Etude d'un compilateur VLISP optimisant

Jérôme CHAILLOUX

Mai 1979

RESUME :

Nous décrivons dans ce papier, un compilateur LISP très optimisant. Le dialecte VLISP est utilisé à la fois comme langage source et comme langage d'écriture du compilateur. Ce compilateur est prévu pour fonctionner sur tous les interprètes VLISP existants (du PDP 10 au micro Intel 8080). Il engendre des instructions pour la machine virtuelle VM#2 ce qui assure sa portabilité et sa simplicité. L'exécution des fonctions ainsi compilées est en moyenne 6 fois plus rapide et l'espace mémoire utilisé pour ranger ces fonctions compilées se trouve réduit (de l'ordre de 4 fois).

1.0 INTRODUCTION

Depuis très longtemps, tout l'effort d'amélioration des performances du langage LISP [McCarthy 1962, Allen 1978] a porté sur son interprétation. De nombreuses techniques ont été étudiées en vue d'améliorer le temps de réponse de l'interprète et de gagner de la place en mémoire. Parmi les techniques les plus fructueuses citons :

- la liaison superficielle des variables (shallow binding) [Baker 77] qui permet de gérer l'environement LISP (i.e. les liaisons dynamiques variable-valeur) sans utiliser de A-LIST comme en LISP 1.5 [McCarthy 62]. Ce nouveau type de liaison associe à chaque atome (de type variable) un emplacement mémoire à adresse fixe, la C-VAL. Cette C-VAL contient à tout moment la valeur actuelle de la variable. L'accès à cette valeur est instantané (à une indirection près). En revanche, pour pouvoir traiter correctement la récursivité, la gestion de l'environement à l'entrée et à la sortie de chaque fonction se complique. Il faut en effet :
  (1) à l'entrée de chaque fonction, sauver (dans la pile par exemple) les anciennes valeurs des paramètres formels.
  (2) lier via la C-VAL les paramètres actuels.
  (3) et restaurer au retour de la fonction les anciennes valeurs des paramètres formels.
  Ce type de liaison très rapide ne permet plus de traiter les objets de type FUNARGS du LISP 1.5 [Moses 1970].

- la non-utilisation de doublets de travail (i.e. d'appels de la fonction CONS) pour les besoins propres de l'interprète [Chailloux 78c]. En particulier la fonction EVLIS n'est plus utilisée pour interpréter les λ-expressions ; elle est avantageusement remplacée par une procédure qui fabrique la liste des valeurs dans la pile de travail.

- l'évaluation itérative de fonctions récursives en position terminale (tail-recursion) [Greussay 76] permet un gain substanciel de temps mais surtout évite de faire déborder la pile de travail en cas de fausse récursion (même infinie).

- l'invocation directe de la fonction associée à chaque atome en fonction de son type sans utiliser la P-LIST de l'atome fonction [Chailloux 78a]. Pour ce faire les interprètes VLISP associent à chaque atome un emplacement mémoire fixe qui contient la valeur de la fonction associée à l'atome (la F-VAL) ainsi que son type codé (le F-TYP). Cette liaison superficielle fonctionnelle possède tous les avantages de la liaison superficielle des variables. Elle permet un accès extrêmement rapide aux fonctions et facilite l'implantation des fonctions dynamiques ESCAPE [Greussay 77] ou WHERE [Chailloux 79].

Pour permettre un nouveau saut quantitatif important dans les performances du langage LISP, l'étude d'un compilateur très optimisant s'impose. Les gains espérés (et obtenus) d'un tel compilateur sont de l'ordre de 400 à 1000 % pour les temps et de de 300 à 500 % pour l'espace mémoire.

Toutefois pour laisser ce compilateur dans le seul rôle d'accélérateur qui lui est assigné et pour le rendre totalement invisible de l'utilisateur, les spécifications suivantes ont été définies :

- le compilateur ne doit pas utiliser de déclarations spécifiques. En effet la plupart des compilateurs LISP existant utilisent des déclarations qui indiquent le mode de certaines variables (SPECIAL vs UNSPECIAL) voire leur type (FLOAT, FIX ...). De telles déclarations spécifiques au compilateur vont à l'encontre du principe d'invisibilité du compilateur.

- le compilateur doit pouvoir compiler une fonction ou un ensemble de fonctions de manière totalement incrémentale. Toutefois, dans le cas d'un ensemble de fonctions possédant des intéractions mutuelles, il est préférables de livrer au compilateur cet ensemble de fonctions, plutot que de faire compiler ces fonctions les unes après les autres.

