Réécriture et Récursion dans une Fermeture ; Etude sur un LISP à liaison superficielle, Application aux objets

J.P. Briot, P. Cointe, E. Saint – James
LITP & IRCAM, 5 Place jussieu, 75005 Paris

Resumé:

Cet article traite de la gestion correcte des Fermetures (i.e. une fonction possédant un environnement propre de variables rémanentes, dites accointances ou champs), dans un interprète gérant l'environnement par la liaison superficielle.
Nous passons en revue les différentes méthodes proposées jusqu'à présent en montrant leurs limitations, et donnons une solution originale et complète compatible avec la réécriture et la récursion même anonyme.
Nous remarquons que les objets sont aussi des fermetures et nous appliquons donc la solution précédente à la résolution du problème de la mise à jour des champs d'un objet ObjVlisp signalé dans [Cointe84a].

1. LES FERMETURES LISP

1.1 LECTURE

Le premier problème à résoudre est celui de la création d'une fermeture, qui intervient lors de l'évaluation d'un argument ou d'un résultat fonctionnel: c'est le problème du Funarg [Perrot79a]. Pour l'instant il s'agit donc seulement de pouvoir LIRE les bonnes valeurs lors de l'utilisation ultérieure de la fermeture.

1.1.1 Principes

1.1.1.1 *Extension*

Une première solution a été proposée par Patrick Greussay [Greussay77a]; nous l'appellerons "extension".

1.1.1.1.1 *Méthode*

Pour une lambda-expression comme:

(lambda (x) (c x))

où l'on souhaite que "c" soit rémanente, on génère:

(lambda (x c) (setqq c <val de c>) (c x))

1.1.1.1.2 *Incompatibilité*

Cette méthode ne permet pas de créer des fermetures n-aires, qui depuis Vlisp, s'écrivent sous la forme: **(lambda x ...)**

C'est particulièrement génant, car les fermetures sont souvent utilisées pour éviter des résultats multiples, qui donc conduisent parfois à des continuations n-aires. De plus elle suppose que l'erreur "pas assez d'arguments" est inexistante, ou du moins redéfinissable.

1.1.1.2 *Insertion*

Une deuxième solution est due à Pierre Cointe [Cointe81a], nous l'appellerons "insertion".

1.1.1.2.1 *Méthode*

Elle consiste à insérer une autre lambda-expression à l'intérieur de la première. Avec le même exemple:

(lambda (x) ((lambda (c) (c x)) '<val de c>))

1.1.1.2.2 *Incompatibilité*
Cette méthode est incompatible avec la primitive de récursion anonyme **self**, nous discuterons plus loin des modifications à lui apporter pour y remédier. Son principe de base est néanmoins celui que nous conserverons.

1.1.2 Réalisation

Le problème à présent est de déterminer les variables rémanentes.

1.1.2.1 *Déclaration*

La solution généralement employée consiste à demander à l'utilisateur de déclarer lui-même ces variables. Un résultat ou argument fonctionnel devra s'écrire alors:
(closure <variables à clôturer> <lambda-exp>)

1.1.2.1.1 *Capture*
Greussay et Cointe définissent la fonction **closure** comme Fexpr. Cette stratégie induit une capture de variables sur les paramètres de la fonction closure elle-même, problème que précisément les fermetures doivent faire disparaître!
Nous en donnons ci-dessous le code avec et sans utilisation de la macro backquote et de la liaison d'arbre. Par la suite nous n'utiliserons quasiment plus ces extensions, malgré leur pouvoir expressif, par souci de compatibilité.

```
(df closure (Lvar lamb)
  (list (car lamb) (cadr lamb)
    (cons (cons 'lambda (cons Lvar (cddr lamb)))
      (mapcar (lambda (x) (kwote (eval x)))
        Lvar))))

(df closure (Lvar (lambda lpar . body))
  `(lambda ,lpar
     ((lambda ,Lvar ,.body)
       ,.(mapcar (lambda (x) (kwote (eval x)))
           Lvar))))
```

