Réflexivité dans les langages fonctionnels

Notes de cours

Claude MICHEL

Mars 1997

Table des matières

Réflexivité dans les langages fonctionnels

1 Introduction

Ce cours qui s'inscrit dans le module sur la réflexivité vient compléter les notions présentées lors de la présentation des langages à objets réflexifs. Ces derniers exploitent la notion d'objet pour y introduire le concept de méta, soit par un prolongement de l'instanciation — les méta-classes — soit par des interprètes des mécanismes d'une dénotation — les méta-objets. Chaque méta peut alors être vu comme une implémentation de sa dénotation et ne reflète que les mécanismes d'un objet ou d'un sous-ensemble d'objets.

La réflexivité dans les langages fonctionnels se différencie non seulement par le contexte linguistique, mais aussi, et surtout, par le modèle de réflexivité qu'elle met en jeu. Bien que postérieure à l'introduction des méta-classes dans Smalltalk, c'est avec les travaux de B. C. Smith [Smith 83] sur une variante de Lisp que le concept de réflexivité est vraiment né. Ses travaux, dans lesquels il définit l'architecture des systèmes réflexifs, sont reconnus comme étant le point de départ des recherches sur la réflexivité.

Parce qu'ils sont historiquement les premiers à avoir bénéficié du concept de réflexivité et parce que, par leurs qualités intrinsèques, ils laissent toute liberté au programmeur de se concentrer sur les mécanismes linguistiques essentiels, les langages fonctionnels apparaissent comme la plate-forme idéale pour l'étude de la réflexivité.

Dans ce cours, nous allons exposer les principaux éléments propres à la réflexivité dans les langages fonctionnels avant de décrire l'implémentation d'un interprète réflexif d'un sous-ensemble de Scheme.

2 Les langages fonctionnels

Les langages fonctionnels sont ceux dont l'application de fonctions constitue le mécanisme central. Parmi ces langages, nous nous intéresserons ici plus particulièrement à Lisp et à l'un de ses dérivés, Scheme.

Ces deux langages se caractérisent par des implémentations à base d'interprètes. La compilation qui a été introduite par la suite ne remet par en cause le modèle interprétatif sous-jacent. Ils restent essentiellement des langages dynamiques.

Le fonctionnement de ces langages s'organise autour du couple (*eval*, *apply*). Alors que le premier des éléments de ce couple s'attache à l'évaluation des expressions du langage, le second ne s'occupe que de l'application des fonctions. L'interprétation se traduit par des appels mutuels de ces deux fonctions, appels initiés par la boucle principale de l'interprète, la boucle *read-eval-print*.

3 Aspects réflexifs de Lisp et Scheme

Lisp et Scheme sont dotés de caractéristiques qui leur confèrent certaines des propriétés des langages réflexifs. Cela ne signifie pas qu'ils sont réflexifs, mais plus précisément, qu'ils sont dotés de quelques capacités réflexives.

L'un de ces aspects concerne la représentation des lambdas de Lisp sous forme de liste. L'utilisation de cette représentation des lambdas dans les premiers Lisps donne au programmeur la possibilité de les manipuler au même titre que toute liste. Cette propriété est a ranger dans la réflexivité structurelle i.e. la capacité qu'offre un langage de manipuler les objets du langage dans une représentation qui appartient à ce même langage.

Un second aspect est constitué par la fonction *call/cc* de Scheme. Elle permet d'accéder à la continuation du calcul et constitue un outil puissant de construction de structure de contrôle. Cette dernière propriété appartient à la réflexivité opératoire i.e. la capacité offerte à un langage de modifier son calcul.

Notons que chaque implémentation de l'un ou l'autre de ces langages va encore plus loin dans l'introduction de capacités réflexives qui vont alors de fonctions qui autorisent la manipulation de l'environnement à diverses fonctions de manipulation des objets du langage. Notons aussi que la plupart de ces extensions concernent la réflexivité structurelle.

4 Modèle de réflexivité

La réflexivité dans les langages fonctionnels est régie par le modèle de tour virtuellement infinie d'interprètes. Il s'agit d'un empilement d'interprètes qui s'interprètent les uns les autres, le niveau zéro de la tour étant constitué par les programmes de l'utilisateur.

Une telle tour est virtuellement infinie. Pour chaque niveau de la tour, constitué d'un interprète, il existe un niveau supérieur, constitué par l'interprète de cet interprète.

Un tel empilement d'interprète n'a de sens que dans la mesure où l'utilisateur du système peut modifier chaque niveau de la tour. Les interprètes réflexifs sont donc dotés d'outils linguistiques qui en autorisent la modification.

L'examen des différents interprètes réflexifs permet d'identifier deux constructions principales :

1. les lambdas réflexives et
2. le *exec-at-meta*.

La première de ces constructions utilise l'application des fonctions pour passer d'un niveau d'interprétation à un autre, tandis que la seconde se sert de l'évaluation. À l'aide de l'une de ces deux constructions le programmeur peut modifier l'interprétation d'un programme. Notons, qu'à priori, les modifications ainsi effectuées sont dynamiques et n'altèrent pas l'interprète à proprement parler. Seul le cours de l'interprétation se trouve modifié pour la durée d'exécution de l'une de ces formes linguistiques.

Ce modèle de réflexivité s'oppose donc à celui prôné par les langages à objets sur deux points. Les tours que l'on peut observer ici sont constituées par un empilement d'interprètes et non de multiples tours constituées d'un empilement de méta-objets. Par ailleurs la programmation réflexive ne s'appuie pas sur la substitution à un interprète local des mécanismes de l'objet d'un interprète défini par l'utilisateur, le méta-objet, mais repose sur l'exécution de code utilisateur au niveau de l'interprète. Cette dernière substitution est dynamique et éphémère contrairement à la première.