- les fonctions à compiler doivent être correctes (i.e. interprétées sans erreur). Ce n'est pas au compilateur de détecter les erreurs (statiques ou dynamiques) mais à l'interprète, le compilateur se cantonnant uniquement dans son rôle d'accélérateur. Aucun diagnostic d'erreur n'est donc produit par le compilateur.

- le compilateur doit limiter au maximum le nombre de constructions non compilables. Cette limitation ne recouvre actuellement que la redéfinition dynamique de fonctions statiques ou standard.

- le compilateur lui-même doit être de taille réduite pour pouvoir être utilisé dans les interprètes VLISP sur micro-processeurs à mots de 8 bits [Chailloux 78a], mais doit pouvoir compiler de très gros systèmes écrits en VLISP tels que le système d'amélioration de programmes PHENARETE [Wertz 78] ou le système de méta-évaluation CAN [Gossens 77].

2.0 LA MACHINE VM#2

Le compilateur que nous décrivons, va fabriquer des instructions pour la machine virtuelle VM#2. Cette machine, issue de la machine VCMC1 [Chailloux 78b], permet la transportabilité du compilateur et l'indépendance par rapport aux représentations internes réelles des objets LISP.

VLM 02

La machine VM#2 est une machine rudimentaire ne manipulant que des pointeurs et possédant une pile.

Ses instructuctions (de longeur variable) utilisent jusqu'à 3 opérandes qui peuvent être :

- des accumulateurs (ou registres)
  notés A1 A2 ... An-1 An

- une pile unique (de contrôle et de donnée)
  notée stack

- des références aux objets VLISP eux-mêmes
  notées ' objet VLISP et NIL

Les instructions sont exécutées séquentiellement sans continuation explicite.

La représentation externe de ces instructions est celle classique du LAP (LISP ASSEMBLY PROGRAM) : chaque instruction est représentée par une liste dont le CAR est l'opérateur et le CDR les opérandes. Les étiquettes sont représentées par des atomes littéraux placés devant les intructions qu'elles référencent.

Voici la description de quelques instruction VM#2. La plupart possèdent 2 types d'opérandes l'un source l'autre destination. L'opérande stack est le seul dont la signification change en fonction de son type : utilisé

comme opérande source, il signifie dépiler, et utilisé comme opérande destination il signifie empiler.

Transfert de pointeurs

(MOVE source destination)	source → destination

Intructions de manipulation de listes

(CAR source destination)	(CAR source) → destination
(CDR source destination)	(CDR source) → destination
(CONS source destination)	(source . destination) → destination
(XCONS source destination)	(destination . source) → destination

Instructions de contrôle
(PC représente le compteur ordinal de la machine VM#2)

(JUMP étiquette)	étiquette → PC
(CALL étiquette)	PC → stack étiquette → PC
(JUMP stack)	stack → PC

Instructions de test de type

(JTNIL source étiquette)	si source = NIL alors étiquette → PC
(JFNIL source étiquette)	si source ≠ NIL alors étiquette → PC
(JTLIST source étiquette)	si LISTP(source) alors étiquette → PC
(JFLIST source étiquette)	si ATOM(source) alors étiquette → PC

Instructions de comparaisons de pointeurs

(JEQ source destination étiqette)
si source = destination alors étiquette → PC
(JNEQ source destination étiquette)
si source ≠ destination alors étiquette →C

3.0 ORGANISATION GENERALE DU COMPILATEUR

On peut découper le compilateur en différentes phases logiques (qui ne reflettent pas les phases réelles qui sont entremélées)

1 ou plusieurs fonctions VLISP

PHASE 1
- expansion des MACRO/MACMP
- recherche du type des fonctions
- recherche du type des variables (libres, liées, SPECIAL ..)
- détermination du lieu de stockage des arguments (registres, pile ou C-VAL)

1 ou plusieurs fonctions VLISP intermédiaires

```
                        - allocation des registres
PHASE 2                 - génération du code brut VM#2

   liste d'instructions
       brutes VM#2

                        - améliorations locales
PHASE 3                 - retournement de code

   liste d'instructions
     VM#2 définitive

                        - traduction des instructions VM#2
PHASE 4                   en instructions réellement executables
                        - améliorations spécifiques

       instructions d'une machine
       réelle de type :

       PDP 10          ou
         SOLAR 16         ou
           VCMC II           ou
             INTEL 8080         ou
               ZILOG 80            ou
                 ....                ..
```

4.0 PHASE 1 : la prélecture

La première phase du compilateur est une phase de prélecture et de transformation de la ou des fonctions d'entrée VLISP en de nouvelles fonctions dans lesquelles :

1 - toutes les fonctions VLISP de type MACRO sont expansées. On aperçoit donc la différence fondamentale entre le traitement de ce type de fonctions effectué par l'interprète qui à chaque appel re-évalue le corps de la MACRO, de celui effectué par le compilateur qui ne compile que le résultat de l'expansion de cette MACRO, cette expansion n'ayant lieu qu'une seule fois durant la première phase de la compilation.