1.1.2.1.2 *Macro-insertion*
La seule méthode fiable est de définir **closure** comme une macro:

```
(dm closure (closure Lvar lamb)
  (list 'closure1
    (kwote lamb) (kwote Lvar) (cons 'list Lvar)))

(de closure1 (lamb Lvar Lval)
  (list (car lamb) (cadr lamb)
    (cons (cons 'lambda (cons Lvar (cddr lamb)))
      (mapcar 'kwote Lval))))
```

1.1.2.2 *Automatisation*

En liaison lexicale, l'utilisateur n'a pas à déclarer les variables qu'il souhaite rémanentes, car l'environnement est une donnée comme les autres: l'évaluation d'une lambda-expression peut alors facilement consister en la construction d'un objet composé de la lambda-expression et de l'environnement courant. Cette souplesse d'utilisation peut se réaliser en liaison superficielle, par un balayage du corps de la lambda-expression, afin de déterminer ses variables libres, opération classique en lambda-calcul. On peut donc définir **lambda** comme une fonction faisant la nomenclature de ses variables libres, et se macro-générant en un appel à la fonction **closure** ci-dessus:

```
(dm lambda call
  (let ((libres (varlib call)))
    (list 'closure1
      (kwote call) (kwote libres) (cons 'list libres))))
```

Voilà une version correcte mais non optimisée de **varlib** qui ne prend pas en compte les paramètres des lambdas internes:

```
(de varlib (lamb)
  (varlib1 (cadr lamb) (cddr lamb) nil))

(de varlib1 (liee e libre)
  (ifn (atom e) (varlib1 liee (cdr e) (varlib1 liee (car e) libre))
    (if (or (constantp e) (memq e libre) (memq e liee)) libre
      (cons e libre))))
```

Le coût de ce balayage est très réduit si l'on définit **lambda** comme une macro écrasante (en remplacant dm par **defmacro** ou **dmd**), puisqu'il n'aura alors lieu qu'une fois.

D'autre part cette technique permet de garder des performances optimales de l'évaluateur dans les cas simples (en particulier une absence totale d'allocation pour l'environnement), alors que la liaison lexicale grève ceux-ci pour accélérer le traitement du cas rare du Funarg.

1.2 ANONYMAT

L'insertion d'une deuxième lambda-expression à l'intérieur de la première est incompatible avec la présence dans celle-ci de la fonction **self**, qui permet l'appel récursif d'une fonction anonyme [Greussay77a].

1.2.1 Itération

Un premier reméde a été proposé par Pierre Cointe [Cointe81a].

1.2.1.1 *Méthode*

Si l'on dispose d'une primitive permettant de redescendre dans la pile - la fonction **lastcall** de Vlisp10 [Chailloux78a], par exemple - il suffit de définir **self** comme allant y chercher l'avant-dernière lambda-expression active, la dernière étant celle des variables rémanentes.

1.2.1.2 *Croisement*

Malheureusement, Jean-Louis Durieux [Durieux82a] a démontré que l'ordre des blocs de contrôle dans la pile d'évaluation ne donnait pas nécessairement l'historique exact du calcul en cours en cas de récursivités terminales croisées. Toute implantation de **self** par simple descente dans la pile, comme celle décrite par Patrick Greussay et Jérôme Chailloux [Chailloux80a], est donc fausse.

Deux recours sont possibles:

L'un, proposé par Emmanuel Saint-James et utilisé par Patrick Greussay pour Vlisp-C [Greussay84a], consiste à permuter les F-vals du dernier bloc de contrôle et de celui qui indique la récursivité terminale éventuellement croisée. Cette méthode est fiable, mais demande une écriture à l'intérieur de la pile, qui du coup ne mérite plus de s'appeler ainsi! Nous préférons donc l'éviter.