5 Les lambdas réflexives

Les lambdas réflexives constituent l'outil le plus courant pour la programmation réflexive. La plupart des interprètes en font usage.

Une lambda réflexive est une lambda dont l'application intervient non à son niveau d'appel mais au niveau de son interprète. Le corps d'un telle lambda s'exécute dans un environnement constitué de la liste de ses paramètres d'appel non-évalués, de l'environnement lexical courant ainsi que de la continuation du calcul. Ce triplet correspond à ce qu'il convient d'appeler les registres de l'interprète dont il est directement extrait.

En tant que lambda, les lambdas réflexives s'apparentent à des lambdas. Leur structure ne se différencie que par le type. Elles sont, au même titre que les lambdas, applicables. Enfin, lorsque ces mêmes lambdas sont traités en objets de première classe, les lambdas réflexives sont elles aussi des objets de première classe.

Par contre, les lambdas réflexives s'exécutent un niveau au dessus de leur niveau d'appel, avec des paramètres d'appel non évalués et deux paramètres supplémentaires (l'environnement lexical courant et la continuation du calcul).

La construction des lambda réflexive s'obtient soit par typage, soit par transformation d'une lambda classique.

Exemple de construction par typage (3-Lisp [Smith 83]):

```
(define boundp
```

```
    (lambda reflect ((v) env cont)
                (cont (bound-in-env v env))))
```

Exemple de construction par transformation (I_r[Jefferson 92]):

```
(set! boundp
     (compound-to-reifier
        (lambda (e r k)
           (k (bound-in-env (car e) r)))))
```

Les lambdas réflexives sont donc :

- un mécanisme d'extension du langage (qui s'apparente à l'introduction de formes spéciales),
- un moyen de se substituer localement à l'interprète courant,
- une forme de réflexivité locale (i.e. une modification locale des mécanismes de l'interprète).

Notons que les lambdas réflexives ne sont pas des méta-lambdas : une lambda réflexive ne réifie pas les mécanismes liés à l'application d'une lambda particulière à laquelle elle serait liée. En cela elles se différencient notablement des méta-objets de certains langages à objets.

6 Modèles de calcul

La réflexivité nécessite un modèle de calcul adéquat propre à en faciliter la manipulation.

Parmis les modèles d'interprétation utilisés par les différents interprètes réflexifs, nous en distinguerons deux :

- l'évaluation d'expressions qui correspond au modèle le pus couramment utilisé.
- les transitions d'états utilisés par Bawden [Bawden 88] pour démontrer l'indépendance de la réflexivité du modèle d'interprétation choisi.

Afin d'offrir un contrôle du calcul, les interprètes réflexifs utilisent le concept de continuations. Il s'agit du même type de continuations que celles que le *call/cc* Scheme permet de manipuler. Pour cela les interprètes réflexifs sont écrit non en style direct (le style habituel) mais en continuation explicite. Chaque fonction de l'interprète accepte un paramètre supplémentaire qui représente la continuation du calcul.

L'interprète peut être vu comme une machine dotée de trois registres:

- l'expression à évaluer représentée sous la forme d'une liste de symboles.
- l'environnement lexical courant représenté soit par une A-liste soit par des fermetures lexicales.

- la continuation du calcul représentée par une fermeture lexicale d'un paramètre, le résultat du calcul précédent.

C'est à travers la manipulation de ces trois registres que les calculs réflexifs sont exprimés.

7 Construction d'un interprète réflexif

Afin de vous donner une meilleure compréhension de ce qu'est un interprète réflexif, nous allons maintenant décrire la construction d'un interprète réflexif d'un sous-ensemble de Scheme et dont la réflexivité est basée sur les lambda réflexives.

Cette construction va se dérouler progressivement en quatre étapes:

- La définition d'un interprète méta-circulaire.
- La construction d'une tour finie d'interprètes.
- L'introduction de la réflexivité dans notre interprète.
- Le passage à une tour infinie.

Le langage interprété L est :

exp ::= (*exp*+) |	; application
(**lambda** (*ident**) *exp*+) |	; abstraction
(**if** *exp exp exp*) |	; conditionnelle
(**set!** *ident exp*) |	; affectation
(**quote** exp) |	; quotation
ident |	; identificateur
constant	; constante

Notons que L est complètement inclus dans Scheme. Un interprète de L écrit en L, un interprète méta-circulaire de L, peut donc directement être interprété par un interprète Scheme sans qu'il soit nécessaire de faire intervenir un troisième interprète i.e. un interprète de L écrit en Scheme. Cet artifice ne doit pourtant pas nous faire oublier que dans le cas général, l'écriture de ce troisième interprète est indispensable.

7.1 Construction d'un interprète méta-circulaire

Un interprète méta-circulaire est un interprète de L écrit en L. Nous ne décrirons pas complètement ici un tel interprète. Vous pouvez vous référez à votre cours de Lisp ou à l'article de Jefferson [Jefferson 92] pour obtenir la définition complète d'un tel interprète.

L'écriture d'un interprète méta-circulaire permet d'expliciter les mécanismes de l'interprète de L dans L. C'est à travers les définitions de l'interprète méta-circulaire que l'utilisateur pourra agir sur ces mécanismes.