2 - toutes les fonctions spéciales MACMP sont également expansées. Ces fonctions sont utilisées pour redéfinir des fonctions standard sous forme de MACRO internes au compilateur en vue de réduire la taille du compilateur et d'alléger son travail .

exemple d'expansion réalisée au moyen des MACMP

```
(CAAR x)            ☞   (CAR (CAR x))

(COND                   (IF p1
  (p1 e1)                   e1
  (p2)              ☞       (OR p2 (IF p3
  (p3 e2 e3)                           (PROGN e2 e3)
  (T e4 ... eN))                       (PROGN e4 ... eN)))
```

3 - on effectue la détermination du type de chacunes des fonctions rencontrées. A chaque fonction est associée un type de définition qui peut être soit statique (i.e. défini au niveau de la boucle

principale de l'interprète) soit dynamique (i.e. défini temporairement durant l'exécution). Une des limitations du compilateur apparaît ici clairement : le mode de lancement des fonctions statiques est extrememement rapide (e.g. au moyen d'un CALL F-VAL) alors que le lancement des fonctions dynamiques doit parfois faire appel à la fonction interprète EVAL. Il n'est donc pas possible de compiler correctement des fonctions qui ont à la fois une définition statique et dynamique (par exemple des fonctions standard redéfinie dynamiquement d'un autre type).

4 - on effectue la détermination du statut de toutes les variables. Chaque variable peut être d'une part libre ou liée dans la fonction où elle apparait et d'autre part suceptible ou non d'apparaître libre dans une autre fonction (on nomme ces variables SPECIAL ou UNSPECIAL). L'accès a une variable dépend du type de celle-ci : on accède à une variable SPECIAL comme l'interprète i.e. via sa C-VAL ; alors qu'on accède à une variable UNSPECIAL directement par un registre.

exemple soit à compiler la fonction :

```
(DE DELQ (L A)
   (COND
      ((NULL L) NIL)
      ((EQ (CAR L) A) (DELQ (CDR L) A))
      (T (CONS (CAR L) (DELQ (CDR L) A)))))
```

le compilateur produit après la première phase la fonction :

```
(DE DELQ (L A)
   (IF (NULL L)
      NIL
      (IF (EQ (CAR L) A)
         (DELQ (CDR L) A)
         (CONS (CAR L) (DELQ (CDR L) A)))))
```

DELQ est considérée comme une SUBR à 2 arguments

L et A sont considérées comme des variables UNSPECIAL

A l'appel de la fonction DELQ, les registres A1 et A2 doivent contenir les valeurs actuelles des paramètres formels L et A.

5.0 PHASE 2 : génération du code brut

Cette phase engendre le code brut, sous forme d'une liste d'instructions VM#2. Cette génération essaie d'utiliser au mieux les ressources de type registre ou pile, de la machine cible VM#2 en tenant à jour les contenus actuels de chacun des registres et l'état courant de la pile.

La génération proprement dite utilise un certain nombre de primitives récursives dont voici un apperçu :

(ADD L) rajoute l'instruction L à la liste des intructions en cours de formation.

(COMMUT f) teste si la fonction f est commutative. Ce test s'effectue par consultation de la base de données du compilateur.

(CMP x r) engendre les intructions nécessaires au calcul de l'expression x dont la valeur doit se trouver dans le registre r. S'il est possible d'obtenir le résulat dans le registre spécifié, CMP actualise le contenu courant du registre r, sinon CMP ramène NIL.

(CMPX x) engendre les instructions nécessaires au calcul de l'expression x. CMPX doit ramener le numéro du registre qui contient le résultat du calcul.

.....

(SAVE) sauvegarde l'état courant de la liste des instructions, des registres et de la pile.