L'autre, proposé par Jean-Louis Durieux, consiste, pour chaque invocation d'une lambda-expression, à affecter un mot mémoire par la lambda-expression. En particulier, la variable **self** elle-même pourra être ce mot mémoire. C'est cette technique que nous retiendrons. Il faut remarquer que **self** est alors un paramètre implicite de toute lambda-expression, et donc doit être rajouté explicitement dans la liste des variables liées lors du calcul des variables libres:

```
(de varlib (lamb)
  (varlib1 (cons 'self (cadr lamb)) (cddr lamb) nil))
```

Dans un cas comme dans l'autre, l'avant-dernière lambda-expression active n'est pas infailliblement déterminable à l'aide de la pile; nous abandonnerons donc cette méthode. Elle possède de plus un autre défaut, commun avec une autre solution que nous allons étudier à présent.

1.2.2 Redéfinition

1.2.2.1 *Méthode*

Une autre solution proposée par Pierre Cointe [Cointe81a], consiste à ajouter la variable **self** dans la liste des variables rémanentes, pour qu'elle soit liée à la lambda-expression englobante, au moyen d'une liste circulaire.

1.2.2.2 *Inconvénient*

Cette solution est fiable, mais du coup, tout appel ultérieur de **self** appelera cette lambda-expression, même si entretemps une autre est devenue la dernière. En réalité, Cointe (comme Greussay dans Vlisp.C [Greussay84a]) utilise en fait non pas **self** mais une autre variable, **cself** (**SELF** en Vlisp.C), ce qui oblige l'utilisateur à distinguer entre une lambda-expression susceptible de devenir une fermeture et les autres.

Les problèmes rencontrés ne sont donc pas triviaux. Aussi, c'est vers l'élimination de la lambda-expression clôturant les variables rémanentes qu'il faut s'orienter, pour ne pas perturber le comportement des autres aspects du système.

1.2.3 Elimination

Notre problème est de lier dynamiquement les variables rémanentes à leurs valeurs. Or, il n'y a pas que les "Exprs" pour réaliser une telle opération: il y a également les "Fexprs". Celles-ci sont de toute façon plus indiquées que les Exprs dans ce contexte, puisque les valeurs des variables rémanentes ont été évaluées au moment de la création de la fermeture: on a même du générer force appels à **quote** pour empêcher des doubles évaluations. L'utilisation des Fexprs permet donc:

- une restauration de l'environnement très rapide, car sans évaluation,
- une fermeture de taille extrêmement réduite, en fait moins encombrante encore qu'en liaison lexicale, puisque une liste des seules valeurs est construite, et non une liste des couples variables/valeurs. La liste des variables pourra être partagée par une classe de fermetures, on en verra une application immédiate dans le modèle objet.

```
(dm closure (closure Lvar lamb)
  (list 'closure1
    (kwote lamb) (kwote Lvar) (cons 'list Lvar)))

(de closure1 (lamb Lvar Lval)
  (list (car lamb) (cadr lamb)
    (cons (cons 'flambda (cons Lvar (cddr lamb)))
      Lval)))
```

La Fexpr **flambda** pourra être remplacée par la fonction **letdicq**[1] (ou **letvq**) qui réalise une liaison d'arbre dynamique [Cointe84b] permettant d'utiliser en Lisp la structure de "dictionnaire" Smalltalk (cf. infra les objets ObjVlisp).

```
(de closure1 (lamb Lvar Lval)
  (list (car lamb) (cadr lamb)
    (mcons 'letvq Lvar (kwote Lval) (cddr lamb))))
```

Enfin, pour être compatible avec la construction **self**, il suffit de limiter celle-ci à l'appel de la dernière Expr active et non de la dernière lambda-expression active quelque soit son type (Expr ou Fexpr). En pratique, cette limitation n'en est pas une, car une Fexpr récursive est inconcevable: n'évaluant pas ses arguments, un appel récursif entraînerait nécessairement une récursion infinie.

1. letdic signifiant let-dictionary.

1.3 ECRITURE

Jusqu'à présent, nous avions abordé la problématique des fermetures dans un univers totalement applicatif: seuls les arguments et résultats fonctionnels posaient problème. Mais les fermetures sont utiles également dans un contexte impératif. L'exemple le plus simple est celui du générateur de symboles (le **gensym** Lisp): son compteur doit être incrémenté à chaque appel, cette valeur devant être retrouvée à l'appel suivant. Il s'agit donc ici d'accéder en écriture aux variables rémanentes d'une fermeture.