Les deux fonctions essentielles de l'interprète sont l'évaluation et l'application des fonctions. La fonction *evaluate* réalise l'évaluation des expressions de L. Sa définition est :

```
(set! evaluate
      (lambda (e r k)
        ((if (constant? e)
             evaluate-constant
             (if (variable? e)
                 evaluate-variable
                 (if (if? e)
                     evaluate-if
                     (if (assignment? e)
                         evaluate-assignment
                         (if (abstraction? e)
                             evaluate-abstraction
                             evaluate-combination))))) e r k)))
```

La fontion *evaluate-combination* réalise l'application du premier des éléments contenus dans le paramètre e à ses suivants. Sa définition est :

```
(set! evaluate-combination
      (lambda (e r k)
        (evaluate (operator-part e) e
          (lambda (proc)
            (evaluate-operands (operands-part e) r
              (lambda (args)
                (apply-proc proc args k)))))))
```

Ces deux fonctions sont écrites de manière relativement classique sinon qu'elles utilisent un troisième paramètre, la continuation (k). La fonction *evaluate-combination* met en évidence l'usage de la continuation dans l'expression du calcul. L'enchaînement des calculs y est souligné par l'utilisation de continuations. Ainsi cette fonction commence par évaluer le premier élément de l'expression qui lui est passé en paramètre. La valeur de cet élément sera passé à la continuation qui déclenchera l'évaluation des autres éléments de l'expression.

7.2 Construction d'une tour finie d'interprètes

La construction d'une tour finie d'interprètes constitue la première étape vers la réalisation de notre interprète réflexif.

La construction de cette tour finie s'effectue simplement «à la main». La première étape consiste à charger le fichier contenant les définitions de l'interprète depuis l'interprète Scheme choisi :

```
(load "interp-meta-circulaire.scheme")
```

L'interprète peut alors être lancé sur un niveau :

```
(initialize-global-env)
(openloop 0)
```

Nous nous trouvons alors dans la boucle *read-eval-print* de l'interprète constitué d'une tour contenant un unique niveau d'interprétation. Si les opérations précédentes sont réitérés :

```
(loadfile "interp-meta-circulaire.scheme")
(initialize-global-env)
(openloop 1)
```

nous nous retrouvons alors dans la boucle *read-eval-print* d'un second interprète. La tour d'interprètes est alors constituée de deux interprètes. En répétant l'opération, l'utilisateur peut introduire autant de niveau qu'il le désire. Ces manipulations conduisent à l'obtention d'une tour finie contenant un nombre déterminé de niveaux.

7.3 Introduction de la réflexivité

Une tour finie d'interprète, tant qu'elle n'est pas dotée des lambdas réflexives, n'a d'autre effet que de ralentir la machine ! Pour qu'une telle tour tende vers quelque chose d'utile, il faut doter notre interprète d'un outil qui permet à l'utilisateur d'effectuer des calculs sur l'interprétation de ses programmes. Pour cela, nous utiliserons les lambdas réflexives. C'est à travers cet outil linguistique que la réflexivité va prendre place.

La première des choses à introduire est un constructeur de lambda réflexive. Nos lambdas réflexives vont être construites par transformation de lambdas classiques. Ces dernières sont représentées par des listes dont le premier élément donne le type de l'objet :

```
(compound <paramètres> <corps> <environnement>)
```

Une lambda réflexive est obtenue par transformation du type de l'objet de *compound* en *reifier*. Le symbole *reifier* servira donc de marqueur pour les listes qui représentent des lambdas réflexives. La fonction qui transforme une lambda classique en lambda réflexive est :

```
(set! compound-to-reifier
      (lambda (compound)
         (cons 'reifier (cdr compound))))
```

Dans un deuxième temps, *evaluate-combination* doit être modifiée afin de prendre en compte les lambdas réflexives en les distinguant des lambdas classiques. L'application d'une lambda réflexive diffère en effet de celle d'une lambda classique par le changement de niveau qu'elle opère avant d'évaluer le corps de la lambda. La fonction *evaluate-combination* est modifiée de la manière suivante:

```
(set! evaluate-combination
      (lambda (e r k)
         (evaluate (operator-part e) r
            (lambda (proc)
               (if (reifier? proc) ; Application des lambdas réflexives
                  ((reifier-to-compound proc) (cdr e) r k)
                  (evaluate-operands (operands-part e) r
                     (lambda (args)
                        (apply-proc proc args k))))))))
```

Après avoir évalué le premier élément de *e*, la fonction *evaluate-combination* regarde si sa valeur correspond à une lambda réflexive ou non. Dans ce dernier cas le traitement est identique à la première version. Les lambdas réflexives sont traitées dans la première branche du **if**. La ligne de code correspondante détient tous les éléments qui rendent l'interprète réflexif : elle appelle la lambda réflexive de la même manière qu'elle appelle une fonction de l'interprète. Ainsi, la lambda réflexive n'est pas évaluée par l'interprète courant mais est exécutée au sein de ce même interprète. Avec cette simple ligne, le code de l'utilisateur est remonté d'un niveau dans la tour d'interprètes.

Les paramètres passés à la lambda réflexive correspondent aux registres de l'interprète. Le premier de ces paramètres correspond à l'expression que l'interprète doit maintenant évaluer. Le *car* de e, la lambda réflexive, ayant déjà été évalué, c'est donc le *cdr* de e qui est passé à la lambda réflexive. Le deuxième paramètre correspond à l'environnement lexical courant qui est mis à la disposition de la lambda réflexive sous la forme d'une A-liste. Le dernier paramètre est la continuation du calcul i.e. l'endroit où le calcul doit reprendre après l'application de la lambda réflexive. Cette continuation est matérialisée par une fonction d'un unique paramètre, le résultat de l'application de la lambda réflexive.