(RESTORE) restaure le dernier état de la liste des instructions, des registres et de la pile, sauvés par la primitive (SAVE)

exemples de description de compilation :
(les instructions VM#2 sont soulignées)

compiler
(CAR x) dans le registre r

devient
(ADD (CAR (CMPX x) r))

compiler l'appel d'une SUBR à 2 arguments
de la forme (subr2 x1 x2)
(les valeurs des arguments doivent se trouver
respectivement dans les registres A1 et A2)

devient

```
(SAVE)
(CMP x1 A1)
(IF (CMP x2 A2)
   ()
   (RESTORE)
   (ADD (MOVE (CMPX x1) stack))
   (CMP x2 A1)
   (IF (COMMUT subr2)
      (ADD (MOVE stack A2))
      (ADD (MOVE A1 A2) (MOVE stack A1))))
(ADD (CALL subr2))
```

exemple de la génération du code brut de la fonction DELQ résultante de la phase 1

```
                                (DE DELQ (A L)
DELQ  (JFNIL A1 G101)              (IF (NULL L)
      (MOVE NIL A1)                    ()
      (JUMP G102)
G101  (CAR A1 A3)                      (IF (EQ (CAR L) A)
      (JNEQ A3 A2 G103)
      (CDR A1 A1)                          (DELQ (CDR L) A)
      (CALL DELQ)
      (JUMP G102)
G103  (MOVE A3 stack)                      (CONS (CAR L) (DELQ (CDR L) A)))))
      (CDR A1 A1)
      (CALL DELQ)
      (CONS stack A1)
G102  (JUMP stack)
```

6.0 PHASE 3 : améliorations locales

Cette phase permet d'améliorer localement la liste d'intructions engendrée durant la phase 2. Ces améliorations sont syntaxiques et sont naturellement réalisées automatiquement.

Voici les principales améliorations effectuées :

(1) Résolution des chaînes de JUMP

```
        (JUMP etq1)                          (JUMP etq2)
          .....          ☞                     .....
etq1    (JUMP etq2)                  etq1    (JUMP etq2)
```

(2) Inversion des sauts conditionnels (la condition inverse de Jcond est notée J-cond)

```
        (Jcond ... etq1)
        (JUMP etq2)      ☞          (J-cond ... etq2)
etq1                                etq1
```

(3) Elimination des intructions redondantes

```
        (JFNIL op etq)
        (MOVE NIL op)        ☞      (JFNIL op etq)

        (JUMP etq)
etq       ....               ☞    etq   .....
```

(4) Elimination des récursions terminales

```
        (CALL etq)
        (JUMP stack)         ☞      (JUMP etq)
```

(5) Amélioration des boucles par retournement de code

```
        (JUMP x) ou début du code          (JUMP x)
etq1                                  etq3
        ┌─────┐                          ┌─────┐
        │ B 1 │                          │ B 2 │
        └─────┘                          └─────┘
(Jcond ... etq2)     ☞          etq1
        ┌─────┐                          ┌─────┐
        │ B 2 │                          │ B 1 │
        └─────┘                          └─────┘
(JUMP etq1)                         (J-cond   etq3)
                                    (JUMP etq2)
```

Le début du code de la fonction DELQ vue précédemment est :

```
DELQ    (JFNIL A1 G101)          [1]
        (MOVE NIL A1)            [2]
        (JUMP G102)              [3]
G101    (CAR A1 A2)              [4]
        (JNEQ A3 A2 G103)        [5]
        (CDR A1 A1)              [6]
        (CALL DELQ)              [7]
        (JUMP 102)               [8]
G103      .....
```

On peut réaliser différentes améliorations :

- l'instruction [2] est enlevée comme instruction redondante (3) :

```
DELQ  (JFNIL A1 G101)          [1]
      (JUMP G102)              [3]
G101  (CAR A1 A3)              [4]
      (JNEQ A3 A2 G103)        [5]
      (CDR A1 A1)              [6]
      (CALL DELQ)              [7]
      (JUMP G102)              [8]
G103     .....
```

- l'instruction [3] disparait en inversant le test [1] (2) :

```
DELQ  (JTNIL A1 G102)          [1]
      (CAR A1 A3)              [4]
      (JNEQ A3 A2 G103)        [5]
      (CDR A1 A1)              [6]
      (CALL DELQ)              [7]
      (JUMP G102)              [8]
G103    .....
```

- l'étiquette G102 est équivalente à stack [1] et [8] (1) :

```
DELQ  (JTNIL A1 stack)         [1]
      (CAR A1 A3)              [4]
      (JNEQ A3 A2 G103)        [5]
      (CDR A1 A1)              [6]
      (CALL DELQ)              [7]
      (JUMP stack)             [8]
G103     .....
```

- la fausse récursion [7] [8] est éliminée (4) :

```
DELQ  (JTNIL A1 stack)         [1]
      (CAR A1 A3)              [4]
      (JNEQ A3 A2 G103)        [5]
      (CDR A1 A1)              [6]
      (JUMP DELQ)              [8]
G103     .....
```