Aucune des solutions proposées jusqu'à maintenant n'est correcte. Nous allons les passer en revue et montrer pourquoi.

1.3.1 Totale

1.3.1.1 *Principe*

Une première méthode, proposée par Pierre Cointe [Cointe81a], consiste à opérer une rafale de modifications physiques sur la totalité de la liste des valeurs des variables rémanentes, l'opération d'affectation (**set**) elle-même restant inchangée.

A partir de ce principe de base, plusieurs versions sont possibles.

1.3.1.2 *Equilibrée*

La première est d'opérer les modifications une fois par appel de la fermeture, c'est à dire juste avant de sortir de celle-ci.

1.3.1.2.1 *Méthode*

Au moment de la création de la fermeture, on enchasse son corps par la fonction **protect** (ou **unwind-protect**) et la rafale d'auto-modifications. Cette série d'auto-modifications s'effectue par une application de **map** sur la fonction **rplaca-cval**. Cette stratégie permet une construction indépendante du nombre de modifications à réaliser et une exécution très efficace en compilant **rplaca-cval**.

```
(de closure1 (lamb L_var L_val)
  (list (car lamb) (cadr lamb)
    (cons (list flambda L_var
            (list 'protect (cons 'progn (cddr lamb))
              (list 'map ''rplaca-cval
                (kwote L_val) (kwote L_var))))
      L_val)))

(de rplaca-cval (L_val L_var)
  (rplaca L_val (cval (car L_var))))
```

1.3.1.2.2 *Inconvénients*

Cette méthode a comme premier défaut d'empêcher tout optimisation des récursivités terminales, "l'evlis-tail-récursion" n'étant plus possible du fait de la présence systématique de la rafale de modifications.

Mais ce qui est beaucoup plus grave, c'est la neutralisation de la mise à jour des variables rémanentes en cas d'appel récursif. Supposons par exemple que l'on veuille modifier la définition de Fibonnacci pour en faire une fermeture possédant comme variable rémanente la liste des valeurs déjà calculées. Au moment de l'appel récursif sur "n-1" il y aura bien calcul et mémorisation de la valeur pour "n-2", mais cette mémorisation ne se fait pas sentir sur la liaison de la variable rémanente de l'appel précédent, du fait de la restauration (qui sera suivie d'une (mauvaise) mémorisation!) de l'ancienne valeur de la variable au sortir de l'appel récursif!

1.3.1.3 *Abandon*

Le problème vient de la restauration des anciennes valeurs des variables rémanentes en cas d'appel récursif; il convient de l'éviter.

1.3.1.3.1 *Méthode*

Pour ce faire, il suffit qu'en cas d'appel récursif, la forme appelée ne soit pas la fermeture elle-même, mais la lambda-expression originelle, celle dans laquelle ne figure pas la Fexpr liant (et déliant!) les variables rémanentes.

1.3.1.3.2 *Inconvénient*
Cette méthode convient au problème de Fibonnacci, mais n'est pas suffisante en cas d'appels croisés avec d'autres lambda-expressions possédant des variables locales ou rémanentes de même nom que les variables rémanentes de notre fermeture: ces dernières n'auront pas les valeurs correctes au moment de l'appel récursif.

1.3.1.4 *Suraffectation*

1.3.1.4.1 *Méthode*
Une autre méthode serait de réaffecter toutes les variables rémanentes à chaque retour d'appel récursif (simple ou non) d'une fermeture. Ceci ne peut pas se faire en Lisp sans un changement de l'évaluateur, mais peut se faire facilement en ObjVlisp (Smalltalk) en modifiant la fonction **send** assurant l'activation d'un objet, nous le verrons plus loin.

1.3.1.4.2 *Inconvénient*
Cette méthode a malheureusement un ennui: si la fermeture utilise des variables locales de même nom que des variables rémanentes, l'effet de l'affectation sera masqué.

Notre problème est donc double:

- réaliser les affectations au bon moment,
- réaliser les affectations sur les bonnes variables.