Notons qu'avant d'appeler la lambda réflexive, cette dernière est à nouveau transformée en lambda classique. Cette opération est nécessaire pour éviter que la lambda réflexive ne soit remontée indéfiniment de niveau en niveau. En effet, si la lambda réflexive est conservée telle quelle, l'interprète I_{n+1} de l'interprète courant I_n va, lorsqu'il devra interpréter l'appel de la lambda réflexive, non pas opter pour la seconde branche du **if** mais pour la première (puisque la lambda réflexive est toujours une lambda réflexive). Ce faisant, il la remontera d'un niveau supplémentaire. Ce processus se réitérera ainsi indéfiniment. Pour que la lambda réflexive soit bien considérée comme une lambda appelée par I_n il faut donc la transformée en lambda classique avant de l'appeler.

7.4 Exemples d'utilisation

La tour finie d'interprètes, dotée des lambdas réflexives, est déjà suffisante pour se faire une idée de l'utilisation de la réflexivité dans les langages fonctionnels. Pour autant qu'un nombre suffisant de niveaux soit inséré dans la tour, la presque totalité de la réflexivité est d'ores et déjà utilisable.

Dans notre démarche, nous nous sommes servie d'un interprète d'un langage minimal. La réflexivité, parce qu'elle met les mécanismes de l'interprète à la disposition de l'utilisateur, permet d'en étendre aisément les possibilités. À l'aide d'une lambda réflexive, il est tout a fait possible d'augmenter l'interprète de nouvelles fonctionnalités. Un exemple souvent donné est celui de la fonction *boundp* qui teste si un symbole est lié dans l'environnement courant. Ceci ne peut être accompli qu'en accédant à l'environnement courant et nécessite donc une lambda réflexive. Avec notre interprète, cette fonction s'écrit:

```
(set! boundp
      (compound-to-reifier
         (lambda (e r k)
```

```
(get-pair (car e) r
          (lambda (pair) (k #t))
          (lambda () (k #f))))))
```

La lambda réflexive *boundp* extrait le premier élément de *e*, le symbole dont on veut savoir si il est lié dans l'environnement courant, et le passe à la fonction *get-pair* qui recherche ce symbole dans l'environnement. Selon le résultat de la recherche, cette dernière appellera l'une de ses continuations qui transmettra le résultat de la recherche à la continuation du calcul. C'est lors de cet appel que s'effectuera le retour au niveau d'appel de la lambda réflexive.

La lambda réflexive *boundp* s'utilise de la manière suivante:

```
(boundp a)  → #f
(let ((a 1)) (boundp a)) → #t
```

Un autre exemple d'utilisation des lambda réflexive est l'écriture d'un *call/cc* en L.

```
(set! call/cc
      (compound-to-reifier
        (lambda (e r k)
          (evaluate (car e) r
            (lambda (proc)
              (apply-proc proc
                          (list (compound-to-reifier
                                  (lambda (e1 r1 k1)
                                    (evaluate (car e1) r1 k))))))))))
```

La définition de *call/cc* laisse appraître deux lambda réflexive. La première est utilisée pour passer la continuation du calcul en paramètre de la fonction passée au *call/cc*. La seconde appelera la continuation du calcul *k* depuis l'interprète (c'est pour cela qu'une lambda réflexive est utilisée i.e. pour que *k* soit appelé au bon niveau) après avoir évalué l'expression *e1* dans l'environnement lexical *r1*.

À priori, les lambdas réflexives sont conçues pour modifier localement le processus d'interprétation. L'exemple de *boundp* illustre ce point : une fois *boundp* exécuté, l'interprétation reprends son cours normal. Nous appellerons ce type de réflexivité, la réflexivité locale.

Pourtant L, notre langage, permet d'effectuer des effets de bord. Et l'environnement global contient les définitions de l'interprète. En utilisant un effet de bord au sein d'une lambda réflexive, il est donc possible de modifier les définitions de l'interprète. Les définitions de l'interprète et l'interprète étant causalement connectés, cette modification doit se traduire par une modification effective et durable du fonctionnement de l'interprète. Prenons pour exemple le code suivant :

```
((compound-to-reifier
   (lambda (e r k)
     ((lambda (old-evaluate)
        (set! evaluate
              (lambda (e r k)
                (display e)
                (newline)
```

```
                        (old-evaluate e r k))))
            evaluate))))
```

L'exécution du code précédent modifie la définition de *evaluate* de telle manière que l'ensemble des expressions évaluées soient affichées à l'écran. Une fois retourné au niveau d'appel, toutes les expressions évaluées seront affichées à l'écran. La modification effectuée ne correspond pas à l'évaluation de la lambda réflexive, mais va au delà.

Ce type de réflexivité, parce qu'il permet une modification globale des mécanismes de l'interprète, est appelée réflexivité globale.

La réflexivité globale offre beaucoup plus de possibilités que la réflexivité locale. Elle rend possible la modification de la totalité des mécanismes de l'interprète. Il devient alors envisageable de changer complètement de langage interprété.

Enfin, parce que son pouvoir expressif est plus important, la réflexivité globale permet des applications comme l'introduction d'un mécanisme de trace des accès aux variables ou encore l'introduction des abstractions quasi-statiques [Simmons II 93]. Ces applications, qui nécessitent la modification de la gestion de l'environnement, ne sont pas envisageables dans le cadre de la réflexivité locale.

7.5 Passage à une tour infinie

L'interprète muni d'une tour finie ne peut être considéré comme un interprète réflexif complet. Il est en effet limité par le nombre de niveaux que l'utilisateur a introduit à la main dans la tour. Il est donc toujours possible, en utilisant le nombre adéquat de lambdas réflexives, d'atteindre Scheme. Pour rendre impossible la sortie de l'interprète et s'approcher au mieux du modèle, il faut rendre la tour infinie.

Le passage d'une tour finie à une tour infinie est plus délicat qu'il n'y parait et demande une attention particulière. Plus précisément, le caractère infini de la tour doit être implémenté dans un monde fini.

