets LISP.

Bibliographie

DERSHOWITZ N. & MANNA Z. (1977) : *The Evolution of Programs : Automatic Program Modification*, I.E.E.E. Transactions on Software Engineering, Vol. SE-3, n°6, pp. 377-385

ELGOT C.C. & SNYDER L. (1977) : *On the many facets of lists*, IBM T.J Watson Research Center, RC-6449

GOLDSTEIN I. (1974) : *Understanding Simple Picture Programs*, M.I.T. A.I.-Laboratory AI-TR-294

GOOSSENS D. (1978) : *A System for Visual-Like Understanding of LISP programs*, Université Paris 8-Vincennes, Département d'Informatique, RT-1-78

IGARASHI S., LONDON R.L. & LUCKHAM D.C. (1975) : *Automatic Program Verification 1 : Logical Basis and its Implementation*, Acta Informatica, Vol. 4, pp. 145-182

KING M.C. (1975) : *A New Approach to Program Testing*, Proc. ACM Int. Conf. on Reliable Software, Los Angeles, CA, pp. 228-233

RICH Ch. & SHROBE H.E. (1976) : *Initial Report on a LISP Programmer's Apprentice* M.I.T. AI-Laboratory AI-TR-354

SANDEWALL E. (1975) : *Ideas about Management of LISP data bases*, M.I.T. AI-Laboratory, AI-Memo n°332

WERTZ H. (1976) : *Understanding LISP Programs in Improving LISP Programs*, G.I.-6 Jahrestagung, Stuttgart, sept-okt 1976, Springer Verlag, ed. E. J. Neuhold, pp. 427-441

WERTZ H. (1977) : *Understanding and Improving LISP programs*, (extended abstract) Proc. 5th IJCAI, M.I.T. août 22-25, p. 377

WINOGRAD T. (1975) : *Breaking the Complexity Barrier Again*, SIGPLAN Notices, Vol. 10 n°1, Jan. 75, pp. 13-22

CHAILLOUX J. (1976) : *Manuel VLISP-10, Université Paris 8-Vincennes*, Département d'Informatique, RT 17-76.

GREUSSAY P. (1977) : *Contribution à la définition interprétative et à l'implémentation des λ-langages*, Thèse, Université Paris 7