Nous allons exposer à présent notre méthode:

1.3.2 **Ponctuelle**

Les exemples précédents montrent la nature du problème: le décalage temporel entre l'environnement local temporaire et la liste des valeurs de la fermeture, retard de la fermeture lors d'appels récursifs ou retard de l'environnement local lors de leurs retours. Cela nous rappelle un problème bien connu (le Funarg) que les fermetures voulaient justement résoudre!

Il nous faut donc éviter la construction récursive d'environnements temporaires lors d'appels récursifs, il n'est en effet plus possible de les faire coïncider.
Toute affectation sur les variables rémanentes doit s'accompagner immédiatement de la mise à jour de la liste des valeurs de celles-ci. La mise à jour va ainsi se faire ponctuellement, au coup par coup, et non pas en totalité comme jusqu'à présent.

1.3.2.1 *Redéfinition*

Pour ce faire, il suffit de redéfinir toutes les fonctions d'affectation. Cette redéfinition va permettre de simuler totalement la technique d'affectation de la liaison lexicale.

1.3.2.2 *Distinction*

Le seul problème est de savoir distinguer les variables rémanentes des autres: sans quoi, autant recourir définitivement à la liaison lexicale! Plus exactement, si une variable n'apparaissait pas dans la liste des variables à clôturer, alors elle n'est évidemment pas rémanente; en revanche, si elle y apparaissait, elle ne l'est pas non plus nécessairement, car elle peut avoir été rendue locale par une lambda-expression interne: cette distinction est donc dynamique, non pas statique.

Les variables locales ont une importante propriété: elles ne sont jamais indéfinies. La solution va donc consister à rendre indéfinies les variables rémanentes à l'entrée de la fermeture, en mémorisant leurs valeurs de clôture ailleurs que dans leurs "Cvals". Ce faisant, en voulant résoudre le problème de l'écriture, on est conduit à reposer le problème de la lecture.

1.3.2.3 *Lecture*

L'évaluation d'une variable rémanente va maintenant provoquer l'appel de l'erreur "variable indéfinie". En redéfinissant dynamiquement cette erreur, comme consultant la liste des variables clôturées, on peut donner la valeur de la variable. Toutefois,

si la variable en question n'apparaît pas dans la liste des variables clôturées, il peut s'agir, soit d'une variable effectivement indéfinie, soit d'une variable clôturée d'une fermeture appelante. Dans les deux cas, il suffit d'invoquer la définition précédente de l'erreur "variable indéfinie".

Cette redéfinition dynamique va se faire à l'entrée de la fermeture.

1.3.2.4 *Ecriture*

De la même manière, on redéfinit la fonction primitive d'affectation (**set**, les autres s'en déduisant) à l'entrée de la fermeture, afin qu'elle teste si la variable est indéfinie, et

- sinon: affecte la Cval,
- si oui: teste l'appartenance de la variable à la liste de variables clôturées, d'où deux nouveaux sous-cas:
  - si oui modifie physiquement la liste des valeurs,
  - sinon appelle la definition précédente de **set**, pour remonter dans les fermetures englobantes, eventuellement jusqu'à la définition usuelle de **set**.

2. LES OBJETS OBJVLISP

Le modèle Objvlisp définit une machine abstraite définie méta-circulairement [Cointe85a]. L'un des principes de base est d'installer cette machine sur différents Lisp en conservant la liaison dynamique, cette démarche pose naturellement le problème de la mise à jour des champs d'un objet dans le cas de transmissions récursives, déjà signalé dans [Cointe84a].