L'examen du comportement d'une tour finie permet de déduire deux choses:

- Le changement de niveau dans la tour a lieu lors de l'appel d'une lambda réflexive.
- L'interprète Scheme est atteint lorsqu'une lambda réflexive est appelée depuis l'interprète L qui se trouve juste au dessous.

Remarquons enfin que pour tout programme qui se termine, il existe un niveau n dans la tour tel que l'ensemble des niveaux allant de n à l'infini sont identiques. En effet, pour qu'un niveau ait été modifié, il faut que l'utilisateur y ait accédé. Or un programme qui se termine ne peut parcourir une tour infinie. Les interprètes des niveaux allant de n à ∞ sont donc exempt de toute modification effectuée par l'utilisateur. Par conséquent, ils sont tous définis de manière identique.

Une tour infinie est composée d'interprètes modifiés par l'utilisateurs pour les niveaux allant de 0 à $n-1$, et d'une infinité d'interprètes identiques pour les niveaux allant de n à ∞. L'implémentation d'une tour finie telle qu'elle a été décrite dans les sections précédentes convient parfaitement à l'implémentation des premiers niveaux de la tour (les niveaux allant de 0 à $n-1$). Dans notre cas, implémenté une tour infinie revient à implémenter les niveaux allant de n à ∞.

Il est évident qu'une infinité de niveaux ne peut être directement implémenté mais uniquement simulé. Notre tour ne sera donc que virtuellement infinie, l'essentiel étant que l'utilisateur ait l'impression qu'elle est infinie. Parce que tous les niveaux sont identiques et effectuent la même tache, un seul interprète est nécessaire pour simuler les niveaux allant de n à ∞. Cet interprète est un interprète méta-circulaire écrit en Scheme et qui interprète du L. Appelons I_{mc} cet interprète.

Notre tour est donc composée d'une tour finie pour les niveaux allant de 0 à $n-1$, et d'un unique interprète méta-circulaire I_{mc} qui simule les niveaux allant de n à ∞.

Le rôle de I_{mc} ne peut pourtant pas se limiter à la simulation des derniers niveaux de la tour. Lorsqu'une lambda réflexive est appelée depuis le niveau $n-1$, elle atteint I_{mc} si rien n'est fait pour l'en empêcher. Ce dernier doit donc insérer un niveau supplémentaire dans la tour à chaque fois qu'il interprète un appel à une lambda réflexive. En insérant un nouvel interprète à son niveau actuel et en remontant d'un niveau, il garantie ainsi qu'il ne sera jamais modifié par l'utilisateur et lui donnera l'illusion de manipuler une tour infinie.

L'insertion d'un niveau dans la tour soulève un certain nombre de problèmes. Les uns sont liés au nouvel état de I_{mc} lorsqu'il monte d'un niveau dans la tour et les autres à l'état de l'interprète inséré.

Lorsqu'il remonte d'un niveau, I_{mc} doit recevoir de nouvelles valeurs pour chacun de ses registres $(e\ r\ k)$. Pour connaître ces valeurs, il faut déterminer l'état d'un interprète I_{n+1} qui interprète un interprète I_n qui interprète un appel à une lambda réflexive (si si ...). À cet instant, l'activité de I_{n+1} se traduit par l'exécution de la fonction *evaluate-combination*. Plus précisément, I_n vient de déterminer que le premier élément de e est une lambda réflexive et se prépare à l'appeler comme une fonction de l'interprète. I_{n+1} est donc en train d'évaluer le code suivant:

```
'((reifier-to-compound proc) (cdr e) r k)
```

L'environnement lexical courant est aussi déterminé par l'activité de l'interprète. À chaque appel de fonction, un nouvel environnement lexical est construit à partir des paramètres de la fonction. Dans le cas de *evaluate-combination* ces paramètres sont $(er\ k)$. Mais l'appel d'un lambda réflexive n'a pas lieu depuis le corps de *evaluate-combination* mais depuis celui de la lambda qu'elle construit pour servir de continuation à *evaluate*. Cette continuation étant construite dans l'environnement lexical de *evaluate-combination*, son environnement lexical sera constitué de celui de *evaluate-combination* augmenté de son propre environnement lexical i.e. de $(e\ r\ k)$ et de $(proc)$.

Déterminer la continuation est un peu plus délicat et nous ne nous étendrons pas ici sur la justification de sa valeur. Vous trouverez une explication plus poussée dans [desRivières 84]. Jim des Rivières y montre qu'il suffit de prendre pour valeur de k une continuation qui représente la dernière partie de la boucle *read-eval-print* (soit *print*).

Pour résumer, lorsque I_n exécute l'expression correspondant à l'appel d'une lambda réflexive, I_{n+1} exécute l'expression suivante:

```
(evaluate '((reifier-to-compound proc) (cdr e) r k)
          <environnement-lexical {e r k}, {proc}>
          <continuation de read-eval-print>)
```

La fonction *evaluate-combination* de I_{mc} doit être modifiée en conséquence. L'appel d'un lambda réflexive se traduit alors par :

```
(set! evaluate-combination
   (lambda (e r k)
     (evaluate (operator-part e) r
               (lambda (proc)
                 (if (reifier? proc)
                   (call-reifier proc e r k)
                   (evaluate-operands (operands-part e) r
                        (lambda (args)
                           (apply-procedure proc args k))))))))
```