- et enfin on procède à un retournement du code [6] [8] (5) :

```
G104  (CDR A1 A1)              [6]
DELQ  (JTNIL A1 stack)         [1]
      (CAR A1 A3)              [4]
      (JEQ A3 A2 G104)         [5]
G103     ......
```

7.0 PHASE 4 : génération code machine réelle

Cette dernière phase va transformer la liste d'intructions définitive VM#2 issue de la phase 3 pour pouvoir être effectivement exécutée. Plusieurs stratégies sont possibles en fonction du type de la machine utilisée. La liste d'instructions va pouvoir :

- être exécutée par une machine VM#2. Malheureusement cette machine n'existe pas à l'heure actuelle.
- être exécutée par un simulateur de la machine VM#2. Il existe actuellement un simulateur de la machine VM#2 écrit en VLISP 10.
- être interprétée par une autre machine (comme le SOLAR 16, le PDP 11 ou l'Intel 8080)
- être macro-générée dans le langage machine d'une autre machine existante ,si le taux d'expansion n'est pas trop élévé (comme pour le PDP10). Cette dernière phase de macro-génération peut contenir un autre ensemble d'améliorations locales propres à la machine cible.

exemple de la macro-génération en langage machine PDP 10 de la fonction DELQ

```
G104  (CDR A1 A1)                G104  (HRRZ A1 :MEM A1)
DELQ  (JTNIL A1 stack)           DELQ  (JUMPE A1 :VPOPJ)
      (CAR A1 A3)                      (HLRZ A3 :MEM A1)
      (JEQ A3 A2 G104)                 (CAIN A3 0 A2)
                                       (JRST 0 G104)
      (MOVE A3 stack)                  (PUSH P A3)
      (CDR A1 A1)                      (HRRZ A1 :MEM A1)
      (CALL DELQ)                      (PUSHJ P DELQ)
      (CONS stack A1)                  (POP P A8)
                                       (HRL A1 A8)
                                       (EXCH A1 :MEM FREE)
                                       (EXCH FREE A1)
      (JUMP stack)                     (POPJ P)
```

8.0 CONCLUSION

La réalisation d'un compilateur LISP optimisé se révèle donc possible malgrés les nombreuses contraintes de départ grâce aux énormes possibilités de manipulations d'expressions symboliques du langage VLISP. Le principal défaut de ce type de compilateur (inhérent à tous les compilateurs LISP) est de devoir fonctionner en association avec un interprète et de ne pas pouvoir construire des modules pouvant être exécutés independamment.
Il n'en reste pas moins que ce compilateur étend le champs des possibilités de VLISP pour ses utilisateurs et ses implanteurs, grâce à son rôle d'accélérateur, de compacteur et d'analyseur de fonctions VLISP.

9.0 BIBLIOGRAPHIE

ALLEN J. : The Anatomy of LISP, Mc Graw Hill, 1978.

BAKER Jr, H.G. : Shallow binding in LISP 1.5, M.I.T. Artificial Intelligence Laboratory, Working Paper 138, January 1977.

CHAILLOUX J. : VLISP 8 un système LISP pour micro-processeur à mots de 8 bits, RT 21-78, Département d'Informatique, Université de Paris VIII - Vincennes, Juillet 1978.

CHAILLOUX J. : a VLISP interpreter on the VCMC1 machine, LISP Bulletin #2, July 1978.

CHAILLOUX J. : VLISP 10 . 3 , Manuel de Références, RT 16-78, Université de Paris VIII - Vincennes, Août 1978.

CHAILLOUX J. : VLISP 8 . 2 , Manuel de Références, RT 11-79, Université de Paris VIII - Vincennes, Avril 1979.

GOOSSENS D. : CAN, Département d'Informatique, Université de Paris VIII - Vincennes, 1977.

GREUSSAY P. : Iterative interpretations of tail-recursive LISP procedures, Département d'Informatique, TR 20-76, Université de Paris VIII - Vincennes, Septembre 1976.

GREUSSAY P. : Contribution à la définition interprétative et à l'implémentation des LAMBDA-langages, Thèse, Université de Paris VII, Novembre 1977.

McCARTHY J. et al. : LISP 1.5 Programmer's manual, the M.I.T. Press, Cambridge, Mass., 1962.

MOSES J. : the function of FUNCTION in LISP, Memo 199, M.I.T. Artificial Intelligence Laboratory, June 1970.

WERTZ H. : Un système de compréhension, d'amélioration et de correction de programmes incorrects, Thèse de 3ème cycle, Université Paris VI, Juillet 1978.