2.1 DEFINITION FONCTIONNELLE D'UN OBJET

Nous reprenons le modèle des objets ObjVlisp tel qu'il est présenté dans [Cointe84a]:

1. les objets ObjVlisp sont implémentés comme des atomes Lisp ayant pour valeur fonctionnelle une fermeture auto-référencée; le champ **oself** dénote le nom de l'objet, le champ **mclass** celui de sa classe,
2. l'activation d'un objet s'opère dans le contexte de ses champs dont les valeurs sont installées DYNAMIQUEMENT par la fonction letdic; letdic admet pour arguments la liste des champs détenue par la classe "(fields mclass)" et la liste de leurs valeurs détenue par l'objet "(f-dico oself)",

```
(synonym 'send 'funcall)

(de UN_OBJET (-selector- . -args-)
  (if (eq oself 'UN_OBJET) (apply (lookup -selector- mclass) -args-)
    (let ((oself 'UN_OBJET) (mclass 'UNE_CLASSE))
      (letdic (fields mclass) (f-dico oself)
        (protect (apply (lookup -selector- mclass) -args-)
          (rewrite oself mclass))))))

(defmacro oself (-selector- . -args-)
  '(funcall (lookup ,-selector- mclass) ,.-args-))

(setq oself 'oself mclass 'mclass)
```

3. la transmission de message ObjVlisp se ramène à l'évaluation d'une forme LISP i.e. à un simple appel de fonction (send = funcall). Le couple <eval , apply> est préservé, l'implémentation du modèle ObjVlisp ne nécessite donc pas la création d'un évaluateur particulier,

4. la mise à jour de la fermeture s'effectue automatiquement en fin de transmission par la fonction *rewrite*. L'écriture d'un champ utilise donc les fonctions d'affectation Lisp standards (*set* etc...),

5. le "pseudo-objet" **oself** est AUSSI un objet fonctionnel[2] , MAIS qui lance la recherche de la méthode sans réinstaller l'environnement local, toute transmission s'effectuant dans le contexte du receveur,

6. pour détecter les cas de récursivité non anonyme, le test de récursivité "(eq oself 'UN_OBJET)" est effectué pour CHAQUE objet,

7. au toplevel ObjVlisp, les pseudos-champs **oself** et **mclass** sont définis comme des constantes (ils s'admettent comme propre valeur).

2.2 REECRITURE DES CHAMPS D'UN OBJET

Comme dans le cas des fermetures, la technique visant à mettre à jour la valeur des champs en fin de transmission ne permet pas d'assurer une concordance permanente entre les Cvals des champs vus comme des variables globales Lisp et les valeurs de ces champs stockés dans la liste "(f-dico oself)".

2.2.1 Mises à jour automatiques

La première solution consiste à redéfinir la fonction **send** (initialement synonyme de **funcall**), pour assurer cette concordance:

```
(defmacro send message_passing
  '(progn
     (rewrite oself mclass)
     (protect (funcall ,.message_passing)
       (etirwer oself mclass))))

(de rewrite (inst mclass)
  (mapc 'rplaca-cval (f-dico inst) (fields mclass)))

(de etirwer (inst mclass)
  (deset (fields mclass) (f-dico inst)))

(f-dico 'oself ())
(f-dico 'mclass ())
```

Deux traitements d'entrée et de sortie interviendront donc avant et après l'activation de l'objet, et avant l'évaluation des arguments car ils doivent s'appliquer à l'objet appelant:

rewrite met à jour les valeurs des champs (donc la fermeture) à partir de l'environnement local déterminé par les variables Lisp de même nom,

etirwer symétriquement, réécrit l'environnement de l'objet dans l'environnement local.

On attribue aux pseudo-objets *oself* et *mclass* des fermetures (nulles) pour traiter l'activation du premier objet au top-level. Les deux réécritures ont alors lieu sur la liste de variables de *mclass* et la liste de valeurs de *oself*, toutes deux nulles.

Comme nous l'avons déjà signalé en 1.3.1.4, cette première solution n'est que partielle, elle induira des réécritures inexactes en cas de variables locales de même nom que les champs.

---

2. Voilà une autre définition de **oself** qui fige définitivement la recherche de la méthode:
```
(defmacro oself (-selector- . -args-)
  (cons (lookup (eval -selector-) mclass)
    -args-))
```

2.2.2 Simulation du lexical

Cette solution est l'adaptation ObjVlisp de la technique décrite en 1.3.2, elle repose sur la redéfinition dynamique des fonctions de consultation et d'écriture des champs. Nous revenons donc à un modèle d'accès direct dans la fermeture, mais cette fois traité implicitement[3] par le biais de nos redéfinitions.