La fonction *call-reifier* se charge d'insérer un niveau supplémentaire dans la tour et de relancer le calcul :

```
(set! call-reifier
   (lambda (proc e r k)
        ; Incrémente le compteur de niveaux
      (set! depth (+ depth 1))
         ; Construit le nouvel environnement global de Imc
         ; qui contient l'environnement global du niveau inséré
      (set! global-env (extend (copy-env initial-env)
                               '(global-env)
                               global-env))
        ; Reprend le calcul un niveau au dessus
      (evaluate '((reifier-to-compound proc) (cdr e) r k)
                <environnement-lexical {e r k}, {proc}>
                <continuation de read-eval-print>)))
```

Il reste à clarifier la construction de l'environnement lexical ainsi que celle de la continuation. Cette construction n'est pas aussi directe qu'il y parait : I_{mc} est un interprète écrit en Scheme qui doit interpréter du L. Les données qu'il manipule sont donc nécessairement en Scheme. Cependant, l'interprète inséré est un interprète de L écrit en L. Il est donc indispensable d'opérer une conversion des données de Scheme en L avant de les passer à l'interprète inséré.

Nous allons voir maintenant deux techniques de conversion des données:

- La conversion à la volée.
- La conversion à la demande.

Chacune d'entre elles présente avantages et inconvénients et conditionne fortement l'écriture de I_{mc}.

Conversion à la volée

La conversion à la volée laisse I_{mc} manipuler ses registres comme des objets de L. Ainsi, lorsque l'interprète méta-circulaire doit remonter d'un niveau, il dispose des données dont il a besoin dans le langage manipulé par les interprètes des niveau inférieurs.

La difficulté réside dans la manipulation des continuations de l'interprète. Les autres registres, l'expression et l'environnement lexical, disposent d'une même représentation en Scheme et en L. Les continuations doivent elles être construites et appelées comme des objets de L. Le procédé consiste à découper les fonctions de l'interprète en séparant les fonctions de l'interprète des constructions des continuations et en utilisant une fonction dédiée pour chaque appel de continuation. Cette dernière devra alors traduire l'appel d'une continuation en L par l'appel d'une fonction Scheme équivalente.

Prenons pour exemple la fonction *evaluate-if*. Dans l'interprète méta-circulaire, cette fonction s'écrit :

```
(set! evaluate-if
      (lambda (e r k)
          (evaluate (test-part e) r
                    (lambda (v)     ; construction d'une continuation
                        (if v
                            (evaluate (then-part e) r k)
                            (evaluate (else-part e) r k))))))
```

La zone grisée correspond à la construction d'une continuation qui est passée à la fonction *evaluate*. Si la fonction *evaluate-if* est implémentée telle quelle, elle sera construite comme un objet de Scheme. Afin de la construire comme un objet de L, il faut substituer aux lignes de code en gris une fonction chargée de construire la continuation comme un objet de L:

```
(set! evaluate-if
      (lambda (e r k)
          (evaluate (test-part e) r
                    (make-evaluate-if-cc e r k))))
```

La fonction *make-evaluate-if-cc* construit la continuation comme un objet de L. Sa définition est :

```
(set! make-evaluate-if-cc
      (lambda (e r k)
```

```
        (list 'compound ; Construit la continuation comme un objet de L
              '(v)
              '(if v
                   (evaluate (then-part e) r k)
                   (evaluate (else-part e) r k))
              (extend empty-env
                      '(e r k)
                      (list e r k)))))
```

Les lambdas de L étant représentées par des listes, *make-evaluate-if-cc* construit une liste qui en L représente la continuation. Afin que l'interprète puisse appeler cette continuation, il lui faut disposer d'une fonction Scheme équivalente:

```
(set! evaluate-if-cc
      (lambda (v e r k)
        (if v
            (evaluate (then-part e) r k)
            (evaluate (else-part e) r k))))
```

À chaque fois qu'une continuation construite par I_{mc} devra être appelée, un appel à la fonction *evaluate-if-cc* lui sera substitué. Notons que les paramètres de cette dernière correspondent à la valeur passée à la continuation suivi du contenu de l'environnement lexical capturé lors de la construction de la continuation.

La substitution par des fonctions de Scheme des continuations de L est assurée par la fonction *call-cc*. Tous les appels à des continuations depuis l'interprète I_{mc} sont remplacées par un appel à la fonction *call-cc*. Par exemple, la fonction *evaluate-constant* qui a pour définition:

```
(set! evaluate-constant
      (lambda (e r k) (k (constant-part e))))
```

est réécrite en:

```
(set! evaluate-constant
      (lambda (e r k) (call-cc k (constant-part e))))
```

La fonction *call-cc* recherche dans une table la fonction Scheme équivalente.

La ligne qui réalise l'appel d'une fonction réflexive (dans *evaluate-combination*) est remplacée par

```
(call-reifier proc e r k)
```

avec

```
(set! call-reifier
      (lambda (proc e r k)
        (set! depth (+ depth 1))
        ((lambda (ge)
           (set! global-env (extend (copy-env initial-env)
                                    '(global-env)
                                    (list ge)))) global-env)
```

```
    (evaluate '((reifier-to-compound proc) (operands-part e) r k)
              (extend empty-env
                      '(proc e r k)
                      (list proc e r k))
              (make-openloop-cc depth))))
```

Étant donné que e, r et k sont des objets de L, ils peuvent être directement utilisés par l'interprète inséré.

Il faut noter que l'ensembles des modifications des définitions de I_{mc} ne sont pas visibles par l'utilisateur. Le *call-reifier* interdisant aux lambdas réflexives l'accès à ce niveau, l'utilisateur ne peut en examiner les définitions.

La conversion à la volée ralentit notablement l'interprète méta-circulaire. I_{mc} doit continuellement convertir des objets de L en Scheme et construire toutes les continuations qu'il manipule en L. Cependant, cette approche dispose d'un avantage notable : elle n'est pas dépendante du Scheme utilisé.

Conversion à la demande

Dans la conversion à la demande, la conversion des données de Scheme à L ne s'effectue que lorsqu'un niveau supplémentaire est inséré dans la tour. I_{mc} manipule ses données en Scheme et ne souffre plus du coût des conversions continuelles de Scheme à L et vice versa.

Un tel système de conversion repose entièrement sur l'implémentation de Scheme utilisée. La norme R4RS ne dispose pas des fonctionnalités nécessaires au support de ce système. Par contre, la plupart des implémentations disposent de fonctions d'introspections qui leur permettent de supporter ce système de conversion.

En Stk [Gallesio 96], la conversion à la demande repose sur les fonctions *procedure-body*, *uncode*, *environment->list* et *procedure-environment*.

La fonction *call-reifier* s'écrit alors:

```
(set! call-reifier
      (lambda (proc e r k)
         (set! depth (+ depth 1))
         ((lambda (ge)
             (set! global-env (extend (copy-env initial-env)
                                      '(global-env)
                                      (list ge)))) global-env)
         (evaluate '((reifier-to-compound proc) (operands-part e) r k)
                   (extend empty-env
                           '(proc e r k)
                           (cons proc (Stk-to-Ill e r k))0)
                   (make-openloop-cc depth))))
```

La conversion à la demande repose entièrement sur la fonction *Stk-to-Ill*. Cette fonction, que nous ne dévoilerons pas ici, traduit des objets de Scheme en objets de L.

Étant donné que les fonctions de I_{mc} ne subissent pas de découpage, l'interprète est plus efficace que dans le cas de la conversion à la volée. Cependant, ce type de conversion est fortement dépendante de l'implémentation du Scheme utilisé. Il nécessite la réécriture de la fonction de conversion lorsque l'interprète est porté sur une autre plate-forme.

8 Techniques d'optimisation

L'interprète réflexif qui vient d'être décrit, s'il est conforme au modèle initial décrit par B. C. Smith [Smith 83], souffre d'un lourd handicap : il se révèle particulièrement inefficace. L'empilement d'interprètes qui est au centre de son architecture se traduit par une accumulation des niveaux d'interprétation. Cette dernière s'oppose par essence à une exécution rapide des programmes réflexifs.

Nous allons décrire succinctement deux techniques qui permettent de traiter ce problème d'inefficacité.

8.1 Limitation du modèle

La technique la plus courante d'optimisation des performances des interprètes réflexifs consiste à limiter le modèle de réflexivité. Cette limitation consiste à réduire la réflexivité à la réflexivité locale. La suppression de la réflexivité globale se traduit par des interprètes identiques à tous les niveaux de la tour puisque non modifiables par l'utilisateur. La tour peut alors être implémentée par un unique interprète méta-circulaire qui monte et descend de niveau en niveau pour simuler l'ensemble de la tour.

Les définitions des fonctions des interprètes de chaque niveaux étant identiques, la connexion causale n'est plus complète. Elle se réduit aux registres de l'interprète. Un niveau est alors complètement décrit par les éléments de l'interprète susceptibles d'êtres modifiés, directement ou indirectement, par l'utilisateur. Ces éléments sont la continuation et l'environnement global (lorsqu'il est spécifique à chaque niveau, certains interprètes partageant un environnement global unique).

L'implémentation d'une telle tour est réalisée à l'aide d'un unique interprète qui monte et descend dans la tour au gré du code qu'il doit interpréter. Il est associé à une pile de continuation qui associe à chaque niveau de la tour la dernière continuation de l'interprète avant qu'il ne redescende d'un niveau.

Le changement de niveau est déterminé par la nature du code à interpréter : l'interprète monte d'un niveau lorsqu'il doit exécuter du code utilisateur (principalement, lors de l'appel d'une lambda réflexive), et redescend lorsqu'il doit interpréter son propre code (dont il connaît l'implémentation).

Cette approche permet de réduire le nombre de niveaux d'interprétation accumulés à un instant t. Elle est cependant restrictive sur les possibilités offertes par la réflexivité : il n'est désormais plus possible de modifier l'interprète. La réflexivité n'agit plus que sur le cours de l'interprétation.

8.2 Évaluation partielle

L'évaluation partielle

L'évaluation est une technique de transformation de programme qui permet de spécialiser un programme en fonction de la connaissance d'une partie de ses paramètres d'entrée. À un programme $P(e_1...e_n)$, elle fait correspondre un programme $P_s(e'_1...e'_n)$ où e'_i est soit désigné comme étant inconnu, soit comme étant une valeur v_i connue de e_i. Les programmes P et P_s sont alors liés par l'équation suivante

$$P(e_1...e_n) = P_s(e'_1...e'_n) \quad avec \quad e_i = e'_i \quad et \quad e_i = v_i \quad si\ v_i\ existe$$

Le programme spécialisé s'obtient en effectuant tous les calculs qu'il est possible de faire statiquement i.e. avant l'interprétation, à partir de la connaissance du programme à spécialiser, P, et des valeurs de certains de ses paramètres d'entrée. Par exemple, si pour l'expression $(+\ a\ b\ c)$ les valeurs de de a et de b sont connues et valent, respectivement, 3 et 4, alors la somme peut être remplacée par $(+\ 7\ b)$.