Notre technique est de lier tous les champs à la valeur "indéfinie", et de redéfinir dynamiquement la fonction de traitement d'erreur associée - par exemple en Le_Lisp [Chailloux84a] - la clause **errudv** de la fonction **syserror**, de manière à détourner cette erreur sur une consultation dans la fermeture. De même la fonction d'affectation primitive **set** (on suppose les autres fonctions d'affectation définies à partir d'elle) sera redéfinie dynamiquement.
Voici donc la nouvelle valeur fonctionnelle d'un objet:

```
(de UN_OBJET (-selector- . -args-)
  (let ((oself 'UN_OBJET) (mclass 'UNE_CLASSE))
    (let ( (syserror
             (closure '(syserror oself mclass)
               (lambda (fun msg expr))
                 (ifn (eq msg 'errudv) (initialsyserror fun msg expr)
                   (let ((l (get_closure expr (fields mclass) (f-dico oself))))
                     (ifn l (syserror fun msg expr)
                       (car l)))))))
           (set (var val)
             (closure '(set oself mclass)
               (lambda (var val)
                 (if (boundp var) (initialset var val)
                   (let ((l (get_closure val (fields mclass) (f-dico oself))))
                     (ifn l (set var val)
                       (rplaca l val)
                       val))))))  )
      (letdic (fields mclass) (undef_fields mclass)
        (apply (lookup -selector- mclass) -args-)))))
```

La fonction **undef_fields** construit la liste des valeurs indéfinies (*_undef_* en Le_Lisp) associée à la liste des variables, qui sera liée au lieu de la liste des valeurs détenue par l'objet. Elle pourra être avantageusement remplacée par une nouvelle propriété des classes, étant constante et commune à toutes ses instances.
La fonction **get_closure** ramène la sous-liste des valeurs rémanentes associée à la variable recherchée (et () sinon), pour permettre sa lecture ou son écriture.

```
(de undef_fields (mclass)
  (makelist (length (fields mclass)) '_undef_))

(de get_closure (var L_var L_val)
  (cond
    ((null L_var)
     ())
    ((eq var (car L_var))
     L_val)
    (t
     (get_closure var (cdr L_var) (cdr L_val)))))
```

3. à la différence d'un accès explicite aux champs par des primitives spécialisées, par exemple en LOOPS [Bobrow83a].

2.2.2.1 *Redéfinition dynamique de fonctions*

Cette technique de redéfinition dynamique de fonctions est une des caractéristiques fondamentales du projet AGLAE, et notamment des langages Meta-Vlisp [Saint-James86a, Saint-James86b] et Meta-Smalltalk [Briot84a, Jeanjean86a]. Dans le modèle Meta-Vlisp, les valeurs fonctionnelles sont en Cval, un simple **let** suffit alors.
Dans notre implémentation en Le_Lisp présentée ici, nous évitons l'emploi de la primitive **flet** qui ne nous permet pas aisément de redéfinir dynamiquement une fonction par une construction dynamique (en l'occurrence la construction d'une fermeture). Nous simulons donc ces valeurs fonctionnelles en Cval par la redéfinition de **syserror** et **set** vers une indirection explicite à leurs Cvals respectives. Ainsi:

```
(synonym 'initialsyserror 'syserror)        ; sauvegarde de la définition initiale

(setq syserror                              ; la définition au top-level
  (lambda (fun msg expr) (initialsyserror fun msg expr)))

(de syserror (fun msg expr)                 ; indirection explicite: Fval en Cval
  (funcall syserror fun msg expr))

                                            ; et de même pour la fonction set
(synonym 'initialset 'set)

(setq set
  (lambda (var val) (initialset var val)))

(de set (var val)
  (funcall set var val))
```

2.2.2.2 *Définition récursive d'une fermeture*

Nous constatons que la définition d'un objet (et de la gestion de sa fermeture) est récursive, puisqu'elle utilise pour la redéfinition de la fonction **syserror** (de même pour **set**) une construction d'une fermeture. Cette fermeture mémorise la définition précédente, l'objet (**oself**) et sa classe (**mclass**), pour pouvoir retrouver les environnements clôturés lors d'une "remontée" des redéfinitions successives.
Lors d'une utilisation de cette technique pour implémenter des fermetures en Lisp, le risque de régression infinie est résolu, puisque ces deux fermetures sont en lecture seule, dont la solution (cette fois non récursive) est donnée en 1.2.3.