Généralement, l'évaluation partielle est appliquée à des fonctions complexes dont certains des paramètres d'entrée sont connus à l'avance. Par exemple, prenons une fonction qui concatène deux listes:

```
(define (my-append l1 l2)
   (if (null? l1)
       l2
       (cons (car l1) (my-append (cdr l1) l2))))
```

La fonction *my-append* spécialisée pour *l1* ayant pour valeur '(1 2 3) donne:

```
(define (my-append1 l1 l2)
   (cons '1 (cons '2 (cons '3 l2))))
```

Mais cette même fonction spécialisée pour *l2* ayant pour valeur '(1 2 3) donne:

```
(define (my-append2 l1 l2)
   (if (null? l1)
       '(1 2 3)
       (cons (car l1) (my-append2 (cdr l1) l2))))
```

Dans ce dernier cas le gain n'est plus significatif. Cet exemple montre que l'utilisation de l'évaluation partielle n'est pas triviale et demande une certaine compréhension des mécanismes qui la régisse ainsi que du fonctionnement du programme à spécialiser.

L'évaluation partielle montre toutes ses capacités lorsqu'elle est appliquée à des mécanismes interprétatifs. Prenons pour exemple une petite calculatrice en notation polonaise. *calc* accepte deux paramètres, un programme qui définit les opérations à effectuer et une liste de valeurs. Sa définition est:

```
(define (calc prgm values)
  (letrec
```

```
((compute
   (lambda (stack exprs env)
     (if (null? exprs)
         (car stack)
         (let ((expr (car exprs)))
           (cond ((number? expr)
                   (compute (cons expr stack) (cdr exprs) env))
                  ((memq expr '(+ - * /))
                   (compute (cons
                              (case expr
                                ((+) (+ (car stack) (cadr stack)))
                                ((+) (+ (car stack) (cadr stack)))
                                ((+) (+ (car stack) (cadr stack)))
                                ((+) (+ (car stack) (cadr stack))))
                              (cddr stack))
                            (cdr exprs) env)))
                  (else
                    (let ((pair (assq expr env)))
                      (compute (cons (cdr pair) stack) (cdr exprs) env)))))))))
  (compute '(0) (cdr prgm) (extend '() (car prgm) values)))
```

L'évaluation partielle de *calc* avec un premier paramètre égal à:

```
'((a b) a 2 * b 3 * +)
```

et un second paramètre étant spécifié comme inconnu donne:

```
(define (calc-1 prgms values)
  (+ (* 3 (car (cdr values))) (* 2 (car values))))
```

L'évaluation partielle a supprimé tous les mécanismes interprétatifs. Il ne subsiste du programme initial que l'extraction des valeurs du second paramètre. Cet exemple montre clairement l'intérêt de l'évaluation partielle dans le cadre des interprètes. C'est en cela que son application aux interprètes réflexifs est prometteuse.

Application de l'évaluation partielle à la réflexivité

L'utilisation de l'évaluation partielle dans le cadre de la réflexivité n'en est qu'à ses balbutiements. Elle repose sur la capacité d'un évaluateur partiel à compiler un langage L lorsqu'il est associé à un interprète de L. Nous ne donnerons ici qu'une ébauche de son utilisation.

Un tour d'interprète peut être vu comme une série d'interprètes qui s'interprètes les uns les autres.

$$(\ldots(I_{n+1}(I_n(I_{n+1}(\ldots(I_1(I_0\ldots)\ldots)\ldots)\ldots)$$

Un évaluateur partiel peut être utilisé pour réduire la tour d'un niveau :

$$PE(I_n, I_{n-1}) = I_{n,n-1}$$

En réitérant le procédé de niveau en niveau on obtient :

$$PE(...(I_{n+1}(I_n(I_{n-1}(...(I_1(I_0...)...)...) = I_{...,n+1,n,n-1,...,1,0}$$

Le résultat est la réduction de la tour à un seul niveau d'interprétation.

Le gain potentiel en performance est important : la réduction de la tour conduit à la suppression des niveaux d'interprétation source de l'inefficacité des systèmes réflexifs.

Il n'y a pratiquement pas de pertes sinon que le code, une fois compilé, ne peut être directement modifié.

Cependant, l'évaluation partielle est une technique encore jeune qui souffre de nombreuses limitations. En particulier, l'absence d'un traitement complet des effets de bord ne permet pas d'envisager son utilisation dans le cadre d'un interprète réflexif complet.

9 Conclusion

L'objectif de ce cours était de vous donner un aperçu de la réflexivité dans le monde des langages fonctionnels. À la fin de ce cours, vous devriez être capable d'écrire un petit interprète réflexif. Au besoin, vous pouvez completer vos connaissances par lecture de l'article de Jefferson [Jefferson 92].

Bibliographie

[Bawden 88] Alan Bawden. – Reification without evaluation. *the 1988 ACM Conference on Lisp and Functional Programming.* ACM, pp. 342–351. – Snowbird, Utah, Juillet 1988.

[desRivières 84] Jim des Rivières et Brian Cantwell Smith. – The Implementation of Procedurally Reflective Languages. *Proceedings of the Symposium on Lisp and Functional Programming*, éd. par New York: ACM Press, pp. 331–347. – 1984.

[Gallesio 96] Erick Gallesio. – *STk Reference Manual (Version 3.1).* – Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S), Université de Nice-Sophia Antipolis, Juillet 1996.

[Jefferson 92] Stanley Jefferson et Daniel P. Friedman. – A Simple Reflective Interpreter. *Proceedings of the International Workshop on Reflection and Meta-Level Architectures*, pp. 24–35. – Novembre 1992.

[Simmons II 93] John Wiseman Simmons II et Daniel P. Friedman. – First Class-interpreters: Illustrating the Limits Imposed by Representation in a Reflective Language. *OOPSLA Workshop on Reflection and Meta-Level Archictectures.* ACM. – 1993.

[Smith 83] Brian Cantwell Smith. – Reflection and Semantics in Lisp. *Conference Record of the ACM Symposium on Principles of Programming Languages.* pp. 23–35. – ACM Press, 1983.