Il est enfin possible de minimiser le coût de construction de ces fermetures, dans le modèle objet si l'on postule de limiter l'environnement d'un objet à ses seules variables rémanentes. La redéfinition récursive et mémorisante est alors remplacée par une simple redéfinition à partir de la définition initiale (*initialsyserror* et *initialset*).

3. CONCLUSION

Nous espérons avoir tout d'abord présenté le problème de la réécriture dans une fermeture en liaison dynamique, puis montré qu'il est possible de faire coexister dans un système, l'efficacité de la liaison superficielle avec l'exactitude de la liaison lexicale sans pénaliser pour autant le développement d'un système objet "dynamique".

4. REFERENCES

[Bobrow83a] D. Bobrow and M. Stefik, *The LOOPS Manual*. December 83.

[Briot84a] J.P. Briot and E. Saint-James, "Intégration de SMALLTALK dans le projet AGLAE.," *Globule*, (41)(Novembre 84).

[Chailloux78a] J. Chailloux, "VLISP 10.3, Manuel de Référence," RT-16-78, Université Paris 8 - Vincennes (Avril 1978).

[Chailloux80a] J. Chailloux, *Le modèle VLISP : Description, Implémentation et Evaluation (thèse de 3ième cycle)*, Universités Paris 6-7, LITP 80-20 (Avril 1980).

[Chailloux84a] J. Chailloux, *Le_Lisp de l'INRIA. Le Manuel de référence*, INRIA (Mai 1984).

[Cointe81a] P. Cointe, "Fermetures dans les λ–Interprètes : Application aux Langages LISP, PLASMA et SMALLTALK (thèse de 3ième cycle)," LITP 82-11, Université Paris VI, Paris (Décembre 1981).

[Cointe84b] P. Cointe, "Une extension de Vlisp par les Objets," *Science of Computer Programming 4*, (161) pp. 291-322 North Holland, (1984).

[Cointe85a] P. Cointe, "Le modèle OBJVLISP une plate-forme pour l'expérimentation des formalismes Objets," pp. 737-754 dans *5ème congrès AFCET-RFIA* , Vol. 2, INRIA AFCET, Grenoble (27-29 Novembre 1985).

[Cointe84a] P. Cointe, "Implémentation et interprétation des langages objets, application aux langages Formes, ObjVlisp et Smalltalk (thèse d'Etat).," LITP 85.55, Université Paris-VI, IRCAM (17 Décembre 84).

[Durieux82a] J.L. Durieux, "Sémantique des Liaisons nom-valeur," dans *Comptes Rendus des Journées GRECO-GROPLAN*, Bergerac (Mars 1982).

[Greussay77a] P. Greussay, *Contribution à la définition interprétative et à l'implémentation des λ-langages (thèse d'Etat)*, Universités Paris 6-7, LITP 78-2 (Novembre 1977).

[Greussay84a] P. Greussay, "Commandes de Vlisp-C," Département Informatique, Université Paris 8 - Vincennes (1984).

[Jeanjean86a] P. Jeanjean, "Analyse syntaxique de Smalltalk par redéfinition dynamique de fonctions virtuelles.," *Globule*, (48)(Janvier 86).

[Perrot79a] J. F. Perrot, "LISP et λ–calcul," pp. 277-301 dans *λ–Calcul et Sémantique formelle des langages de programmation*, ed. B. Robinet, LITP-ENSTA, Paris (1979).

[Saint-James86a] E. Saint-James, "Interprétation méta-récursive du scribe LISP," LITP *Rapport à paraître*, Paris (1986).

[Saint-James86b] E. Saint-James, "Une interprétation méta-récursive du lecteur LISP," LITP *Rapport à paraître*, Paris (1986).